УДК 1(091)
DOI: 10.17223/1998863X/52/3

**О.А. Доманов**

# ФОРМАЛИЗАЦИЯ КОНТЕКСТОВ В ТЕОРИИ ТИПОВ С ЗАПИСЯМИ И МОДУЛЯМИ

*Представлен подход к формализации контекстов в семантике естественного языка, основанный на теории типов с записями и модулями. Описаны общие принципы формализации и ее конкретная реализация в системе Agda. Показана связь данного подхода с ситуационной семантикой Дж. Барвайса и Дж. Перри. Приведены примеры описания референциальной непрозрачности контекстов, а также понятий абстрактного и конкретного смыслов из ситуационной семантики. Код формализации свободно доступен.*
Ключевые слова: *семантика естественного языка, теория типов, Agda, пропозициональные установки, ситуационная семантика.*

Одна из основных проблем теоретико-типовой семантики [1] состоит в отсутствии адекватных методов работы с контекстами. Для восполнения этого дефицита Р. Купером была предложена теория типов с записями [2, 3]. Она, однако, обладает ограниченными возможностями в формализации таких феноменов, как референциальная непрозрачность контекстов. В то же время многие функциональные языки программирования, основанные на теории типов, содержат развитые модульные системы, позволяющие селективно скрывать или открывать информацию тех или иных фрагментов программы. В данной статье мы рассмотрим возможности, которые открываются для формализации контекстов в теории типов с записями и модулями. Формализация будет проведена на языке программирования с зависимыми типами Agda. Использование языка программирования для работы в теоретико-типовой семантике не случайно. Теория типов тесно связана как с логикой, так и вычислениями, в частности с программированием. В контексте теоретико-типовой семантики мы можем понимать смысл как нечто вычисляемое, поэтому многие семантические проблемы оказываются связанными с разработкой способов (или программ) вычисления смысла или референта. Agda, как и многие системы для работы с доказательствами, не создавалась специально для нужд семантики естественного языка, она, скорее, ориентирована на математику и логику. По этой причине ее ресурсы, возможно, недостаточно адекватны проблеме, поэтому одна из задач исследования состоит в тестировании пределов этой адекватности. Конечной целью (не достигаемой, разумеется, в данной краткой статье) является разработка универсального инструментария для работы с семантическими проблемами.

Код Agda с описываемой формализацией доступен по адресу: https://github.com/odomanov/ttsemantics/.

## Необходимые сведения о синтаксисе Agda

Нам не потребуется Agda во всем объеме, поэтому в данном разделе описано лишь то, что необходимо для дальнейшего изложения (для более подробного введения см., напр.: [4, 5]). Agda является функциональным языком

программирования, основанным на варианте теории типов П. Мартин-Лёфа [6, 7], и во-многом следует ее обозначениям. Программы на Agda состоят из файлов в кодировке Юникод, что позволяет использовать в них большинство математических символов, таких как кванторы, различные стрелки и пр. Типы в Agda понимаются как множества, поэтому запись `A : Set` означает, что `A` является типом (существует также иерархия типов, но она нам не понадобится). Запись `a : A` означает, что `a` относится к типу `A`, т.е. является *термом*, или *элементом*, типа `A`. При определениях сначала декларируется тип определяемого, а затем после равенства записывается само определение, например:

```
   a : A
a = ...
```

Тип функций перехода из типа `A` в тип `B` обозначается как `A → B`. Функция определяется действием на свои аргументы. Например, если тип `A` состоит из двух элементов `Tom` и `Jerry`, то функция в тип натуральных чисел может быть определена следующим образом:

```
   f : A → Nat
f Tom = 1
f Jerry = 3
```

Здесь сначала задается тип функции, а затем – ее значения на элементах типа (Agda требует, чтобы функция была определена на всех этих элементах). При определении функции предполагается, что ее аргументы следуют за ее именем, однако Agda допускает и инфиксную, а также смешанную нотацию:

```
   _+_ : Nat → Nat → Nat
```

Здесь подчерки обозначают места для переменных; применение так определенной функции можно записывать обычным образом как `1 + 4` (но также, при желании, и как `_+_ 1 4`).

Если аргументы функции зависят друг от друга, то соответствующие термы могут быть указаны явно. Например:

```
   f : (n : Nat) → Fin n → C
```

Здесь `Fin n` – (зависимый) тип чисел, меньших `n` (конечное множество).

Согласно соответствию Карри–Ховарда типы соответствуют пропозициям, и существование элемента типа можно рассматривать как существование доказательства пропозиции (и обратно). Поэтому предикаты или пропозициональные функции определяются как функции в `Set`. Например, `A → Set` – это одноместный предикат на типе `A`, `A → B → Set` – двуместный на типах `A`, `B` и т.д. Благодаря наличию этого соответствия доказательства пропозиций обычно выглядят в Agda следующим образом:

```
   prf : Proposition
prf = ...
```

Здесь в первой строчке декларируется существование доказательства для `Proposition`, а затем записывается, чему это доказательство равно. Если имя доказательства несущественно и далее не используется, то допустимо писать:

```
   _ : Proposition
_ = ...
```

Я ниже буду часто пользоваться такой записью, приводя при этом доказательства почти без пояснений – для целей данной статьи их конкретный вид обычно не важен, важно лишь наличие.

