ОГЛАВЛЕНИЕ

Стр.

ВВЕДЕНИЕ

В настоящее время все больше внимания уделяется разработке программ гарантированного качества. Поскольку цена любой ошибки определяется частотой ее проявления, сложностью поиска причины ее возникновения и сложностью ее устранения, существуют классы программ, для которых критически важно гарантировать отсутствие максимально широкого класса ошибок. К таким программам относятся, например, все программы из стека разработки программного обеспечения: интерпретаторы, трансляторы, оптимизаторы, генераторы и компоновщики кода.

В последние несколько лет в среде разработчиков компиляторов этому факту уделяется все больше внимания. Например, разработаны верфицированный компилятор языка программирования C [1] и кодогенератор с доказанно корректными оптимизациями [2].

При этом, однако, все известные средства верифицированной трансляции ставят целью доказательство корректности производимых преобразований, предполагая абсолютную корректность этапа выбора машинных инструкций и последующей компоновки.

Целью настоящей работы является разработка метода формальной верификации статической и динамической компоновки блоков ассемблерного кода с сохранением семантики с точностью до ABI.

В первой главе представлено введение в генерацию кода на языке ассемблера, ABI, компоновку (линковку) и формально верифицированные трансляторы, рассмотрены существующие решения в данных областях, их возможности и ограничения.

Во второй главе представлены основные этапы разработки метода формальной верификации компоновки блоков ассемблерного кода.

В третьей главе представлен конечный вариант разработанного метода формальной верификации статических и динамических компоновщиков, позволяющий доказать, что для корректно заполненной таблицы смещений (GOT) вызов элемента таблицы процедурной линковки (PLT) эквивалентен непосредственному вызову функции.

Весь программный код, представленный в настоящей работе, оформлен в стиле Literate Agda[3].

ГЛАВА 1

ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Для программ, относящихся к стеку разработки программного обеспечения, критически важно гарантировать отсутствие как можно большего числа возможных ошибок, ведь цена ошибки в этой области чрезвычайно высока. В настоящее время уже существуют различные проекты, направленные на избежание ошибок в программах стека разработки.

В данной главе представлены обзор существующих проектов и введение в кодогенерацию и компоновку программ.

1.1 От исходного кода к исполнению на CPU

Готовый исходный код проходит через несколько стадий трансляции перед непосредственным исполнением на CPU:

- трансляция из первоначального языка программирования в первое промежуточное представление;
- последовательная трансляциия между промежуточными представлениями;
- трансляция в ассемблер с выбором ABI;
- трансляция из языка ассемблера в машинный код;
- упаковка машинного кода в объектные файлы;
- отображение объектных файлов в память;
- линковка загруженного в память кода.

На рисунке 1.1 изображена диаграмма всевозможных преобразований программы. Вершинами представленного графа являются состояния программы и/или исполнителя (в зависимости от формализма языка программирования, например, в лямбда-исчислении всем состоянием исполнителя является вся программа, в то время как в языке ассемблера состоянием исполнителя являются регистры и память). В направлении оси абсцисс располагаются ребра, вверху графа представляющие собой оптимизации программы, а внизу — исполнение кода на CPU[1]. В направлении оси ординат распола-

[1]С точки зрения теории языков программирования вычисление на CPU представляет собой последовательное применение к состоянию вычислителя некоторого простого набора редукций, а оптимизации кода на первоначальном языке программирования можно рассматривать как некоторый набор нетривиальных правил редукций.

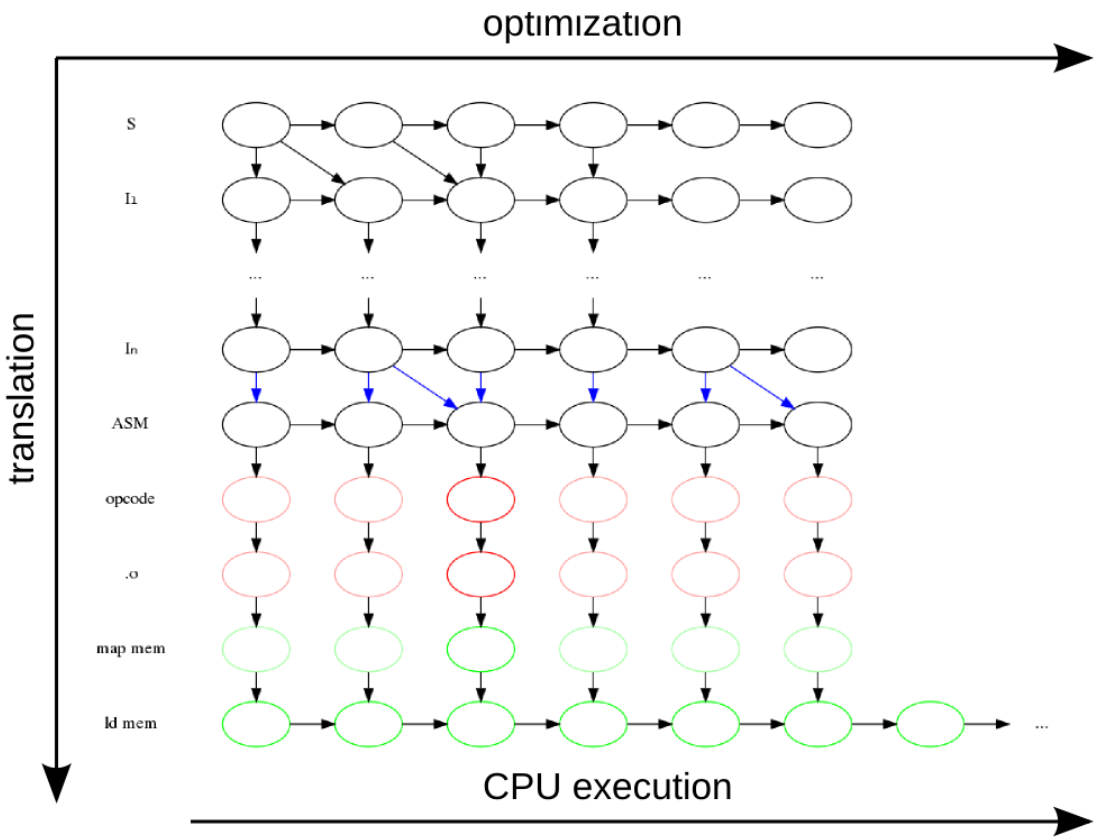

Рис. 1.1 — Диаграмма преобразований программы

гаются ребра, соответствующие различным этапам трансляции кода между следующими представлениями:

- S — код на первоначальном языке программирования;
- I_1, …, I_n — код на некотором промежуточном языке;
- *ASM* — код на языке ассемблера;
- *opcode* — машинный код;
- *.o* — скомпилированные файлы с исполняемым кодом;
- *mapmem* — код, загруженный в память;
- *ldmem* — пролинкованный код в памяти.

Конечной целью разработки доказанно корректного стека разработки программного обеспечения является доказательство коммутативности диаграммы, представленной на рисунке 1.2. Заметим, однако, что в этой диаграмме участвует корректность спецификации машинного кода производителями CPU, а также технический процесс производства конкретного устройства, на котором производятся вычисления, что не поддается формальному доказательству. Поэтому большая часть усилий разработчиков доказанно коррект-

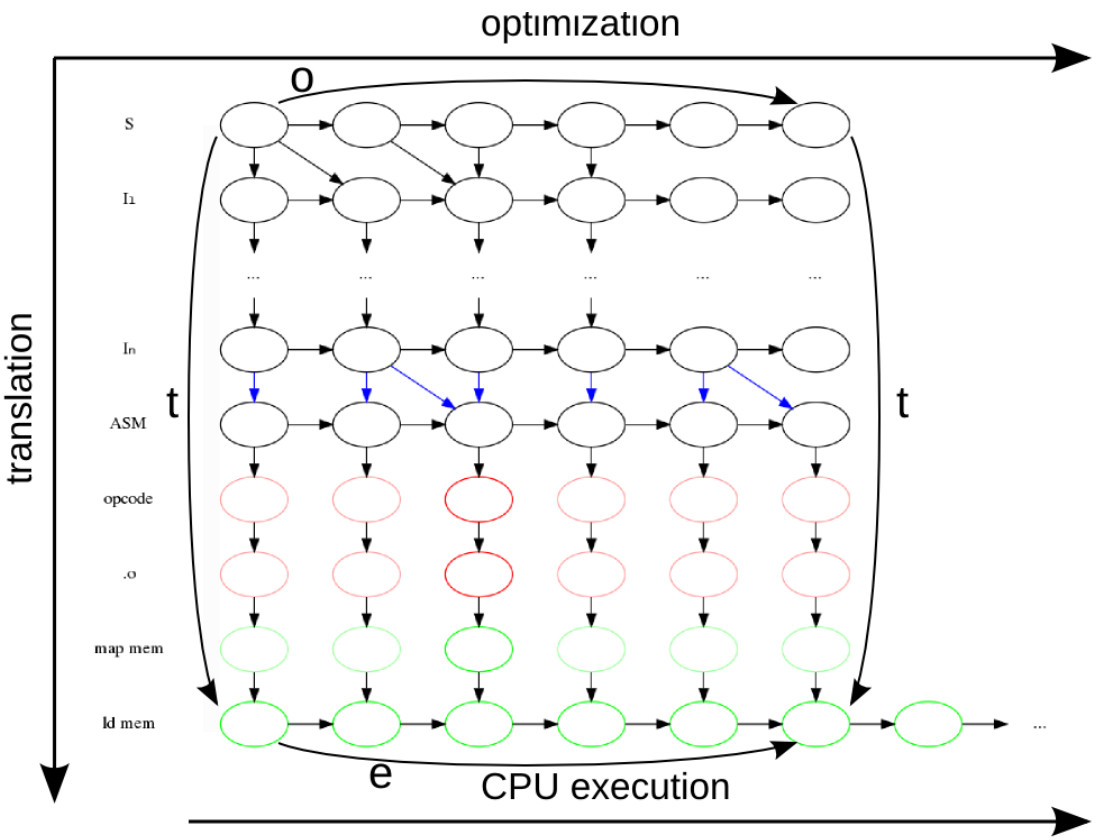

Рис. 1.2 — Коммутативная диаграмма преобразований программы

ных программ стека разработки программного обеспечения полностью игнорирует нижнюю половину диаграммы, начиная с уровня *opcode*.

Целью настоящей работы является доказательство коммутативности диаграммы, охватывающей область от уровня *ASM* до уровня *ldmem* (рисунок 1.3) в предположенях, что:

- бинарный код, исполняемый CPU, соответствует семантике заданного ассемблера;
- конкретное вычислительное устройствое произведено в полном соответствии с техническим процессом и не имеет ошибок;
- операционная система настраивает таблицу трансляции виртуальных адресов в соотвествии с расположением ассемблерного кода в виртуальной памяти.

