САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-Механический факультет
Кафедра Системного Программирования

Использование proof assistants для описания операционных семантик

Курсовая работа студента 445 группы
Тарана Кирилла Сергеевича

Научный руководитель:
к.ф.-м.н. Булычев Д. Ю.

Заведующий кафедрой:
д.ф.-м.н., профессор Терехов А. Н.

Санкт-Петербург
2013 г.

Содержание

Введение

В данной работе будут рассмотрены возможности систем для интерактивного доказательство теорем (далее *proof assistants*) применительно к описанию операционных семантик — дедуктивных систем, описывающих поведение программ, написанных на языке программирования.

Операционные семантики — один из способов формально описать семантику языка. С их помощью можно описывать интерпретаторы, компиляторы, статические анализаторы и прочие программы, необходимые для работы с языками программирования, таким образом, что о них можно рассуждать формально. В свою очередь, формализация разрабатываемого объекта даёт возможность снизить количество ошибок при реализации, верфицировать полученную программу (т.е. доказать её корректность), а в некоторых случаях и автоматизировать сам процесс разработки.

В то же время, *сертификационное программирование* — направление, в котором при разработке программы используются формальные описания и доказательства её полезных или других интересных свойств. Появилось это направление в связи с развитием средств для интерактивного доказательства и автоматической проверки теорем. Относительно новое поколение таких средств основано на *зависимых типах* и позволяет совместить программу и доказательство её корректности в одном тексте.

Цель курсовой работы — оценить применимость proof assistants в области разработки языков программирования, в частности для:

- описания операционной семантики;
- автоматизированной генерации интерпретатора по операционной семантике (встроенными средствами выбранного инструмента);
- доказательства каких-либо свойств об интерпретации (по возможности автоматического);

1 Обзор proof assistants

В данной работе сделан упор на поколении proof assistants, в основе реализации которых лежит язык программирования с зависимыми типами.

Такие системы позволяют кодировать утверждения теорем и их доказательства средствами самого языка, используя соответствие Карри-Говарда. Поэтому proof assistants на зависимых типах являются примером применения стройной математической теории в программировании.

1.1 Зависимые типы

> Система типов — это гибко управляемый синтаксический метод доказательства отсутствия в программе определённых видов поведения при помощи классификации выражений языка по разновидностям вычисляемых ими значений.
>
> "Types and Programming Languages" [1]

Механизм *зависимых типов* является расширением параметрического полиморфизма.

При наличии параметрического полиморфизма в языке мы можем работать с типами, параметризованными другими типами. В качестве примера приведём `List A` — тип списка с элементами типа `A`.

Код, определяющий этот тип данных:

```
data List (A : Set) : Set where
  cons : A -> List A -> List A
  nil  : List A
```

Примером параметрического полиморфизма могут служить также *generics* в Java и C#. Подробнее про параметричекий полиморфизм можно прочитать в [2].

Зависимые же типы могут быть параметризованы не только типами, но и термами (значениями). Классический пример зависимого типа: `Vector A (n : N)` — список фиксированной длины n.

Код, определяющий этот тип данных:

```
data Vector (A : Set) : N -> Set where
  cons : (n : N) -> A -> Vector A n -> Vector A (S n)
  nil  : Vector A O
-- S - конструктор ненулевых натуральных чисел
-- S n == n + 1
```

Возможность использования термов в качестве типовых параметров позволяет записывать предикаты как типы, а значит, можно использовать предикаты как типы аргументов функций.

Частично примером может служить функция `head` с типом `(n : N) -> Vector A (S n) -> A`, возвращающая первый элемент списка типа `Vector`. Эта функция ожидает в качестве аргумента терм типа `Vector A (S n)`, что исключает воможность вызвать `head` от пустого списка.

Числа `(S n)` здесь являются конструктивными доказательствами предиката "список не пуст". Этот предикат задан неявно, как часть определения типа `Vector`, но возможно и явное задание в виде обособленного типа вроде `Predicate (p : Proof)` с доказательствами его выполнения в виде элементов типа `Proof`.