Помимо определения или вычисления типы и их элементы могут быть постулированы:

```
  postulate
_runs : A → Set
p : Tom runs
```

Здесь постулировано существование предиката `runs` на типе `A`, а также доказательства пропозиции `Tom runs`. Постулаты, вообще говоря, нарушают принципы конструктивной математики, на которой основана теория Мартин-Лёфа. Например, мы можем постулировать существование доказательства дизъюнкции, не предполагая существования доказательства конкретного члена этой дизъюнкции. В случае семантики, однако, это не является недостатком, поскольку люди могут иметь подобные убеждения, и данное свойство позволяет нам их моделировать.

Данная запись демонстрирует также роль отступов в Agda. Команда `postulate` начинает блок, который отмечается отступом, величина которого фиксируется первой строкой после `postulate`. После этого строки с тем же отступом относятся к начатому блоку, а любая строка с меньшим отступом заканчивает блок. Таким образом, две декларации выше относятся к одному блоку `postulate`. Аналогично любое выражение считается продолжающимся на следующие строки, если они записаны с отступом.

Различные значения в Agda могут определяться как неявные в случаях, когда Agda в состоянии найти их самостоятельно. Эти значения помещаются в фигурные скобки. Например, функция

```
func : {A B : Set} → (f : A → B) → C
```

имеет три аргумента: типы `A` и `B` и функцию `f : A → B`. Однако указания последнего аргумента достаточно для того, чтобы вычислить два первых, поэтому они выше определены как неявные. При применении функции `func` они не указываются.

## Модульная система

Программа на Agda может содержать множество функций, переменных, констант и пр. Было бы неудобно придумывать им всем различные имена. Agda обладает простой, но мощной системой, позволяющей гибко управлять использованием имен. Для этого программа разбивается на модули, в каждом из которых имена должны быть уникальными. Чтобы снаружи модуля `M` обратиться к имени `name`, находящемуся внутри него, следует использовать полное имя `M.name`. Модули могут быть вложены с произвольной глубиной, так что имена также могут иметь несколько уровней: `M1.M2.M3.name`. Модуль задается следующей директивой:

```
module M where
```

Эта директива открывает блок, в котором записываются декларации модуля. Для того чтобы сделать имена объектов модуля доступными вне него не по полным, а по сокращенным именам, модуль может быть открыт:

```
open M
```

При этом Agda содержит средства управления тем, какие именно имена могут быть открыты или, наоборот, скрыты. Для этого существуют модификаторы `using`, `hiding` и `renaming`. Например, команда

```
open M using (n1; n2) renaming (n3 to new3)
```

делает доступными только объекты `n1`, `n2` и `n3`, при этом последний – под именем `new3`. Команда

```
open M hiding (n2; n3)
```

открывает все имена, кроме `n2` и `n3`. И так далее. Модули можно открывать внутри других модулей, при этом если модуль `M2` открывается внутри модуля `M1`, то конструкция

```
module M1 where
 open M2 public
```

означает, что при открытии модуля `M1` имена из `M2` также становятся доступными (с возможными модификаторами `using`, `hiding`, `renaming`).

Модули могут иметь параметры. Например,

```
module M (A : Set) (x : A) where
 f : B
 g : A → B
```

означает, что все, определенное внутри модуля `M`, зависит также от дополнительных аргументов типов `Set` и `A`. То есть при открытии модуля `M` функции, которые становятся доступными, имеют следующие типы:

```
f : (A : Set) → (x : A) → B
g : (A : Set) → (x : A) → A → B
```

Такой параметризованный модуль может быть открыт с явно указанными параметрами, например:

```
open M Nat n
```

В этом случае становятся доступными функции типов

```
f : B
g : Nat → B
```

В результате модульная система предоставляет нам гибкий инструмент для открытия и сокрытия имен, который поможет моделировать прозрачность или непрозрачность контекстов. Для этого нам понадобится также понятие записи.