Строго говоря, первый и последний пункты данного списка поддаются формальной верификации:

- первый — при наличии исходного кода интерпретатора бинарного кода, компилируемого в транзисторы производителями CPU;

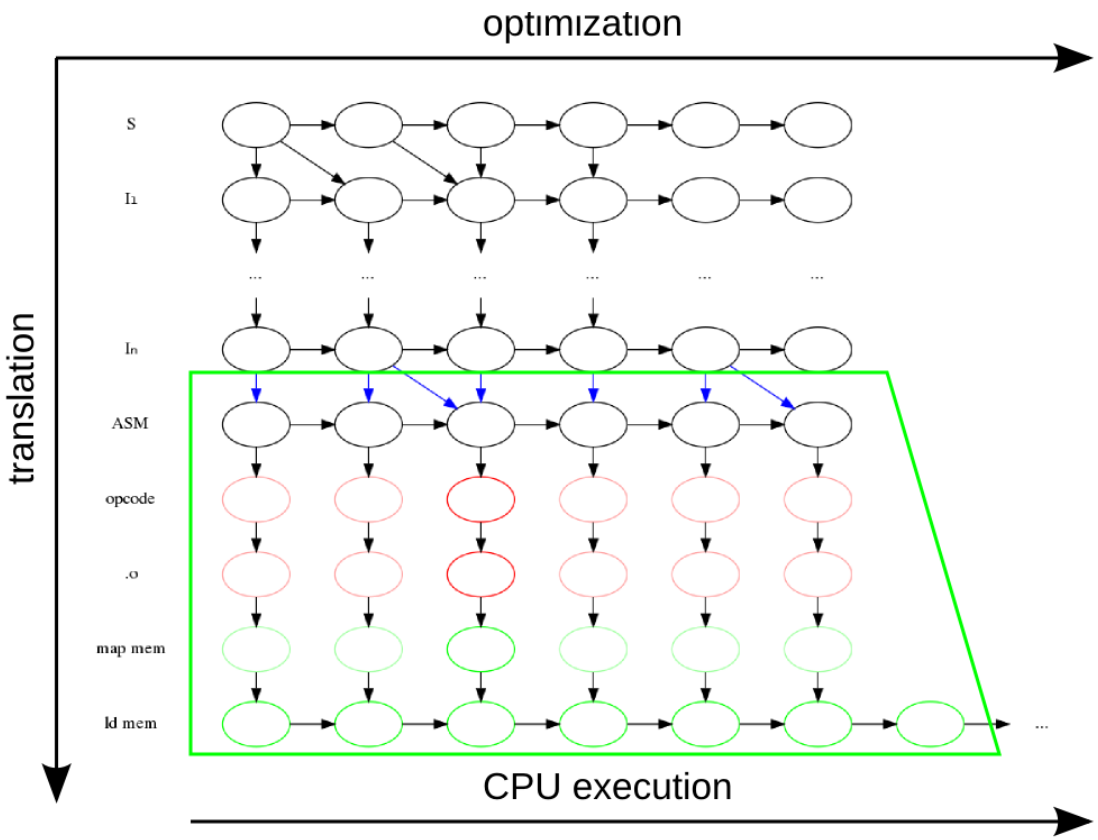

Рис. 1.3 — Рассматриваемая в настоящей работе область диаграммы

- последний — при наличии доказанно корректного ядра операционной системы (такие проекты существуют [4]) с доказанно корректной подсистемой виртуальной памяти (такие проекты автору данной работы не известны).

Таким образом, при наличии транслятора из первоначального языка программирования в язык ассемблера, для которого доказана коммутативность диаграммы, охватывающей область от уровня S до уровня ASM (рисунок 1.4), благодаря тривиальной теореме из алгебры/теории категорий[2] в указанных выше предположениях будет достигнута конечная цель разработки доказанно корректного стека разработки программного обеспечения, указанная выше.

1.2 Основные определения

Определим используемые в дальнейшем термины.

[2] $(g \circ f \equiv k \circ h) \rightarrow (m \circ k \equiv s \circ i) \rightarrow (m \circ g \circ f \equiv s \circ i \circ h)$

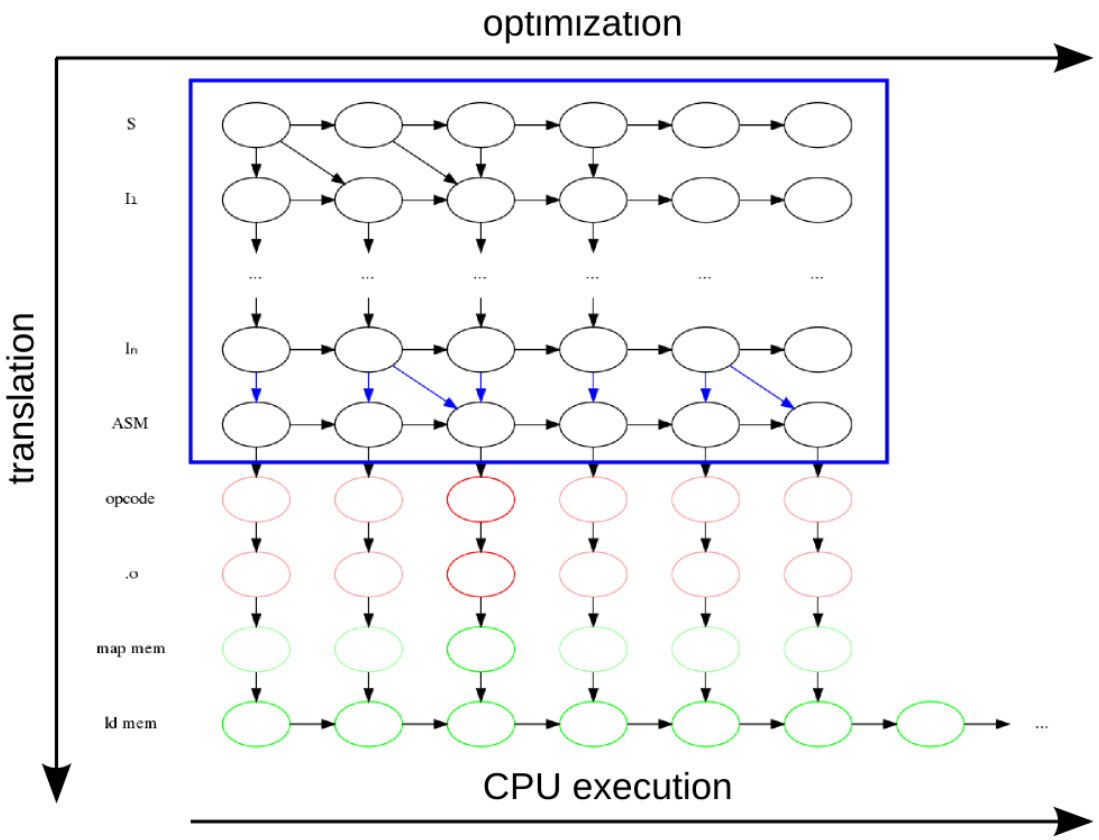

Рис. 1.4 — Рассматриваемая в доказанно корректном компиляторе область диаграммы

Язык ассемблера — машинно-ориентированный язык, предназначенный для представления в удобочитаемой форме программ, записанных в машинном коде. Каждая команда языка ассемблера соответствует какой-либо команде исполнителя. Для краткости будем называть язык ассемблера *ассемблером*.

SSA-представление (static single assignment form) — промежуточное представление кода, в котором каждой переменной значение присваивается не более одного раза.

Компоновка (линковка) — сборка исполняемого файла из нескольких объектных файлов, в процессе которой происходит проставление необходимых ссылок на внешние функции и переменные. При *статической* линковке код всех исходных объектных файлов включается в результирующий файл с уже заполненными ссылками на переменные и функции. При *динамической* линковке в объектные файлы добавляется только необходимая метаинформация и, возможно, дополнительные фрагменты исходного кода, а непосредственное заполнение ссылок производится при загрузке программы и библиотек *динамическим загрузчиком*.

К дополнительным фрагментам исходного кода, добавляемых при динамической линковке, относятся таблицы *GOT (global offset table)* и *PLT (procedure linkage table)*.

Простой динамической линковкой будем называть динамическую линковку, при которой вызовы внешних функций заменяются вызовом соответствующих им элементов PLT, а сам элемент PLT состоит из одной инструкции: непрямой `jmp` на соответствующий элемент GOT.

1.3 Существующие проекты

В настоящем разделе рассмотрены некоторые проекты, доказывающие коммутативность диаграмм, охватывающих различные уровни от *S* до *ASM* включительно на рисунке 1.1.

1.3.1 Software Foundations

Software Foundations [5] — курс по математическим основам надежного программного обеспечения. Покрывает основные понятия логики, доказательство теорем с помощью Coq[6], функциональное программирование и статические системы типов, не требуя при этом предварительного углубленного изучения логики или языков программирования.

Отличительной особенностью этого курса является то, что он полностью формализован и ориентирован на чтение и выполнение упражнений внутри интерактивной сессии Coq.

1.3.2 Iron Lambda

Iron Lambda [7] — набор доказательств на Coq для функциональных языков программирования различной сложности. Покрывает несколько распространенных систем типов. В их число входят просто типизированное лямбда-исчисление, SystemF и некоторые их расширения.

1.3.3 CompCert

CompCert [1] — исследовательский проект, направленный на разработку формально верифицированных компиляторов, применимых для встраиваемого программного обеспечения. Такие компиляторы предоставляют machine-checked доказательства того, что генерируемый ими код имеет ту же семантику, что и исходный код.

Основным результатом этого проекта является верифицированный компилятор языка C, покрывающий почти весь стандарт ISO C90 и генерирующий эффективный код для архитектур PowerPC, ARM и x86.

1.3.4 Vellvm

Vellvm (Verified LLVM) [2] представляет собой фреймворк для доказательства утверждений о программах, записанных в промежуточном представлении LLVM. Содержит формализацию промежуточного представления LLVM, его системы типов и свойств его SSA-представления.

1.3.5 Typed Assembly Language

В статье [8] описаны статически типизированный язык ассемблера (TAL) и сохраняющая типы трансляция в него из SystemF. TAL основан на обычном RISC ассемблере, но его система типов обеспечивает поддержку языковых абстракций высокого уровня (например, замыкания, кортежи и пользовательские типы данных). При этом TAL не накладывает почти никаких ограничений на оптимизации более низкого уровня, таких как аллокация регистров и выбор инструкций.

Типизированный ассемблер можно применять для написания верифицированных компиляторов и доказательства различных теорем про нетривиальные ассемблеры (с оптимизациями, преобразованиями между ABI и так далее). Однако, в указанной статье это направление исследований не рассматривается.

1.4 Предлагаемый метод решения поставленной задачи

Поставленную задачу планируется решить, формализовав типизированный ассемблер, содержащий необходимое для реализации простой динамической линковки подмножество инструкций ассемблера x86-64, с помощью реализованного ассемблера формализовать подмножество ABI, касающееся линковки, ввести понятие эквивалентности блоков кода и доказать требуемую эквивалентность.

1.5 Выводы по главе

В данной главе было представлено введение в кодогенерацию и линковку, а так же был дан краткий обзор существующих проектов, покрывающих различные уровни процесса трансляции исходного кода в исполняемый машинный код.

ГЛАВА 2

ОСНОВНЫЕ ЭТАПЫ РАЗРАБОТКИ МЕТОДА

В этой главе представлены основные этапы разработки фреймворка, позволяющего формализовать подмножество ассемблера, необходимого для описания элементов PLT, определить понятие эквивалентности блоков кода и доказывать эквивалентность структурно различных блоков кода.

2.1 Основные определения

Определим набор основных сущностей, используемых в дальнейшем, и опишем типы, позволяющие гарантировать корректность исполнения правильно типизированных программ.

Исполнение кода производит некоторые манипуляции с данными, поэтому основной используемой сущностью являются значения каких-либо типов. Значения могут располагаться либо в регистрах, либо в памяти. Для упрощения реализации динамически аллоцируемая память не рассматривается, вся требуемая память предварительно аллоцируется в секции данных бинарных файлов.

Все значения, расположенные либо в памяти, либо в регистрах, должны иметь некоторый тип. Также стоит описывать состояние набора регистров, так как блоки кода могут рассчитывать на наличие в регистрах значений конкретных типов. То же верно и для состояния памяти.

```
data Type : Set
RegTypes : Set
DataType : Set
```

Состояние регистров представим списком типов, причем каждому регистру должен соответстовать элемент списка. В этом случае именем регистра является индекс в этом списке.

Аналогичные рассуждения верны и для состояния памяти.

```
RegTypes = List Type
DataType = List Type
```

Рассматриваемыми значениями являются только блоки кода и указатели на некоторое значение, расположенное в памяти. При этом корректность ис-

полнения типизируемого блока кода возможно гарантировать, описав в его типе состояние регистров и памяти, которые должны быть на момент начала исполнения блока. Для простоты считаем, что типы значений, расположенных в памяти, никогда не изменяются, поэтому тип блока описывает только требуемое состояние регистров.

```
data Type where
  _* : Type → Type
  block : (Γ : RegTypes) → Type
```

2.2 Первая реализация: SSA

Первым вариантом реализации является реализация типизированного ассемблера в SSA-представлении и доказательство на его основе эквивалентности блока кода PLT и соответствующей ему функции.

SSA-представление предполагает, что регистры никогда не изменяют свой тип, а инструкции, требующие этого (например, загрузка значения из памяти в регистр), просто добавляют еще один регистр в контекст. При этом предполагается, что процессор при исполнении этого кода способен сам разобраться, как сопоставить регистры из контекста реальным регистрам.

Так же в данной реализации память считается неизменяемой.