Описания подобных приёмов можно почерпнуть из [3].

Другая идиома, которую открывают зависимые типы — *представления* (*views*). Она заключается в особом описании типов с параметрами таким образом, что сопоставление элементов этих типов с образцом сообщает дополнительную информацию об элементе-параметре.

Пример:

```
data Parity : N -> Set where
  even : (k : N) -> Parity (k * 2)
  odd  : (k : N) -> Parity (k * 2 + 1)
```

Далее мы можем использовать элементы типа `Parity` для некоторого числа n, чтобы разложить аргумент n некоторой функции либо на `k * 2`, либо на `k * 2 + 1` в зависимости от значения `parity` для n:

```
f : N -> ...
f n with parity n -- parity : (n : N) -> Parity n
f .(k * 2)     | even k = ...
f .(k * 2 + 1) | odd  k = ...
-- точка означает автоматически выводимый шаблон
```

В дальнейшем эта техника будет использована для описания операционной семантики.

1.2 Соответствие Карри-Говарда

Соответствие Карри-Говарда — наблюдаемое структурное сходство между компьютерными программами и математическими доказательствами. Его суть состоит в том, что любой типизированной системе соответствует некоторая логика, и наоборот.

Именно благодаря этому соответствию мы можем использовать зависимые типы для выражения условий. Можно сказать, что высказывания есть типы, а их доказательства являются программами.

Известная книга Б. Пирса по теории типов так объясняет зависимые типы: “В конструктивных логиках доказательство утверждения P состоит в демонстрации конкретного *свидетельства* в пользу P. Карри и Говард заметили, что это свидетельство во многом похоже на вычисление. Например, доказательство утверждения $P \Rightarrow Q$ можно рассматривать как механическую процедуру, которая, получая доказательство P, строит доказательство Q.” [1]

В рамках соответствия Карри-Говарда следующие структурные элементы рассматриваются как аналогичные:

высказывание P	тип `P`
доказательство высказывания P	терм типа `P`
утверждение P доказуемо	тип `P` обитаем
конъюнкция $P \wedge Q$	кортеж `P` $\times$ `Q`
дизъюнкция $P \vee Q$	размеченное объединение `P` $+$ `Q`
импликация $P \Rightarrow Q$	функциональный тип `P` $\to$ `Q`
истинная формула	тип с единственным элементом
ложная формула	тип без элементов
квантор всеобщности $\forall$	зависимое произведение $\prod$
квантор существования $\exists$	зависимая сумма $\sum$

1.3 Особенности конкретных систем

Прежде чем рассмотреть различные proof assistants, необходимо определить понятия *интерактивного* и *автоматического* построения доказательства. *Интерактивным* способом мы будем обозначать взаимодействие пользователя с системой на протяжении всего процесса с целью управления доказательством, а *автоматическим* способом — взаимодействие, полностью исключающее вмешательство пользователя, кроме формулирования цели (утверждения, теоремы).

1.3.1 Coq

Одна из самых известных и зрелых систем для интерактивного доказательства теорем — Coq[1]. Как и все инструменты, рассмотренные в данной работе, Coq использует функциональный язык программирования с зависимыми типами — Gallina — для описания структур данных, определения функций и типов.

Интерактивное и автоматическое доказательство теорем в Coq осуществляется с помощью *тактик* — программ для обработки контекста доказательства (контекст доказательства содержит переменные-гипотезы подобно контексту вычислений, содержащему переменные-значения). Тактики можно писать на специальном языке Ltac; вдобавок, есть встроенная библиотека тактик. Некоторые тактики способны полностью автоматически доказывать теоремы для алгоритмически разрешимых теорий (например, *omega* для арифметики Пресбургера).

Присутствует автоматизация и в автоматическом выводе типов — можно опускать некоторые типы в сигнатурах функций, которые Coq может вычислить самостоятельно.