## Записи, модули и референциальная непрозрачность

Согласно Мартин-Лёфу, основное суждение теории типов `a : A` представляет собой суждение знания. Его смысл (который, как выражается Мартин-Лёф, мы «поясняем») состоит в том, что утверждается существование элемента `a`, относящегося к типу `A`. В таком виде теория типов близка ситуационной семантике [8]. Это обстоятельство, а также роль в нем структуры записей отмечены Купером [3]. Я поэтому буду называть моделируемые контексты ситуациями (хотя их связь с ситуационной семантикой, разумеется, должна быть исследована отдельно). Записи (records) являются стандартной структурой многих языков программирования. С точки зрения теории типов они являются обобщением зависимых Σ-типов или типом кортежей, в котором проекции имеют имена. В упрощенном виде, которого нам для дальнейшего достаточно, тип записи в Agda записывается следующим образом:

```
    record R Δ : Set where
field
f1 : A1
f2 : A2
...
```

Здесь R – имя записи, Δ – параметры, от которых запись может зависеть, f1, f2, … – имена полей записи, т.е. термов типов A1, A2, … (каждый из них может зависеть от предыдущих). Терм записи представляет собой кортеж, состоящий из термов, соответствующих полям, и записывается как

```
    r = record { f1 = a1; f2 = a2; ...}
```

Здесь a1, a2, …– термы типов A1, A2, … С точки зрения теории типов поля являются проекциями Σ-типов, т.е. функциями типов (без учета параметров):

```
    f1 : R → A1
f2 : R → A2
...
```

В частности, для указанного выше терма записи r выполнены равенства

```
    f1 r ≡ a1
f2 r ≡ a2
...
```

(≡ и ≢ обозначают пропозициональное, т.е. доказываемое, равенство и неравенство). Так определенные, записи сходны с модулями. И действительно, Agda для каждой записи определяет соответствующий параметризованный модуль с тем же именем:

```
    module R {Δ} (r : R Δ) where
f1 : A1
f1 = R.f1 r
f2 : A2
f1 = R.f2 r
...
```

(вспомним, что объекты параметризованных модулей являются функциями от параметров). Поэтому записи (фактически, соответствующие им модули) также могут быть открыты, причем допускают использование модификаторов. Более того, при задании записи мы можем добавлять декларации в ее модуль, например:

```
    record R Δ : Set where
field
f1 : A1
f2 : A2
b : B
b = ...
```

Или даже:

```
    record R Δ : Set where
field
f1 : A1
f2 : A2
open M public
```

В последнем случае при открытии записи R будут также доступны определения из модуля M.

Системы знаний, частным случаем которых являются контексты или ситуации, могут быть моделированы либо как модули, либо как записи. Если мы хотим просто вычислить нечто внутри некоторого контекста, то нам достаточно задать модуль с соответствующими определениями и провести в нем необходимые действия. Если же нам требуется метаязык, т.е. высказывания о самих контекстах, композиции контекстов и т.д., то они должны быть определены как термы некоторого типа. В этом случае необходимо формализовать типы контекстов как записи.

Рассмотрим моделирование убеждений (belief reports) на классическом примере референциальной непрозрачности, принадлежащем Куайну [9]: некий Ральф знает двух человек – назовем их человеком в шляпе и человеком на пляже – как двух разных людей, причем первого считает шпионом, а второго нет; мы, однако, знаем, что это один и тот же человек по имени Орткут.

Прежде всего постулируем существование типа Man и предиката «быть шпионом»:

```
   postulate
Man : Set
_is-a-spy : Man → Set
```

Если ситуации определены как записи, то мы можем их комбинировать. Действительно, пусть задан простейший тип ситуаций «Имеется объект типа A»:

```
   record SitSmth (A : Set) : Set where
field
smth : A
```

Как видно, этот тип записей имеет единственное поле, представляющее собой объект типа A. Частным случаем таких ситуаций являются ситуации типа «Имеется человек»:

```
   record SitMan : Set where
field
sitsmth : SitSmth Man
open SitSmth sitsmth renaming (smth to man) public
```

Единственным полем этой ситуации является ситуация типа SitSmth с параметром, равным Man. Эта ситуация открыта публично с переименованием поля (фактически, функции) smth в поле (функцию) man. Мы можем далее использовать эту ситуацию как строительный материал для ситуации «Имеется шпион»:

```
   record SitSpy : Set where
field
sitMan : SitMan
open SitMan sitMan public
field
man-spy : man is-a-spy
```

Эта ситуация состоит из ситуации «Имеется человек» и доказательства того, что этот человек шпион. При этом ситуация SitMan открывается, делая доступным объект man, который затем используется. Подобным образом мы

можем составлять сложные ситуации из простых, открывая (или не открывая) их, добавляя новую информацию и т.д.

