# Теории типов — лекционные заметки

Alexander Gryzlov[1] and Alexander Kuklev[2,3] ‹a@kuklev.com›

[1]IMDEA Software Institute, Spain
[2]Radboud University Nijmegen, Software Science
[3]JetBrains Research

## 1. Вводная лекция

Добро пожаловать на курс по теориям типов!

Первым делом нужно рассказать, что вообще такое теории типов. А для этого нужно сперва немного поговорить о языках и системах записи, начиная с естественных языков, таких как тот, на котором написан этот текст.

### 1.1. Специализированные языки

Естественные языки очень универсальны. Кроме собственно коммуникации между людьми, они позволяют описывать факты и действия, доказательства и алгоритмы, абстрактные концепции и материальные устройства, идеализированные физические системы и реальные экспериментальные установки. Вместе с тем, естественные языки имеют фундаментальный недостаток, особенно критичный во времена LLM: они допускают разумно звучащую белиберду. Словесные описания длинных алгоритмов, доказательств или спецификаций бизнес-процессов/бизнес-логики зачастую содержат незаметные или намеренно замаскированные уязвимости, ошибки, неоднозначности и упущения. Проблемы такого рода продолжают находить спустя годы и десятилетия в текстах, внимательно вычитанных сотнями экспертов.

- В канадском споре Rogers Comm. и Aliant Telecom смысл 14-страничного договора зависел от запятой в положении о сроке действия и прекращении договора. Изначально спор мог стоить Rogers около 1 млн долларов; CRTC сначала приняла одну интерпретацию, затем, после анализа французской версии соглашения, изменила решение.
- Выдающийся математик Владимир Воеводский стал лауреатом Филдсовской премии за свои работы в области теории мотивов. Частью фундамента этой работы был технический аппарат, описанный им в статье “Cohomological Theory of Presheaves with Transfers” в 1992 году. У статьи сотни цитирований; начиная с 1993 года несколько групп математиков изучали эту статью на семинарах и использовали её в своей работе, но годами никто не замечал ошибки. Её заметил сам автор — спустя целых восемь лет, и ошибка была существенной. Не только доказательство одной из ключевых лемм содержало ошибку — сама формулировка леммы не подлежала спасению. Потратив ещё шесть лет, Воеводский сумел заменить её более слабой и более сложной леммой, которой всё же хватало для всех приложений. Именно этот опыт стал причиной его интереса к машинной проверке доказательств и прорывным идеям, о которых мы будем много говорить в ходе этого курса.

Справляться с фундаментальным недостатком естественных языков позволяют специализированне формальне языки, устраняющих неоднозначности и навязывающих структурированность и систематичность изложения. Неоднозначность становится невыразимой, а лакуны и одумки становятся явными. Специализированные языки должны быть пригодными для алгоритмической обработки и в то же время максимально удобными для чтения и понимания человеком — удобнее естественного языка для квалифицированного читателя.

Немало именно таких языков появилось задолго до появления компьютеров. Один из старейших примеров — позиционная запись чисел цифрами:

- записать, прочитать и понять в голове число 2524 значительно проще, чем «две тысячи пятьсот двадцать четыре»,
- запись цифрами автоматически исключает числа-одумки («двадцать восемь, двадцать девять, двадцать десять»),
- удачная специализированная запись открывает новые возможности — арифметические операции в столбик. Это как раз пример “алгоритмической обработки”, ставшей возможной благодаря удачному специализированному способу записи.

Ещё один важный пример — химическая запись ($C_2H_5OH$, anyone?), значительно поспособствовавшая в своё время пониманию химических реакций. А вот запись чисел римскими цифрами — это пример-предостережение, как специализированный язык может погрязнуть в избыточной сложности в результате отсутствия какой-либо идеи. В данном случае — концепции нуля.

## 1.2. Языки программирования

Наверняка, самый знакомый вам пример — языки программирования. Программы на них — это просто конечные последовательности символов заданного наперёд алфавита. Но не любые, а лишь те, которые допускает компилятор/интерпретатор языка. А семантика языка программирования (то есть исчерпывающее смысловое значение текстов на нём) тоже в конечном итоге определяется интерпретатором/компилятором и поведением машины, на которой программа исполняется.

Для многих языков программирования написана книга под названием “стандарт языка”, где описана формальная грамматика языка (в терминах строгих синтаксических правил) и семантика языка, как правило, в терминах того, как программа должна в точности исполняться на идеализированной/стандартизированной машине.[1] Отметим, однако, что почти для всех реальных языков программирования стандарт языка — это больше декларация о намерениях, чем исчерпывающее описание.

Исключение составляют языки программирования, возникшие не из практических нужд, а из математических абстракций — в частности, языки на базе комбинаторной логики/λ-исчисления. Почти все такие языки программирования низкоуровневые, то есть больше похожи на машинный код — невозможно понять смысл программы нетривиальной длины, читая код; результат работы программы (даже если это просто число) как правило нуждается в расшифровке, а для написания нетривиальных программ нужен ещё слой сокращений-макросов, своего рода вторичный язык.

[1] Это называется операционной семантикой. Существуют и другие подходы, среди которых нужно отдельно отметить денотационную и аксиоматическую семантику. О том и другом мы ещё подробно поговорим в следующих лекциях.

## 1.3. System T: Особенный язык программирования

Первый высокоуровневый “математический” язык программирования был побочным результатом математической работы 1958 года Курта Гёделя о непротиворечивости арифметики первого порядка. Автор не планировал разрабатывать язык программирования, не интересовался машинной имплементацией и совершенно не задумывался о практическом удобстве — устройство языка диктовалось соображениями логической структуры арифметики.

