Доманов О. А. Признаки и типы в теоретико-типовой семантике естественного языка // Философия. Журнал Высшей школы экономики. — 2024. — Т. 8, № 1. — С. 188–214.

Олег Доманов*

Признаки и типы в теоретико-типовой семантике естественного языка**

Получено: 12.04.2023. **Рецензировано:** 11.08.2023. **Принято:** 09.01.2024.

Аннотация: Признаки и типы являются двумя возможными способами классификации явлений, относящихся к формализации грамматики и семантики естественного языка. Признаки часто используются в лингвистически ориентированных теориях. Однако они плохо согласуются с теоретико-типовой семантикой из-за понятия подтипа, к которому они приводят. В статье предлагается способ согласования этих двух подходов путем определения типов, основанных на классификации по признакам. Способ демонстрируется на примере формализации небольшого фрагмента английского языка. Построена общая формальная теория синтаксиса и семантики такого фрагмента, которая также имеет самостоятельное значение. Формализация проводится в языке Агда (Agda). Агда служит одновременно как: (1) метаязык, на котором формализуется синтаксис естественного языка, и (2) семантический/онтологический язык, на котором интерпретируется естественный язык. Это позволяет формализовать интерпретацию как функцию, переводящую выражения Агды, представляющие синтаксис, в формулы Агды, образующие семантику. Понятие подтипа опирается на понятие коэрсии или приведения типов, причем определение типов на основе признаков позволяет определить коэрсию автоматически. Механизм аргументов экземпляров (*instance arguments*) Агды позволяет во многих случаях проводить коэрсию автоматически. В конце статьи приведены примеры формализации языковых выражений, демонстрирующие работу построенной теории. Несмотря на то, что Агда как язык ориентирована прежде всего на математику, она содержит средства, позволяющие эффективно использовать ее в исследованиях естественного языка в рамках теоретико-типовой семантики.

Ключевые слова: теоретико-типовая семантика, естественный язык, теория типов, Agda, подтипы, коэрсия.

DOI: 10.17323/2587-8719-2024-1-188-214.

При формализации семантики естественного языка в теории типов (Chatzikyriakidis & Luo, 2020; Modern Perspectives..., 2017; Ranta, 1994) нарицательные существительные (англ. *common nouns*) могут представляться разными способами, которые являются комбинацией двух основных (о различных способах интерпретации нарицательных имен см., например, Chatzikyriakidis & Luo, 2017):

*Доманов Олег Анатольевич, к. филос. н., старший научный сотрудник, Институт философии и права Сибирского отделения Российской академии наук (Новосибирск), domanov@philosophy.nsc.ru, ORCID: 0000–0003–0057–3901.

(1) Мы можем следовать Монтегю (Montague, 1974a,b) и представлять такие имена как предикаты или подмножества, то есть как функции вида $e \to t$, где e — тип объектов, а t — тип истинностных значений.
(2) Мы можем следовать Ранта (Ranta, 1994) и представлять такие имена как типы.

Первый способ приводит к проблемам в случае копредикации, то есть применения разных предикатов к одному и тому же аргументу. Рассмотрим, например, предложение «Джон взял и прочитал три книги». Предикат «взять» применим к физическим объектам, а «прочитать» — к тем, которые возможно прочитать (будем называть их информационными). Например, книгу в Интернете можно прочитать, но нельзя взять. Что, в таком случае, означает «три разных книги»? Они физически или информационно разные? Если Джон взял и прочитал три книги, то были ли это три книги, различные физически и информационно, или три экземпляра одной и той же книги, или трилогия, собранная физически в одном томе? (Ср. также двусмысленность предложения «Мои книги продаются во всех магазинах».) Можем ли мы различить эти ситуации? Анализ показывает, что грамматика Монтегю не справляется с данной задачей (Bahramian et al., 2017: 2–6). Это же касается и ее усовершенствованных версий (таких как Heim & Kratzer, 1998), в которых предикаты представляются частичными функциями.

Теория типов также сталкивается с трудностями в подобных ситуациях, однако эти трудности иного рода. Глаголы в теоретико-типовой семантике формализуются как зависимые типы, причем тип, от которого они зависят, соответствует области применимости глагола. Если глагол «взять» применим к типу физических объектов, а глагол «прочитать» — к информационным, то книга должна одновременно относиться и к тем, и к другим. Однако в теории типов это невозможно. В теоретико-типовой семантике используются так называемые современные теории типов MTT (*Modern Type Theories*). К ним относят, например, исходную теорию Мартин-Лёфа (MLTT: Martin-Löf, 1984), единую теорию типов UTT (*Unified Theory of dependent Types*: Luo, 1994), а также исчисление индуктивных конструкций CIC (*Calculus of Inductive Constructions*). Их характерной особенностью является использование канонических объектов, посредством которых определяется тип. Это означает, что объект при появлении в теории получает единственный тип, который не может быть впоследствии изменен. Объекты, следовательно, не могут

относиться одновременно к нескольким типам. Понятие подтипа отсутствует в теориях МТТ и должно быть специально определено. При этом «наивный» способ определения посредством правила вида

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \subseteq B}{\Gamma \vdash a : B}$$

(называемый также подтипизацией включения, *subsumptive subtyping*) некорректен, поскольку a не может одновременно относиться к типам A и B.

В то же время, типы в семантике естественного языка возникают с самого начала — как ограничения на применимость предикатов, таких как глаголы или прилагательные. Действительно, такие выражения как предложение Ноама Хомского «Colorless green ideas sleep furiously» («Бесцветные зеленые идеи яростно спят»), будучи синтаксически правильными, не являются семантически корректными (то есть не имеют смысла) из-за неверного применения предикатов к своим аргументам (бесцветные идеи не могут быть зелеными, спать нельзя яростно и т. д.). Кроме того, квантификацию также естественно понимать как пробегающую по множеству объектов, к которым применимо выражение под квантором, а не по всему множеству объектов в мире. Универсум дискурса, как и человеческий универсум вообще, состоит из многих типов, различающихся по их способности участвовать в той или иной деятельности. Мы видим, однако, что типы, как они вводятся в теориях МТТ, не схватывают это семантическое несоответствие в случае естественного языка. Требуется система подтипов, адекватная семантике естественного языка. Мы должны иметь возможность для каждого типа указать, что к его объектам применимы те или иные предикаты, несмотря на то, что изначально они определены, возможно, на других типах.

Помимо упомянутой выше подтипизации включения, подтипы могут вводиться в теорию и другими способами. Мы можем изначально считать типы подмножествами общего множества (типа) объектов. Это последнее соответствует «домену дискурса», то есть тому, что мы вообще различаем и о чем способны говорить. Так, Тьери Кокан (Coquand, 1992) строит модель для теории типов как единое множество объектов (к которым, в частности, относятся и типы — это потенциально противоречивая система). Другой способ состоит в представлении подмножеств через отношение идентичности (Bahramian et al., 2017). Он, однако, также требует единого домена (типа), на котором устанавливаются эквивалентности, а это делает проблематичной саму идею

представления существительных посредством типов, а не предикатов (ср. также о представлении CN как сетоида: Luo, 2012a). Возможно также использовать одновременно как типы, так и предикаты (Retoré, 2013), при этом, однако, возникает отдельная проблема их согласования (Chatzikyriakidis & Luo, 2017).

По-видимому, наиболее подходящим для естественного языка способом моделирования подтипов является коэрсия или приведение типов. Это понятие было разработано в компьютерных науках и затем перенесено в лингвистику, где используется для описания согласования в различных языковых конструкциях (см., например, Asher & Luo, 2013). При коэрсии определяется функция $c : A \to B$, которая интерпретируется как приведение типа A к типу B. После этого, если нам требуется подставить объект в предикат, требующий типа B, мы можем использовать вместо него объект типа A, который пересчитывается в B с помощью функции c. Луо с соавторами (Luo et al., 2013) показывают, что коэрсия является консервативным расширением теории типов, в отличие от остальных рассмотренных ими способов. В частности, она хорошо работает не только в функциональных языках, но и в математических системах, таких как Агда (Agda) или Coq (ibid.: 4). Подробнее о коэрсии в MTT и ее использовании в формальной семантике можно посмотреть в Luo, 2012b.