Система убеждений Ральфа может быть теперь представлена как модуль

```
   module RalphBelief where
postulate
sitSpy : SitSpy
open SitSpy sitSpy public
```

Модуль `SitSpy` здесь открыт, поэтому мы можем (внутри модуля RalphBelief) записать утверждение о том, что человек, в существование которого верит Ральф, является шпионом, просто как «`man is-a-spy`», причем его доказательством будет просто доступное Ральфу `man-spy`:

```
   _ : man is-a-spy
_ = man-spy
```

Без открытия модуля мы должны были бы писать

```
   _ : SitMan.man (SitSpy.sitMan sitSpy) is-a-spy
_ = SitSpy.man-spy sitSpy
```

Таким образом, вне модуля `RalphBelief` – т.е. для нас, вне системы убеждений Ральфа – некоторые доказательства и даже формулировка пропозиций могут оказаться невозможными, если мы не разделяем суждений, в которые верит Ральф. Мы можем, однако, проводить доказательства в локальном контексте, что записывается следующим образом:

```
   _ : Σ[ x ∈ Man ] x is-a-spy
 _ = man , man-spy
 where
 open RalphBelief
```

Здесь `Σ[ x ∈ A ] B` обозначает тип (Σx : A)B, т.е. экзистенциальную квантификацию (или, при другой интерпретации, – подмножество элементов множества `A`, таких что `B`). Таким образом, доказываемая пропозиция означает «Существует человек, который является шпионом» (а ее доказательством является пара, состоящая из человека и доказательства того, что он шпион). Директива `where` позволяет задать локальный контекст, в котором могут быть определены дополнительные типы, их термы, а также открыты модули. В данном случае, мы открываем модуль убеждений Ральфа, т.е. проводим доказательство «внутри» этих убеждений. Вне контекста Ральфа, т.е. для нас, доказательство превращается в следующее:

```
   _ : Σ[ x ∈ Man ] x is-a-spy
 _ = RalphBelief.man , RalphBelief.man-spy
```

Мы видим, что модульная система позволяют нам разделять наборы убеждений и контролировать то, какая информация доступна нам и Ральфу. Тем самым моделируется контролируемым образом непрозрачность доксических контекстов, которая составляла проблему для Куайна. Действительно, в полном виде убеждение Ральфа можно записать следующим образом:

```
   record SitTwo : Set where
field
{man-in-a-hat} : Man
{man-on-the-beach} : Man
hspy : man-in-a-hat is-a-spy
```

```
bspy : ¬ man-on-the-beach is-a-spy
h≠b : man-in-a-hat ≠ man-on-the-beach

module RalphBelief where
postulate sit : SitTwo
open SitTwo sit public
```

Здесь `SitTwo` – это ситуация, в которой есть два различных человека – в шляпе и на пляже – один из которых шпион, а другой нет. Конкретное строение ситуации `SitTwo` зависит от наших предпочтений, удобства и пр. Например, мы могли бы определить ситуацию `SitMan` как индексированную натуральным числом и содержащую несколько человек (в частном случае – одного). В этом случае `SitTwo` могла бы содержать ситуацию `SitMan 2` вместо двух элементов типа `Man`. Язык предоставляет нам определенную гибкость. Приведенное определение является одним из простейших.

Аналогично предыдущему модуль `RalphBelief` моделирует убеждения Ральфа. Внутри него мы можем легко доказать:

```
   _ : man-in-a-hat is-a-spy
_ = hspy

_ : Σ[ x ∈ Man ] ¬ x is-a-spy
_ = man-on-the-beach , bspy
```

Пусть теперь наша система убеждений выглядит следующим образом:

```
   module OurBelief where
 open RalphBelief using (man-in-a-hat; man-on-the-beach)
 postulate
 Ortcutt : Man
 o≡h : Ortcutt ≡ man-in-a-hat
 o≡b : Ortcutt ≡ man-on-the-beach
```

Здесь в `RalphBelief` открываются только элементы `man-in-a-hat` и `man-on-the-beach`, что означает, что мы разделяем с Ральфом знание о существовании этих людей (для него и для нас это одни и те же люди), но не убеждение об их различии, а также принадлежности или непринадлежности к шпионам. Но зато мы знаем, что оба эти человека являются на самом деле одним и тем же человеком по имени Орткут. Поскольку информация о принадлежности к шпионам не содержится в нашем контексте, мы не можем, например, доказать, что Орткут шпион. Однако мы можем это сделать, если откроем `RalphBelief` полностью (что означает, что мы полностью согласимся с Ральфом):

```
   _ : Ortcutt is-a-spy
 _ = subst (λ x → x is-a-
spy) (sym (proj₂ (proj₂ hs))) hspy
 where
 open RalphBelief
 hs : Σ[ x ∈ Man ] x is-a-spy × Ortcutt ≡ x
 hs = man-in-a-hat , hspy , o≡h
```

Аналогичным образом мы можем доказать, что Орткут не может быть шпионом, что показывает, что наша с Ральфом объединенная система убеж-

дений противоречива (хотя в каждой из них противоречие доказать невозможно). Впрочем, мы можем вывести противоречие более коротким путем, если предварительно докажем (находясь в нашем контексте), что человек в шляпе совпадает с человеком на пляже:

```
   h≡b : man-in-a-hat ≡ man-on-the-beach
h≡b = trans (sym o≡h) o≡b


_ : ⊥
_ = contradiction h≡b (RalphBelief.h≢b)
```

Вместе с тем, как нетрудно видеть, открытие дополнительно, например, только `hspy` (т.е. наше согласие с Ральфом в том, что человек в шляпе шпион) не приводит к противоречию.