Coq также позволяет разрабатывать и верифицировать программы на основе одного исходного кода, т.к. обладает механизмом извлечения кода на языках OCaml, Haskell и Scheme из исходных текстов Coq. При этом верификационная информация стирается для большей производительности.

[1] http://coq.inria.fr

1.3.2 Agda

Система Agda[2] более молодая и имеет меньше средств для автоматизации доказательств.

В отличие от Coq механизма тактик в Agda нет, как и разделения доказательства на фазы формулировки и собственно доказательства. Вместо этого теоремы определяются как простые функции (из соответствия Карри-Говарда следует, что функции и доказательства теорем суть одно и то же). Однако отсутствие тактик в некоторой степени компенсируется *рефлексией* в последних версиях системы, т.е. возможностью внутри доказательства обратиться к утверждению.

В Agda реализована подсистема Agsy для автоматического поиска *type inhabitant*, т.е. значения заданного типа. С помощью неё можно автоматически искать доказательства теорем, но этот инструмент недостаточно зрелый и имеет ряд ограничений.

Кроме того, автоматически можно разбивать аргументы функций на конструкторы, что также автоматизирует построение доказательства.

[2] http://wiki.portal.chalmers.se/agda

2 Операционные семантики

Операционная семантика [4] программы задаёт отношение на множестве состояний и определяется набором правил вывода.

Обозначения следующие:

$< S; s > \to s'$	программа S переводит состояние s в s'
$s[k \mapsto v]$	модификация состояния s присвоение значения v переменной k
$\dfrac{A \qquad B}{C}$	составное правило A и B — посылки, а C — заключение

2.1 Язык While

Примеры правил будут приведены для операционных семантик *большого шага* и *малого шага* для языка While [4]. Вкратце опишем синтаксис этого языка.

Для языка While определены следующие синтаксические категории и мета-переменные:

- категория **Num** из нумералов и мета-переменная n для них;
- категория **Var** из переменных вместе с мета-переменной x;
- категория **Aexp**, состоящая из арифметических выражений, и мета-переменная a;
- категория **Bexp**, содержащая логические выражения, и мета-переменная b для них;
- категория **Stm** инструкций языка While и мета-переменная S.

Далее приведём синтаксис в форме Бэкуса — Наура:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$$

$$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

$$S ::= x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b_1 \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } b \texttt{ do } S$$

2.2 Примеры правил

Приведём правила операционной семантики *большого шага*:

ass (":=")	$\langle x := a, s\rangle \to s[x \mapsto a]$
seq (";")	$\dfrac{\langle p_1, s_0\rangle \to s_1 \qquad \langle p_2, s_1\rangle \to s_2}{\langle p_1; p_2, s_0\rangle \to s_2}$
skip	$\langle skip, s\rangle \to s$
if	1. $s_0[b] = true \Rightarrow \dfrac{\langle p_1, s_0\rangle \to s_1}{\langle \text{if b then } p_1 \text{ else } p_2, s_0\rangle \to s_1}$ 2. $s_0[b] = true \Rightarrow \dfrac{\langle p_2, s_0\rangle \to s_2}{\langle \text{if b then } p_1 \text{ else } p_2, s_0\rangle \to s_2}$
while	1. $s_0[b] = true \Rightarrow$ $\dfrac{\langle p, s_0\rangle \to s_1 \qquad \langle \text{while b do p}, s_1\rangle \to s_2}{\langle \text{while b do p}, s_0\rangle \to s_2}$ 2. $s_0[b] = false \Rightarrow \langle \text{while b do p}, s_0\rangle \to s_0$

Здесь *ass*, *skip*, *while-ff* — элементарные правила, не требующие предпосылок, остальные — составные.

Таким образом, каждой программе соответствует *семантическое дерево*, листья которого — применения элементарных правил, а ветви — посылки к составным.