Хотя типы в семантике возникают естественным образом, в современных грамматических теориях, таких как генеративная грамматика (Митренина и др., 2018) или HPSG (*Head-Driven Phrase Structure Grammar*: Head-Driven Phrase Structure Grammar, 2021) более привычным средством классификации являются не типы, а признаки (*features*). Например, глаголы или прилагательные применимы к существительным, если те обладают нужными признаками: красными могут быть физические, но не идеальные объекты, спать могут одушевленные существа и т. д. Признаки составляют систему, называемую структурой признаков (*Feature Structure*: Carpenter, 1992). Разумеется, признаки сами по себе позволяют разделить объекты по типам — например, можно полагать, что каждое множество признаков определяет отдельный тип. Однако при этом объекты могут обладать несколькими признаками, что делает возможным отнесение их к разным типам, а значит невозможным прямое использование теорий MTT. Возможна ли, тем не менее, теория типов, содержащая подтипы и способная формализовать системы признаков, как они существуют в упомянутых грамматиках и семантиках? Типизация по признакам позволяет, вообще говоря, определить

коэрсию: если объект относится к типу, определяемому некоторым множеством признаков f, то он также должен относиться к типам, определяемым множествами, меньшими f. Если мы определим типы так, чтобы это условие выполнялось автоматически, мы получим систему типов с коэрсией, пригодную для использования в семантике естественного языка. Именно этот метод используется в данной статье. Мы определим систему признаков, на ее основе — систему типов с коэрсией и затем посмотрим на примере небольшого фрагмента английского языка, каким образом она может использоваться в семантике. Формализация будет проводиться в языке Агда. Я представляю здесь основные идеи и структуры, полный код можно найти по адресу `https://github.com/odomanov/ttsemantics/blob/v2/Agda/MontagueTT/MontagueTTfeatures.agda`.

основные обозначения языка агда

Агда является функциональным языком с зависимыми типами Agda Documentation, 2014; Norell, 2009. Он основывается на теории типов UTT. Я предполагаю некоторое знакомство читателя с Агдой, однако буду достаточно подробно пояснять встречающиеся выражения так, чтобы, даже ничего о ней не зная, можно было бы составить впечатление об общих идеях.

Мы будем использовать Агду в двойной роли. С одной стороны, она выступает метаязыком для фрагмента естественного языка. Мы формализуем грамматику этого языка, представив его выражения (существительные, глаголы, предложения и пр.) в виде выражений Агды. С другой стороны, Агда предоставит интерпретацию для этого языка. Это означает, что формулы Агды (которые являются теоретико-типовыми формулами) будут служить смыслом выражений формализованного естественного языка. Благодаря этой двойной роли, интерпретация оказывается просто функцией, которая переводит одни выражения языка Агда в другие (фактически, мы получим не одну функцию, а их набор для разных синтаксических категорий).

Рассмотрим основные обозначения языка Агда. Прежде всего, нам нужно знать, что выражение `a : A` означает, что объект (или элемент, или терм) `a` относится к типу `A`. Типы функций записываются как `(x : A) → B`, причем `B` может зависеть от `x` (см. ниже о зависимых типах). Эта запись означает, что для каждого элемента типа `A` имеется процедура получения элемента типа `B`. Поэтому тип функций записывают также как `∀(x : A) → B`. Если Агда может восстановить тип `A` из текущего контекста, то запись можно сократить до `∀ x → B`. Если `B` не зависит от `x`,

то тип функций можно записать проще как `A → B`. Элементы этого типа, т. е. функции, обозначаются как `λ(x : A) → b`. Для них также используются сокращенная запись `λ x → b`. Применение функции `f` к аргументу `a` записывается как `f a`. Аргумент функции может быть имплицитным, то есть не указываемым при применении функции (Агда, в таком случае, должна сама быть способна его восстановить). Имплицитные аргументы обозначаются фигурными скобками с аналогичными сокращениями: `{x : A} → B`, `∀{x : A} → B`, `∀{x} → B`, `λ{x : A} → b` и `λ{x} → b`. Аналогично λ-исчислению, функции многих аргументов образуются функциями одного аргумента, значениями которых являются функции. При этом скобки можно опускать и вместо `A → (B → C)` писать `A → B → C`.

Типы сами являются элементами универсумов типов («типов типов»). Для предотвращения парадоксов Агда содержит иерархию универсумов $\texttt{Set}_0$, $\texttt{Set}_1$,..., нумеруемых уровнями. Универсумы нижних уровней принадлежат универсумам верхних уровней: $\texttt{Set}_i : \texttt{Set}_{i+1}$. Универсум нижнего уровня $\texttt{Set}_0$ (который мы будем в основном использовать) обозначается также просто как `Set`. Соответственно, запись `A : Set` означает, что `A` является типом (в данном случае, нулевого уровня).

Агда допускает зависимые типы, то есть типы, зависящие от других типов (такие как тип жителей города, который зависит от типа городов). Например, тип, зависящий от `A`, представляет собой функцию `A → Set`, т. е. функцию, которая каждому элементу типа `A` сопоставляет некоторый тип (элемент универсума `Set`).

Нам понадобится тип пар или Σ-тип. Он обозначается `Σ A B`, где `B` имеет тип функции `A → Set`. Этот тип состоит из пар `(a , b)`, таких, что `a : A` и `b : B a`. Если мы понимаем `B` как пропозициональную функцию, то `b` представляет собой доказательство пропозиции `B a`, и Σ-тип соответствует экзистенциальной пропозиции, поскольку ее доказательствами служат пары из `a` и доказательства, что `B a`. Функции проекции $\texttt{proj}_1$ и $\texttt{proj}_2$ извлекают, соответственно, первый и второй элементы пары. Если `B` на самом деле не зависит от `A`, то тип пар представляет собой декартово произведение и обозначается `A × B`. Для Σ-типа имеется также обозначение `Σ[ x ∈ A ] B x`, напоминающее пропозицию с экзистенциальным квантором.

Остальные обозначения будут вводится по мере изложения. Я также буду упрощать некоторые записи, если это не мешает пониманию; полные определения можно посмотреть в указанном выше файле.

Теория типов подчиняется принципу «пропозиция как тип». Это означает, что (1) пропозиция рассматривается как множество/тип ее

доказательств, (2) тип рассматривается как пропозиция, утверждающая непустоту данного типа (так, что каждый элемент типа является доказательством этой пропозиции). Это приводит к формальной эквивалентности типов, множеств и пропозиций. В частности, отношения часто определяются как зависимый тип соответствующей местности. Например, двуместное отношение `R` определяется как тип, зависящий от двух аргументов, так, что каждый его элемент `p : R x y` является доказательством того, что `x` и `y` находятся в данном отношении (то есть, что `R x y`). В частности, коэрсию мы определим как следующий тип:

```
data _⊆_ : Set → Set → Set₁  where
  coerce : {A : Set} → {B : Set} → (A → B) → (A ⊆ B)
```

Здесь слово `data` используется для определения новых типов, в данном случае типа с именем `_⊆_` (подчерки в имени типа обозначают места для аргументов, то есть данный тип можно использовать как `A ⊆ B`). Как видно, этот тип определяется как функция типа `Set → Set → Set₁`, значением которой является элемент универсума `Set₁`, то есть тип (уровня 1). Пропозиция `A ⊆ B` означает, что существует коэрсия из типа `A` в тип `B`. После слова `where` перечисляются конструкторы элементов определяемого типа (то есть конструкторы доказательств пропозиции `A ⊆ B`). В данном случае мы имеем только один конструктор `coerce`, который представляет собой функцию, имеющую в качестве аргумента функцию `A → B`. Таким образом, мы можем построить элемент типа `A ⊆ B` (то есть доказательство `A ⊆ B`), если существует функция `f : A → B`. Этот элемент строится конструктором `coerce` и имеет вид `coerce f`. Именно так мы будем ниже строить коэрсии для разных типов.