2.2.1 Блоки, инструкции и значения

Как говорилось ранее, значения, лежащие в памяти, никогда не изменяются. При этом все определения, касающиеся кода, могут на ссылаться на память, потому целесообразно поместить эти определения в модуль с фиксированным состоянием памяти.

```
module FixedHeap (Ψ : DataType) where
```

Не все инструкции изменяют одинаковые части состояния исполнителя. Некоторые инструкции изменяют состояние регистров, некоторые определяют, какой код будет исполняться дальше. Второй тип инструкций будем называть *управляющими инструкциями*.

Принимая во внимание тот факт, что с точки зрения компоновки необходимо рассматривать лишь переходы между различными частями загруженного в память кода, удобно ввести понятие *базового блока кода* — последовательности инструкций, не содержащей никаких переходов, и впоследствии рассматривать лишь переходы между этими блоками. В терминах различных видов инструкций определение блока описывается как последовательность инструкций, меняющих регистры, заканчивающаяся управляющей инструкцией.

Блок должен описывать, в каком состоянии исполнителя он будет корректно исполнен и как он его изменит. Изменением состояния исполнителя является список добавляемых регистров.

```
data Block (Γ : RegTypes) : (Δ : RegTypes) → Set
```

Управляющая инструкция должна знать, при состоянии регистров она исполняется, чтобы гарантировать, что блок, на который происходит переход, будет исполнен корректно.

```
data ControlInstr (Γ : RegTypes) : Set
  where
```

Определим несколько управляющих инструкций, которые понадобятся в дальнейшем:

- вызов соответствующего блока кода, находящегося в памяти по данному адресу;

  ```
  call  : (f : block Γ ∈ Ψ) → ControlInstr Γ
  ```

- передача управления на блок кода, адрес которого записан в ячейке памяти по данному адресу;

  ```
  jmp[_] : (f : (block Γ) * ∈ Ψ) → ControlInstr Γ
  ```

- передача управления на блок кода, расположенный в памяти по данному адресу.

  ```
  jmp  : (f : block Γ ∈ Ψ) → ControlInstr Γ
  ```

Стоит отметить, что последняя инструкция не требуется для реализации простой динамической компоновки и приведена здесь в качестве примера возможной инструкции.

Определим тип, описывающий инструкции. Инструкция, как и управляющая инструкция, должна знать, при каком состоянии регистров она исполняется. Но, в отличие от управляющей инструкции, она может его менять.

Поэтому тип инструкции должен дополнительно описывать, как он меняет состояние регистров. В данном случае это будет описываться списком регистров, добавляемых к уже имеющимся.

```
data Instr (Γ : RegTypes) : (Δ : RegTypes) → Set
```

Для реализации простой динамической компоновки требуются только управляющие инструкции, описанные выше. Тем не менее, стоит определить хотя бы одну инструкцию для демонстрации того, как код может выглядеть в данной реализации. В качестве такой инструкции была выбрана инструкция загрузки значения в регистр. Но для ее описания требуется определить, чем же являются значения.

```
data Value : Type → Set where
```

Как уже говорилось ранее, возможными значениями являются:

- блоки кода;

```
block : {Γ Δ : RegTypes} → Block Γ Δ → Value (block Γ)
```

- указатели на значения, лежащие в памяти.

```
ptr  : ∀ {τ} → τ ∈ Ψ → Value (τ *)
```

Теперь можно определить описанную выше инструкцию. Она принимает произвольное значение типа τ и добавляет к имеющимся регистрам один регистр этого типа.

```
data Instr (Γ : RegTypes) where
  mov  : ∀ {τ} → Value τ → Instr Γ [ τ ]
```

Опишем конструирование базовых блоков кода.

```
data Block (Γ : RegTypes) where
```

Базовый блок — это конструкция, построенная из следующих примитивов:

- блок, совершающий переход куда-либо в соответствии с результатом исполнения управляющей инструкции;

```
  ⇝  : ControlInstr Γ → Block Γ []
```

- блок, исполняющий инструкцию и продолжающий исполнение заданным блоком в измененном состоянии исполнителя.

```
  _·_ : ∀ {Δ₁ Δ₂} → Instr Γ Δ₁ → Block (Γ ++ Δ₁) Δ₂
      → Block Γ (Δ₂ ++ Δ₁)
```

Для удобства определим тип блока, для которого заранее не известно, на какое состояние исполнителя он рассчитывает и как он его меняет. Это может

быть полезным, например, при определении функции, загружающей произвольный блок кода из памяти.

```
NewBlk = Σ RegTypes (λ Γ → Σ RegTypes (Block Γ))
```

2.2.2 Память

Определим структуру, представляющую память. Тип этой структуры описывает, значения каких типов и на каких позициях находятся в памяти.

```
data Data : DataType → Set where
```

Определение памяти практически аналогично определению односвязного списка. Разница состоит только в том, что значение, добавляемое к списку, может ссылаться на какие-либо значения из этого списка, что указано в его типе.

```
[]  : Data []
_,_ : ∀ {τ Ψ} → (H : Data Ψ) → Value Ψ τ → Data (τ :: Ψ)
```

Для работы с памятью необходимо определить разыменование указателя, то есть получение значения по указателю на него:

```
deref : ∀ {l Ψ} → Data Ψ → l * ∈ Ψ → l ∈ Ψ
deref [] ()
deref (vs , block x) (here ())
deref (vs , ptr p)  (here refl) = there p
deref (vs , x)      (there p)  = there (deref vs p)
```

Для каждой возможной сущности в старом наборе данных определено корректное преобразование в сущность в новом наборе данных:

```
module WeakeningLemmas {Ψ Ψ' : DataType} (ss : Ψ ⊆ Ψ') where
```

- значение из старого набора данных корректно преобразуется в значение в новом наборе данных (реализация приведена ниже);

  ```
  wk−value : ∀ {τ} → Value Ψ τ → Value Ψ' τ
  ```

- инструкция из старого набора данных корректно преобразуется в инструкцию в новом наборе данных;

  ```
  wk−instr : ∀ {Γ Δ} → Instr Ψ Γ Δ → Instr Ψ' Γ Δ
  wk−instr (mov x) = mov (wk−value x)
  ```

- управляющая инструкция из старого набора данных корректно преобразуется в управляющую инструкцию в новом наборе данных;

```
wk-cinstr : ∀ {Γ} → ControlInstr Ψ Γ → ControlInstr Ψ′ Γ
wk-cinstr (call f) = call (ss f)
wk-cinstr jmp[ f ] = jmp[ ss f ]
wk-cinstr (jmp f) = jmp (ss f)
```

- блок кода из старого набора данных корректно преобразуется в блок кода в новом наборе данных.

```
wk-block : ∀ {Γ Δ} → Block Ψ Γ Δ → Block Ψ′ Γ Δ
wk-block (⇝ x) = ⇝ (wk-cinstr x)
wk-block (x · b) = wk-instr x · wk-block b
```

Реализация описанной выше функции `wk-value`:

```
wk-value (block x) = block (wk-block x)
wk-value (ptr x)  = ptr (ss x)
```

Определим функцию для загрузки значения из памяти по указанному адресу:

```
load : ∀ {l Ψ} → Data Ψ → l ∈ Ψ → Value Ψ l
load (vs , x) (here refl) = wk-value there x
load (vs , x) (there p)  = wk-value there (load vs p)
```

Так же для удобства определим функцию, загружающую произвольный блок кода:

```
loadblk : ∀ {Γ Ψ} → Data Ψ → block Γ ∈ Ψ → NewBlk Ψ
loadblk Ψ f with load Ψ f
loadblk Ψ f | block x = _ , _ , x
```

2.2.3 Исполнение кода

Далее определим, что происходит непосредственно во время исполнения кода.

Стек вызовов можно представить как список адресов блоков в памяти. Для простоты определим его как список блоков, а не их адресов.

```
CallStack : DataType → Set
CallStack Ψ = List (NewBlk Ψ)
```

Контекстом исполнения назовем пару из стека вызовов и указателя на блок, который должен исполняться вслед за текущим. Это важно, потому что результат исполнения некоторых инструкций зависит от того, какой блок

расположен вслед за текущим. Например, такой инструкцией является вызов функции, при котором на стеке вызовов оказывается адрес возврата.

```
CallCtx : DataType → Set
CallCtx Ψ = CallStack Ψ × NewBlk Ψ
```

Теперь можно описать, что именно делают определенные выше инструкции в некотором определенном контексте исполнения и фиксированном состоянии памяти.

```
module InCallCtx {Ψ : DataType} (H : Data Ψ) (cc : CallCtx Ψ)
  where
```

Результатом исполнения управляющей инструкции является измененный контекст исполнения.

```
exec-control : ∀ {Γ} → ControlInstr Ψ Γ → CallCtx Ψ
```

Вызов функции добавляет адрес возврата на стек и следующим исполняемым блоком делает загруженную из памяти функцию:

```
exec-control (call f) = proj₂ cc :: proj₁ cc , loadblk H f
```

Непрямой `jmp` не меняет стек и передает управление на блок, загруженный из памяти по разыменованному указателю:

```
exec-control jmp[ f ] = proj₁ cc , loadblk H (deref H f)
```

`jmp` не меняет стек и передает управление так же, как это делает вызов функции:

```
exec-control (jmp f) = proj₁ cc , loadblk H f
```

Результатом исполнения блока является не только измененный контекст исполнения, но и измененное состояние регистров. Но в данной реализации рассматриваются только переходы между блоками и никак не учитывается состояние регистров, что отражено в описании результата исполнения блока кода.

```
exec-block : ∀ {Γ Δ} → Block Ψ Γ Δ → CallCtx Ψ
exec-block (↝ x) = exec-control x
exec-block (i · b) = exec-block b
```

2.2.4 Эквивалентность блоков кода

Используя описанные выше определения, можно ввести понятие эквивалентности блоков кода: два блока считаются *эквивалентными в одном кон-*

тексте исполнения, если для обоих блоков указана последовательность исполнения, приводящая к одному и тому же блоку с одинаковым контекстом исполнения.

```
data BlockEq {Ψ : DataType} (H : Data Ψ) (CC : CallCtx Ψ)
  : {Γ₁ Γ₂ Δ₁ Δ₂ : RegTypes}
  → Block Ψ Γ₁ Δ₁ → Block Ψ Γ₂ Δ₂ → Set
  where
```

Два блока эквивалентны, если:

- они одинаковы;

```
  equal : ∀ {Γ Δ} → {B : Block Ψ Γ Δ} → BlockEq H CC B B
```

- исполнение первого из них приводит к блоку, эквивалентному второму;

```
  left : ∀ {Δ₁ Δ₂ Δ₃ Γ₁ Γ₂ Γ₃}
    → {A : Block Ψ Γ₁ Δ₁} → {B : Block Ψ Γ₂ Δ₂}
    → {C : Block Ψ Γ₃ Δ₃}
    → proj₂ (exec-block H CC C) ≡ _ , _ , A
    → BlockEq H CC A B
    → BlockEq H CC C B
```

- исполнение второго из них приводит к блоку, эквивалентному первому;

```
  right : ∀ {Δ₁ Δ₂ Δ₃ Γ₁ Γ₂ Γ₃}
    → {A : Block Ψ Γ₁ Δ₁} → {B : Block Ψ Γ₂ Δ₂}
    → {C : Block Ψ Γ₃ Δ₃}
    → proj₂ (exec-block H CC C) ≡ _ , _ , B
    → BlockEq H CC A B
    → BlockEq H CC A C
```

- исполнение обоих блоков меняет контекст исполнения и приводит к эквивалентным блокам.

```
  ctxchg : ∀ {Δ₁ Δ₂ Δ₁′ Δ₂′ Γ₁ Γ₂ Γ₁′ Γ₂′}
    → {CC′ : CallCtx Ψ}
    → {A′ : Block Ψ Γ₁′ Δ₁′} {B′ : Block Ψ Γ₂′ Δ₂′}
    → BlockEq H CC′ A′ B′
    → {A : Block Ψ Γ₁ Δ₁}
    → exec-block H CC A ≡ proj₁ CC′ , _ , _ , A′
    → {B : Block Ψ Γ₂ Δ₂}
```

```
  → exec-block H CC B ≡ proj₁ CC′ , _ , _ , B′
  → BlockEq H CC A B
```

2.2.5 Компоновка кода

В случае простой динамической компоновки все используемые функции уже загружены в память и GOT корректно заполнен, поэтому PLT состоит всего из одной инструкции.

```
plt-stub : ∀ {Γ Ψ} → (block Γ) ∗ ∈ Ψ → Block Ψ Γ []
plt-stub got = ⇝ (jmp[ got ])
```