Тем не менее, результатом стал высокоуровневый строго-типизированный язык программирования System T на базе λ-исчисления и самой простой формы рекурсии, где числа — это числа, их последовательности — это последовательности, а программы напрямую работают с такими объектами (без постоянного кодирования и декодирования) и используют достаточно высокоуровневые примитивы, чтобы нетривиальные алгоритмы в самом деле можно было писать руками и читать глазами.

Удивительная особенность этого языка программирования — поразительная элегантность и простота его описания. В то время как стандарт языка C — это книга на 284 страницы, System T полностью описывается небольшим набором правил, который легко вмещается на полстраницы (см. ниже) и исчерпывающе описывает синтаксис, семантику и систему типов языка.

Правила отталкиваются именно от системы типов: они описывают, какие типы данных встречаются в языке, и для каждого типа даётся исчерпывающий набор операторов порождения и разбора значений этого типа; кроме того, описывается действие операторов разбора на операторы порождения.

$$\frac{}{\Gamma, x : T \vdash x : T}\text{(var)} \qquad \frac{}{\text{Unit}}\text{(unit)} \qquad \frac{}{\Gamma \vdash () : \text{Unit}}$$

$$\frac{X \qquad Y}{X \to Y}\text{(func)} \qquad \frac{\Gamma, x : X \vdash f : Y}{\Gamma \vdash \lambda(x : X)\, f : X \to Y} \qquad \frac{\Gamma \vdash f : X \to Y \qquad \Gamma \vdash x : X}{\Gamma \vdash f x : Y}$$

$$\frac{X \qquad Y}{X \times Y}\text{(pair)} \qquad \frac{\Gamma \vdash x : X \qquad \Gamma \vdash y : Y}{\langle x, y \rangle : X \times Y} \qquad \frac{\Gamma \vdash p : X \times Y}{\Gamma \vdash p_{\text{fst}} : X} \qquad \frac{\Gamma \vdash p : X \times Y}{\Gamma \vdash p_{\text{snd}} : Y}$$

$$\frac{T}{T^*}\text{(list)} \qquad \frac{}{\Gamma \vdash \text{nil} : T^*} \qquad \frac{\Gamma \vdash \text{head} : T \qquad \Gamma \vdash \text{tail} : T^*}{\Gamma \vdash \text{head}, \text{tail} : T^*}$$

$$\frac{\Gamma \vdash \text{list} : X^* \qquad \Gamma \vdash \text{init} : Y \qquad \Gamma, \text{head} : X, \text{tail} : X^*, \text{prev} : Y \vdash \text{step} : Y}{\Gamma \vdash \text{reduce}(\text{list})(\text{step})(\text{init}) : Y}$$

$$(\lambda(x : X)\, f(x))\ x' \rightsquigarrow f(x') \qquad \langle x, y \rangle_{\text{fst}} \rightsquigarrow x \qquad \langle x, y \rangle_{\text{snd}} \rightsquigarrow y$$

$$\text{reduce}(\text{head}, \text{tail})(\text{step})(\text{init}) \rightsquigarrow \text{step}(\text{head})(\text{tail})(\text{reduce}(\text{tail})(\text{step})(\text{init}))$$

$$\text{reduce}(\text{nil})(\text{step})(\text{init}) \rightsquigarrow \text{init}$$

Такой подход называется типотеоретическим описанием языка, а языки, описанные таким образом, называются теориями типов (в узком смысле). Математическая дисциплина, изучающая их, тоже называется теорией типов (в широком смысле). Типотеоретическое описание языка не только очень элегантно и экономично, но ещё и очень удобно для того, чтобы математически рассуждать о свойствах программ, доказывать или опровергать их свойства.

Большую часть языков программирования (C, Java) невозможно описать так экономно, но можно разрабатывать языки программирования, отталкиваясь

от возможностей типотеоретического описания. Разработчик всё ещё исходит из практических нужд программирования, но вынужден действовать в очень строгих рамках возможного. Эти рамки очень часто, more often than not, подсказывают элегантные дизайнерские решения. Иногда они не просто элегантные, а вообще единственно возможные. Как и в математике, стирается грань между исследованием и разработкой. Разработчик языка одновременно “придумывает” новые конструкции языка и “открывает“ правильный способ их реализации, вытекающий из самой природы вещей.

## 1.4. Гораздо больше, чем языки программирования

Хотя теории типов неразрывно связаны с computer science, их область применимости гораздо шире, чем программирование. Собственно, успешных мейнстримных языков программирования с исчерпывающим типотеоретическим описанием до сих пор нет, хотя мы и приближаемся к этому.

Теория типов произвела грандиозную революцию в области машинопроверяемых доказательств и оснований математики. Типотеоретический подход подсказал элегантные способы улучшить языки для формулирования и доказательства математических теорем. Эти языки, называемые исчислениями построений/конструктивными теориями типов, всё ещё родственны языкам программирования, а точнее включают в себя тотальные функциональные языки программирования наряду с языком доказательств.

Однако теории типов могут быть совершенно не связанными с программированием. Так, в 2018-2023 годах были получены типотеоретические описания ω-категорий и родственных объектов, что привело к существенному прогрессу в соответствующем разделе математики.

С другой стороны, основанные на теории типов языки программирования могут быть обобщены до языков описания взаимодействующих систем (см. KeymaeraX,[2] VeriDrone,[3] SpaceEx[4]). В том числе гибридных систем, где часть компонентов действует согласно детерминистическому алгоритму, а часть лишь удовлетворяет определённым (непрерывным и/или вероятностным) моделям поведения, описываемым в терминах кусочно-дифференциальных уравнений. Такие языки уже сейчас служат основой для формальных доказательств безопасности беспилотных автомобилей и летательных аппаратов, газопроводов и реакторов.