Таким образом, теория типов с записями и модульной системой позволяет нам разделять системы убеждений и корректно описывать эффекты непрозрачности контекстов.

## Семантические вычисления: абстрактный и конкретный смысл

Рассмотрим еще один пример, а именно понятия абстрактного и конкретного смыслов, как они определяются в ситуационной семантике.

В ситуационной семантике для существования смысла (meaning) требуется наличие трех условий [10. P. 610–613]: 1) ситуации говорения кем-то некоторых высказываний, 2) описываемой или фокальной ситуации – той, о которой идет речь в ситуации говорения, 3) референций, устанавливаемых говорящим (speaker's connections) и определяющих, что имеет в виду говорящий, употребляя те или иные языковые выражения. Дополнительно могут понадобиться другие ситуации, например ресурсные, которые непосредственно не присутствуют, но необходимы для вычисления значений, такие как общие знания, ситуации из прошлого и пр. Таким образом, о смысле можно говорить, лишь предполагая ситуацию, в которой кто-то говорит о чем-то, употребляя слова определенным образом.

Для описания этой структуры зададим следующие типы:

```
   postulate
 Human LOC TIM Referable : Set
```

и предикаты

```
   postulate
_speaks-to_/_,_         : Human → Human → LOC → TIM → Set
_utters_/_,_           : Human → String → LOC → TIM → Set
_refers-to_via_/_,_ : Human → Referable → String → LOC → TIM → Set
```

Здесь `Human` – тип людей, `LOC` и `TIM` – типы мест и времен, которые я не буду далее конкретизировать, `String` – встроенный тип, обозначающий строку символов, а `Referable` – специальный тип, которым будем обозначать все, на что может существовать референция в речи (см. ниже).

Предикаты означают:

| | |
|---|---|
| `h1 speaks-to h2 / l , t` | «Человек `h1` говорит (что-то) человеку `h2` в месте `l` в момент времени `t`» |

`h utters S / l , t` «Человек `h` произносит фразу `S` в месте `l` в момент времени `t`»

`h refers-to E via S / l , t` «Человек `h` отсылает к сущности `E` посредством фразы `S` в месте `l` в момент времени `t`».

Это позволяет нам определить ситуацию говорения:

```
   record SitUttering : Set where
field
{l} : LOC
{t} : TIM
{speaker} : Human
{listener} : Human
{phr} : String
spk : speaker speaks-to listener / l , t
utt : speaker utters phr / l , t
```

Эта ситуация содержит место и время говорения, говорящего (спикера), слушателя, произносимую фразу `phr` и доказательства того, что говорящий обращается к слушателю и произносит данную фразу. Фигурные скобки вокруг полей означают, что их значения являются имплицитными, и Agda будет пытаться самостоятельно их вычислить. Это позволит нам в дальнейшем для определения термов `SitUttering` указывать лишь поля `spk` и `utt`, остальные Agda будет способна восстановить по типам последних.

Рассмотрим пример [10. P. 610]. Пусть Мелисса говорит Наоми: «Человек у двери» (A man is at the door). Когда она это говорит, то: 1) обращается к Наоми; 2) производит отсылки (референции) к человеку и двери, также присутствующим в ситуации. Соответственно, для формализации определим дополнительно типы дверей, объектов и ситуаций:

```
   postulate
Door Object SIT : Set
```

Вообще говоря, термы почти всех типов одновременно относятся к типу `Referable`, поскольку о них можно говорить. Agda (как и теория типов Мартин–Лёфа) не позволяет относить термы к нескольким типам одновременно и не содержит встроенного механизма приведения типов (коэрсии). Однако его можно определить посредством подходящей функции. Например, коэрсия `Door <: Referable` означает, что определена функция, которая для каждого элемента `Door` вычисляет соответствующий ему элемент `Referable` (см. модуль `Coercion` по адресу выше). Я буду далее считать, что все необходимые приведения определены, причем термы `Human` и `Door` относятся также к `Object`, а термы `Object` и `SIT` – к `Referable` (соответственно, `Human` и `Door` также относятся к `Referable`). Приведение типа в тексте будет записываться как `⟪ a ⟫`, что обозначает терм необходимого типа, вычисленного по `a` на основе приведения (приведенное от типа, к которому относится `a`, к нужному в данном месте типу). Agda при этом самостоятельно находит нужные для приведения функции. Кроме этого, для всякого типа ситуации `Sit` всегда определено `Sit <: SIT`, т.е. всякая ситуация одновременно относится к типу `SIT`.

Определим далее тип ситуации референции:

```
   record SitRef : Set where
field
situ : SitUttering
open SitUttering
field
referent : Referable
phrase : String
referring : (speaker situ) refers-to referent via phrase
/ l situ , t situ
```

Она состоит из ситуации говорения, референта типа `Referable`, фразы `phrase` (которая не обязательно совпадает с `phr` из `SitUttering`, а может быть, например, ее частью) и доказательства того, что говорящий данной фразой указывает на этот референт. Ситуация референции – это ситуация, в которой кто-то говорит, делая отсылку к некоторому референту.