При динамической компоновке в память добавляются дополнительные значения. Сначала опишем их типы.

```
pltize : DataType → DataType
```

На каждый блок кода компоновщиком генерируются:

```
pltize (block Γ :: Ψ)
```

- блок PLT, тип которого совпадает с типом блока, на который происходит переход;

```
  = block Γ
```

- элемент GOT, тип которого — указатель на блок, на который происходит переход.

```
  :: block Γ ∗
```

При этом сам блок кода остается на месте.

```
  :: block Γ :: (pltize Ψ)
```

Все остальное при этом остается неизменным.

```
pltize (x :: Ψ) = x :: (pltize Ψ)
pltize [] = []
```

При этом все, что загружалось в память при статической компоновке, будет загружено и при динамической компоновке.

```
plt-⊆ : ∀ {Ψ} → Ψ ⊆ pltize Ψ
plt-⊆ {x = block Γ} (here refl) = there $ there (here refl)
plt-⊆ {x = x ∗ } (here refl) = here refl
plt-⊆ {block Γ :: ψs} (there i) = there $ there (there (plt-⊆ i))
plt-⊆ {ψ ∗ :: ψs} (there i) = there (plt-⊆ i)
```

Опишем, какие значения определенных выше типов появляются в памяти.

```
pltize–data : ∀ {Ψ} → Data Ψ → Data (pltize Ψ)
pltize–data [] = []
```

На каждый блок кода из исходного набора значений, находящихся в памяти, генерируется:

```
pltize–data (vs , block f) = ((pltize–data vs
```

- сам блок кода;

```
, block (wk–block plt–⊆ f))
```

- элемент GOT, указывающий на блок, лежащий перед ним в памяти;

```
, ptr (here refl))
```

- блок PLT, ссылающийся на элемент GOT, лежащий перед ним в памяти.

```
, block (plt–stub (here refl))
```

Указатели на значения корректно преобразуются при добавлении в память таблиц GOT и PLT.

```
pltize–data (vs , ptr x) = pltize–data vs , ptr (plt–⊆ x)
```

Зная адрес, по которому блок располагался в памяти при статической компоновке, можно получить адрес, по которому при динамической компоновке будут располагаться соответствующие ему:

- элемент таблицы PLT;

```
plt : ∀ {Γ Ψ} → (block Γ) ∈ Ψ → (block Γ) ∈ pltize Ψ
plt (here refl) = here refl
plt {Ψ = block Δ :: Ψ} (there f) = there (there (there (plt f)))
plt {Ψ = x * :: Ψ} (there f) = there (plt f)
```

- элемент таблицы GOT.

```
got : ∀ {Γ Ψ} → (block Γ) ∈ Ψ → (block Γ) * ∈ pltize Ψ
got (here refl) = there (here refl)
got {Ψ = block Δ :: Ψ} (there f) = there (there (there (got f)))
got {Ψ = x * :: Ψ} (there f) = there (got f)
```

При смене компоновки со статической на динамическую меняется и код. Все вызовы функций заменяются на вызовы соответствующих элементов PLT, а для каждого перехода исправляется адрес перехода в соответствии с изменением адреса функции.

```
pltize–code : ∀ {Ψ Γ Δ} → Block Ψ Γ Δ → Block (pltize Ψ) Γ Δ
pltize–code (⇝ (call f)) = ⇝ (call (plt f))
pltize–code (⇝ jmp[ f ]) = ⇝ (jmp[ plt–⊆ f ])
```

```
pltize-code (⇝ (jmp f)) = ⇝ (jmp (plt-⊆ f ))
pltize-code (i · b) = wk-instr plt-⊆ i · pltize-code b
```

2.2.6 Доказательства

Все требуемые примитивы определены, теперь можно доказывать эквивалентность различных видов компоновки. Сначала докажем несколько лемм.

Блок кода эквивалентен непрямому `jmp`, если по указанному адресу находится указатель на этот блок кода:

```
jmp[]-proof : ∀ {Ψ Γ Δ} → {CC : CallCtx Ψ}
  → {H : Data Ψ}
  → {A : Block Ψ Γ Δ}
  → (f : (block Γ) ∗ ∈ Ψ)
  → loadblk H (deref H f) ≡ _ , _ , A
  → BlockEq H CC A (⇝ jmp[ f ])
jmp[]-proof {Ψ} {CC = CC} {H = H} {A = A} f p = right p equal
```

Вызов функции меняет контекст исполнения так, как ожидается: на стек добавляется адрес возврата, а исполнение передается на вызываемый блок:

```
call-proof : ∀ {Ψ Γ} → (CC : CallCtx Ψ) → {A : NewBlk Ψ}
  → {H : Data Ψ}
  → (f : (block Γ) ∈ Ψ)
  → loadblk H f ≡ A
  → exec-block H CC (⇝ (call f))
  ≡ ((proj₂ CC :: proj₁ CC) , A)
call-proof CC f p rewrite p = refl
```

Загрузка блока PLT, относящегося к заданной функции, действительно загружает блок PLT. К сожалению, из-за несовершенства определений эта лемма осталась недоказанной.

```
loadplt : ∀ {Ψ Γ} → (H : Data (pltize Ψ))
  → (f : block Γ ∈ Ψ)
  → loadblk H (plt f) ≡ Γ , [] , ⇝ jmp[ got f ]
loadplt H f = {! !}
```

Блок PLT для любой функции выглядит заданным образом:

```
jmp[]–plt–stub : ∀ {Ψ Γ} → (f : block Γ ∈ Ψ)
  → plt–stub (got f) ≡ ⇝ jmp[ got f ]
jmp[]–plt–stub f = refl
```

Загрузка блока, в типе которого указано ограничение на состояние регистров, загружает блок, который действительно ограничивает состояние регистров именно так. Эта лемма тоже осталась недоказанной.

```
loadblk–Γ : ∀ {Ψ Γ} → (H : Data Ψ) → (f : block Γ ∈ Ψ)
  → proj₁ (loadblk H f) ≡ Γ
loadblk–Γ H f = {! !}
```

Блок PLT эквивалентен самой функции. Эта лемма не доказана.

```
plt–fun–eq : ∀ {Γ Ψ}
  → (H : Data (pltize Ψ))
  → (cc : CallCtx (pltize Ψ))
  → (f : block Γ ∈ Ψ)
  → BlockEq H cc
    (proj₂ $ proj₂ (loadblk H (plt–⊆ f)))
    (plt–stub (got f))
plt–fun–eq H cc f with jmp[]–plt–stub f | loadblk–Γ H (plt–⊆ f)
plt–fun–eq H cc f | refl | r = {! !}
```

И, наконец, доказательство того, что в любом контексте исполнения вызов функции напрямую эквивалентен вызову соответствующего PLT.

```
proof : ∀ {Γ Ψ}
  → (H : Data (pltize Ψ))
  → (f : block Γ ∈ Ψ)
  → (cc : CallCtx (pltize Ψ))
  → BlockEq H cc
    (wk–block plt–⊆ (⇝ (call f)))
    (⇝ (call (plt f)))
proof {Γ = Γ} {Ψ = Ψ} H f ctx = ctxchg after–call just–call plt–call
  where
  newblock–f  = loadblk H (plt–⊆ f)
  called–block = proj₂ $ proj₂ newblock–f

  just–call : exec–block H ctx (⇝ (call $ plt–⊆ f)) ≡
```

```
    proj₂ ctx :: proj₁ ctx , newblock–f
just–call = call–proof ctx (plt–⊆ f) refl

plt–call : exec–block H ctx (↝ (call $ plt f)) ≡
    proj₂ ctx :: proj₁ ctx , _ , _ , ↝ jmp[ got f ]
plt–call = call–proof ctx (plt f) (loadplt H f)

after–call : BlockEq H (proj₂ ctx :: proj₁ ctx , newblock–f)
                called–block
                (↝ jmp[ got f ])
after–call = plt–fun–eq H (proj₂ ctx :: proj₁ ctx , newblock–f) f
```

2.2.7 Проблемы этого решения

Приведенное выше решение обладает рядом недостатков:

- во-первых, SSA не подходит для описания модели, касающейся ABI;
- во-вторых, память рассматривается неизменяемой, что сильно сужает класс программ, которые возможно написать;
- в-третьих, текущее определение памяти не позволяет блокам ссылаться друг на друга, что не позволяет писать рекурсивные функции.

2.3 Наборы изменений

Эта секция описывает, как решить первую из проблем предыдущей реализации.

Ранее для описания типа инструкций использовался список регистров, добавляемых к уже имеющимся. Это приводило к тому, что никакая инструкция не могла изменить уже добавленные ранее регистры. Указанную проблему можно решить, введя тип, описывающий изменение списка регистров, и соответственно изменив тип инструкции.

2.3.1 Описание изменений регистров

Определим тип, описывающий одно изменение списка фиксированной длины: в таком списке можно только менять элементы, что и требуется от регистров. Так как не любое изменение можно применить к произвольному списку, его тип должен ограничивать, к какому списку его можно применять.

```
module ListChg (A : Set) where
  data Chg (Γ : List A) : Set where
```

Для того, чтобы описать изменение элемента в списке, необходимо указать, какая позиция меняется и на что.

```
chg : ∀ {τ} → τ ∈ Γ → A → Chg Γ
```

Сами по себе изменения не несут особого смысла: необходимо указать, как они применяются к спискам.

```
chgapply : (Γ : List A) → Chg Γ → List A
chgapply (_ :: Γ) (chg (here refl) σ) = σ :: Γ
chgapply (τ :: Γ) (chg (there p)  σ) = τ :: chgapply Γ (chg p σ)
```

Блок кода последовательно применяет изменения к списку регистров, значит, в его типе должен быть описан набор изменений.

Набор изменений является общим для изменений различных видов. То, к чему применяется изменение, будем называть *контекстом*.

```
module Diff
  {Ctx : Set}
  {Chg : Ctx → Set}
  (chgapply : (Γ : Ctx) → Chg Γ → Ctx)
  where
```

Так же, как и для изменения, тип набора изменений ограничивает контекст, к которому его можно применять.

```
data Diff (Γ : Ctx) : Set where
```

Набор изменений — это:

- либо пустой набор;

```
dempty : Diff Γ
```

- либо это изменение, добавленное перед уже имеющимся набором.

```
dchg : (c : Chg Γ) → Diff (chgapply Γ c) → Diff Γ
```

Для набора изменений определены несколько функций:

- применение набора к контексту — это последовательное применение всех изменений из набора;

```
dapply : (Γ : Ctx) → Diff Γ → Ctx
dapply Γ dempty = Γ
dapply Γ (dchg c d) = dapply (chgapply Γ c) d
```

- объединение двух наборов изменений;

```
dappend : ∀ {Γ} → (d : Diff Γ)
  → Diff (dapply Γ d) → Diff Γ
dappend dempty b = b
dappend (dchg c a) b = dchg c (dappend a b)
```

- лемма, доказывающая, что объединение с пустым набором не меняет набор;

```
dappend–dempty–lemma : ∀ {Γ} → (d : Diff Γ)
  → dappend d dempty ≡ d
dappend–dempty–lemma dempty = refl
dappend–dempty–lemma (dchg c d)
  rewrite dappend–dempty–lemma d = refl
```

- лемма, доказывающая, что применение объединения наборов эквивалентно последовательному применению этих наборов.

```
dappend–dapply–lemma : ∀ S → (d₁ : Diff S)
  → (d₂ : Diff (dapply S d₁))
  → dapply S (dappend d₁ d₂)
  ≡ dapply (dapply S d₁) d₂
dappend–dapply–lemma S dempty d₂ = refl
dappend–dapply–lemma S (dchg c d₁) d₂
  = dappend–dapply–lemma (chgapply S c) d₁ d₂
```

2.3.2 Блоки и инструкции

Используя определенный тип, переопределим типы блоков и инструкций.

```
module FixedHeap (Ψ : DataType) where
```

Ранее тип блока описывал список добавляемых регистров. Теперь он описывает набор изменений, применяемых к уже имеющимся регистрам.

```
data Block (Γ : RegTypes) : Diff Γ → Set
```

Управляющие инструкции не меняют регистров, поэтому их определение останется неизменным.