Агда определяет специальный тип имплицитных аргументов, которые называются аргументами экземпляра (*instance arguments*). Агда ищет значения для таких аргументов не просто в контексте, а в специальном хранилище объектов, заранее декларированных как экземпляры (*instances*). Такие аргументы обозначаются двойными фигурными скобками. Мы будем использовать этот механизм для автоматического поиска коэрсии. Для этого определим функцию

```
⟪_⟫ : {A : Set} {B : Set} → A → {{A ⊆ B}} → B,
```

в которой имеется аргумент экземпляра `{{A ⊆ B}}`. Если мы теперь декларируем в качестве экземпляра какой-то элемент этого типа, например `c : A ⊆ B`, затем возьмем элемент `a : A` и используем выражение `⟪ a ⟫` там, где требуется элемент типа `B`, то Агда обратится к хранилищу

экземпляров, найдет там коэрсию `c` и пересчитает `a` из типа `A` в тип `B`. Тем самым мы можем использовать функцию «_», не заботясь о типе аргумента. Например, для функции `f : B → C` мы можем писать `f « a »`, даже если `a` относится не к типу `B`, а к какому-то другому, из которого имеется коэрсия в `B`. Агда совершит преобразование автоматически.

Как сказано ранее, мы будем определять типы через множества признаков. Тип признаков обозначим `Features`, а тип множества признаков — `FS`. Признаки могут быть реализованы различными способами, важно лишь, чтобы для `FS` было определено отношение подмножества

```
_⊆ᶠ_ : FS → FS → Set.
```

В примере ниже `FS` представлены упорядоченными списками признаков, а конструкторы отношения `⊆ᶠ` декларированы как экземпляры так, чтобы его можно было использовать в аргументах экземпляров для автоматического поиска.

Прежде чем рассматривать признаки и основанные на них подтипы, построим теорию синтаксиса и семантики обобщенного языка. Она имеет самостоятельное значение. С точки зрения конкретной реализации, семантика и синтаксис определены как модули, входными параметрами которых служат определенные ниже лексическая структура `LexStructure` и модель `Model`. Первая содержит лексику языка, а вторая — ее интерпретацию на типах, которые мы определим, основываясь на признаках.

ФОРМАЛЬНАЯ МОДЕЛЬ СИНТАКСИСА И СЕМАНТИКИ

Определим грамматику фрагмента (английского) языка, для которого мы будем строить семантику. Она содержит следующие синтаксические категории:

`CN`	нарицательные имена, возможно модифицированные
`PN`	собственные имена
`VI`	непереходные глаголы
`VT`	переходные глаголы
`Adj`	прилагательные
`Det`	детерминативы (артикли и пр.)
`NP`	именная группа
`VP`	глагольная группа
`AP`	группа прилагательного
`S`	предложения

Под `CN` здесь понимается группа выражений, состоящих из нарицательных существительных, а также их возможных модификаций,

из которых мы рассмотрим два вида: (1) модификация прилагательным («большая собака») и (2) модификация придаточным предложением («собака, которая бежит», «dog that runs»). В отличие от выражений группы NP, они обозначают множество объектов и не могут, вообще говоря, служить подлежащими (особенно в английском языке, которым мы сейчас ограничиваемся).

Определим, как выглядят выражения нашего фрагмента для всех этих категорий, а затем построим функцию интерпретации, переводящую эти выражения в формулы языка Агда, то есть в теоретико-типовые выражения. В частности, CN интерпретируются как типы, PN и NP — как элементы некоторых типов, VP и AP — как функции от одной переменой (одноместный предикат), а S — как пропозиция/тип. Ограничения на возможные аргументы для функций VP, AP и др. мы будем включать в синтаксис. Это, в частности, делает невозможными синтаксически корректные, но семантически неверные предложения, такие как предложение Хомского выше. Поэтому, строго говоря, наш синтаксис уже включает некоторую семантическую информацию, однако это требование можно ослабить позднее, чего мы в данной статье делать не будем. Такое ослабление, в частности, приведет к тому, что описанные ниже функции интерпретации потребуется определять как частичные.

Условия на лексику языка собраны в структуре LexStructure следующего вида (знаки -- начинают комментарий; все, что следует после них, игнорируется Агдой):

```
record LexStructure : Set₁ where
  field
    nameCN namePN nameVI nameVT nameAdj : Set   -- слова синтаксических
                                                -- категорий
    argPN  : namePN  → nameCN                   -- аргументная структура
    argVI  : nameVI  → nameCN
    argVT  : nameVT  → nameCN × nameCN
    argAdj : nameAdj → nameCN
    _<:0_  : nameCN  → nameCN → Set             -- коэрсия на синтаксическом
                                                -- уровне
```

Использованный здесь тип record можно рассматривать как тип кортежей с поименованными полями (или таблицу, в которой поля представляют собой имена столбцов). Таким образом, каждый элемент типа LexStructure является кортежем, который содержит, во-первых, слова соответствующих категорий nameCN, namePN и пр., а, во-вторых, их

аргументную структуру `argPN`, `argVI` и пр. Так, для каждого `namePN` указано имя `nameCN` — т. е. множество, к которому принадлежит объект, обозначаемый данным собственным именем; для каждого `nameVI`, `nameVT` и `nameAdj` — имена `nameCN`, к которым применимы соответствующие глаголы и прилагательные (напомню, что запись `nameCN × nameCN` обозначает декартово произведение или тип пар). Помимо этого, `LexStructure` содержит предикат `_<:0_` на именах `nameCN`, который представляет коэрсию между ними на уровне синтаксиса (см. ниже).

Таким образом, каждый объект типа `LexStructure` задает лексикон некоторого языка вместе с информацией о сочетаемости его слов в виде аргументной структуры и отношения коэрсии.

Определим теперь перечисленные выше синтаксические категории. Мы определим их как типы, конструкторы которых конструируют выражения соответствующих категорий. Начнем с категории `CN`. В упрощенном виде определение для нее могло бы выглядеть следующим образом:

```
data CN : Set where
  cn-n  : nameCN → CN
  cn-ap : (cn : CN) → AP cn → CN
  rcn   : (cn : CN) → VP cn → CN
```

Здесь имеются три конструктора для выражений этой категории. Прежде всего, каждому имени лексикона `nameCN` соответствует элемент `CN` (т. е. каждое имя `nameCN` само по себе образует элемент `CN`). Кроме этого, наш фрагмент языка содержит `CN`, составленные из прилагательного (точнее, группы прилагательного `AP`) и другого `CN` (например, «green ball»), а также из `CN` и глагольной группы `VP` (например, «ball that rolls»). Выражения `AP cn` и `VP cn` в этом определении обозначают, как мы увидим далее, типы групп прилагательного и глагола, индексированные объектами `cn : CN`. Так `cn-ap cn ap` конструирует элемент `CN` из `cn` и `ap`, которое относится к типу `AP cn`. Мы, однако, хотим допустить несовпадение, разрешаемое с помощью коэрсии, так, чтобы группа прилагательного могла относиться к типу `AP cn₁`, где `cn₁` связано с `cn` коэрсией. По этой причине фактическое определение должно быть более развернуто:

```
data CN : Set where
  cn-n  : nameCN → CN
  cn-ap : {cn₁ : CN} → AP cn₁ → (cn₂ : CN) → {{cn₂ <: cn₁}} → CN
  rcn   : (cn₁ : CN) {cn₂ : CN} → VP cn₂ → {{cn₁ <: cn₂}} → CN
```