Ситуация, описываемая фразой «Человек у двери» выглядит следующим образом:

```
   postulate
_at-the-door_ : Human → Door → Set

record SitManDoor : Set where
 field
 {man} : Human
 {door} : Door
 mandoor : man at-the-door door
```

Ее строение определяется семантическими соображениями, т.е. смыслом рассматриваемой фразы, который, согласно теоретико-типовой семантике [1], записывается как тип

```
   (Σ x : Human) (Σ y : Door) x at-the-door y.
```

В нашей интерпретации это соответствует приведенному выше типу записи `SitManDoor`. Ее конкретный терм определяется контекстом (см. ниже).

Наконец, построим ситуацию, изображающую наши убеждения. Прежде всего, в ней имеется экземпляр ситуации `SitManDoor`, которую описывает Мелисса:

```
   postulate sitManDoor : SitManDoor
open SitManDoor sitManDoor using (door)
```

Здесь открыто только поле `door`, но не поля `man` и `mandoor`, что предполагает, что мы не знаем точно, на какого человека указывает Мелисса, но знаем, о какой двери идет речь.

Для информации, доступной нам только посредством контекста Мелиссы, определим для удобства

```
   Melissa-man        = SitManDoor.man        sitManDoor
Melissa-mandoor = SitManDoor.mandoor sitManDoor
```

Далее, постулируем

```
   postulate
Melissa Naomi : Human
sitUttering-mandoor : SitUttering
s≡m : SitUttering.speaker sitUttering-mandoor ≡ Melissa
l≡n : SitUttering.listener sitUttering-mandoor ≡ Naomi
```

```
p≡md : SitUttering.phr sitUttering-
mandoor ≡ "A man is at the door"
loc = SitUttering.l sitUttering-mandoor
tim = SitUttering.t sitUttering-mandoor
postulate
mrefm : Melissa refers-to ⟪ Melissa-man ⟫ via "man"
/ loc , tim
mrefd : Melissa refers-
to ⟪ door ⟫ via "door" / loc , tim
```

Таким образом, в нашей ситуации присутствуют Мелисса, Наоми и экземпляр ситуации `SitUttering`, в которой говорящий, слушающий и фраза совпадают, соответственно, с Мелиссой, Наоми и «A man is at the door», а кроме того – доказательства (т.е. наше знание) того, что Мелисса словами man и door обозначает человека и дверь из ситуации `sitManDoor`, открытой ранее.

Эти определения позволяют нам проводить доказательства, например для пропозиции «(Имеется) некий человек у (некой) двери»:

```
   _ : Σ[ x ∈ Human ] Σ[ y ∈ Door ] x at-the-door y
_ = Melissa-man , door , Melissa-mandoor
```

Или строить производные типы, такие как «Тип людей, на которых ссылается Мелисса»:

```
_ : Σ[ x ∈ Human ] Σ[ p ∈ String ] Σ[ l ∈ LOC ] Σ[ t ∈ TIM ]
 Melissa refers-to ⟪ x ⟫ via p / l , t
```

(несложно показать, что `Melissa-man` принадлежит данному типу).

Наконец, мы можем различить понятия абстрактного и конкретного смыслов, т.е. смысла без учета и с учетом контекста [10. P. 612]. Согласно Девлину, абстрактный смысл (abstract meaning) устанавливает связь между типами ситуации говорения и фокальной ситуации (о которой идет речь), а конкретный смысл (meaning-in-use) – связь между конкретной ситуацией говорения и объектом, на который в ней указывается (вероятно, лучше было бы говорить не о смысле, а о референте). При этом конкретный смысл является инстанциацией абстрактного. Таким образом, о смысле можно говорить только по отношению к ситуациям, которые содержат в себе ситуацию говорения, а также некоторую связь, позволяющую вычислять смысл в данной ситуации. Таким образом, для описания этой структуры нам требуется выделить класс ситуаций, в которых существует смысл, причем для этих ситуаций процедуры вычисления смысла могут существенно различаться. Agda предоставляет механизм для группировки типов в классы со сходными свойствами. При этом для классов не требуется введения каких-то новых конструкций, они определяются как записи. Для этого сначала определим класс смысл-ситуаций (т.е. тех, в которых мы можем говорить о смысле):

```
   record SitMeaning (Sit : Set) : Set where
field
sitmu : Sit → SitUttering
meaning : Sit → Referable
```

Это класс типов `Sit`, такой что каждый тип, относящийся к этому классу должен определять две функции, позволяющие вычислять: 1) ситуацию говорения; 2) смысл, высказанный в данной ситуации (для простоты, здесь счита-

ется, что смысл совпадает с референцией, т.е. относится к типу `Referable`). Как именно вычисляется смысл, зависит от типа ситуации и определяется для каждого из них отдельно. Рассмотрим для примера два таких типа.

