Доказательство корректности работы императивных программ с помощью Coq

Руководитель : Дашков Евгений Владимирович
Автор : Деркач Илья

1 Введение

Coq — это интерактивное программное средство доказательства теорем, включающее собственный язык функционального программирования с развитой системой типов. Данное средство позволяет формулировать теоремы, их доказательства и проверять рассуждения на корректность. Проверка типов в Coq гарантирует логическую корректность выполненных шагов в любой момент рассуждения. Вдобавок к этому среда предоставляет высокоуровневые средства для разработки доказательств, включая большую библиотеку общих определений и лемм, мощные тактики для полуавтоматического доказательства некоторых классов утверждений.

В отличие от прошлого семестра, работа с Coq в текущем была связана не с доказательством математических теорем, а с доказательством утверждений о программах. Для этого мне пришлось узнать больше о тактиках в Coq, способах формализаций простейших программ и их свойств. Мне удалось как познакомиться с логиками, упрощающих работу с утверждениями о программах, так и использовать их для решения задач. Моей конечной целью было расширение языка с простейшими инструкциями до императивного языка с оперативной памятью и кучей. Кроме того, от меня требовались формализация и доказательство корректной работы некоторых программ, реализованных на полученном языке. Параллельно мне удалось доказать некоторые свойства о полученном языке и формализовать правила вывода утверждений для некоторых его инструкций.

Работа делится на два файла. Один из них вспомогательный — в нем определяются частичные и тотальные функции, которые используются как часть состояния программы в определенный момент времени. Второй файл непосредственно содержит определение императивного языка программирования, другие вспомогательные определения и утверждения, а также доказанные утверждения о программах. Ниже я постарался написать сопроводительный текст, который демонстрирует основные идеи для решения моей задачи. Описание состоит из четырех частей. В первой рассказывается о том, как реализуется модель памяти в построенном языке. Все, что описано в этой части, содержится в вспомогательном файле `Maps.v`. Во второй части демонстрируется способ реализации императивного языка программирования. Третья сосредоточена на формализации утверждений о программах и логике вывода этих утверждений. Последняя часть содержит в себе описания доказанных свойств и утверждений, связанных с построенным языком. Также в конце текста содержится ссылка на репозиторий с кодом работы.

Помимо ссылки на исходный код, в последнем разделе указаны работы, на которые я опирался при решении задачи. В электронной книге [2] я почерпнул идеи реализации частичных функций и примеры синтаксиса для императивного языка. В книге [3] мною были найдены пригодившиеся в работе теоретическое описание работы с кучей и правил вывода троек Хоара.

2 Реализация задачи

2.1 Функции как модель памяти

Перед тем, как доказывать спецификацию каких-то программ, нужно определиться с тем, какие инструкции будет поддерживать наш язык программирования. В этой части я опишу средства, с помощью которых мы будем реализовывать команды в программе, а в следующей части предоставлю полное описание инструкций и их синтаксис.

Любая программа на нашем простом императивном языке будет совершать действие над памятью, которая будет представлена двумя видами. Первый тип будет хранить значение переменных в каждый момент времени. По умолчанию переменная — это любая строка. Второй тип будет аналогом кучи в современных языках программирования: по указателю мы сможем работать с числом, хранящемуся по этому "адресу". В качестве указателей будем

рассматривать натуральные числа. Каждый из двух видов памяти будет хранить натуральные значения. Отсюда становится понятно, что основное требование к объекту, моделирующему память, — умение сопоставить числу или строке (в зависимости от типа памяти) другое число. Также мы должны иметь возможность менять сопоставляемое значение в некоторый момент времени. Эти требования можно выполнить довольно естественным с помощью функций, именно их мы возьмем в качестве моделей памяти для нашего языка программирования. Функции будут "автоматически"выдавать сопоставляемое число в силу своего определения.

Читателю может показаться странным, что мы хотим рассматривать два вида памяти вместо одного. Однако объяснение этому решению достаточно просто. Во-первых, как было сказано выше, память реализуется с помощью некоторой функции `map`, поэтому желание иметь только один вид памяти вынуждает ее домен иметь тип `Nat` ∪ `String`. Подобная конструкция может усложнить доказательства, но это не основная причина подобного разделения. Во-вторых, этому способствует наше желание улавливать утечки памяти при работе с кучей. Так, при обращении программы к незарезервированной памяти, она должна падать. Подобное поведение должно фиксироваться в утверждениях о программах и, соответственно, в доказательствах. От обращения же к переменным такого поведения не требуется: мы считаем, что переменные не надо декларировать перед их использованием (как, например, это делается в Python). Наконец, принципиальное отличие кучи от памяти для переменных обусловлено тем, что выделенный объем памяти в куче зачастую зависит от вывода и бывает не ограничен какой-то константой. Такое поведение может быть причиной различных ошибок, неудовлетворяющих спецификацию программы. Поэтому для реализации кучи будем использовать частичную функцию (далее `heap`), а при реализации "оперативной памяти"будем использовать тотальную функцию (далее `store`). Давайте разберемся, чем они отличаются друг от друга.