Здесь в конструкторах `cn-ap` и `rcn` типы могут не совпадать, и чтобы это учесть, эти конструкторы дополнительно содержат аргументы экземпляров вида `{{cn₁ <: cn₂}}`, содержащих коэрсию между соответствующими именами. Коэрсия `_<:_` является расширением `_<:0_` с базовых имен `nameCN` на все `CN` и определяется следующим образом:

```
data _<:_ : CN → CN → Set where
  instance cnm  : ∀ {n₁ n₂} → {{n₁ <:0 n₂}} → cn-n n₁ <: cn-n n₂
  instance cap  : ∀ {cn₁ cn₂ ap coe} → cn-ap {cn₁} ap cn₂ {{coe}} <: cn₂
  instance crcn : ∀ {cn₁ cn₂ vp coe} → rcn cn₁ {cn₂} vp {{coe}} <: cn₁
  c∘ : ∀ {cn₁ cn₂ cn₃} → cn₁ <: cn₂ → cn₂ <: cn₃ → cn₁ <: cn₃
```

Это отношение имеет четыре конструктора, три из которых декларированы как экземпляры (директивой `instance`). Конструктор `cnm` строит доказательство отношения `<:` для базовых имен `nameCN` (которые строятся с помощью конструктора `cn-n`) на основе отношения `<:0`, взятого из `LexStructure`. Конструктор `cap` строит доказательство коэрсии для `CN`, построенных с помощью группы прилагательного (например, коэрсии «зеленый шар» в «шар»). Конструктор `crcn` строит аналогичную коэрсию для относительных выражений (например, коэрсию «шар, который катится» в «шар»). Наконец, конструктор `c∘` отвечает за композицию двух коэрсий (он не декларирован как экземпляр по чисто техническим причинам — для увеличения быстродействия программы).

Кроме детерминативов `Det` все остальные синтаксические категории определяются как зависимые от `CN`. Как я уже говорил выше, это приводит к тому, что всякое выражение языка имеет интерпретацию, то есть все синтаксически правильные языковые выражения автоматически оказываются семантически правильными.

Для краткости я буду говорить далее, что выражения `VP cn`, `NP cn` и т. д. относятся к типу `cn`, надеясь, что из контекста понятно, что речь не идет о типах теории типов.

Глагольные группы могут быть двух видов:

```
data VP : (cn : CN) → Set where
  vp-vi : (n : nameVI) → VP (cn-n (argVI n))
  vp-vt : (n : nameVT) → {cn₂ : CN} → NP cn₂
          → {{coe : cn₂ <: cn-n (proj₂ (argVT n))}}
          → VP (cn-n (proj₁ (argVT n)))
```

Конструктор `vp-vi` образует элемент `VP` из непереходного глагола, а `vp-vt` — из переходного глагола, примененного к именной группе. Типы образованных глагольных групп определяются функциями `argVI` и `argVT`. Кроме того, в случае `vp-vt` требуется коэрсия, приводящая тип объекта `NP cn₂` к типу `cn-n (proj₂ (argVT n))`, требуемому для аргумента глагола.

Именные группы определяются следующим образом:

```
data NP : (cn : CN) → Set where
  np-pn  : (n : namePN) → NP (cn-n (argPN n))
  np-det : Det → (cn : CN) → NP cn
```

Первый конструктор строит группу из собственного имени, а второй — из детерминатива `Det` и выражения категории `CN`, например, «the dog», «every black dog». При этом категория детерминативов определяется как

```
data Det : Set where
  a/an every no the : Det
```

Группы прилагательных в нашем фрагменте языка имеют только одну форму, образованную из единственного прилагательного, взятого из лексикона:

```
data AP : (cn : CN) → Set where
  ap-a : (n : nameAdj) → AP (cn-n (argAdj n))
```

Наконец, категория предложений также имеет один конструктор, соответствующий предложениям вида NP VP:

```
data S : Set where
  s-nv  : ∀{cn₁} → NP cn₁ → ∀{cn₂} → VP cn₂ → {{coe : cn₁ <: cn₂}} → S
```

Поскольку выражения `NP cn₁` и `VP cn₂` могут иметь разные типы, нам требуется коэрсия `cn₁ <: cn₂` (то есть глагол должен быть применим к множеству объектов, как минимум включающему область определения `NP cn₁`).

Итак, мы определили вид всех выражений нашего фрагмента языка. Перейдем теперь к их интерпретации. Для этого, прежде всего, нам потребуется модель, которая представлена следующей структурой:

```
record Model (nam : LexStructure) : Set₁ where
  open LexStructure nam      -- делает доступными nameCN и пр.
  field
    valCN  : nameCN → Set
```

```
valPN  : (n : namePN)  → valCN (argPN n)
valVI  : (n : nameVI)  → valCN (argVI n) → Set
valVT  : (n : nameVT)  → valCN (proj₁ (argVT n))
                       → valCN (proj₂ (argVT n)) → Set
valAdj : (n : nameAdj) → valCN (argAdj n) → Set
val<:0 : ∀{n₁ n₂} → {{n₁ <:0 n₂}} → valCN n₁ ⊆ valCN n₂
```

Модель содержит функции интерпретации для каждого из имен нашего лексикона, то есть для имен, содержащихся в структуре `LexStructure`. Как видно, слова категории `CN` интерпретируются как типы (множества). Слова категории `PN` интерпретируются как элементы соответствующих множеств `valCN cn`, где аргумент `cn` вычисляется с помощью функции `argPN` из `LexStructure`. Глаголы и прилагательные интерпретируются как соответствующие функции. Наконец, модель также задает интерпретацию для отношения синтаксической коэрсии `<:0`, которая интерпретируется как отношение семантической коэрсии ⊆ для соответствующих множеств `valCN n₁` и `valCN n₂`.

Данная модель задает интерпретацию лексикона, т. е. базовых выражений языка. Для всех остальных выражений она строится рекурсивно как описано ниже. Мы определим функции интерпретации отдельно для каждой категории и будем обозначать их ⟦`cn_`⟧, ⟦`pn_`⟧ и т. д. для, соответственно, категорий `CN`, `PN` и т. д. Кроме того, нам потребуется функция ⟦`coe_`⟧ для интерпретации коэрсии, то есть для перевода синтаксического отношения `<:` в семантическое отношение ⊆. Для удобства, я буду также использовать следующие два сокращения для коэрсии функции и Σ-типа:

```
⟪→ f ⟫ = λ x → f ⟪ x ⟫
⟪Σ⟫ A B = Σ A ⟪→ B ⟫
```

Итак, определим функцию интерпретации для категории `CN` (ниже, как обычно в Агде, в первой строке декларируется, что некоторый терм относится к такому-то типу — в данном случае, к типу `CN → Set`, — а в последующих определяется, чему этот терм равен — в данном случае определяются значения функции ⟦`cn_`⟧ для всех возможных аргументов):

```
⟦cn_⟧ : CN → Set
⟦cn cn-n n ⟧ = valCN n
⟦cn cn-ap ap cn {{coe}} ⟧ = ⟪Σ⟫ ⟦cn cn ⟧ ⟦ap ap ⟧ {{⟦coe coe ⟧}}
⟦cn rcn cn₁ {cn₂} vp {{coe}} ⟧ = ⟪Σ⟫ ⟦cn cn₁ ⟧ (⟦vp vp ⟧ {cn₂}) {{⟦coe coe ⟧}}
```

Здесь интерпретация выражений `cn-n n` берется из модели, а два остальных вида выражений категории `CN` интерпретируются как Σ-тип. Действительно, функция интерпретации для `AP` выглядит следующим образом:

```
⟦ap_⟧ : {cn : CN} → AP cn → (⟦cn cn ⟧ → Set)
⟦ap ap-a n ⟧ = valAdj n
```

Таким образом, `AP cn` интерпретируется как функция вида `⟦cn cn ⟧ → Set`, что и требуется для Σ-типа выше. В итоге, мы видим, что, например, выражение «зеленый шар» интерпретируется как тип пар, состоящих из шара и доказательства того, что шар зеленый[1]. Аргумент экземпляра `{{⟦coe coe ⟧}}` обеспечивает коэрсию, возможно требующуюся при несовпадении типов прилагательного и выражения, к которому оно применяется. Этот аргумент содержит коэрсию, вычисленную с помощью функции `⟦coe_⟧`, которая выглядит следующим образом:

```
⟦coe_⟧ : {cn₁ cn₂ : CN} → (cn₁ <: cn₂) → (⟦cn cn₁ ⟧ ⊆ ⟦cn cn₂ ⟧)
⟦coe cnm ⟧  = val<:0
⟦coe cap ⟧  = coerce proj₁
⟦coe crcn ⟧ = coerce proj₁
⟦coe c∘ c₁₂ c₂₃ ⟧ = coerce (ggetfunc c₂₃ ∘ ggetfunc c₁₂)
```

Как видно, она переводит синтаксическое отношение `<:` в семантическое отношение ⊆. Для этого требуются функции коэрсии, которые в случае `cnm` берутся из модели, а в случае `cap` и `crcn` равны первой проекции Σ-типа (это так, поскольку первая проекция задает коэрсию из `(Σ A B)` в `A`). Функция `ggetfunc` извлекает функцию коэрсии из имеющегося доказательства отношения `<:`. Ее определение можно посмотреть в указанном выше файле.