Тип инструкции изменим так же, как изменился тип блока: список добавляемых регистров заменим на описание набора изменений.

```
data Instr (Γ : RegTypes) : Chg Γ → Set
```

В этой реализации инструкция не добавляет регистр, а описывает изменение конкретного регистра из списка на значение нового типа.

```
data Instr (Γ : RegTypes) where
  mov  : ∀ {r τ} → (r∈Γ : r ∈ Γ) → Value τ
       → Instr Γ (chg r∈Γ τ)
```

И, наконец, определим конструкторы блока.

```
data Block (Γ : RegTypes) where
  ⇝  : ControlInstr Γ → Block Γ dempty
  _·_ : ∀ {c d} → Instr Γ c → Block (chgapply Γ c) d
      → Block Γ (dchg c d)
```

2.4 Вторая реализация: метаассемблер

В области формальных доказательств существуют серьезные проблемы с переиспользованием доказательств: даже при небольшом изменении основных определений все доказательства приходится менять. При этом определения многих сущностей косвенно используют определения конкретных инструкций ассемблера, не вдаваясь в детали, и являются общими для всех языков ассемблера. Эти определения можно было бы определить независимо от конкретного языка ассемблера, используя конкретные реализации инструкций как параметры. Набор этих определений будем называть *метаассемблером*.

От конкретного ассемблера можно абстрагироваться, переформулировав все определения так, чтобы они использовали понятие блока кода. Тогда реализовать метаассемблер можно, определив несколько модулей, принимающих параметрами сущности конкретного ассемблера. Таких модулей реализовано четыре:

- Модуль `Blocks`, принимающий параметрами типы инструкций и управляющих инструкций. Определяет понятие блока.

- Модуль `Values`, принимающий параметром тип блока. Определяет все основные сущности: значения, память и регистры.
- Модуль `ExecBlk`, принимающий параметрами типы инструкций и управляющих инструкций, а также функции, определяющие, как эти инструкции меняют состояние исполнителя. Определяют функцию, определяющую, как блок изменяет состояние исполнителя.
- Модуль `Eq`, принимающий параметрами тип блока и функцию, определяющую, как блок изменяет состояние исполнителя. Определяет понятие эквивалентности блоков.

С помощью этих модулей можно легко получать все нужные определения, имея минимальное определение ассемблера, просто импортировав нужные модули с нужными параметрами.

2.4.1 Основные определения

В этой версии память больше не считается неизменной, поэтому тип состояния исполнителя должен описывать не только регистры, но и память.

```
record StateType : Set where
  constructor state
  field
    memory : DataType
    registers : RegTypes
open StateType public
```

Определение набора изменений регистров аналогично используемому ранее.

Так как почти нигде в коде не используется применение изменений к набору регистров отдельно от всего состояния исполнителя, удобно определить функцию для применения изменений сразу ко всему состоянию исполнителя.

```
sdapply : (S : StateType) → Diff (registers S) → StateType
sdapply (state h r) d = state h (dapply r d)

SDiff = λ S → Diff (registers S)
```

2.4.2 Модуль Blocks

Типы инструкций и управляющих инструкций аналогичны используемым ранее. Отличием является только то, что в качестве первого аргумента они содержат не типы регистров, а тип состояния исполнителя.

```
module Blocks
  (ControlInstr : StateType → Set)
  (Instr : (S : StateType) → Chg (registers S) → Set)
  where
```

Определение блока аналогично приведенным в предыдущих секциях.

```
data Block (S : StateType) : Diff (registers S) → Set where
  ⇝  : ControlInstr S → Block S dempty
  _·_ : ∀ {c d} → Instr S c → Block (sdapply S (dchg c dempty)) d
      → Block S (dchg c d)
```

2.4.3 Модуль Values

Параметром этого модуля, как уже было сказано, является тип блока.

```
module Values
  (Block : (S : StateType) → Diff (registers S) → Set)
  where
```

Определение значений аналогично используемому ранее.

```
data Value (Ψ : DataType) : Type → Set where
  block : ∀ {Γ} → {d : Diff Γ}
    → Block (state Ψ Γ) d
    → Value Ψ (block Γ)
  ptr : ∀ {τ} → τ ∈ Ψ → Value Ψ (τ *)
```

Первая реализация не позволяла блокам ссылаться друг на друга. Эту проблему можно решить, если определить память, используя два параметра типа: первый описывает, на что значения могут ссылаться, а второй описывает, что в памяти в действительности располагается.

```
data IData (Ψ : DataType) : DataType → Set where
  []  : IData Ψ []
  _::_ : ∀ {τ Δ} → Value Ψ τ → IData Ψ Δ → IData Ψ (τ :: Δ)
```

При этом корректно заполненной память считается тогда, когда эти параметры совпадают.

```
Data : DataType → Set
Data Ψ = IData Ψ Ψ
```

Определим функцию для загрузки значений из памяти:

```
load : ∀ {Ψ τ} → τ ∈ Ψ → Data Ψ → Value Ψ τ
load p memory = iload [] p memory
  where
  ++[]–lemma : {A : Set} (a : A) (as bs : List A)
    → as ++ a :: bs ≡ (as ++ [ a ]) ++ bs
  ++[]–lemma a [] bs = refl
  ++[]–lemma a (x :: as) bs rewrite ++[]–lemma a as bs = refl

  iload : ∀ {Ψ τ} ψs → τ ∈ Ψ → IData (ψs ++ Ψ) Ψ
    → Value (ψs ++ Ψ) τ
  iload ψs (here refl) (x :: _) = x
  iload {ψ :: Ψ} ψs (there p) (x :: h)
    rewrite ++[]–lemma ψ ψs Ψ = iload (ψs ++ [ ψ ]) p h
```

Регистры — список значений, ссылающихся на память, в типе которого описано, значения каких типов он хранит.

```
data IRegisters (Ψ : DataType) : RegTypes → Set where
  []  : IRegisters Ψ []
  _::_ : ∀ {τ τs} → Value Ψ τ → IRegisters Ψ τs
    → IRegisters Ψ (τ :: τs)

Registers : StateType → Set
Registers S = IRegisters (memory S) (registers S)
```

Ниже приведены вспомогательные функции для работы с значениями.

```
unptr : ∀ {Ψ τ} → Value Ψ (τ *) → τ ∈ Ψ
unptr (ptr x) = x

unblock : ∀ {Ψ Γ} → Value Ψ (block Γ)
  → Σ (Diff Γ) (Block (state Ψ Γ))
unblock (block x) = _ , x
```

```
_∈B_ : ∀ {S d} → Block S d → Data (memory S) → Set
_∈B_ {S} b Ψ = Σ (block (registers S) ∈ memory S)
  $ λ ptr → (unblock (load ptr Ψ)) ≡ _ , b
```

2.4.4 Модуль ExecBlk

Перед тем, как описывать исполнение кода, необходимо определить набор сущностей, влияющих на результат исполнения, но не не известных на момент компиляции. К ним относятся instruction pointer и стек вызовов.

```
module Exec−Context (Ψ : DataType) where
```

Назовем *типизированным instruction pointer-ом* индекс блока, рассчитывающего на указанное состояние регистров, находящийся в памяти.

```
IPRT : RegTypes → Set
IPRT Γ = block Γ ∈ Ψ
```

В таком случае *instruction pointer-ом* в обычном понимании является зависимая пара из типа состояния регистров и типизированного им instruction pointer-а.

```
IP = Σ RegTypes IPRT
```

В этой версии вместо сохранения в стеке вызовов самих блоков будем хранить указатели на них.

```
CallStack = List IP
```

Параметрами модуля `ExecBlk` являются типы инструкций и управляющих инструкций, а также функции, описывающие результат их исполнения. изменятся. Опишем типы этих параметров.

```
module ExecBlk
```

Сигнатуры инструкций и управляющих инструкций уже были описаны ранее.

```
(Instr : (S : StateType) → Chg (registers S) → Set)
(ControlInstr : StateType → Set)
```

Результат исполнения инструкции зависит от того, какие значения в данный момент находятся в памяти и регистрах, и определяет, на какие значения они изменятся.

```
(exec–instr : {S : StateType}
  → {c : Chg (registers S)} → Instr S c
  → Values.Data
    (Blocks.Block ControlInstr Instr)
    (memory S)
  → Values.Registers
    (Blocks.Block ControlInstr Instr)
    S
  → Values.Data
    (Blocks.Block ControlInstr Instr)
    (memory $ sdapply S (dchg c dempty))
  × Values.Registers
    (Blocks.Block ControlInstr Instr)
    (sdapply S (dchg c dempty)))
```

Результат исполнения управляющей инструкции зависит от того, какие значения находятся в памяти, что находится в стеке вызовов и чему равен instruction pointer, и определяет, как изменится стек вызовов и типизированный типом текущего состояния регистров instruction pointer.

```
(exec–control : {S : StateType}
  → ControlInstr S
  → Values.Data
    (Blocks.Block ControlInstr Instr)
    (memory S)
  → CallStack (memory S)
  → IP (memory S)
  → CallStack (memory S)
  × IPRT (memory S) (registers S))
where
open Blocks ControlInstr Instr public
open Values Block public
```

Результатом исполнения блока является совокупность результатов всех входящих в блок инструкций. Это означает, что результат исполнения блока зависит от того, какие значения находятся в памяти и регистрах, чему равен instruction pointer и что находится в стеке вызовов и определяет, какой блок

будет исполняться следующим и как изменятся стек вызовов и значения в памяти и регистрах.

Однако, для некоторых блоков (например, блоков, заканчивающихся условным переходом или вызовом функции) важно их расположение в памяти: за ними должен располагаться блок кода, имеющий подходящий тип. Это не было учтено при реализации блоков, из-за чего корректно определить функцию `exec-block` оказалось затруднительно.

```
exec−block : {S : StateType} {d : Diff (registers S)} {b : Block S d}
  → (Ψ : Data (memory S))
  → b ∈B Ψ
  → Registers S → CallStack (memory S)
  → (Σ (Diff $ dapply (registers S) d) (Block $ sdapply S d))
  × (Data (memory $ sdapply S d)
  × (Registers (sdapply S d)
  × CallStack (memory $ sdapply S d)))
exec−block {b = b} Ψ p Γ cs = {! !}
```

2.4.5 Модуль Eq

Параметрами этого модуля являются тип блока и функция, определяющая результат исполнения блока.

```
module Eq
  (Block : (S : StateType) → Diff (registers S) → Set)
  (exec−block : {S : StateType} {d : Diff (registers S)} {b : Block S d}
    → (Ψ : Values.Data Block (memory S))
    → Values._∈B_ Block b Ψ
    → Values.Registers Block S → CallStack (memory S)
    → (Σ (Diff $ dapply (registers S) d) (Block $ sdapply S d))
    × (Values.Data Block (memory $ sdapply S d)
    × (Values.Registers Block (sdapply S d)
    × CallStack (memory $ sdapply S d))))
  where
  open Values Block
```

Определение эквивалентности блоков почти аналогично приведенному ранее. Отличием является то, что стек вызовов теперь считается меняющимся после исполнения любого блока.

```
data BlockEq :
  {S₁ S₂ : StateType} → {d₁ : SDiff S₁} {d₂ : SDiff S₂} →
  (Ψ₁ : Data (memory S₁)) (Ψ₂ : Data (memory S₂)) →
  (Γ₁ : Registers S₁) (Γ₂ : Registers S₂) →
  (CC₁ : CallStack (memory S₁)) (CC₂ : CallStack (memory S₂)) →
  Block S₁ d₁ → Block S₂ d₂ → Set
  where
```

Два блока эквивалентны, если:

- они одинаковы;

```
  equal : ∀ {S} → {d : SDiff S}
    → {Ψ : Data (memory S)} {CC : CallStack (memory S)}
    → {B : Block S d} {Γ : Registers S}
    → BlockEq Ψ Ψ Γ Γ CC CC B B
```

- исполнение первого из них приводит к блоку, эквивалентному второму;

```
  left  : ∀ {S₁ S}
      → {d₁ : SDiff S₁} {d₂ : SDiff (sdapply S₁ d₁)}
      → {d : SDiff S}
      → {A₁ : Block S₁ d₁} {A₂ : Block (sdapply S₁ d₁) d₂}
      → {B : Block S d}
      → (Ψ₁ : Data (memory S₁))
      → (Ψ₂ : Data (memory (sdapply S₁ d₁)))
      → (Ψ : Data (memory S))
      → (ip₁ : A₁ ∈B Ψ₁) (ip₂ : A₂ ∈B Ψ₂)
      → (ip : B ∈B Ψ)
      → (Γ₁ : Registers S₁) (Γ₂ : Registers (sdapply S₁ d₁))
      → (Γ : Registers S)
      → (CC₁ : CallStack (memory S₁))
      → (CC₂ : CallStack (memory $ sdapply S₁ d₁))
      → (CC : CallStack (memory S))
      → exec-block Ψ₁ ip₁ Γ₁ CC₁ ≡ (_ , A₂) , Ψ₂ , Γ₂ , CC₂
```

```
      → BlockEq Ψ₁ Ψ Γ₁ Γ CC₁ CC A₁ B
      → BlockEq Ψ₂ Ψ Γ₂ Γ CC₂ CC A₂ B
```