Начнем с определенного выше типа ситуаций `SitRef`. Будем считать, что в этих ситуациях смысл (фактически, референт) совпадает с `SitRef.referent`. Для отнесения данного типа ситуаций к классу `SitMeaning` нам нужно определить функции `sitmu` и `meaning`. В Agda это делается с помощью директивы `instance`:

```
    instance
SMRef : SitMeaning SitRef
sitmu {{SMRef}} = SitRef.situ
meaning {{SMRef}} s = ⟪ SitRef.referent s ⟫
```

Здесь прежде всего декларируется, что `SMRef` относится к типу `SitMeaning SitRef`, для чего должны быть определены две функции из `SitRef`. Эти функции определяются в двух последних строчках, причем значением `sitmu` является ситуация говорения из `SitRef`, а значением `meaning` – поле `referent` из `SitRef`. После этого определения мы способны вычислить смысл для любой ситуации типа `SitRef`. Например, определим референциальную ситуацию «Мелисса, говоря с Наоми, обозначает словом „дверь“ дверь»:

```
    MrD : SitRef
MrD = record
 { situ = sitUttering-mandoor
 ; referent = ⟪ door ⟫
 ; phrase = "door"
 ; referring =
 subst (λ x → x refers-
to ⟪ door ⟫ via "door" / loc , tim)
 (sym s≡m) mrefd
 }
```

Здесь в качестве ситуации говорения используется постулированная выше ситуация `sitUttering-mandoor`, а в качестве референта и фразы – дверь из ситуации `SitManDoor` (приведенная к требуемому здесь типу `Referable`). В качестве `referring` должно стоять доказательство того, что говорящий из ситуации `situ` обозначает словом «дверь» дверь, т.е. терм типа

```
    (SitUttering.speaker situ) refers-
to ⟪ door ⟫ via "door" / loc , tim
```

Этого терма у нас нет, но мы можем его построить из `mrefd` и `s≡m`; выражение в определении `MrD` выше и есть такое построение.

Мы можем теперь доказать, что смысл в ситуации `MrD` равен `door`:

```
  _ : meaning MrD ≡ ⟪ door ⟫
_ = refl
```

Или, не используя промежуточное определение `MrD`:

```
  _ : meaning record
{ situ = sitUttering-mandoor
; referent = ⟪ door ⟫
; phrase = "door"
; referring =
subst (λ x → x refers-
to ⟪ door ⟫ via "door" / loc , tim)
(sym s≡m) mrefd
 }
≡ ⟪ door ⟫
_ = refl
```

Мы можем здесь видеть разницу между абстрактным и конкретным смыслами. Действительно, при отнесении `SitRef` к `SitMeaning` мы указали, что смысл для ситуации `s` равен ⟪ `SitRef.referent s` ⟫, что соответствует *абстрактному смыслу* «Референт ситуации референции». В то же время для конкретной ситуации `MrD` мы получили *конкретный смысл*, равный ⟪ `door` ⟫, т.е. определенному объекту.

Это различие еще более явно во втором примере, который мы рассмотрим. Пусть Мелисса произносит слово «Я». Каков его абстрактный и конкретный смысл?

Определим ситуацию «Говорение „Я“» следующим образом:

```
   record SitI : Set where
field
situ : SitUttering
open SitUttering
field
uI : phr situ ≡ "I"
```

Отнесем ее к классу `SitMeaning`:

```
   instance
SMI : SitMeaning SitI
sitmu {{SMI}} = SitI.situ
meaning {{SMI}} s = ⟪ SitUttering.speaker (SitI.situ s) ⟫
```

Как видно, мы определяем здесь абстрактный смысл: «Я» обозначает говорящего в ситуации говорения.

Пусть теперь у нас есть конкретная ситуация, в которой Мелисса говорит «Я»:

```
   postulate
sitUttering-I : SitUttering
s≡mI       : SitUttering.speaker sitUttering-I ≡ Melissa
p≡mdI : SitUttering.phr sitUttering-I ≡ "I"
```

Мы можем для нее вычислить конкретный смысл и увидеть, что он совпадает с Мелиссой:

```
_ : meaning record {situ = sitUttering-I; uI = p≡mdI} ≡ ⟪Melissa⟫
_ = ≡-coerce s≡mI
```

## Заключение

Мы видим, что теория типов с записями и модулями позволяет описывать важные аспекты зависимости от контекстов. Вместе с тем этот результат можно рассматривать лишь как предварительный. Agda сама по себе не содержит некоторых инструментов, необходимых для полноценной работы с контекстами или ситуациями. Например, в ней нет адекватных средств сравнения контекстов, установления между ними отношений включения и т.д. Способы их добавления в рамках теории типов требуют прояснения. Тем не менее использование методов функционального программирования, как можно надеяться, способно существенно обогатить инструментарий формальной семантики.