Аналогично интерпретируются относительные конструкции `rcn`. Действительно, для категории `VP` мы имеем:

```
⟦vp_⟧ : {cn₁ : CN} → VP cn₁ → {cn₂ : CN} → {{⟦cn cn₂ ⟧ ⊆ ⟦cn cn₁ ⟧}}
      → ⟦cn cn₂ ⟧ → Set
⟦vp vp-vi n ⟧ x = valVI n ⟪ x ⟫
⟦vp vp-vt vt {cn₂} np {{coe}} ⟧ x =
    ⟦np np ⟧ {cn₂} λ y → valVT vt ⟪ x ⟫ (⟪ y ⟫ {{⟦coe coe ⟧}})
```

[1] См. подробнее о принципах теоретико-типовой грамматики: Ranta, 1994: 65; Chatzikyriakidis & Luo, 2020. Нужно заметить, что это лишь одна из возможных интерпретаций прилагательных. Например, такие выражения как «ложный опенок» должны интерпретироваться иначе.

Она также интерпретируется как функция вида ⟦cn cn₂ ⟧ → Set, как и требуется для Σ-типа. Аналогично предыдущему, выражение «шар, который катится» интерпретируется как тип пар, состоящих из шара и доказательства того, что шар катится. В случае vp-vi n функция интерпретации берется из модели, случай же vp-vt следует рассмотреть подробнее. Нам нужно вспомнить, как интерпретируется категория NP в грамматике Монтегю. Ее интерпретация имеет тип $(e \to t) \to t$, так чтобы при применении этой функции к интерпретации VP, то есть к терму типа $e \to t$, получалось предложение, которое у Монтегю имеет тип t. Мы поступим аналогично и будем интерпретировать NP следующим образом:

```
⟦np_⟧ : {cn₁ : CN} → NP cn₁ → {cn₂ : CN} → {{⟦cn cn₁ ⟧ ⊆ ⟦cn cn₂ ⟧}}
      → (⟦cn cn₂ ⟧ → Set) → Set
⟦np np-pn n ⟧           ⟦vp⟧ = ⟦vp⟧ ⟪ valPN n ⟫
⟦np np-det d cn ⟧ {cn₂} ⟦vp⟧ = ⟦det d ⟧ cn {cn₂} ⟦vp⟧
```

Как видно, NP интерпретируется как функция вида (⟦cn cn ⟧ → Set) → Set. При ее применение к интерпретации VP вида ⟦cn cn ⟧ → Set мы получим элемент Set, то есть пропозицию. Эта часть аналогична грамматике Монтегю. Однако, в отличие от последней, мы не понимаем собственные имена как функции, аналогичные NP, мы берем их интерпретации непосредственно из модели. Тем не менее, интерпретация np-pn n, как видно выше, представляет такую функцию.

Что касается именных фраз с детерминативом, с ними мы также поступаем аналогично Монтегю. Функция интерпретации для детерминативов выглядит следующим образом:

```
⟦det_⟧ : Det → (cn : CN) → {cn₁ : CN} → {{⟦cn cn ⟧ ⊆ ⟦cn cn₁ ⟧}}
       → (⟦cn cn₁ ⟧ → Set) → Set
⟦det a/an ⟧  cn vp = Σ ⟦cn cn ⟧ ⟪→ vp ⟫
⟦det every ⟧ cn vp = ∀(x : ⟦cn cn ⟧) → vp ⟪ x ⟫
⟦det no ⟧    cn vp = ∀(x : ⟦cn cn ⟧) → ¬ vp ⟪ x ⟫
⟦det the ⟧   cn vp = Σ[ Aₚ ∈ Pointed ⟦cn cn ⟧ ] vp ⟪ theₚ Aₚ ⟫
```

Здесь мы снова должны вспомнить, что у Монтегю детерминативы имеют тип

$$(e \to t) \to (e \to t) \to t.$$

Это функции, аргументами которых являются CN (которые у Монтегю соответствуют функциям $e \to t$) и VP. Поскольку у нас CN интерпретируются не как функции, а как типы, то детерминативы интерпретируются как функции вида (cn : CN) → (⟦cn cn ⟧ → Set) → Set. Аргумент

экземпляра позволяет учесть возможную коэрсию. Что касается конкретных интерпретаций различных детерминативов, то они соответствуют грамматике Монтегю. Неопределенный артикль `a/an`, определяется как Σ-тип, то есть как квантор существования, слово `every` — как универсальный квантор, `no` — как универсальный квантор с отрицанием. Определенный артикль определяется для типов `Pointed`, которые представляют собой тип с выделенным элементом, который здесь обозначен как $\texttt{the}_{\texttt{p}}$ (см. пример ниже). Словами эта интерпретация может быть передана так: «Имеется тип с выделенным элементом, такой, что этот элемент выполняет предикат `vp`».

Наконец, предложение интерпретируется как тип или пропозиция:

```
⟦s_⟧ : S → Set
⟦s s-nv {cn₁} np {cn₂} vp {{coe}} ⟧ =
    ⟦np np ⟧ {cn₁} (⟪→ ⟦vp vp ⟧ {cn₂} ⟫ {{⟦coe coe ⟧}})
```

Аналогично Монтегю, эта интерпретация представляет собой применение функции для `NP` к аргументу `VP`. При этом учитывается коэрсия.

Итак, мы построили синтаксис и семантику фрагмента грамматики и семантики английского языка в общем виде. Они, вообще говоря, не зависят от того, как мы определяем интерпретацию `CN`, то есть функцию `valCN`, а также от отношения коэрсии между ними. Нам нужно теперь построить такую интерпретацию, которая опиралась бы на понятие признака. Сделаем это на конкретном примере.

ПРИМЕР

Перейдем к конкретному примеру и рассмотрим как работает данная формализация при определении типов через признаки. Прежде всего, определим информацию, необходимую для лексикона `LexStructure`:

```
data nameCN  : Set where Human Dog Animate Object : nameCN
data namePN  : Set where Alex Mary Polkan : namePN
data nameVI  : Set where run : nameVI
data nameVT  : Set where love : nameVT
data nameAdj : Set where black : nameAdj

argPN : namePN → nameCN
argPN Alex = Human
argPN Mary = Human
argPN Polkan = Dog
```

```
argVI : nameVI → nameCN
argVI run = Animate

argVT : nameVT → nameCN × nameCN
argVT love = Animate , Object

argAdj : nameAdj → nameCN
argAdj black   = Object
```

Как видно, мы определяем четыре имени для `CN` — `Human`, `Dog`, `Animate` и `Object` — и три собственных имени `Alex`, `Mary` и `Polkan`. Функция `argPN` показывает, что `Alex` и `Mary` относятся к типу `Human`, а `Polkan` — к типу `Dog`. Кроме того мы определяем непереходный глагол `run`, определенный на `Animate`, и переходный глагол `love`, определенный на `Animate` и `Object` (то есть его субъектом выступают термины `Animate`, а дополнением — термины `Object`). И, наконец, мы определяем прилагательное `black`, применимое к `Object`.