Дальнейшим обобщением являются языки, способные описывать экспериментальные установки и физические системы, в которых могут играть роль квантовые эффекты — область, где естественные языки как раз теряют применимость. В последние пять лет достигнут огромный прогресс в области экспериментальных теорий типов, естественным образом описывающих квантовые теории поля и теории струн (Myers et al., 2024; Sati and Schreiber, 2026). Такие теории призваны предоставить язык, позволяющий достичь концептуального понимания в проблеме измерения, проблеме локального реализма и, в конечном итоге, проблеме времени.

[2] https://keymaerax.org/
[3] https://ucsd-pl.github.io/veridrone/
[4] http://spaceex.imag.fr/

# 2. Языки и алгебраические теории

Типизированные языки задаются грамматическими правилами вида:

$$\frac{s_1 : T_1 \qquad \cdots \qquad s_n : T_n}{e : T}$$

Грамматическое правило описывает, как выражения складываются из подвыражений. Под горизонтальной чертой указывается описываемое выражение и – через двоеточие – его тип. Над горизонтальной чертой указываются несколько (их может быть ноль) подвыражений, используемых в выражении. В естественных языках типы — это части речи (существительное) или типы функциональных групп (напр. “новый алый шарф” — номинальная группа).[5]

В качестве примера давайте рассмотрим язык алгебраических выражений со скалярами и векторами из эвклидова векторного пространства над ними:

$$\frac{a : \text{Scalar} \qquad v : \text{Vector}}{av : \text{Vector}} \qquad \frac{v : \text{Vector} \qquad u : \text{Vector}}{v \cdot u : \text{Scalar}}$$

$$\frac{v : \text{Vector} \qquad u : \text{Vector}}{v + u : \text{Vector}} \qquad \frac{v : \text{Vector}}{-v : \text{Vector}} \qquad \frac{}{0 : \text{Vector}}$$

$$\frac{a : \text{Scalar} \qquad b : \text{Scalar}}{a + b : \text{Scalar}} \qquad \frac{a : \text{Scalar}}{-a : \text{Scalar}} \qquad \frac{}{0 : \text{Scalar}}$$

$$\frac{a : \text{Scalar} \qquad b : \text{Scalar}}{ab : \text{Scalar}} \qquad \frac{}{1 : \text{Scalar}}$$

В типизированных языках выражения отождествляются со своими деревьями вывода. Поэтому правила не содержат скобок, приоритизации и ассоциативности операторов. Преобразование выражений в текст и обратно (парсинг текста в синтаксические деревья) — это отдельная область, называемая теорией формальных языков, и её мы тут рассматривать не будем.

Типизированные языки могут иметь бесконечные ряды типов и правил. Характерный пример — нетипизированное λ-исчисление, где тип выражения определяется количеством свободных переменных в нём, и стало быть, типов бесконечно много. Грамматические правила там получаются из нижеследующих схем путём подстановки натуральных чисел в качестве индексов:

$$\frac{f : \text{Term}_n \qquad x : \text{Term}_n}{fx : \text{Term}_n}(\text{app}) \qquad \frac{t : \text{Term}_{n+i+1}}{\lambda_i t : \text{Term}_{n+i}}(\text{abs}) \qquad \frac{}{x_n : \text{Term}_n}(\text{var}) \qquad \frac{t : \text{Term}_n}{t : \text{Term}_{n+1}}(\text{sub})$$

Использование нумерованных переменных не слишком удобно на практике: людям удобнее воспринимать именованные переменные. В связи с этим был разработан особый формат записи грамматических правил, называемый естественным выводом. Вместо того чтобы писать в типе выражения количество переменных, перед значением записывается контекст — список именованных переменных, используемых в нём.

$$\frac{\Gamma \vdash f : \text{Term} \qquad \Gamma \vdash x : \text{Term}}{\Gamma \vdash fx : \text{Term}}(\text{app}) \qquad \frac{\Gamma, x \vdash t : \text{Term}}{\Gamma \vdash (x \mapsto t) : \text{Term}}(\text{abs})$$

$$\frac{}{x \vdash x : \text{Term}}(\text{var}) \qquad \frac{\Gamma \vdash t : \text{Term}}{\Gamma, x \vdash t : \text{Term}}(\text{sub})$$

[5] Типизированные языки используются и в лингвистике. См. Категориальные грамматики Ламбека — это формальные системы для математически строгого анализа синтаксиса и семантики естественных языков.

Для полноты картины давайте приведём ещё язык теории множеств:

$$\frac{}{x, y \vdash x = y : \mathrm{Prop}}(\text{eq}) \qquad \frac{}{x, y \vdash x \in y : \mathrm{Prop}}(\text{in}) \qquad \frac{\Gamma \vdash p : \mathrm{Prop}}{\Gamma, x \vdash p : \mathrm{Prop}}\text{sub}$$

$$\frac{\Gamma, x \vdash p : \mathrm{Prop}}{\Gamma \vdash \exists(x)\, p : \mathrm{Prop}}(\text{exists}) \qquad \frac{\Gamma, x \vdash p : \mathrm{Prop}}{\Gamma \vdash \forall(x)\, p : \mathrm{Prop}}(\text{forall})$$

$$\frac{\Gamma \vdash p : \mathrm{Prop} \qquad \Gamma \vdash q : \mathrm{Prop}}{\Gamma \vdash p \wedge q : \mathrm{Prop}}(\text{and}) \qquad \frac{\Gamma \vdash p : \mathrm{Prop} \qquad \Gamma \vdash q : \mathrm{Prop}}{\Gamma \vdash p \to q : \mathrm{Prop}}(\text{implies})$$

$$\frac{\Gamma \vdash p : \mathrm{Prop} \qquad \Gamma \vdash q : \mathrm{Prop}}{\Gamma \vdash p \vee q : \mathrm{Prop}}(\text{or}) \qquad \frac{\Gamma \vdash p : \mathrm{Prop}}{\Gamma \vdash \neg p : \mathrm{Prop}}(\text{not})$$

Отметим, что естественная дедукция — всего лишь способ записи. Имеются механические способы перевода правил естественной дедукции в обычные синтаксические правила — de Bruijn и co-de Bruijn translations.