- исполнение второго из них приводит к блоку, эквивалентному первому.

```
  right : ∀ {S₁ S}
    → {d₁ : SDiff S₁} {d₂ : SDiff (sdapply S₁ d₁)}
    → {d : SDiff S}
    → {A₁ : Block S₁ d₁} {A₂ : Block (sdapply S₁ d₁) d₂}
    → {B : Block S d}
    → (Ψ₁ : Data (memory S₁))
    → (Ψ₂ : Data (memory (sdapply S₁ d₁)))
    → (Ψ : Data (memory S))
    → (ip₁ : A₁ ∈B Ψ₁) (ip₂ : A₂ ∈B Ψ₂)
    → (ip : B ∈B Ψ)
    → (Γ₁ : Registers S₁) (Γ₂ : Registers (sdapply S₁ d₁))
    → (Γ : Registers S)
    → (CC₁ : CallStack (memory S₁))
    → (CC₂ : CallStack (memory $ sdapply S₁ d₁))
    → (CC : CallStack (memory S))
    → exec-block Ψ₁ ip₁ Γ₁ CC₁ ≡ (_ , A₂) , Ψ₂ , Γ₂ , CC₂
    → BlockEq Ψ Ψ₁ Γ Γ₁ CC CC₁ B A₁
    → BlockEq Ψ Ψ₂ Γ Γ₂ CC CC₂ B A₂
```

2.4.6 Ассемблер x86-64

Определения инструкций и управляющих инструкций аналогичны приведенным ранее.

```
  data ControlInstr (S : StateType) : Set where
    jmp call : block (registers S) ∈ memory S → ControlInstr S
    jmp[_]  : block (registers S) * ∈ memory S → ControlInstr S

  data Instr (S : StateType) : Chg (registers S) → Set where
    mov : ∀ {τ σ} → (r : σ ∈ registers S)
      → Values.Value
        (Blocks.Block ControlInstr Instr)
```

```
      (memory S) τ
    → Instr S (chg r τ)
```

Описание результатов исполнения определенных инструкций аналогично приведенному ранее.

```
exec-control : {S : StateType}
  → ControlInstr S
  → Values.Data
    (Blocks.Block ControlInstr Instr)
    (memory S)
  → CallStack (memory S) → IP (memory S)
  → CallStack (memory S) × IPRT (memory S) (registers S)
exec-control {state memory registers} (jmp x) Ψ cs ip = cs , x
exec-control {state memory registers} (call x) Ψ cs ip = ip :: cs , x
exec-control {state memory registers} (jmp[ x ]) Ψ cs ip
  = cs
  , (Values.unptr
    (Blocks.Block ControlInstr Instr)
    $ Values.load (Blocks.Block ControlInstr Instr) x Ψ)
```

Функция `exec-instr` не была реализована, так как к моменту ее определения стало понятно, что приведенные выше определения обладают рядом недостатков, которые не позволят решить поставленную задачу.

```
exec-instr : {S : StateType}
  → {c : Chg (registers S)} → Instr S c
  → Values.Data
    (Blocks.Block ControlInstr Instr)
    (memory S)
  → Values.Registers
    (Blocks.Block ControlInstr Instr)
    S
  → Values.Data
    (Blocks.Block ControlInstr Instr)
    (memory $ sdapply S (dchg c dempty))
  × Values.Registers
    (Blocks.Block ControlInstr Instr)
```

```
        (sdapply S (dchg c dempty))
exec-instr = {! !}
open ExecBlk Instr ControlInstr exec-instr exec-control
open Eq Block exec-block
```

2.4.7 Проблемы этого решения

Несмотря на то, что это решение не обладает описанными ранее недостатками, оно имеет ряд проблем.

- Размеры возможных значений никак не учитываются. Ничто не мешает загрузить в регистр блок кода.
- Блок кода не накладывает никаких ограничений на свое расположение в памяти, но может рассчитывать на то, что после него лежит подходящий блок. Это означает, что существует возможность после блока, завершающегося вызовом функции, поместить блок данных, что приведет к некорректному состоянию стека вызовов.
- Нет никакой динамической аллокации памяти, которая нужна хотя бы для того, чтобы можно было сохранить значения регистров.
- Инструкции не накладывают ограничений на состояние стека вызовов. Это означает, что инструкция `ret` может быть исполнена при пустом стеке вызовов, что некорректно.

2.5 Выводы по главе

В этой главе были рассмотрены основные этапы разработки метода формальной верификации компоновки блоков ассемблерного кода:

- представлен набор основных сущностей и описание их типов;
- представлена реализация ассембелера в SSA-представлении, для него введено понятие эквивалентности блоков кода, описано преобразование кода при простой динамической линковке и доказана эквивалентность вызова функции напрямую и соответствующего ей элемента таблицы PLT, а также указаны проблемы этого решения;

- описан способ применить указанный в первой реализации метод для ассемблера не в SSA-представлении с помощью описания набора изменений регистров;
- введено понятие метаассемблера, представлена его первая реализация, попытка применения его к ассемблеру `x86-64` и указаны проблемы выбранных определений.

ГЛАВА 3

РАЗРАБОТАННЫЙ МЕТОД И ЕГО ПРИМЕНЕНИЕ

В этой главе представлен разработанный вариант реализации метода доказательства эквивалентности ассемблерных кодов, не обладающий недостатками предыдущих реализаций и позволяющий доказывать эквивалентность программ, скомпилированных статически и динамически.

3.1 Основные определения

Одной из проблем прошлых решений является неучитывание размеров возможных значений, из-за чего возможно написать код, загружающий в регистр значение, которое не может быть загружено. Решить эту проблему можно, поделив возможные значения на два класса:

- значения, которые могут располагаться в регистрах;

```
data RegType : Set

RegTypes = List RegType
```

- значения произвольного размера, которые могут располагаться в памяти.

```
data Type : Set

DataType = List Type
```

Еще одной проблемой являлось полное отсутствие динамически аллоцируемой памяти, в качестве которой можно использовать стек данных.

Для удобства будем считать стек вызовов и стек данных разными сущностями, хотя на практике обычно используется один стек.

```
DataStackType : Set
CallStackType : Set
```

Состояния обоих стеков, как и состояния регистров и памяти, входят в состояние исполнителя.

```
record StateType : Set where
  constructor statetype
  field
```

```
    registers : RegTypes
    memory  : DataType
    datastack : DataStackType
    callstack : CallStackType
```

Значениями, размер которых равен размеру регистра, являются указатели и целые числа.

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

К значениям, которые могут располагаться в памяти, относятся как значения размера регистра, так и блоки кода. Тип блока кода описывает состояния регистров и обоих стеков, при которых он может быть корректно исполнен.

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

Стек данных может хранить только значения размера регистра. Его тип можно описать списком типов находящихся в нем значений.

```
  DataStackType = List RegType
```

Стек вызовов хранит указатели на блоки в памяти. Параметрами типа блока являются типы регистров, тип стека данных и тип стека вызовов. При этом сохраненный в стеке вызовов указатель на блок может быть исполнен только при снятии его со стека. Это означает, что добавить указатель на блок в стек вызовов можно только в тот момент исполнения, когда тип стека вызовов совпадает с тем, на который рассчитывает блок. Это позволяет описать тип стека вызовов как список пар типов регистров и стека данных, считая, что тип стека вызовов задан неявно.

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

3.2 Наборы изменений

Определение изменений для регистров используется из предыдущих реализаций.

```
    module RegDiff where
```

```
open ListChg RegType public
open Diff chgapply public
```

В отличие от предыдущих реализаций, в тип инструкций должны входить и стек вызовов, и стек данных. Это означает, что для них необходимо определить наборы изменений.

```
module StackDiff (A : Set) where
  data Chg (S : List A) : Set where
```

Возможными изменениями стека являются:

- добавление значения на вершину стека;

```
    push : (i : A) → Chg S
```

- снятие значения с вершины стека, если стек не пуст.

```
    pop  : ∀ {Γ S'} → S ≡ Γ :: S' → Chg S
```

Определим, как изменения применяются к стеку, и используем определенный ранее тип набора изменений.

```
chgapply : (S : List A) → Chg S → List A
chgapply cs (push x) = x :: cs
chgapply (_ :: S') (pop refl) = S'
open Diff chgapply public
```

Общим набором изменений состояния исполнителя будет являться структура, описывающая изменения регистров и двух стеков.

```
record Diff (S : StateType) : Set where
  constructor diff
  field
    rdiff  : RegDiff.Diff (StateType.registers S)
    dsdiff : StackDiff.Diff RegType (StateType.datastack S)
    csdiff : StackDiff.Diff (RegTypes × DataStackType)
      (StateType.callstack S)
open Diff public
```

Определим вспомогательные функции и типы для конструирования и применения изменений состояния исполнителя:

- конструирование пустого набора изменений;

```
dempty : ∀ {S} → Diff S
dempty = diff
  RegDiff.dempty
```

```
    StackDiff.dempty
    StackDiff.dempty
```

- применение набора изменений к состоянию исполнителя;

```
dapply : (S : StateType) → Diff S → StateType
dapply (statetype r m d c) (diff rd dd cd) =
  statetype
  (RegDiff.dapply r rd)
  m
  (StackDiff.dapply RegType d dd)
  (StackDiff.dapply (RegTypes × DataStackType) c cd)
```

- объединение двух наборов изменений;

```
dappend : ∀ {S} → (d : Diff S) → Diff (dapply S d) → Diff S
dappend (diff rd dd cd) (diff rd′ dd′ cd′) =
  diff
  (RegDiff.dappend rd rd′)
  (StackDiff.dappend RegType dd dd′)
  (StackDiff.dappend (RegTypes × DataStackType) cd cd′)
```

- изменение стека данных;

```
DataStackChg : StateType → Set
DataStackChg S
  = StackDiff.Chg RegType (StateType.datastack S)
```

- изменение стека вызовов;

```
CallStackChg : StateType → Set
CallStackChg S
  = StackDiff.Chg
    (RegTypes × DataStackType)
    (StateType.callstack S)
```

- изменение набора регистров;

```
RegChg : StateType → Set
RegChg S = RegDiff.Chg (StateType.registers S)
```

- изменение, производимое инструкцией: одна инструкция может изменить либо регистры, либо стек, либо и то, и другое (например, при pop значения со стека в регистр);

```
data SmallChg (S : StateType) : Set where
  onlyreg   : RegChg S → SmallChg S
  onlystack : DataStackChg S → SmallChg S
  regstack  : RegChg S → DataStackChg S → SmallChg S
```

- конструирование набора изменений состояния исполнителя по одному изменению регистров;

```
regChg : ∀ {S} → RegChg S → Diff S
regChg c =
  diff
  (RegDiff.dchg c RegDiff.dempty)
  StackDiff.dempty
  StackDiff.dempty
```

- конструирование набора изменений состояния исполнителя по одному изменению стека данных;

```
dsChg : ∀ {S} → DataStackChg S → Diff S
dsChg c =
  diff
  RegDiff.dempty
  (StackDiff.dchg c StackDiff.dempty)
  StackDiff.dempty
```

- конструирование набора изменений состояния исполнителя по одному изменению, производимому инструкцией;

```
sChg : ∀ {S} → SmallChg S → Diff S
sChg (onlyreg r) = regChg r
sChg (onlystack d) = dsChg d
sChg (regstack r d) =
  diff
  (RegDiff.dchg r RegDiff.dempty)
  (StackDiff.dchg d StackDiff.dempty)
  StackDiff.dempty
```

- конструирование набора изменений состояния исполнителя по одному возможному изменению стека вызовов.

```
csChg : ∀ S → Maybe (CallStackChg S) → Diff S
csChg S nothing = dempty
```

```
csChg S (just c) =
  diff
  RegDiff.dempty
  StackDiff.dempty
  (StackDiff.dchg c StackDiff.dempty)
```

3.3 Метаассемблер

Аналогично приведенному в предыдущей главе, метаассемблер состоит из четырех модулей.