Кроме операционной семантики большого шага (она же *натуральная* семантика) существует понятие операционной семантики *малого шага*, или *структурной*. В этой семантике переходам между состояниями разрешено иметь остаточную программу, за счёт чего упрощаются правила и исчезают правила с двумя посылками.

ass (":=")	$\langle x := a, s\rangle \to s[x \mapsto a]$
seq (";")	1. $\dfrac{\langle p_1, s_0\rangle \to s_1}{\langle p_1; p_2, s_0\rangle \to s_1^{\underline{p_2}}}$ 2. $\dfrac{\langle p_1, s_0\rangle \to s_1^{\underline{p_1'}}}{\langle p_1; p_2, s_0\rangle \to s_1^{\underline{p_1'; p_2}}}$
skip	$\langle skip, s\rangle \to s$
if	1. $s[b] = true \Rightarrow \langle \text{if b then } p_1 \text{ else } p_2, s\rangle \to s^{\underline{p_1}}$ 2. $s[b] = false \Rightarrow \langle \text{if b then } p_1 \text{ else } p_2, s\rangle \to s^{\underline{p_2}}$
while	$\langle \text{while b do p}, s\rangle \to s^{\underline{\text{if b then (p ; while b do p) else skip}}}$

В семантике малого шага в каждой вершине семантического дерева скрыт список остаточных программ, а само дерево вырождено в список; новые узлы этого дерева появляются только благодаря правилам seq. В некотором смысле такое представление программы более структурировано, отсюда и второе название этой семантики.

От натуральной семантики можно перейти к структурной, проигнорировав во всех правилах остаточную программу. В обратную сторону это также верно. В книге [4] доказано, что операционные семантики большого и малого шагов для языка While эквивалентны.

3 Реализация интерпретатора операционной семантики

Рассмотрим поэтапно описание языка While на Agda.

3.1 Синтаксис языка While

Абстрактный синтаксис языка определяется с помощью нескольких простых алгебраических типов данных без параметров. Исходный текст является фактически трансляцией записи Бэкуса — Наура в Agda.

В первую очередь, определим типы данных для алгебраических и логических выражений.

```
data Aexp : Set where
  const : N -> Aexp
  var   : V -> Aexp
  plus  : Aexp -> Aexp -> Aexp
  minus : Aexp -> Aexp -> Aexp
  star  : Aexp -> Aexp -> Aexp
```

Здесь использован тип `V`; предполагается, что он определён заранее. Например, это может быть синоним для типа строк: `V = String`.

```
data Bexp : Set where
  const : Bool -> Bexp
  conj  : Bexp -> Bexp -> Bexp
  neg   : Bexp -> Bexp
  lt    : Aexp -> Aexp -> Bexp
  eq    : Aexp -> Aexp -> Bexp
```

Затем определим тип операторов языка:

```
data Stm : Set where
  skip   : Stm
  comp   : Stm  -> Stm  -> Stm
  assign : V    -> Aexp -> Stm
  while  : Bexp -> Stm  -> Stm
  branch : Bexp -> Stm  -> Stm -> Stm
```

3.2 Семантика языка While

Для задания правил семантик нам понадобятся некоторые семантические операции вроде присвавивания значения переменной или вычисления арифметического выражения.

Не будем приводить их определения, но укажем типы:

`_|>_ : Binding -> State -> State`	установка значения переменной
`_<|_ : State -> V -> N`	получение значения переменной
`_[[_]] : State -> Aexp -> N`	вычисление выражения
`_<<_>> : State -> Bexp -> Bool`	вычисление выражения

Здесь `State` — тип состояний вычисления, определённый как список переменных вместе с их значениями:

```
record Binding : Set where
  constructor _,_
  field
    key   : V
    value : N

State = List Binding
```

Правила операционной семантики мы зададим с помощью зависимого типа `Transition`. Значение этого типа является семантическим деревом некоторой программы и одновременно является доказательством того, что программа *переводит* одно состояние в другое, т.е. завершает свою работу в некотором состоянии при заданном начальном состоянии. Конструкторы `Transition` кодируют метки вершин семантического дерева.

Воспользуемся приёмом, описанным выше, и определим нужный нам тип как *представление*, тогда мы сможем получать информацию о виде посылок правила при сопоставлении этого правила с образцом.