Естественная дедукция значительно упрощает описание типизированных λ-исчислений, языков аксиоматических теорий и, совершенно особенно, языков доказательств, для которых она, собственно, и была придумана.

## 2.1. Алгебраические теории

Грамматика языка ещё не описывает его семантики, однако во некоторых очень важных случаях для исчерпывающего описания семантики достаточно добавить в язык тождества — правила о том, в каких случаях выражения равны между собой. Тождества тоже записываются как правила с горизонтальной чертой. Сверху — переменные, снизу — равенство двух выражений. Язык вместе с набором тождеств называется алгебраической теорий.

Например, множества с одной ассоциативной операцией (например, целые числа со сложением) суть модели следующей алгебраической теории:

$$\frac{x : S \qquad y : S}{xy : S} \qquad \frac{x : S \qquad y : S \qquad z : S}{(xy)z = x(yz)}$$

Модели этой теории, кстати, называются в математике полугруппами. Если ещё добавить нейтральный элемент — получатся моноиды. Умножение матриц тоже ассоциативная операция, но тут нам понадобится типизированная теория, где типы характеризуют размеры матриц.

$$\frac{x : M_n^m \qquad y : M_k^n}{\Gamma \vdash xy : M_k^m} \qquad (xy)z = x(yz)$$

Такие типизированные моноиды называются категориями, и являются одной из самых часто встречающихся и богатых структур в математике. Разумеется, в общем случае индексы $m$ и $n$ не обязаны быть положительными целыми числами, а могут быть представителями любого множества (точнее, класса).

Лупы, группы, кольца, решётки, алгебры Ли — всё это также алгебраические теории. Модели алгебраических теорий образуют категории, обладающие массой элегантных свойств. Если тот или иной класс математических объектов удаётся охарактеризовать в терминах алгебраических теорий, это даёт глубокое структурное понимание объектов и широчайший инструментарий математических средств для работы с ним. Среди прочего, для каждой алгебраической теории существует уникальная универсальная модель, а для каждого множества порождающих

существует свободно-порождённая ими модель. Так, например, свободный моноид на $n$ порождающих — это просто множество строк алфавита из n букв вместе с операцией конкатенации.

Ценные свойства алгебраических теорий вытекают из того, что операция построения свободно-порождённой модели является монадой (что бы это ни значило), и порождённая этой монадой категория — в точности категория всех моделей алгебраической теории с гомоморфизмами между ними.

Когда мы говорили о типизированных языках, мы требовали, чтобы каждое правило вывода содержало лишь конечное число посылок (подвыражений), а сами грамматические правила были эффективно перечислимы, т.е. существовала компьютерная программа, которая последовательно генерирует все правила. Эти ограничения гарантируют, что выражения языка тоже эффективно перечислимы. Это соответствует интуитивным представлениям о том, что такое язык — перечислимость выражений означает, что в конечном итоге все они могут быть записаны как конечный текст на конечном алфавите.

Тем не менее, ценные математические свойства алгебраических теорий сохраняются и в случае, если допускаются неперечисимо большие языки. Первый пример — параметризованные языки. Мы можем взять язык, где для каждого вещественного числа $s : \mathbb{R}$ имеется операция скалярного умножения

$$\frac{v : \mathrm{Vector} \qquad u : \mathrm{Vector}}{v + u : \mathrm{Vector}} \qquad\qquad \frac{v : \mathrm{Vector}}{sv : \mathrm{Vector}} \text{ for each } s : \mathbb{R}$$

Нам потребуются также параметризованные тождества, гарантирующие ассоциативность и дистрибутивность скалярного произведения. Модели этой теории — в точности векторные пространства над вещественными числами.

Другой пример — теории с оператором, принимающим бесконечное число аргументов. Скажем, оператор $\lim(a_n)$ предела последовательности.[6] Модели такой теории — компактные Хаусдорфовы пространства.

В обоих случаях язык теории оказывается несчётным, но это не мешает построить её монаду. Такие теории называются инфинитарно-алгебраическими.

## 2.2. Внутренние языки категорий

В 1970ые годы было обнаружено, что важный класс категорий (декартово замкнутые категории) — суть в точности модели алгебраической теории STLC (просто типизированного λ-исчисления). Таким образом, все построения, доступные в декартово-замкнутых категориях благодаря их структуре, могут быть описаны на языке STLC, а все их свойства вытекают из тождеств STLC. Так сформировалось понятие внутреннего языка класса категорий — алгебраической теории, моделями которой являются категории этого класса.

Вскоре Seely (1984) сформулировал аналогичный результат для локально декартово замкнутых категорий, см. Clairambault and Dybjer (2014) для полного доказательства. Проблема в том, что их “внутренним языком” является экстенсиональная теория типов $\lambda_{\Pi\Sigma\equiv}$, не являющаяся алгебраической теорией. Тут нужно перейти от строгих категорий к $\infty$-категориям, где свойства композиции соблюдаются не строго, а с точностью до эквивалентности. У локально декартово-замкнутых $\infty$-категорий в самом деле есть внутренний язык — унивалентная теория типов $\lambda_{\Pi\Sigma\simeq}$ (Kapulkin, 2017).

[6] Для полной общности нужно использовать ультрафильтры вместо последовательностей.

В ходе этого курса мы будем говорить о внутренних языках других классов категорий, важных с типотеоретической точки зрения, см. рис. 1.