3.3.1 Модуль Blocks

Ранее управляющие инструкции описывали только состояние исполнителя, требуемое для выполнения инструкции. С добавлением в состояние исполнителя стека вызовов становится возможным описание изменений, производимых управляющей инструкцией. По типу управляющей инструкции видно, что в результате исполнения может измениться только, возможно, стек вызовов.

Тип инструкции тоже задает, какие части состояния исполнителя он может изменить: это либо регистры, либо стек данных, либо и то, и то.

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

Определение блока аналогично приведенному ранее.

```
data Block (S : StateType) : Diff S → Set where
  ⇝ : ∀ {c} → ControlInstr S c → Block S (csChg S c)
  _·_ : ∀ {c d}
    → Instr S c
```

```
      → Block (dapply S (sChg c)) d
      → Block S (dappend (sChg c) d)
```

3.3.2 Модуль Values

```
module Values
  (Block : (S : StateType) → Diff S → Set)
  where
```

Определения значений аналогичны используемым ранее, но поделены на два класса, соответствующие значениям размера регистра и значениям произвольного размера.

```
data RegValue (Ψ : DataType) : RegType → Set where
  ptr : ∀ {τ} → τ ∈ Ψ → RegValue Ψ (τ *)
  int : ℕ → RegValue Ψ int

data Value (Ψ : DataType) : Type → Set where
  atom : ∀ {τ} → RegValue Ψ τ → Value Ψ (atom τ)
  block : ∀ {Γ DS CS d}
    → Block (statetype Γ Ψ DS CS) d
    → Value Ψ (block Γ DS CS)
```

Определим вспомогательные функции для работы со значениями:

- получение блока из значения типа `block`;

```
  unblock : ∀ {Ψ Γ DS CS} → Value Ψ (block Γ DS CS)
    → Σ (Diff (statetype Γ Ψ DS CS))
      (Block (statetype Γ Ψ DS CS))
  unblock (block b) = _ , b
```

- получение указателя на `τ` из значения типа `τ *`.

```
  unptr : ∀ {Ψ τ} → Value Ψ (atom (τ *)) → τ ∈ Ψ
  unptr (atom (ptr x)) = x
```

Определение набора регистров аналогично приведенному ранее.

```
data Registers (Ψ : DataType) : RegTypes → Set where
  []  : Registers Ψ []
  _::_ : ∀ {τ τs}
    → RegValue Ψ τ
```

```
      → Registers Ψ τs
      → Registers Ψ (τ :: τs)
```

Определим вспомогательные функции для работы с регистрами:

- загрузка значения из заданного регистра;

```
fromreg : ∀ {Ψ Γ τ} → Registers Ψ Γ → τ ∈ Γ → RegValue Ψ τ
fromreg (x :: Γ) (here refl) = x
fromreg (x :: Γ) (there p) = fromreg Γ p
```

- запись значения в заданный регистр.

```
toreg : ∀ {Ψ Γ σ τ}
  → Registers Ψ Γ
  → (r : σ ∈ Γ)
  → RegValue Ψ τ
  → Registers Ψ (RegDiff.chgapply Γ (RegDiff.chg r τ))
toreg (x :: Γ) (here refl) v = v :: Γ
toreg (x :: Γ) (there r) v = x :: (toreg Γ r v)
```

Состояние памяти определяется аналогично приведенному ранее.

```
data IData (Ψ : DataType) : DataType → Set where
  []  : IData Ψ []
  _::_ : ∀ {τ τs} → Value Ψ τ → IData Ψ τs → IData Ψ (τ :: τs)

Data : DataType → Set
Data Ψ = IData Ψ Ψ

load : ∀ {Ψ τ} → Data Ψ → τ ∈ Ψ → Value Ψ τ
load {Ψ} {τ} = iload
  where
  iload : ∀ {Γ} → IData Ψ Γ → τ ∈ Γ → Value Ψ τ
  iload [] ()
  iload (x :: H) (here refl) = x
  iload (x :: H) (there p) = iload H p
```

Определим вспомогательные функции для работы с памятью:

- загрузка блока кода из памяти по указателю на блок;

```
loadblock : ∀ {Ψ Γ CS DS} → Data Ψ → block Γ DS CS ∈ Ψ
  → Σ (Diff (statetype Γ Ψ DS CS))
```

```
      (Block (statetype Γ Ψ DS CS))
  loadblock Ψ f = unblock $ load Ψ f
```

- загрузка указателя на `τ` из памяти по указателю на `τ *`.

```
  loadptr : ∀ {Ψ τ} → Data Ψ → atom (τ *) ∈ Ψ → τ ∈ Ψ
  loadptr Ψ p = unptr $ load Ψ p
```

Стек данных — список значений размера регистра, в типе которого указано, значения каких типов в нем находятся.

```
  data DataStack (Ψ : DataType) : List RegType → Set
    where
    []    : DataStack Ψ []
    _::_  : ∀ {τ DS} → RegValue Ψ τ
          → DataStack Ψ DS
          → DataStack Ψ (τ :: DS)
```

Типизированный instruction pointer — указатель на блок кода в памяти.

```
  IPRT : DataType
    → RegTypes
    → DataStackType
    → CallStackType
    → Set
  IPRT Ψ Γ DS CS = block Γ DS CS ∈ Ψ
```

Стек вызовов — список типизированных instruction pointer-ов. Ранее было описано, почему в типе стека вызовов не указывается требуемое блоком состояние стека вызовов.

```
  data CallStack (Ψ : DataType) : CallStackType → Set where
    [] : CallStack Ψ []
    _::_ : ∀ {Γ DS CS} → IPRT Ψ Γ DS CS → CallStack Ψ CS
      → CallStack Ψ ((Γ , DS) :: CS)
```

Состояние исполнителя — совокупность состояний регистров, памяти и стеков.

```
  record State (S : StateType) : Set where
    constructor state
    field
      registers : Registers
        (StateType.memory S)
```

```
      (StateType.registers S)
    memory  : Data (StateType.memory S)
    datastack : DataStack
      (StateType.memory S)
      (StateType.datastack S)
    callstack : CallStack
      (StateType.memory S)
      (StateType.callstack S)
```

3.3.3 Модуль ExecBlk

```
module ExecBlk
```

Сигнатуры инструкций и управляющих инструкций были описаны ранее.

```
(Instr : (S : StateType) → Diffs.SmallChg S → Set)
(ControlInstr : (S : StateType)
  → Maybe (Diffs.CallStackChg S)
  → Set)
```

Результат исполнения инструкции заивисит от состояния исполнителя и определяет, как изменятся регистры, память и стек данных.

```
(exec-instr : ∀ {S c}
  → Values.State
    (Blocks.Block ControlInstr Instr)
    S
  → Instr S c
  → Values.Registers
    (Blocks.Block ControlInstr Instr)
    (StateType.memory S)
    (StateType.registers
      (Diffs.dapply S (Diffs.sChg c)))
  × (Values.Data
    (Blocks.Block ControlInstr Instr)
    (StateType.memory S)
  × Values.DataStack
    (Blocks.Block ControlInstr Instr)
```

```
      (StateType.memory S)
      (StateType.datastack
        (Diffs.dapply S (Diffs.sChg c)))))
```

Результат исполнения управляющей инструкции тоже зависит от состояния исполнителя и определяет, как изменится стек вызовов и какой блок будет исполняться следующим.

```
(exec-control : ∀ {S c}
  → Values.State
    (Blocks.Block ControlInstr Instr)
    S
  → ControlInstr S c
  → Values.CallStack
    (Blocks.Block ControlInstr Instr)
    (StateType.memory S)
    (StateType.callstack
      (Diffs.dapply S (Diffs.csChg S c)))
  × Σ (Diffs.Diff
    (Diffs.dapply S (Diffs.csChg S c)))
(Blocks.Block ControlInstr Instr
    (Diffs.dapply S (Diffs.csChg S c))))
where
open Diffs
open Blocks ControlInstr Instr
open Values Block
```

Для определения функции `exec-block` потребовалось определить несколько лемм:

- если набор изменений состояния исполнителя построен как набор изменений стека вызовов, то набор изменений регистров пуст;

```
reg-const : ∀ S → (c : Maybe (CallStackChg S))
  → rdiff (csChg S c) ≡ RegDiff.dempty
reg-const S (just c) = refl
reg-const S nothing = refl
```

- если набор изменений состояния исполнителя построен как набор изменений стека вызовов, то набор изменений стека данных пуст;

```
ds–const : ∀ S → (c : Maybe (CallStackChg S))
  → dsdiff (csChg S c) ≡ StackDiff.dempty
ds–const S (just x) = refl
ds–const S nothing = refl
```

- если набор изменений состояния исполнителя построен как набор изменений, производимых инструкцией, то набор изменений стека вызовов пуст;

```
cs–lemma : ∀ S → (c : SmallChg S)
  → csdiff (sChg c) ≡ StackDiff.dempty
cs–lemma S (onlyreg x) = refl
cs–lemma S (onlystack x) = refl
cs–lemma S (regstack x x₁) = refl
```

- применение набора изменений, построенных как набор изменений стека вызовов, к состоянию исполнителя изменяет только стек вызовов, оставляя остальное неизменным.

```
dapply–csChg : ∀ S → (c : Maybe (CallStackChg S))
  → dapply S (csChg S c)
  ≡ statetype
    (StateType.registers S)
    (StateType.memory S)
    (StateType.datastack S)
    (StackDiff.dapply (RegTypes × DataStackType)
      (StateType.callstack S) (csdiff (csChg S c)))
dapply–csChg S (just x) = refl
dapply–csChg S nothing = refl
```

Проблемой предыдущей реализации было то, что для некоторых блоков важно было их расположение в памяти, из-за чего определить, какой блок будет исполняться следующим, не всегда представлялось возможным. Если потребовать, чтобы все управляющие инструкции задавали явно все требуемые значения, не рассчитывая на определенное расположение в памяти, проблема не будет возникать. Такие управляющие инструкции могут отличаться от имеющихся в реальном ассемблере, но каждая из них должна транслироваться в реальный ассемблер с сохранением семантики и возможным добавлением дополнительных переходов.

```
exec-block : ∀ {ST d} → State ST → Block ST d
  → State (dapply ST d)
  × Σ (Diff (dapply ST d)) (Block (dapply ST d))
exec-block {S} (state Γ Ψ DS CS) (Blocks.↝ {c} ci)
  rewrite reg-const S c | ds-const S c
  = (state Γ Ψ DS CS') , next-block
  where
  ecr = exec-control (state Γ Ψ DS CS) ci
  CS' = proj₁ ecr
  next-block : Σ
    (Diff
      (statetype (StateType.registers S) (StateType.memory S)
        (StateType.datastack S)
        (StackDiff.dapply (RegTypes × DataStackType)
          (StateType.callstack S) (csdiff (csChg S c)))))
    (Block
      (statetype (StateType.registers S) (StateType.memory S)
        (StateType.datastack S)
        (StackDiff.dapply (RegTypes × DataStackType)
          (StateType.callstack S) (csdiff (csChg S c)))))
  next-block rewrite sym (dapply-csChg S c) = proj₂ ecr
exec-block {S} (state Γ Ψ DS CS) (Blocks._·_ {c} {d} i b)
  rewrite cs-lemma S c
    | RegDiff.dappend-dapply-lemma
      (StateType.registers S)
      (rdiff (sChg c))
      (rdiff d)
    | StackDiff.dappend-dapply-lemma RegType
      (StateType.datastack S)
      (dsdiff (sChg c))
      (dsdiff d)
    = exec-block (state Γ' Ψ' DS' CS) b
  where
  eir = exec-instr (state Γ Ψ DS CS) i
```

```
Γ′      = proj₁ eir
Ψ′      = proj₁ (proj₂ eir)
DS′ = proj₂ (proj₂ eir)
```