Будем формализовать множество признаков как список, для чего определим признаки и тип списка признаков (`List`):

```
data Feature : Set where
  f-animate f-dog f-human f-object : Feature

FS = List Feature
```

Таким образом, мы определили четыре признака и тип `FS` как тип списка признаков. Кроме того, на `FS` мы определим отношение подмножества `_⊆ᶠ_` (см. файл с полной формализацией).

Будем далее считать, что обладание признаками `f-human` и `f-dog` означает также обладание признаком `f-animate`, а обладание последним — обладание `f-object`. Тогда мы можем определить следующие множества признаков (`[]` означает пустой список, а `f :: fs` — список, образованный присоединением признака `f` к списку `fs`):

```
FO = f-object :: []
FA = f-object :: f-animate :: []
FH = f-object :: f-animate :: f-human :: []
FD = f-object :: f-animate :: f-dog :: []
```

Их связь с именами для `CN` определяется функцией:

```
FSet : nameCN → FS
FSet Human   = FH
FSet Dog     = FD
FSet Animate = FA
FSet Object  = FO
```

Эти определения подготавливают нашу интерпретацию. Ниже, будучи интерпретированным, имя `Object` будет обозначать тип элементов, обладающих признаком `f-object`, имя `Animate` — тип элементов, обладающих признаками `f-animate` и `f-object`, имя `Dog` — тип элементов, обладающих признаками `f-dog`, `f-animate` и `f-object`, и, наконец, имя `Human` — тип элементов, обладающих признаками `f-human`, `f-animate` и `f-object` (то есть все собаки и люди являются одушевленными, а все одушевленные также объектами).

Функция `FSet` позволяет нам определить (синтаксическую) коэрсию `_<:0_` на базовых именах `nameCN` через отношение подмножества на множестве признаков (обратите внимание на обращение порядка аргументов в отношении):

```
_<:0_ : nameCN → nameCN → Set
n₁ <:0 n₂ = FSet n₂ ⊆ᶠ FSet n₁
```

Для построения семантики нам нужно определить типы, на которых мы будем интерпретировать выражения категории `CN`. Определим их как зависимый тип, индексированный множествами признаков `FS` (двойные квадратные скобки в `⟦CN⟧` являются частью имени, они используются здесь как обычные символы; Агда в этом отношении очень либеральна):

```
data ⟦CN⟧ : FS → Set where
  base : (n : namePN) → ⟦CN⟧ (FSet (argPN n))
  ⟪_⟫  : ∀ {f₁ f₂} → {{f₂ ⊆ᶠ f₁}} → ⟦CN⟧ f₁ → ⟦CN⟧ f₂
```

Это ключевой момент нашего построения. Тип `⟦CN⟧ fs` это тип всех объектов, обладающих признаками из множества `fs` и только ими. Он имеет элементы двух видов. Прежде всего, это базовые элементы вида `base n`, где `n` — собственное имя из множества `namePN`. Второй вид строится конструктором `⟪_⟫`, который конструирует объекты типа `⟦CN⟧ fs₂` из объектов типа `⟦CN⟧ fs₁`, *если множество признаков `fs₂` является подмножеством `fs₁`*. Таким образом, функция `⟪_⟫` определяет коэрсию на уровне семантики: всякий элемент типа `⟦CN⟧ fs₁` приводится к типу `⟦CN⟧ fs₂` для всех наборов признаков `fs₂`, меньших, чем набор `fs₁`.

Это естественное требование — объекты, обладающие признаками `fs` обладают также и любым набором признаков, меньшим `fs`.

Можно показать, что признаки `fs` однозначно определяют множество `⟦CN⟧ fs`:

```
uniq-⟦CN⟧ : ∀{fs₁ fs₂} → fs₁ ≡ fs₂ → ⟦CN⟧ fs₁ ≡ ⟦CN⟧ fs₂
uniq-⟦CN⟧ refl = refl
```

Здесь `≡` означает пропозициональное, т. е. доказываемое, равенство, в отличие от равенства по определению `=`. Другими словами, если равны `fs₁` и `fs₂`, то равны также `⟦CN⟧ fs₁` и `⟦CN⟧ fs₂` (если имеется доказательство `fs₁ ≡ fs₂`, то можно построить доказательство `⟦CN⟧ fs₁ ≡ ⟦CN⟧ fs₂`). Что касается обратного, то при наличии коэрсии равенство для `⟦CN⟧ fs` не является простым понятием, и из него не обязательно следует равенство соответствующих `fs`. Действительно, коэрсию `A ⊆ B` мы интерпретируем как подмножество, то есть так, что все элементы `A` содержатся также и в `B`. Однако, строго говоря, в теории типов это не может быть верно, поскольку в ней каждый элемент может содержаться лишь в одном множестве (типе). Здесь требуется уточнение понятия равенства, которое мы, однако, проводить не будем.

Введем обозначения:

```
*Human  = ⟦CN⟧ FH
*Dog    = ⟦CN⟧ FD
*Animate = ⟦CN⟧ FA
*Object = ⟦CN⟧ FO
*Alex   = base Alex
*Mary   = base Mary
*Polkan = base Polkan
```

Условимся, что далее, как и здесь, термины со звездочкой будут обозначать семантические сущности, на которых мы интерпретируем, то есть множества (типы) и их элементы. Наше определение `⟦CN⟧` гарантируют, что `*Alex` и `*Mary` принадлежат типу `*Human`, а `*Polkan` — типу `*Dog`. Функция `(_)`, учитывающая коэрсию, позволяет нам использовать термины в качестве элементов разных типов. Проверим, например, что `( *Mary )` входит во все наши типы, кроме `*Dog` (в случаях, подобных ниже, если нам не требуется имя декларируемого терма, мы можем заменить его подчерком):

```
_ : *Human
_ = *Mary
```

```
_ : *Human
_ = ⟪ *Mary ⟫

_ : *Animate
_ = ⟪ *Mary ⟫

_ : *Object
_ = ⟪ *Mary ⟫
```

Постулируем дополнительно предикаты для интерпретации глаголов и прилагательных:

```
postulate
  _*run   : *Animate → Set
  _*love_ : *Animate → *Object → Set
  *black  : *Object → Set
```

Все это позволяет нам определить функции валюации, нужные для модели:

```
valCN : nameCN → Set
valCN n = ⟦CN⟧ (FSet n)

valPN  : (n : namePN) →  valCN (argPN n)
valPN n = base n

valVI  : (n : nameVI) →  valCN (argVI n) → Set
valVI run = _*run

valVT  : (n : nameVT) →  valCN (proj₁ (argVT n)) → valCN (proj₂ (argVT n)) →
     Set
valVT love = _*love_

valAdj : (n : nameAdj) → valCN (argAdj n) → Set
valAdj black = *black

val<:0 : ∀{n₁ n₂ : nameCN} → {{n₁ <:0 n₂}} → valCN n₁ ⊆ valCN n₂
val<:0 = coerce ⟪_⟫
```

Эти определения вполне прозрачны. Последнее из них определяет интерпретацию коэрсии на базовых именах `nameCN`. Как видно, для этого используется функция `⟪_⟫` из определения `⟦CN⟧`. Действительно, она

предоставляет нам нужную функцию коэрсии, связывающую множества ⟦CN⟧ fs для различных fs.

Теперь все готово для рассмотрения примеров языковых выражений.