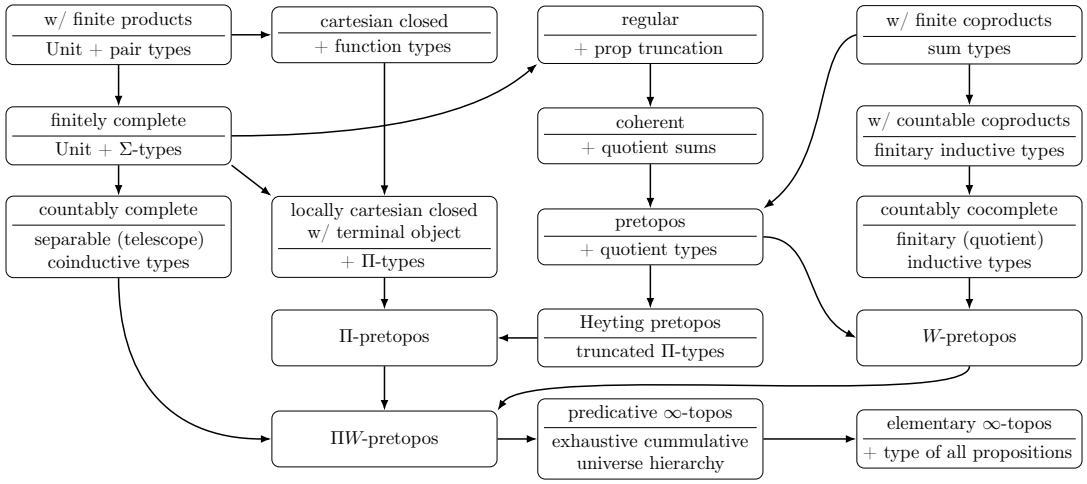


Рис. 1: Основные классы типотеоретически важных $\infty$-категорий

Чтобы быть совсем точными, отметим, что предикативные и элементарные $\infty$-топосы обладают внутренними языками в более слабом смысле. Все эти $\infty$-топосы являются моделями соответствующей теории, но не все модели являются такими $\infty$-топосами. Однако ими являются все конечно-порождённые модели, и в том числе универсальный элементарный $\infty$-топос.

## 2.3. Универсальный элементарный $\infty$-топос

С начала 1920ых годов в качестве общей базы для всей математики использовалась теория множеств Цермело-Френкеля ZFC. К сожалению, эта теория не является в каком-либо смысле канонической — у неё есть масса вариаций или аналогов, которые как будто ничем не хуже, но и не сильно лучше.

В середине 20го века начала развиваться теория категорий — дисциплина, посвящённая главным образом абстрагированию квинтэссенции конкретных свойств объектов. Ключевым свойством ZFC, делающим её пригодной на роль общей базы, является возможность конструировать в ней модели аксиоматических теорий. Это свойство было абстрагировано — в конце 1960ых был охарактеризован класс категорий, обладающий этим свойством. Они называются элементарными топосами, и их внутренний язык (то есть язык универсального элементарного $\infty$-топоса) является каноническим языком математики.

Поиск этого языка — путешествие длиной в 40 лет, близкое к завершению. На данный момент язык реализован в пруф-ассистенте Narya,[7] за исключением одной технической недоделки[8] и одной, правда весьма ключевой, теоремы.[9]

[7] https://narya.readthedocs.io/en/latest/
[8] Не реализованы индуктивные семейства — их реализация там весьма сложна технически.
[9] Доказательство фибрантности унивалентных вселенных.

# 3. STLC и System T — подробный разбор

Во вводной лекции System T появился как первый высокоуровневый «математический» язык программирования, выросший из работы Гёделя о непротиворечивости арифметики. Теперь мы рассмотрим его как тотальный язык, построим для него нормализатор, а затем вернёмся к его исходной роли целевого языка для трансляции арифметических доказательств.

Каждая программа System T завершается (это и есть свойство тотальности). Благодаря этому программы всегда можно полностью нормализовать, а затем сравнивать и анализировать как конечные синтаксические деревья.

Весь материал лекции формализован в Rocq: интерпретация Диалектика находится в репозитории `dialectica-system-t`,[10] а нормализация в его подмодуле `nbe-system-t`,[11] следующем второй главе диссертации Abel (2013). Ядро STLC можно отдельно посмотреть в реализации на Idris.[12]

## 3.1. Конечные типы данных, суммы, пары, функции

Как мы подчеркнули в первой лекции, экономичное описание языков программирования отталкивается от доступных в них типов данных. Самый простой содержательный тип данных — бит, булево значение. Условимся обозначать этот тип $\mathbb{B}$, а его канонические значения $\mathtt{tt}$ и $\mathtt{ff}$. Для всякого типа данных нужно указать синтаксис порождения значений и синтаксис их разбора. Синтаксис порождения просто описывает возможные канонические значения:

$$\frac{}{\Gamma \vdash \mathtt{tt} : \mathbb{B}} \qquad \frac{}{\Gamma \vdash \mathtt{ff} : \mathbb{B}}$$

Синтаксис разбора булевых значений — привычный оператор $\mathsf{if}$:

$$\frac{\Gamma \vdash b : \mathbb{B} \qquad \Gamma \vdash \mathrm{then} : T \qquad \Gamma \vdash \mathrm{else} : T}{\Gamma \vdash \mathsf{if}(\boldsymbol{b})\,\mathrm{then}\,\mathrm{else} : T}$$

То, как “работает” оператор, задают правила вычисления (= $\beta$-reduction):

$$\mathsf{if}(\mathtt{tt})\,\mathrm{then}\,\mathrm{else} \rightsquigarrow \mathrm{then} \qquad \mathsf{if}(\mathtt{ff})\,\mathrm{then}\,\mathrm{else} \rightsquigarrow \mathrm{else}$$

Посторонние значения исключаются правилом уникальности (= $\eta$-expansion):

$$\frac{\Gamma \vdash b : \mathbb{B}}{b \equiv \mathsf{if}(b)\,\mathtt{tt}\,\mathtt{ff}}$$

Стандартные действия над натуральными числами — $n+m$, $n \cdot m$ и $n^m$ — мотивированы операциями над конечными множествами. Так $n^m$ это количество отображений из множества с $m$ элементами во множество с $n$ элементами; $n \cdot m$ это количество (упорядоченных) пар, где первый элемент из множества с $n$ элементами, а второй — из множества с $m$ элементами. Наконец, сумма $n + m$, это размер размеченного объединения множеств с $n$ и $m$ элементами соответственно. Давайте опишем соответствующие операции на типах.