3.3.4 Модуль Eq

Эквивалентность блоков определяется аналогично приведенному выше.

```
module Eq
  (Block : (S : StateType) → Diff S → Set)
  (exec-block : ∀ {ST d} → Values.State Block ST → Block ST d
    → Values.State Block (dapply ST d)
    × Σ (Diff (dapply ST d)) (Block (dapply ST d)))
  where
  open Values Block

  data BlockEq
    : {ST₁ ST₂ : StateType}
    → {d₁ : Diff ST₁} {d₂ : Diff ST₂}
    → (S₁ : State ST₁) (S₂ : State ST₂)
    → Block ST₁ d₁ → Block ST₂ d₂
    → Set
    where
    equal : ∀ {ST}
          → {S : State ST} {d : Diff ST} {A : Block ST d}
          → BlockEq S S A A
    left  : ∀ {ST₁ ST}
          → {d₁ : Diff ST₁} {d₂ : Diff (dapply ST₁ d₁)}
          → {d : Diff ST}
          → {S₁ : State ST₁} {S₂ : State (dapply ST₁ d₁)}
          → {S : State ST}
          → {A₁ : Block ST₁ d₁} {A₂ : Block (dapply ST₁ d₁) d₂}
          → {B : Block ST d}
          → exec-block S₁ A₁ ≡ S₂ , d₂ , A₂
          → BlockEq S₂ S A₂ B
```

```
      → BlockEq S₁ S A₁ B
  right : ∀ {ST₁ ST}
      → {d₁ : Diff ST₁} {d₂ : Diff (dapply ST₁ d₁)}
      → {d : Diff ST}
      → {S₁ : State ST₁} {S₂ : State (dapply ST₁ d₁)}
      → {S : State ST}
      → {A₁ : Block ST₁ d₁} {A₂ : Block (dapply ST₁ d₁) d₂}
      → {B : Block ST d}
      → exec-block S₁ A₁ ≡ S₂ , d₂ , A₂
      → BlockEq S S₂ B A₂
      → BlockEq S S₁ B A₁
```

3.4 Ассемблер x86-64

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

open Blocks ControlInstr Instr
open Values Block
```

Как было сказано ранее, определяемые инструкции могут не совпадать с имеющимися в реальном ассемблере. Такой инструкцией является, например, инструкция `call`. Дополнительным параметром она принимает указатель на блок, который будет добавлен на стек вызовов. В реальный ассемблер эта инструкция может быть транслироваться двумя инструкциями: `call` нужного блока, за которым следует `jmp` на второй указанный блок.

Многие реализованные инструкции не требуются для реализации блока PLT и приведены здесь, чтобы показать возможность корректного определения подобных инструкций.

```
data ControlInstr (S : StateType) where
  call : ∀ {Γ DS}
      → (f : block
        (StateType.registers S)
        (StateType.datastack S)
        ((Γ , DS) :: StateType.callstack S)
```

```
                ∈ StateType.memory S)
            → (cont : block Γ DS (StateType.callstack S)
                ∈ StateType.memory S)
            → ControlInstr S (just $ StackDiff.push (Γ , DS))
        jmp[_] : (ptr : atom
                (block
                (StateType.registers S)
                (StateType.datastack S)
                (StateType.callstack S) *)
                ∈ StateType.memory S)
            → ControlInstr S nothing
        jmp : (f : block
            (StateType.registers S)
            (StateType.datastack S)
            (StateType.callstack S)
            ∈ StateType.memory S)
      → ControlInstr S nothing
        ret  : ∀ {CS}
            → (p : StateType.callstack S
            ≡ (StateType.registers S , StateType.datastack S) :: CS)
            → ControlInstr S (just (StackDiff.pop p))
```

Определенные инструкции являются примером того, как можно реализовать работу с регистрами и стеком данных.

```
data Instr (S : StateType) where
    mov  : ∀ {σ τ}
            → (r : σ ∈ StateType.registers S)
            → RegValue (StateType.memory S) τ
            → Instr S (onlyreg (RegDiff.chg r τ))
    push : ∀ {τ}
            → τ ∈ StateType.registers S
            → Instr S (onlystack (StackDiff.push τ))
    pop  : ∀ {σ τ DS}
            → (r : σ ∈ StateType.registers S)
```

```
                → (p : StateType.datastack S ≡ τ :: DS)
                → Instr S (regstack (RegDiff.chg r τ) (StackDiff.pop p))
```

Функции, определяющие результаты исполнения заданных инструкций, определяются тривиально.

```
exec-control : ∀ {S c}
      → State S
      → ControlInstr S c
      → CallStack
        (StateType.memory S)
        (StateType.callstack (dapply S (csChg S c)))
      × Σ (Diff (dapply S (csChg S c)))
        (Block (dapply S (csChg S c)))
exec-control (state Γ Ψ DS CS) (call f cont)
  = cont :: CS , loadblock Ψ f
exec-control (state Γ Ψ DS CS) (jmp[ p ])
  = CS , loadblock Ψ (loadptr Ψ p)
exec-control (state Γ Ψ DS CS) (jmp f)
  = CS , loadblock Ψ f
exec-control (state Γ Ψ DS (f :: CS)) (ret refl)
  = CS , loadblock Ψ f

exec-instr : ∀ {S c}
  → State S
  → Instr S c
  → Registers
    (StateType.memory S)
    (StateType.registers (dapply S (sChg c)))
  × (Data (StateType.memory S)
  × DataStack
    (StateType.memory S)
    (StateType.datastack (dapply S (sChg c))))
exec-instr (state Γ Ψ DS CS) (mov r x)
  = toreg Γ r x , Ψ , DS
exec-instr (state Γ Ψ DS CS) (push r)
```

```
    = Γ , Ψ , fromreg Γ r :: DS
exec-instr (state Γ Ψ (v :: DS) CS) (pop r refl)
    = toreg Γ r v , Ψ , DS

open ExecBlk Instr ControlInstr exec-instr exec-control
open Eq Block exec-block
```

3.5 Компоновка кода

Преобразование памяти определяется аналогично приведенному в первой реализации. Все, кроме блоков, остается неизменным, а на каждый блок дополнительно добавляются элементы PLT и GOT.

```
pltize : DataType → DataType
pltize [] = []
pltize (atom x :: Ψ) = atom x :: pltize Ψ
pltize (block Γ DS CS :: Ψ)
    = block Γ DS CS
    :: (atom (block Γ DS CS *)
    :: (block Γ DS CS
    :: pltize Ψ))
```

Зная, по какому адресу находилась функция в памяти без GOT и PLT, можно получить адреса в измененной памяти для:

- соответствующего этой функции элемента PLT;

```
plt : ∀ {Γ Ψ DS CS} → block Γ DS CS ∈ Ψ
    → block Γ DS CS ∈ pltize Ψ
plt (here refl) = here refl
plt {Ψ = atom x :: Ψ} (there f) = there $ plt f
plt {Ψ = block Γ DS CS :: Ψ} (there f)
    = there (there (there (plt f)))
```

- соответствующего этой функции элемента GOT;

```
got : ∀ {Γ Ψ DS CS} → block Γ DS CS ∈ Ψ
    → atom (block Γ DS CS *) ∈ pltize Ψ
got (here refl) = there (here refl)
got {Ψ = atom x :: Ψ} (there f) = there $ got f
```

```
got {Ψ = block Γ DS CS :: Ψ} (there f)
  = there (there (there (got f)))
```

- самой функции.

```
func : ∀ {Γ Ψ DS CS} → block Γ DS CS ∈ Ψ
  → block Γ DS CS ∈ pltize Ψ
func (here refl) = there (there (here refl))
func {Ψ = atom x :: Ψ} (there f) = there $ func f
func {Ψ = block Γ DS CS :: Ψ} (there f)
  = there (there (there (func f)))
```

Блок PLT выглядит так же, как и в первой реализации.

```
plt–stub : ∀ {Γ Ψ DS CS} → atom (block Γ DS CS *) ∈ Ψ
  → Block (statetype Γ Ψ DS CS) dempty
plt–stub got = ⇝ jmp[ got ]
```

Опишем важное свойство: элемент GOT корректно заполнен, если в нём действительно находится указатель на соответствующую этому элементу функцию.

```
GOT[_]–correctness : ∀ {Γ Ψ DS CS}
  → (f : block Γ DS CS ∈ Ψ)
  → (H : Data (pltize Ψ))
  → Set
GOT[ f ]–correctness H = loadptr H (got f) ≡ func f
```

3.6 Доказательства

Для доказательства эквивалентности вызовов функции и соответствующего ей элемента PLT потребуются несколько лемм:

- состояние исполнителя в момент непосредственного вызова функции эквивалентно состоянию исполнителя после исполнения непрямого `jmp` по указателю на ее тело;

```
exec–ijmp : ∀ {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)
        exec-ijmp S p = refl
```

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

```
        exec-plt : ∀ {Γ Ψ DS CS}
          → (f : block Γ DS CS ∈ Ψ)
          → (S : State (statetype Γ (pltize Ψ) DS CS))
          → GOT[ f ]-correctness (State.memory S)
          → exec-block S (plt-stub (got f))
          ≡ S , loadblock (State.memory S) (func f)
        exec-plt f S p rewrite sym p = exec-ijmp S (got f)
```

Используя эти леммы, можно доказать, что если GOT заполнен корректно, то верна внешняя эквивалентность блока PLT, использующего соответствующий функции элемент GOT, и самой функции:

```
        proof : ∀ {Γ Ψ DS CS}
          → (f : block Γ DS CS ∈ Ψ)
          → (S : State (statetype Γ (pltize Ψ) DS CS))
          → GOT[ f ]-correctness (State.memory S)
          → BlockEq S S
            (plt-stub (got f))
            (proj₂ $ loadblock (State.memory S) (func f))
        proof f S p = left (exec-plt f S p) equal
```

3.7 Выводы по главе

В данной главе была представлена реализация метода доказательства экиввалентности ассемблерных кодов, не обладающая недостатками первых

решений, и на основе этого доказана эквивалентность вызова функции и соответствующего ей блока PLT. Благодаря удачно подобранным определениям результирующие доказательства получились весьма короткими.

ЗАКЛЮЧЕНИЕ

В настоящей работе был предложен метод формального доказательства эквивалентности результатов статической и динамической компоновки блоков ассемблерного кода, позволяющий доказывать следующие утверждения:

- интенсиональная эквивалентность состояний исполнителя в момент непосредственного вызова функции и после исполнения непрямого `jmp` по указателю на тело функции;
- интенсиональная эквивалентность состояний исполнителя в момент непосредственного вызова функции и после исполнения соответствующего этой функции элемента PLT при условии корректно заполненного GOT;
- внешняя эквивалентность (эквивалентность вызовов) некоторой функции и соответствующего ей элемента PLT при условии корректно заполненного GOT.

При помощи этого метода был разработан фреймворк, формализующий:

- подмножество ассемблера x86-64 на базовых блоках, необходимое для описания элементов PLT (на самом деле больше) со стеком и типизированной статической памятью;
- подмножество ABI, касающееся компоновки с динамическим связыванием.

Существенная часть разработанного фреймворка может быть переиспользована для доказательства утверждений о других ассемблерах благодаря ее обобщению до метаассемблера, позволяющего автоматически получать большинство определений используемых сущностей, требующего от разработчика минимального описания используемого языка:

- имена типов инструкций и управляющих инструкций;
- функции, описывающие семантику исполнения таких инструкций.

Возможные направления дальнейшей работы:

- реализация динамического компоновщика с использованием приведенного фреймворка, корректно преобразующего код и добавляющего таблицы GOT и PLT;

- дальнейшее обобщение разработанного фреймворка в области компоновки (выделение части фреймворка в мета-компоновку);
- реализация ленивой динамической компоновки.

СПИСОК ИСТОЧНИКОВ

1. Leroy Xavier. CompCert. [Электронный ресурс]. URL: http://compcert.inria.fr/.
2. Zdancewic Steve, Martin Milo M. K., Zhao Jianzhou [и др.]. Vellvm. [Электронный ресурс]. URL: http://www.cis.upenn.edu/~stevez/vellvm/.
3. Norell Ulf. Agda. [Электронный ресурс]. URL: http://wiki.portal.chalmers.se/agda/.
4. NICTA. seL4. [Электронный ресурс]. URL: http://sel4.systems/.
5. Pierce Benjamin C., Casinghino Chris, Greenberg Michael. Software Foundations. [Электронный ресурс]. URL: http://www.cis.upenn.edu/~bcpierce/sf/.
6. The Coq development team. The Coq Proof Assistant. [Электронный ресурс]. URL: https://coq.inria.fr/.
7. Lippmeier Ben. Iron Lambda. [Электронный ресурс]. URL: http://iron.ouroborus.net/.
8. From system F to typed assembly language. / J. Gregory Morrisett, David Walker, Karl Crary [и др.] // ACM Trans. Program. Lang. Syst. 1999. Т. 21, № 3. С. 527–568.