Сигнатура типа (для натуральной семантики) выглядит следующим образом:

```
Transition (s₁ : State) : Stm -> State -> Set
```

Параметр s_1 фиксирован для всего типа в целом, что означает, что начальное состояние для программы задаётся извне и не зависит от самой программы. Остальные параметры не фиксированы: возможные программы и состояния, в которых они заканчиваются, зависят от меток дерева (конструкторов).

Правила без посылок кодируются как следующие конструкторы:

```
[skip]    : Transition s₁ skip s₁
[assign]  : ∀ {k v}
  -> Transition s₁ (assign k v) ((k , s₁ [[ v ]]) |> s₁)
```

Поскольку у элементарных правил нет посылок, то и аргументов у этих конструкторов нет.

Остальные конструкторы:

```
[comp]    : ∀ {s₂ s₃ p₁ p₂}
  -> Transition s₁ p₁ s₂
  -> Transition s₂ p₂ s₃
  -> Transition s₁ (comp p₁ p₂) s₃
[if-t]    : ∀ {s₂ b p₁ p₂} -> T (s₁ << b >>)
  -> Transition s₁ p₁ s₂
  -> Transition s₁ (branch b p₁ p₂) s₂
[if-f]    : ∀ {s₂ b p₁ p₂} -> T (not (s₁ << b >>))
  -> Transition s₁ p₂ s₂
  -> Transition s₁ (branch b p₁ p₂) s₂
[while-t] : ∀ {s₂ s₃ p} -> (b : B) -> T (s₁ << b >>)
  -> Transition s₁ p s₂
  -> Transition s₂ (while b p) s₃
```

```
    -> Transition s₁ (while b p) s₃
  [while-f] : ∀ {p} -> (b : B) -> T (not (s₁ << b >>))
    -> Transition s₁ (while b p) s₁
```

При описании структурной семантики программа может завершиться с остаточной программой. Для выражения этого введём вспомогательный тип `Control` для состояний вместе с возможной остаточной программой и заменим второй параметр типа `State` в `Transition` на параметр типа `Control`.

Код, необходимый для определения правил структурной семантики:

```
record Control : Set where
  constructor [>_,_<]
  field
    state   : State
    program : Maybe S

data Transition (s₁ : State) : S -> Control -> Set where
  [skip]    : Transition s₁ skip [> s₁ , nothing <]
  [assign]  : ∀ {k v }
    -> Transition s₁ (assign k v)
      [> (k , s₁ [[ v ]]) |> s₁ , nothing <]
  [comp-j]  : ∀ {s₂ p₁ p₂ p₃}
    -> Transition s₁ p₁ [> s₂ , just p₃ <]
    -> Transition s₁ (comp p₁ p₂) [> s₂ , just (comp p₃ p₂) <]
  [comp-n]  : ∀ {s₂ p₁ p₂   }
    -> Transition s₁ p₁ [> s₂ , nothing <]
    -> Transition s₁ (comp p₁ p₂) [> s₂ , just p₂ <]
  [if-t]    : ∀ (b : B) (p₁ p₂ : S) -> T (s₁ << b >> )
    -> Transition   s₁ (branch b p₁ p₂) [> s₁ , just p₁ <]
  [if-f]    : ∀ (b : B) (p₁ p₂ : S) -> T (not (s₁ << b >>))
    -> Transition   s₁ (branch b p₁ p₂) [> s₁ , just p₂ <]
  [while-t] : ∀ (b : B) (p : S) -> T (s₁ << b >> )
    -> Transition s₁ (while b p) [> s₁ , just p <]
  [while-f] : ∀ (b : B) (p : S) -> T (not (s₁ << b >>))
    -> Transition s₁ (while b p) [> s₁ , nothing <]
```