Предложение «Mary runs» имеет следующий вид:

```
s1 = s-nv (np-pn Mary) (vp-vi run)
```

Агда позволяет вычислить его интерпретацию, которая выглядит ожидаемо:

```
⟦s s1 ⟧ ≡ ⦇ *Mary ⦈ *run
```

Здесь *Mary типа *Human автоматически приводится к типу *Animate, на котором определен предикат *run. Совершенно аналогично для *Polkan, относящегося к типу *Dog:

```
s2 = s-nv (np-pn Polkan) (vp-vi run)

⟦s s2 ⟧ ≡ ⦇ *Polkan ⦈ *run
```

Рассмотрим более сложное предложение «A human runs». Его формализация и интерпретация имеют следующий вид:

```
s3 = s-nv (np-det a/an (cn-n Human)) (vp-vi run)

⟦s s3 ⟧ ≡ Σ *Human λ( x → ⦇ x ⦈ *run)
```

Или, если мы введем обозначение ⦇→ f ⦈ для коэрсии функции f:

```
⦇→_⦈ : ∀{fs₁ fs₂} → (⟦CN⟧ fs₁ → Set) → {{fs₁ ⊆ᶠ fs₂}} → (⟦CN⟧ fs₂ → Set)
⦇→ f ⦈ x = f ⦇ x ⦈

⟦s s3 ⟧ ≡ Σ *Human ⦇→ _*run ⦈
```

Как видно, неопределенный артикль интерпретируется как Σ-тип или экзистенциальный квантор.

Предложение «Every human runs», как и требуется, интерпретируется как формула с универсальным квантором:

```
s4 = s-nv (np-det every (cn-n Human)) (vp-vi run)

⟦s s4 ⟧ ≡ ∀(x : *Human) → ⦇ x ⦈ *run
```

Предложение с определенным артиклем «The human runs» и его интерпретация выглядят следующим образом:

```
s5 = s-nv (np-det the (cn-n Human)) (vp-vi run)

⟦s s5 ⟧ ≡ Σ[ Aₚ ∈ Pointed *Human ] ( theₚ Aₚ ) *run
```

Здесь используется тип `Pointed`, определенный как

```
record Pointed (A : Set) : Set where
  field
    theₚ : A
```

`Pointed A` означает, что тип `A` имеет выделенный элемент, который обозначается $\texttt{the}_\texttt{p}$. Таким образом, интерпретация `s5` гласит: «Существует тип `*Human` с выделенным объектом, и этот выделенный объект бежит». Если мы далее постулируем, что этим выделенным объектом является `*Mary`, а также то, что Мэри бежит, то можем получить доказательство пропозиции `⟦s s5 ⟧`, которое будет представлять собой пару, состоящую из типа $\texttt{H}_\texttt{p}$ с выделенным элементом `*Mary` и доказательства того, что она бежит:

```
_ : ⟦s s5 ⟧
_ = Hₚ , *Mary-run
  where
  Hₚ : Pointed *Human
  Hₚ = record { theₚ = *Mary }

  postulate *Mary-run : ( *Mary ) *run
```

Здесь, как и ранее, функция `(_)` позволяет не заботиться о типах используемых термов; алгоритм поиска экземпляров (*instance search*) проводит коэрсию автоматически. К сожалению, эта автоматика работает не всегда, и в сложных случаях алгоритму приходится помогать. Рассмотрим, например, относительные конструкции. «Human that runs» относится к категории `CN` и выглядит следующим образом:

```
human-that-runs : CN
human-that-runs = rcn (cn-n Human) (vp-vi run)
```

Добавление неопределенного артикля создает категорию `NP`:

```
a-human-that-runs : NP _
a-human-that-runs = np-det a/an human-that-runs
```

Тогда для формализации предложения «Mary loves a human that runs» потребуется добавить экземпляр, обеспечивающий коэрсию:

```
s9 = s-nv (np-pn Mary) (vp-vt love a-human-that-runs
                                {{c∘ ((c∘ crcn (cnm {n₂ = Animate}))) cnm}}
                        )
```

Здесь требуется композиция коэрсий, и Агда не может самостоятельно определить, каков должен быть средний термин этой композиции. Мы подсказываем ей, что это должен быть `Animate`.

Интерпретация предложения `s9` равна:

```
⟦s s9 ⟧ ≡ Σ[ hr ∈ Σ[ h ∈ *Human ] ( h ) *run ] ( *Mary ) *love ( ( proj₁ hr ) )
```

Словами: «Существует человек, который бежит, и Мэри его любит».

Заключение

Таким образом, часто используемая в лингвистике классификация с помощью признаков может быть согласована с теорией типов в случае простого фрагмента естественного языка. Более того, реализованный в Агде механизм имплицитных экземпляров позволяет в простых случаях организовать автоматический пересчет типов. К сожалению, этот механизм создавался, имея в виду другие цели, да и сама Агда направлена прежде всего на нужды математики, а не теории естественного языка. Специализированный, ориентированный на семантику язык, с одной стороны, может быть проще, а с другой — должен удовлетворять дополнительно таким требованиям как допущение подтипов, работа с контекстами для учета референциальной непрозрачности и различия мнений, эффекты интенсиональности и другие. Вопрос о возможности такого «компилятора естественного языка» остается открытым.

Литература

Митренина О. В., *Слюсарь Н. А.*, *Романова Е. Е.* Введение в генеративную грамматику. — 2-е изд. — М. : Ленанд, 2018.

Agda Documentation / Agda. — 2014. — URL: `https://agda.readthedocs.io/`.

Asher N., *Luo Z.* Formalization of Coercions in Lexical Semantics // Proceedings of Sinn und Bedeutung. — 2013. — Vol. 17. — P. 63–80.

Bahramian H., *Nematollahi N.*, *Sabry A.* Copredication in Homotopy Type Theory / IUScholarWorks. — 2017. — URL: `https://hdl.handle.net/2022/21811`.

Carpenter B. The Logic of Typed Feature Structures with Applications to Unification Grammars, Logic Programs and Constraint Resolution. — Cambridge : Cambridge University Press, 1992. — (Cambridge Tracts in Theoretical Computer Science ; 32).

Chatzikyriakidis S., *Luo Z.* On the Interpretation of Common Nouns : Types Versus Predicates // Modern Perspectives in Type-Theoretical Semantics / ed. by S. Chatzikyriakidis, Z. Luo. — 1st ed. — Cham : Springer International Publishing, 2017. — P. 43–70. — (Studies in Linguistics and Philosophy ; 98).

Chatzikyriakidis S., *Luo Z.* Cognitive Science : Logic, Linguistics and Computer Science Set. In 2 vols. Vol. 2. Formal Semantics in Modern Type Theories. — London : Wiley, ISTE, 2020.

Coquand T. Pattern Matching with Dependent Types // Proceedings of the 1992 Workshop on Types for Proofs and Programs / ed. by B. Nordström, K. Petersson, G. Plotkin. — Båstad, 1992. — P. 66–79. — URL: https://wonks.github.io/type-theory-reading-group/papers/proc92-coquand.pdf.

Head-Driven Phrase Structure Grammar : The Handbook / ed. by S. Müller, A. Abeillé, R. D. Borsley, J.-P. Koenig. — Berlin : Language Science Press, 2021. — (Empirically Oriented Theoretical Morphology and Syntax ; 9).

Heim I., *Kratzer A.* Semantics in Generative Grammar. — Cambridge (MA), Oxford : Blackwell Publishers, 1998.

Luo Z. Computation and Reasoning : A Type Theory for Computer Science. — Oxford : Oxford University Press, 1994.

Luo Z. Common Nouns as Types // Logical Aspects of Computational Linguistics : 7th International Conference, LACL 2012, Nantes, France, July 2–4, 2012 / ed. by D. Béchet, A. Dikovsky. — Berlin, Heidelberg : Springer, 2012a. — P. 173–185. — (Lecture Notes in Computer Science ; 7351).

Luo Z. Formal Semantics in Modern Type Theories with Coercive Subtyping // Linguistics and Philosophy. — 2012b. — Vol. 35, no. 6. — P. 491–513.

Luo Z., *Soloviev S.*, *Xue T.* Coercive Subtyping : Theory and Implementation // Information and Computation. — 2013. — Vol. 223. — P. 18–42.