Если у нас есть два типа данных $X$ и $Y$, можно задать их размеченное объединение

[10] https://github.com/clayrat/dialectica-system-t/
[11] https://github.com/clayrat/nbe-system-t/
[12] https://github.com/sequents/code/tree/master/src/Lambda

$(X + Y)$ — обобщение типа $\mathbb{B}$:

$$\frac{\Gamma \vdash x : X}{\Gamma \vdash \mathsf{inl}(x) : (X + Y)} \qquad \frac{y : Y}{\Gamma \vdash \mathsf{inr}(y) : (X + Y)}$$

$$\frac{\Gamma \vdash e : (X + Y) \qquad \Gamma, x : X \vdash \text{left} : T \qquad \Gamma, y : Y \vdash \text{right} : T}{\Gamma \vdash \mathsf{match}(\boldsymbol{e})\,\{\mathsf{inl}\,x \mapsto \text{left}; \mathsf{inr}\,y \mapsto \text{right}\} : T}$$

$$\mathsf{match}(\mathsf{inl}\,x)\,\{\mathsf{inl}(x) \mapsto \text{left}; \mathsf{inr}\,y \mapsto \text{right}\} \rightsquigarrow \text{left}(x)$$

$$\mathsf{match}(\mathsf{inr}\,y)\,\{\mathsf{inl}\,x \mapsto \text{left}; \mathsf{inr}\,y \mapsto \text{right}\} \rightsquigarrow \text{right}(y)$$

$$\frac{\Gamma \vdash e : (X + Y)}{e \equiv \mathsf{match}(\boldsymbol{e})\,\{\mathsf{inl}\,x \mapsto \mathsf{inl}\,x; \mathsf{inr}\,y \mapsto \mathsf{inr}\,y\}}$$

Если у нас есть два типа данных $X$ и $Y$, можно задать тип пар $X \times Y$.

$$\frac{X \qquad Y}{X \times Y}(\text{pair}) \qquad \frac{\Gamma \vdash x : X \qquad \Gamma \vdash y : Y}{\langle x, y \rangle : X \times Y} \qquad \frac{\Gamma \vdash p : X \times Y}{\Gamma \vdash p_{\text{fst}} : X} \qquad \frac{\Gamma \vdash p : X \times Y}{\Gamma \vdash p_{\text{snd}} : Y}$$

$$\langle x, y \rangle_{\text{fst}} \rightsquigarrow x \qquad \langle x, y \rangle_{\text{snd}} \rightsquigarrow y \qquad \frac{\Gamma \vdash p : X \times Y}{p \equiv \langle p_{\text{fst}}, p_{\text{snd}} \rangle}$$

Если у нас есть два типа данных $X$ и $Y$, можно задать тип функций $X \to Y$. Его каноническими элементами будут λ-термы — выражения типа $Y$ нашего же языка с переменной типа $X$. Внутри этих выражений можно использовать в точности те синтаксические конструкции, которые мы ввели — оператор $\mathsf{if}$ и значения $\mathtt{tt}$ и $\mathtt{ff}$, операторы $\mathsf{match}$ и конструкторы $\mathsf{inl}$, $\mathsf{inr}$, пары и проекции, ну и конечно же сами λ-термы и применение функций.

$$\frac{X \qquad Y}{X \to Y}(\text{func}) \qquad \frac{\Gamma, x : X \vdash f : Y}{\Gamma \vdash \lambda(x : X)\, f : X \to Y} \qquad \frac{\Gamma \vdash f : X \to Y \qquad \Gamma \vdash x : X}{\Gamma \vdash f\,x : Y}$$

$$(\lambda(x : X)\, f(x))\ x' \rightsquigarrow f(x') \qquad \frac{\Gamma \vdash f : X \to Y}{f \equiv \lambda(x : X)\, f\,x}$$

Получившийся язык программирования называется расширенным просто типизированным λ-исчислением, и все выразимые на нём типы данных суть конечные множества. Чтобы на нём были выразимы все конечные множества, нам нужно добавить ещё тип $\mathbb{1}$ с одним единственным значением и тип $\mathbb{0}$ вовсе без значений. Это в точности внутренний язык бидекартово замкнутых категорий.

(Обыкновенным) просто типизированным λ-исчислением называется подмножество данного языка, образуемое единичным типом, парами и функциями. Это — язык декартово замкнутых категорий. В отличие от расширенного варианта, в таком языке невозможно сконструировать типы с более чем одним значением, так что он тривиален как язык программирования, если в него не добавить дополнительные дополнительные базовые типы (такие как $\mathbb{B}$).