Однако, теперь тип `Transition` задаёт более “подробное” отношение, чем нужно: оно определено также на тех значениях `Control`, символизирующих “незавершённые” вычисления, которые содержат остаточную программу. Поэтому определим дополнительно транзитивное замыкание этого отношения `Transition*` для выражения “завершённых” вычислений:

```
data Transition* (s₁ : State) : S -> State -> Set where
  cons : ∀ {s₂ s₃ p₁ p₂}
    -> Transition s₁ p₁ [> s₂ , just p₂ <]
    -> Transition* s₂ p₂ s₃ -> Transition* s₁ p₁ s₃
  base : ∀ {s₂ p}
    -> Transition s₁ p  [> s₂ , nothing <]
    -> Transition* s₁ p  s₂
```

В сущности, `Transition*` — тип для списков семантических деревьев таких, что программа каждого следующего дерева является остаточной программой предыдущего.

3.3 Интерпретация программ

Интерпретатором программы мы будем называть функцию `interpret`, строящую семантическое дерево по данному состоянию и заданной программе. Тип этой функции должен быть `(s : State) -> (p : Stm) -> Interpretation s p`, где `Interpretation` — пара, состоящая из конечного состояния и семантического дерева. Отдельный тип введён для того, чтобы выразить зависимость семантического дерева от конечного состояния.

Определения `Interpretation` и `interpret`:

```
record Interpretation (s₁ : State) (p : Stm)
  : Set where
  constructor I_,_I
  field
    state      : State -- для структурной семантики Control
    transition : Transition s₁ p state

interpret : (s : State) -> (p : S)
  -> Interpretation s p
```

Такие типы называют *зависимыми записями*.

Семантическое дерево может быть бесконечного размера из-за таких правил как while. Поэтому функция `interpret` может работать бесконечно. Например, при обработке конструкции while необходимо рекурсивно вызывать `interpret` для одной и той же программы, нарушая принцип *структурной рекурсии*. Поэтому мы не можем гарантировать завершаемость `interpret`.

В то же время Agda по умолчанию производит проверку завершаемости (*termination checking*) для всех своих программ. Следует выключить эту проверку для нашего интерпретатора.

При этом, `interpret` для структурной семантики всегда завершается, т.к. вычисляет только один “шаг” программы, а остальную часть выдаёт в качестве остаточной программы. Необходимо, реализовать также `interpret*` для вычисления замыкания `Transition*`. Для этой функции уже не гарантирована завершаемость, однако в отличие от семантики большого шага мы теперь разделили интерпретатор на две части по признаку завершаемости.

Что касается реализации `interpret`, то существует возможность выполнить её интерактивно, используя механизмы разделения по случаям (*case split*) и поиска значений типа.

Эти механизмы позволяют в полу-автоматическом режиме развить незавершённое уравнение функции:

```
interpret : (s : State) -> (p : S) -> Interpretation s p
interpret s p = {!!} -- {!!} -- цель доказательства
```

в следующий код:

```
interpret : (s : State) -> (p : S) -> Interpretation s p
interpret s₁ (skip) = I s₁ , [skip] I
interpret s₁ (assign k v)
  = I (k , (s₁ [[ v ]])) |> s₁ , [assign] I
interpret s₁ (comp p₁ p₂) with interpret s₁ p₁
... | I s₂ , tr₁ I with interpret s₂ p₂
... | I s₃ , tr₂ I = I s₃ , [comp] tr₁ tr₂ I
interpret s₁ (branch b p₁ p₂) with s₁ << b >>
| inspect (_<<_>> s₁) b
| interpret s₁ p₁
| interpret s₁ p₂
... | true  | Reveal_is_.[_] e | I s₂ , tr₁ I | I s₂ , tr₂ I
  = I s₂ , [if-t] (trueIsTrue e) tr₁ I
... | false | Reveal_is_.[_] e | I s₂ , tr₁ I | I s₂ , tr₂ I
  = I s₂ , [if-f] (falseIsNotTrue e) tr₂ I
interpret s₁ (while  b p) with s₁ << b >>
| inspect (_<<_>> s₁) b
... | false | Reveal_is_.[_] e
  = I s₁ , [while-f] b (falseIsNotTrue e) I
... | true  | Reveal_is_.[_] e with interpret s₁ p
... | I s₂ , tr₁ I with interpret s₂ (while b p)
... | I s₃ , tr₂ I = I s₃ , [while-t] b (trueIsTrue e) tr₁ tr₂ I
```

Однако, к сожалению, эти механизмы не настолько зрелы, чтобы полностью автоматизировать реализацию. Поэтому приходится некоторые элементы реализовывать вручную. Здесь введен не-автоматически весь исходный текст, начинающийся ключевым словом `with` или содержащий конструктор `Reveal_is_.[_]`.