Martin-Löf P. An Intuitionistic Type Theory : Notes by Giovanni Sambin of a Series of Lectures Given in Padua, June 1980. — Napoli : Bibliopolis, 1984. — (Studies in Proof Theory ;)

Modern Perspectives in Type-Theoretical Semantics / ed. by S. Chatzikyriakidis, Z. Luo. — 1st ed. — Cham : Springer International Publishing, 2017. — (Studies in Linguistics and Philosophy ; 98).

Montague R. English as a Formal Language // Formal Philosophy : Selected Papers of Richard Montague / ed., with an introd., by R. H. Thomason. — New Haven, London : Yale University Press, 1974a. — P. 188–221.

Montague R. The Proper Treatment of Quantification in Ordinary English // Formal Philosophy : Selected Papers of Richard Montague / ed., with an introd., by R. H. Thomason. — New Haven, London : Yale University Press, 1974b. — P. 247–270.

Norell U. Dependently Typed Programming in Agda // Advanced Functional Programming : 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures / ed. by P. Koopman, R. Plasmeijer, D. Swierstra. —

1st ed. — Berlin, Heidelberg : Springer-Verlag, 2009. — P. 230–266. — (Lecture Notes in Computer Science ; 5832).

Ranta A. Type-Theoretical Grammar. — Oxford : Clarendon Press, 1994. — (Indices ; 1).

Retoré C. The Montagovian Generative Lexicon ΛTy_n : A Type Theoretical Framework for Natural Language Semantics // Proceedings of TYPES2013 / ed. by R. Matthes, A. Schubert. — Toulouse : TYPES, 2013. — P. 202–229.

Domanov, O. A. 2024. "Priznaki i tipy v teoretiko-tipovoy semantike yestestvennogo yazyka [Features and Types in Type–Theoretical Natural Language Semantics]" [in Russian]. *Filosofiya. Zhurnal Vysshey shkoly ekonomiki [Philosophy. Journal of the Higher School of Economics]* 8 (1), 188–214.

Oleg Domanov
PhD in Philosophy
Senior Research Fellow
Institute of Philosophy and Law
Siberian Branch of the Russian Academy of Science (Novosibirsk, Russia);
orcid: 0000–0003–0057–3901

Features and Types in Type-Theoretical Natural Language Semantics

Submitted: Apr. 12, 2023. **Reviewed:** Aug. 11, 2023. **Accepted:** Jan. 09, 2024.

Abstract: Features and types are two possible ways to classify phenomena related to the formalization of natural language grammar and semantics. Features are frequently used in linguistically oriented theories. However, they do not accord with type theoretical semantics due to the notion of subtype to which they lead. The article suggests a way to coordinate these two approaches through defining types based on the classification by features. An example of formalization for a small fragment of English language is provided for demonstration. General formal theory of syntax and semantics for this language is developed, which is of a separate interest. The language of formalization is Agda. Agda serves simultaneously as (1) a metalanguage that formalizes the syntax of the natural language and (2) a semantical/ontological language that interprets the natural language. This allows to formalize interpretation as a function that maps Agda expressions presenting syntax into Agda expressions comprising semantics. The concept of subtype is based on the notion of coercion. Defining types through features leads to the automatic definition of coercion between them. Agda's mechanism of instance arguments allows in many cases to provide coercion automatically. The article ends with examples of the natural language expression formalizations showing the theory in action. Despite Agda's primary orientation towards mathematics, it contains tools and instruments that render it applicable to natural language studies in the framework of type theoretical semantics.

Keywords: Type Theoretical Semantics, Natural Language, Type Theory, Agda, Subtypes, Coercion.

DOI: 10.17323/2587-8719-2024-1-188-214.

REFERENCES

"Agda Documentation." 2014. Agda. https://agda.readthedocs.io/.

Asher, N., and Z. Luo. 2013. "Formalization of Coercions in Lexical Semantics." *Proceedings of Sinn und Bedeutung* 17:63–80.

Bahramian, H., N. Nematollahi, and A. Sabry. 2017. "Copredication in Homotopy Type Theory." IUScholarWorks. https://hdl.handle.net/2022/21811.

Carpenter, B. 1992. *The Logic of Typed Feature Structures with Applications to Unification Grammars, Logic Programs and Constraint Resolution*. Cambridge Tracts in Theoretical Computer Science 32. Cambridge: Cambridge University Press.

Chatzikyriakidis, S., and Z. Luo, eds. 2017a. *Modern Perspectives in Type–Theoretical Semantics*. 1st ed. Studies in Linguistics and Philosophy 98. Cham: Springer International Publishing.

———. 2017b. "On the Interpretation of Common Nouns: Types Versus Predicates." In *Modern Perspectives in Type–Theoretical Semantics*, 1st ed., ed. by S. Chatzikyriakidis and Z. Luo, 43–70. Studies in Linguistics and Philosophy 98. Cham: Springer International Publishing.

———. 2020. *Formal Semantics in Modern Type Theories*. Vol. 2 of *Cognitive Science* : *Logic, Linguistics and Computer Science Set*. 2 vols. London: Wiley / ISTE.

Coquand, T. 1992. "Pattern Matching with Dependent Types." In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, ed. by B. Nordström, K. Petersson, and G. Plotkin, 66–79. Båstad. https://wonks.github.io/type-theory-reading-group/papers/proc92-coquand.pdf.

Heim, I., and A. Kratzer. 1998. *Semantics in Generative Grammar*. Cambridge (MA) and Oxford: Blackwell Publishers.

Luo, Z. 1994. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford: Oxford University Press.

———. 2012a. "Common Nouns as Types." In *Logical Aspects of Computational Linguistics* : *7th International Conference, LACL 2012, Nantes, France, July 2–4, 2012*, ed. by D. Béchet and A. Dikovsky, 173–185. Lecture Notes in Computer Science 7351. Berlin and Heidelberg: Springer.

———. 2012b. "Formal Semantics in Modern Type Theories with Coercive Subtyping." *Linguistics and Philosophy* 35 (6): 491–513.

Luo, Z., S. Soloviev, and T. Xue. 2013. "Coercive Subtyping: Theory and Implementation." *Information and Computation* 223:18–42.

Martin-Löf, P. 1984. *An Intuitionistic Type Theory: Notes by Giovanni Sambin of a Series of Lectures Given in Padua, June 1980*. Studies in Proof Theory. Napoli: Bibliopolis.

Mitrenina, O. V., N. A. Slyusar', and Ye. Ye. Romanova. 2018. *Vvedeniye v generativnuyu grammatiku [Introduction to Generative Grammar]* [in Russian]. 2nd ed. Moskva [Moscow]: Lenand.

Montague, R. 1974a. "English as a Formal Language." In *Formal Philosophy* : *Selected Papers of Richard Montague*, ed., with an introd., by R. H. Thomason, 188–221. New Haven and London: Yale University Press.

———. 1974b. "The Proper Treatment of Quantification in Ordinary English." In *Formal Philosophy* : *Selected Papers of Richard Montague*, ed., with an introd., by R. H. Thomason, 247–270. New Haven and London: Yale University Press.

Müller, S., et al., eds. 2021. *Head–Driven Phrase Structure Grammar: The Handbook*. Empirically Oriented Theoretical Morphology and Syntax 9. Berlin: Language Science Press.

Norell, U. 2009. "Dependently Typed Programming in Agda." In *Advanced Functional Programming : 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, 1st ed., ed. by P. Koopman, R. Plasmeijer, and D. Swierstra, 230–266. Lecture Notes in Computer Science 5832. Berlin and Heidelberg: Springer-Verlag.

Ranta, A. 1994. *Type–Theoretical Grammar*. Indices 1. Oxford: Clarendon Press.

Retoré, C. 2013. "The Montagovian Generative Lexicon $\Lambda T\,yn$: A Type Theoretical Framework for Natural Language Semantics." In *Proceedings of TYPES2013*, ed. by R. Matthes and A. Schubert, 202–229. Toulouse: TYPES.