***Литература***

1. *Ranta A.* Type-theoretical grammar. Clarendon Press, 1994. 226 p.

2. *Cooper R.* Adapting Type Theory with Records for Natural Language Semantics // Modern Perspectives in Type-Theoretical Semantics. Springer International Publishing, 2017. P. 71–94.

3. *Cooper R.* Austinian Truth, Attitudes and Type Theory // Research on Language and Computation. 2005. Vol. 3, № 2. P. 333–362.

4. *Norell U.* Dependently Typed Programming in Agda // Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures. Berlin ; Heidelberg : Springer-Verlag Berlin Heidelberg, 2009. P. 230–266.

5. *The Agda* Team. Agda Wiki. URL: https://wiki.portal.chalmers.se/agda/

6. *Martin-Löf P.* An intuitionistic theory of types // Twenty-five years of constructive type theory. New York : Oxford Univ. Press, 1998. P. 127–172.

7. *Martin-Löf P.* An intuitionistic type theory : Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. Napoli : Bibliopolis, 1984. 91 p.

8. *Barwise J., Perry J.* Situations and Attitudes. MIT Press, 1983.

9. *Quine W.* Quantifiers and propositional attitudes // Journal of Philosophy. 1956. Vol. 53, is. 5. P. 177–187.

10. *Devlin K.* Situation theory and situation semantics // Handbook of the History of Logic. Elsevier, 2006. P. 601–664.

***Oleg A. Domanov,*** Institute of Philosophy and Law, Siberian Branch of the Russian Academy of Sciences (Novosibirsk, Russian Federation).

E-mail: domanov@philosophy.nsc.ru

*Vestnik Tomskogo gosudarstvennogo universiteta. Filosofiya. Sotsiologiya. Politologiya – Tomsk State University Journal of Philosophy, Sociology and Political Science. 2019. 52. pp. 23–38.*

DOI: 10.17223/1998863X/52/3

**FORMALIZATION OF CONTEXTS IN TYPE THEORY WITH RECORDS AND MODULES**

**Keywords:** natural language semantics; type theory; Agda, propositional attitudes; situation semantics.

Traditional type-theoretical semantics lacks convenient instruments for taking into account contextual information. To cope with this deficiency, a type theory with records was proposed by Robin Cooper. However, it possesses limited capacities as to the formalization of such phenomena as the referential opacity of contexts. At the same time, a lot of functional programming languages based on type theory contain sophisticated module systems providing means for opening or hiding various information from one or another fragment of programs. This article deals with opportunities that type theory with records and modules opens for the formalization of contexts in natural language semantics. The formalization is implemented in the dependently typed functional language Agda. Using programming language for type-theoretical semantics is not without a reason. Type theory is closely con-

nected not only to logic, but also to computing, and programming in particular. In the context of type-theoretical semantics, we can understand meaning as something computable. That is why many semantic problems turn out to be connected with the development of methods (and programs) of meaning or referent calculation. The article outlines basic principles of formalization and its specific realization in Agda. Relation to situation semantics by Jon Barwise and John Perry is demonstrated. Examples of descriptions for contexts' referential opacity as well as the concepts of abstract meaning and meaning-in-use from situation semantics are provided. The Agda formalization code is freely available.

### *References*

1. Ranta, A. (1994) *Type-theoretical grammar*. Clarendon Press.

2. Cooper, R. (2017) Adapting Type Theory with Records for Natural Language Semantics. In: Chatzikyriakidis, S. & Luo, Zh. (eds) *Modern Perspectives in Type-Theoretical Semantics*. Springer International Publishing. pp. 71–94.

3. Cooper, R. (2005) Austinian Truth, Attitudes and Type Theory. *Research on Language and Computation*. 3(2). pp. 333–362. DOI: 10.1007/s11168-006-0002-z

4. Norell, U. (2009) Dependently Typed Programming in Agda. *Advanced Functional Programming*. 6th International School, AFP 2008, Heijen, The Netherlands, May 2008. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg. pp. 230–266.

5. *The Agda Team. Agda Wiki*. (n.d.) [Online] Available from: https://wiki.portal.chalmers.se/agda/.

6. Martin-Löf, P. (1998) An intuitionistic theory of types. In: Sambin, G. & Smith, J.M. *Twenty-five years of constructive type theory*. New York: Oxford University Press. pp. 127–172.

7. Martin-Löf, P. (1984) *An intuitionistic type theory: Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Napoli: Bibliopolis.

8. Barwise, J. & Perry, J. (1983) *Situations and Attitudes*. MIT Press.

9. Quine, W. (1956) Quantifiers and propositional attitudes. *Journal of Philosophy*. 53(5). pp. 177–187.

10. Devlin, K. (2006) Situation theory and situation semantics. In: Gabbay, D.M. & Woods, J. (eds) *Handbook of the History of Logic*. Elsevier. pp. 601–664.