В будущем ожидается, что в Agda появится возможность автоматически вводить `with`-выражения. Как альтернатива, возможно применение специальных *элиминирующих* функций вместо `with`-выражений.

3.4 Доказательство свойств

Как пример утверждений о семантике языка, которые можно формулировать и доказывать с помощью построенного описания, приведём доказанный в [4] факт об эквивалентности семантик большого и малого шагов While.

Для избежания конфликта имён, переименуем `Transition` в `Transition-NS` и `Transition-SS` для натуральной и структурной семантик соответственно. Транзитивное замыкание `Transition*` для структурной семантики переименуем в `Transition-SS*`.

Запишем утверждение об эквивалентности в виде сигнатуры функции:

```
ss=ns : (s : State) -> (p : S) -> ∀ {s-ss s-ns}
    -> (tr-ss : Transition-SS* s p s-ss)
    -> (tr-ns : Transition-NS  s p s-ns)
    -> s-ss ≡ s-ns
```

Расшифровать тип этой функции можно следующим образом: для любых заданных состояний `s`, `s-ss`, `s-ns` и программе `p`, если есть семантические деревья `p` с начальным состоянием `s` и конечными состояниями `s-ns` и `s-ss` для натуральной и структурной семантик соответственно, то эти состояния `s-ns` и `s-ss` равны.

Приведём доказательство этого утверждения для базового случая, когда после интерпретации в структурной семантике отсутствует остаточная программа:

```
ss=ns-base : (s : State) -> (p : S) -> ∀ {s-ss s-ns}
  -> (tr-ss : Transition-SS s p [> s-ss , nothing <])
  -> (tr-ns : Transition-NS s p    s-ns)
  -> s-ss ≡ s-ns

ss=ns-base  .s-ns .skip {.s-ns} {s-ns}
  [skip] [skip] = refl
ss=ns-base   s    .(assign k v)
  ([assign] {k} {v}) [assign] = refl
ss=ns-base  .s-ns .(while  b p) {.s-ns} {s-ns}
  ([while-f] b p x) ([while-f] .b x₁) = refl
```

Стоит заметить, что во всех уравнениях определения достаточно одного только `refl` (свойство рефлексивности равенства) в качестве правых частей. Однако надо правильным образом составить левые части, чтобы типизатор Agda мог убедиться в корректности.

Разработка доказательства производилась с использованием тех же средств интерактивного доказательства, что и построение интерпретатора.

Заключение

В ходе работы стало ясно, что proof assistants вполне применимы для описания операционных семантик. Удалось задать несколько семантик для тестового языка и построить для них интерпретаторы вместе с частичным доказательством их эквивалентности для данного языка.

Получилось выполнить разработку интерактивно с использованием встроенных в Agda средств: разделения по случаям и автоматического поиска решений. Однако полностью автоматически провести построение не удалось. Планируется в дальнейшем опробовать другие средства, предоставляемые proof assistants, для достижения этой цели: тактики, рефлексию и прочее.

Список литературы

[1] B. C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, USA, 2002.

[2] C. Strachey, Fundamental concepts in programming languages, Higher-Order and Symbolic Computation 13 (1/2) (2000) 11–49.

[3] U. Norell, Dependently typed programming in agda, in: A. Kennedy, A. Ahmed (Eds.), TLDI, ACM, 2009, pp. 1–2.

[4] H. R. Nielson, F. Nielson, Semantics with applications: a formal introduction, John Wiley & Sons, Inc., New York, NY, USA, 1992.