## 3.2. System T как язык программирования

Расширенное просто-типизированное λ-исчисление позволяет работать с конечными списками наперёд известного размера ($\mathsf{Byte} := \mathbb{B}^8$), но не позволяет говорить о списках произвольного конечного размера

$$T^* = \mathbb{1} + T + T^2 + T^3 + \cdots$$

Добавление типов списков произвольного размера как раз даст System T в максимальной формулировке — высокоуровневый язык программирования с минималистичной спецификацией. Во вводной лекции мы приводили все правила для типа списков $T^*$ с оператором $\mathsf{reduce}$; в этой лекции, для простоты изложения, мы вернемся к классическому гёделевскому варианту, где System T получается добавлением типов $\mathbb{B}$ и $\mathbb{N}$ (натуральные числа — частный случай списков $\mathbb{N} = \mathbb{1}^*$) и примитивной рекурсии к просто типизированному λ-исчислению (STLC), где имеются только функции и пары. Различия между этими формулировками System T тривиальны с формальной точки зрения.[13]

$$T ::= \mathbb{N} \mid \mathbb{B} \mid \mathbb{1} \mid T \to T \mid T \times T$$

По сравнению со STLC добавляются конструкторы и рекурсор для $\mathbb{N}$:

$$\frac{}{\mathbb{N}}(\text{nat}) \qquad \frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathrm{S}\, n : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N} \qquad \Gamma \vdash \mathit{step} : \mathbb{N} \to T \to T \qquad \Gamma \vdash \mathrm{base} : T}{\Gamma \vdash \mathsf{iterate}(\boldsymbol{n})\, \mathrm{step}\, \mathrm{base} : T}$$

$$\mathsf{iterate}(0)\, \mathrm{step}\, \mathrm{base} \rightsquigarrow \mathrm{base} \qquad \mathsf{iterate}(\mathrm{S}\, n)\, \mathrm{step}\, \mathrm{base} \rightsquigarrow \mathrm{step}\, n\, (\mathsf{iterate}(n)\, \mathrm{step}\, \mathrm{base})$$

Оператор $\mathsf{iterate}$ задаёт *примитивную рекурсию*: он разбирает число, имея на каждом шаге доступ к предшественнику и к результату рекурсивного вызова. Например, сложение записывается как

$$\mathrm{add}(m)(n) = \mathsf{iterate}(n)(\lambda k\, r.\, \mathrm{S}\, r)\, m,$$

и вычисление $\mathrm{add}(2)(2)$ разворачивается в цепочку

$$\mathrm{S}\,(\mathsf{iterate}(1)(\ldots)\,2) \rightsquigarrow \mathrm{S}\,\mathrm{S}\,(\mathsf{iterate}(0)(\ldots)\,2) \rightsquigarrow \mathrm{S}\,\mathrm{S}\,2 = 4.$$

Классическое упражнение состоит в определении предшественника: прямой рекурсии «на единицу назад» у нас нет, но можно рекурсивно строить пару $(k{-}1, k)$ и в конце взять первую компоненту. Это типичный приём: результирующий тип $T$ рекурсора может сам быть парой или функцией. Именно за счёт рекурсии на высших типах System T выразительно богаче примитивно-рекурсивных функций первого порядка: например, функция Аккермана записывается в нём без труда.

При этом язык тотален: *все* программы System T завершаются. Неформальный аргумент состоит в том, что каждый $\mathsf{iterate}$ съедает конкретное натуральное число, а типизированное λ-исчисление само по себе не содержит ни общей рекурсии, ни комбинаторов неподвижной точки. Самая мощная формулировка тотальности через отношение редукции называется сильной нормализацией и требует отдельного доказательства.[14] Сконструированный ниже нормализатор реализует утверждение *для каждого терма существует нормальная форма*; конечность любой последовательности редукций из этого не следует.

[13] Типы $\mathbb{B}$ и $\mathbb{1}$, вообще говоря, тоже необязательны.

[14] Как правило, строящееся методом *логических отношений* (logical relations).

#### Внутренне типизированное (intrinsically typed) представление

В формализации синтаксис и типизация совмещены в одном индуктивном семействе:

```
Inductive tm : cxt -> ty -> Type :=
| tzero : tm Γ tN
| tsuc  : tm Γ tN -> tm Γ tN
| trec  : tm Γ T -> tm Γ (tN ⇒ T ⇒ T) -> tm Γ tN -> tm Γ T
| tvar  : var Γ T -> tm Γ T
| tlam  : tm (S :: Γ) T -> tm Γ (S ⇒ T)
| tapp  : tm Γ (S ⇒ T) -> tm Γ S -> tm Γ T
(* ... *)
```

Конструкторы `tm` непосредственно повторяют правила естественной дедукции: контекст представляет собой список типов, переменные представляют собой типизированные индексы де Брёйна (`var Γ T` свидетельствует, что $T$ лежит в $\Gamma$), а `tlam` расширяет контекст. Некорректно типизированный терм таким способом не строится, поэтому отдельное отношение типизации и лемма о сохранении типов при редукции не нужны.

#### Открытые термы и заблокированные вычисления

Замкнутая программа типа $\mathbb{N}$ всегда вычисляется до числового значения.[15] Интереснее ведут себя открытые термы. Сравним (при нашем определении сложения, где рекурсия ведётся по второму аргументу):

$$x + 2 \rightsquigarrow^* \mathrm{S}\,\mathrm{S}\,x, \qquad 2 + x \rightsquigarrow^* \mathsf{iterate}(x)(\lambda k\, r.\,\mathrm{S}\,r)\,2.$$

Первый терм вычислился до конструкторов; второй *заблокирован*: итератору нечего разбирать, пока $x$ неизвестен. Нормализация открытых термов представляет собой символьное исполнение, а не просто вычисление замкнутых данных. Заблокированные («нейтральные») подвыражения при этом сохраняются в нормальной форме вместе с уже вычисленными фрагментами.

## 3.3. Нормализация вычислением (NbE)

Нормализация нужна, чтобы упрощать и сравнивать программы System T, в том числе программы, которые ниже будут извлечены из доказательств интерпретацией «Диалектика». При упрощении нормализатор заранее вычисляет фрагменты, аргументы которых уже известны, и подставляет полученные значения в окружающий код. В процессе исчезают применения функций, проекции и шаги итерации, которые больше не зависят от входных данных, а в остаточной программе видно, какие неизвестные переменные блокируют дальнейшее вычисление.