Для начала опишем тип `option` (он задан уже за нас в библиотеке Coq)

```
Inductive option (A : Type) : Type :=
| Some : A -> option A
| None : option A
```

Мы видим индуктивно заданный тип, который имеет два конструктора `Some` и `None`. Фактически это задает тип `option A`, элементы которого имеют вид либо `Some a` (где a - элемент A) , либо `None`. Тогда мы можем определить оперативную память как функцию из строк в натуральные числа:

```
Definition total_map := string -> nat.
```

Куча же будет определена с помощью функции из натуральных чисел в `option nat` (фактически натуральные с `None`):

```
Definition partial_map := nat -> option nat.
```

Разобравшись с тем, как определить функции, нужно сделать так, чтобы они выполняли другие наши критерия. Во-первых, должна быть реализована функция `update`, которая обновляет возвращаемое значение по некоторому ключу. Во-вторых, функция `heap` должна возвращать `None` при обращении по ключу, по которому возвращаемое значение не было ранее задано. Для `store` такого условия нет, она может выводить произвольное число. Однако для определенности мы все же зафиксируем, что `store` сопоставляет в этом случае ключу число 0. Это реализуется следующим образом: изначально оперативная память и куча пусты, то есть определяются как `t_empty` и `empty` соответственно:

```
Definition t_empty : total_map := (fun _ => 0).

Definition empty : partial_map := (fun _ => None).
```

Позже, с помощью функции `update` мы изменяем значения `heap` и `store`. Приведем пример `update` для `total_map` (типа функции `store`), `update` аналогично определяется для `partial_map` :

```
Definition t_update (m : total_map)
                    (x : string) (v : nat) :=
  fun x' => if x =? x' then v else m x'.
```

Фактически `update` является оберткой над изначальной функцией, которая возвращает новое значение, если ключ совпадает с обновленным ключом, и значение старой функции, если значение по ключу не обновлялось.

Таким образом, мы реализовали память и обращение с ней для нашего будущего языка. Для доказательства утверждений о программах нам потребуются некоторые свойства выше заданных функций. Приведем для примера одну из них:

```
Lemma t_update_eq : forall (m : total_map) x v,
    (x !-> v ; m) x = v.
```

Сначала опишем новый синтаксис. Кусок леммы `(x !-> v ; m)` — синоним `t_update m x v`. Тогда лемма утверждает, что для любого состояния оперативной памяти верно, что если обновить переменную `x` значением `v`, то при обращании к новому состоянию оперативной памяти по ключу `x` мы получим `v`. Эта достаточно простая лемма будет использоваться нами в некоторых доказательствах, связанных с изменением памяти. Доказательства ее и более сложных лемм, связанных с тотальными и частичными функциями можно, найти в файле `Maps.v`. Также там можно найти нотации, незатронутые в тексте выше. Важно отметить, что подобным способом, можно задавать полиморфные типы функций, например:

```
Definition polymorphic_total_map (A : Type) := string -> A.
```

Именно в полиморфном по значению виде заданы в моей работе описанные выше функции. Основная идея было показана на конкретных типах для большей ясности. К тому же, для реализации нашего языка программирования не требуется полиморфное определение.

2.2 Определение императивного языка программирования

Как было сказано выше, для того, чтобы доказывать факты о программах, нужно формализовать само определение программы. В первом разделе нам удалось определить работу с памятью, в этом же разделе пойдет речь об формализуемых инструкциях и синтаксисе языка программирования. Но перед тем как описывать инструкции, сначала опишем объекты, с которыми эти инструкции будут работать.

В моделируемом нами языке будет поддерживаться работа с типами `nat` и `bool`. Внимательный читатель может подметить, что булевая арифметика и арифметика натуральных чисел уже встроена в Coq, и такие выражения должны обрабатываться автоматически. Однако это не так. Поскольку наш язык императивный, и в нем имеются присваивания, значение выражение может меняться по ходу исполнения программы и зависит от состоятния ее памяти, а потому обычными операциями здесь ограничиться не получится. Подобные выражения будем называть натуральными (и соответственно булевыми) выражениями. Для этого мы определяем два индуктивных типа `aexp` и `bexp`. Для примера ниже приведено определение `aexp`:

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | AId (x : string)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).
```

На человеческом языке выше написано, что выражением типа "натуральное"являются натуральное число, переменная программы, а также сложение, вычитание и умножение двух натуральных выражений. Теперь мы можем записать, например, следующее определение:

```
Definition example : aexp := ANum 5 + X.
```

Однако, мы хотим не только записывать такие выражения, но и оценивать (то есть непосредственно получать натуральные значения этих выражений). Из-за того, что эти выражения содержат переменные, оценка должна зависеть от текущего состояния оперативной памяти. Ниже приведен пример верно работающей оценки натуральных выражений:

```
Fixpoint aeval (st : store) (a : aexp) : nat :=
  match a with
  | ANum n => n
  | AId x => st x
  | APlus a1 a2 => (aeval st a1) + (aeval st a2)
  | AMinus a1 a2  => (aeval st a1) - (aeval st a2)
  | AMult a1 a2 => (aeval st a1) * (aeval st a2)
  end.
```

По сути мы индуктивно по построению "разбиваем"выражение до натуральных чисел и переменных, которые в свою очередь оцениваются понятным образом. Аналогично этому построены булевы выражения.

После определения выражений, можно описать команды языка. Мы знаем, что работа программы — это выполнение заданных инструкций, которые изменяют внутреннее состояние программы. В нашем случае за внутреннее

состояние мы считаем содержимое памяти (как оперативной, так и кучи) и значения кода возврата на текущий момент (`false` в случае ошибки и `true` иначе).

Для начала перечислим список команд, которые должен поддерживать наш язык, и то, как они изменяют внутреннее состояние программы.

Пропуск. Инструкция не меняет внутреннее состояние программы. Нотация: `SKIP`.

Присваивание из натурального выражения. Инструкция меняет содержимое переменной на значение натурального выражения. Всегда отрабатывает корректно. Нотация: `x ::= a`.

Последовательное исполнение команд. Инструкция выполняет две программы: сначала программу слева, потом программу справа. Всегда отрабатывает корректно. Нотация: `c1 ;; c2`.

Цикл. Инструкция выполняет программу в блоке, пока условие выполняется. Всегда отрабатывает корректно. Нотация: `WHILE b DO c END`.

Условие. Инструкция выполняет программу в первом блоке, если условие истино, иначе программу во втором блоке. Всегда отрабатывает корректно. Нотация: `TEST c1 THEN c2 ELSE c3 FI`.

Присваивание из разыменованного указателя. Инструкция меняет содержимое переменной на хранимое по указателю значение. Отрабатывает корректно только в случае, если переданный указатель был до этого выделен. Нотация: `x := *e`.

Присваивание по указаетелю. Инструкция меняет содержимое указателя на значение натурального выражения. Отрабатывает корректно только в случае, если переданный указатель был до этого момента выделен. Нотация: `*e := x`.

Выделение памяти и присваивание в куче. Инструкция принимает список выражений и выделяет требуемое количество подряд идущих ячеек памяти для последующего копирования значений. Всегда отрабатывает корректно. Нотация: `e :=CONS( l )`.

Освобождение памяти. Инструкция освобождает принимаемый указатель. Отрабатывает корректно только в случае, если переданный указатель был выделен. Нотация: `DISPOSE( e )`.

С помощью этого неформального определения команд можно задать индуктивный тип `com`, включающий в себя все возможные программы, порожденные данными инструкциями. Определение этого типа во многом схоже с ранее определенным `aexp`, само определение можно найти в `main.v`.

Также как и с `aexp` мы должны научиться оценивать программы, а именно получать состояние (состояние памяти и код возврата) на выходе при заданном состоянии на входе. Однако мы не сможем сделать функцию подобную `aeval`. Это связано с тем, что `aeval` является `Fixpoint function`, а значит Coq должен уметь находить ответ рекурсивно за конечное время. Для этого Coq при определении должен найти (сам или с нашим указанием) по какому параметру идет уменьшение при рекурсивном вызове. Иными словами, ему требуется найти параметр, который явно покажет, что вычисление функции в любом случае будет конечно. В нашем случае такого параметра нет, так как проблема остановки неразрешима. Поэтому для оценки будем использовать другой подход, использующий предикаты. А именно, мы индуктивно определяем предикат типа :

```
Inductive ceval : com -> store -> heap -> bool -> store -> heap -> bool -> Prop
```

Его семантика (для простоты и удобства записи в коде) выглядит следующим образом:

```
[ st , h , res ] =[ c ]=> [ st' , h' , res' ]
```

Данный предикат означает, что из одного состояния `[ st , h , res ]` можно перейти в `[ st' , h' , res' ]` с помощью программы `c`, где `st` , `h` , `res` — состояние оперативной памяти, кучи и код возврата соответственно. Кодом возврата называется булевое значение, символизирующее в случае значения `true` корректную работу с кучей до данного состояния. Каждый из конструкторов индуктивного типа соответствует инструкции, которая задает правило вывода этого предиката. В коде есть примеры `stack_easy_example` и `heap_easy_example`, демонстрирующие вывод предиката этого типа для некоторых программ и состояний.

2.3 Утверждения о программах. Тройки Хоара

Формальное определение программ хоть и приближает нас к доказательству фактов о программах, но все же является недостаточным условием для этого. Сначала нам нужно определить, что такое утверждение о состоянии программы. Ниже формальное определение:

```
Definition Assertion := store -> heap -> bool -> Prop.
```

Иными словами, это предикат, зависящий от состояния программы. Но нас интересуют больше не само определение высказывания о состоянии программы, а то, как связаны высказывания о состояниях программы в разный момент времени и сам код. Таким образом, мы получаем определение Хоаровской тройки:

```
Definition hoare_triple (P : Assertion) (c : com) (Q : Assertion) : Prop :=
  forall st h res st' h' res',
     [st, h, res] =[ c ]=> [st', h', res'] ->
     P st h res ->
     Q st' h' res'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q) (at level 90, c at next level).
```

Формализация выше позволяет нам выразить следующее интуитивное понимание "следствия"высказываний о программах: при условии `{{ P }} c {{ Q }}` , если в начале выполняется утверждение `P`, то после выполнения программы `c` выполняется утверждение `Q`. Таким образом, с помощью этих троек можно делать утверждения о программах.

Наконец, формализовав все, что нам нужно, мы можем доказывать теоремы о свойствах программ. Однако, в доказательствах нам каждый раз придется раскрывать тройку Хоара, что неудобно, а в некоторых доказательствах очень громоздко. Поэтому для некоторых инструкций построенного языка программирования стоит сделать правила вывода Хоаровских троек. Для операций, не связанных с кучей, такие правила вывода довольно легки для формализации и доказательства. Приведем в качестве примера одно из них:

```
Theorem hoare_if : forall P Q b c1 c2,
  {{fun st h res => P st h res /\ bassn b st h res}} c1 {{Q}} ->
  {{fun st h res => P st h res /\ ~ (bassn b st h res)}} c2 {{Q}} ->
  {{P}} TEST b THEN c1 ELSE c2 FI {{Q}}.
```

Фактически она утверждает, что если выполнены Хоаровские тройки каждой из двух возможных веток условной инструкции, то выводима соответсвующая тройка и для самой условной инструкции. Это интуитивно понятное правило вывода сокращает многие выкладки в доказательствах, позволяя не раскрывать тройки Хоара в доказательстве. Формальное доказательство этого вывода (как и остальных правил) реализовано в файле.

2.4 Программы и их свойства

В этом разделе я собираюсь кратко перечислить те программы и свойства, которые были доказаны с помощью конструкций, заданных выше. Я не буду представлять их подробные доказательства здесь — их можно прочитать в файле `main.v`.

Самой главной задачей являлось доказательство работы программ в соответствии с их спецификацией, которые могут затрагивать сколь угодно большую память. Для них доказательство корректности не так тривиально, так как формализация промежуточных утверждений-инвариантов не очень проста. В качестве такой программы я рассмотрел линейный поиск максимума в массиве. Ниже теорема, утверждающая ее корректность:

```
Theorem GetMaxCorrect:
  forall l,
  {{(fun st h res => (forall (i: nat), i < len l -> (h i = Some (nth i l 0)) ) /\ res = true)}}
  ( X ::= 0;;
    Y ::= 0;;
    WHILE ~(Y = len l) DO
      Z := *Y;;
      TEST Z <= X
        THEN SKIP
        ELSE X ::= Z
      FI;;
```

```
    Y ::= Y + 1
  END)
{{(fun st h res => max_list (st X) (len l) l /\ res = true)}}.
```

Она утверждает, что если в начальный момент времени память рассматриваемых ячеек кучи зарезервированы, то программа завершится корректно, причем в переменной `X` будет хранится максимум массива.

Также мной было доказано утверждение о том, что программа, меняющая содержимое двух указателей местами, отрабатывает корректно(то есть работает ожидаемым образом и не падает во время исполнения):

```
Theorem SwapCorrect:
  forall x y m n,
  {{(fun st h res => h x = Some n /\ h y = Some m /\ res = true)}}
  (X := *x ;; Y := *y ;; *x := Y ;; *y := X)
  {{(fun st h res => h x = Some m /\ h y = Some n /\ res = true)}}.
```

Кроме утверждений о программах мне удалось доказать общие факты о построенном языке. Например, ниже представлено утверждение о том, что любая упавшая в некоторый момент программа в любом случае не закончит работу корректно:

```
Theorem IsCrashFromCrash: forall c st1 h1 st2 h2 res2,
  [st1, h1, false] =[ c ]=> [st2, h2, res2] -> res2 = false.
```

3 Ссылки

1. Git repository: https://github.com/ilyaderkatch/coq_imp

2. Software Foundations (vol. 1, 2).

3. Gordon M. Background reading on Hoare Logic.