Сравнение программ ставит другую задачу. Одна программа может быть записана до или после β-редукции, вычисления проекций и итератора либо η-развёртки, поэтому две её записи не обязаны посимвольно совпадать. Хотелось бы приводить такие записи к одной канонической форме и сравнивать уже результаты нормализации.

Дальше мы будем рассматривать упрощённый вариант System T, используемый в формализации. Напомним, что мы уже работаем в языке без типов сумм; кроме того, в дальнейшем мы не используем η-правила для базовых типов $\mathbb{N}$, $\mathbb{B}$ и $\mathbb{1}$. Свободная переменная базового типа не вычисляется и сохраняется в нормальной форме; переменные и заблокированные на них вычисления мы

[15] Это свойство называется *каноничностью*.

будем называть *нейтральными термами*, или коротко *нейтралями*. Другими словами, η-развёртка выполняется только для функций и пар, а приведённое в первом разделе правило уникальности для булевых значений не входит в набор преобразований, проверяемый нормализатором. Можно построить варианты NbE и для расширенных вариантов равенства, но у них будут другие нормальные формы и другой контракт алгоритма.

Будем писать $t \equiv u$ и говорить, что термы *равны по определению* (definitionally equal),[16] если один можно преобразовать в другой с помощью β-правила для функций, правил вычисления проекций, уравнений iterate и if, η-правил для функций и пар, а также обратных преобразований и замен внутри составных термов.

Почему бы не сравнивать программы непосредственно по их поведению? Для языка с общей рекурсией точная семантическая эквивалентность программ неразрешима. Равенство по определению служит разрешимым компромиссом: в System T каждый терм имеет нормальную форму, и нормализатор даёт

$$t \equiv u \iff \mathrm{norm}(t) = \mathrm{norm}(u).$$

Справа стоит обычное синтаксическое совпадение нормальных форм. Значит, равенство по определению разрешается сравнением нормальных форм. Так нормализация используется для проверки равенства по определению в зависимо типизированных языках, где типы могут содержать программы (об этом подробно в лекции о зависимых типах).

Заметим, что таким образом решается только выбранное *равенство по определению*, а не полная эквивалентность. Например, рассмотренные выше термы $x + 2$ и $2 + x$ равны по теореме о коммутативности сложения, доказательство которой требует индукции. Однако их нормальные формы различны: $\mathrm{S}\,\mathrm{S}\,x$ и $\mathsf{iterate}(x)(\lambda k\, r.\, \mathrm{S}\,r)\,2$. Нормальная форма также не обязана сохранять структуру стоимости и общих подвыражений: β-редукция и η-развёртка могут дублировать синтаксис и уничтожать информацию о переиспользуемых подвыражениях.

Проговорим еще раз, что именно реализует NbE. Тотальность $\mathrm{norm}$ даёт каждому типизированному терму вычислимую каноническую форму, а приведённый выше критерий позволяет разрешать равенство по определению. Нормализатор называется *полным* (full), поскольку нормализует в том числе под λ, в отличие от вычислителя, останавливающегося на слабой головной нормальной форме (WHNF), то есть вычисляющего до первого внешнего конструктора или λ и не заходящего внутрь.

*Сильная нормализация* утверждает больше: любая последовательность редукций должна быть конечной, независимо от того, какое применимое правило и в каком месте терма выбирают на каждом шаге. Тотальный алгоритм $\mathrm{norm}$ сам по себе этого не доказывает. В формализации нет малошагового (small-step) отношения редукции, о последовательностях которого можно было бы сформулировать такую теорему. Вместо этого для функции $\mathrm{norm}$ доказаны здравость (soundness) и полнота (completeness), сформулированные ниже в «Контракте алгоритма».

[16] Также говорят о *конверсии* между такими термами.

### Спецификация выхода: нормальные и нейтральные формы

Выход нормализатора описывается двумя взаимно определёнными грамматиками:

$v ::= 0 \mid \mathrm{S}\, v \mid \mathtt{tt} \mid \mathtt{ff} \mid () \mid \lambda x.\, v \mid \langle v, v \rangle \mid u$ ($u$ входит в $v$ только при базовом типе)

$u ::= x \mid u\, v \mid \mathsf{iterate}(u)\, v\, v \mid \mathsf{if}(u)\, v\, v \mid u_{\mathrm{fst}} \mid u_{\mathrm{snd}}$

Нормальная форма $v$ не содержит применений вида $(\lambda x.\, t)\, u$, к которым можно применить β-правило; все известные рекурсоры и условные конструкции вычислены, а заблокированные элиминации сохранены явно в виде нейтральных термов $u$, то есть цепочек элиминаций, упирающихся в переменную. Нейтрали входят в нормальные формы *только при базовом типе*: нейтраль функционального типа η-разворачивается до λ, а парного до пары. В полученных η-длинных формах каждая функция записана как λ-абстракция, а каждая пара как $\langle \dots, \dots \rangle$; такие формы служат каноническими представителями βη-равенства.

### Идея алгоритма

Для получения нормальной формы можно последовательно применять β-правило, правила проекций и уравнения iterate и if во всём терме, включая подтермы под λ, пока ни одно правило больше не применимо. Такая «сильная редукция» в духе Grégoire and Leroy (2002) работает, но каждый β-шаг требует обхода тела функции, сдвига индексов де Брёйна или переименования связанных переменных и иногда копирования подставляемого аргумента. Повторные обходы усложняют реализацию и создают промежуточный синтаксис. Кроме того, этот способ не выполняет η-развёртку: нейтральную функцию $f$ он оставит как $f$, тогда как каноническим представителем при βη-равенстве является η-длинная форма $\lambda x.\, f\, x$. NbE использует другую схему: «сначала проинтерпретируй, потом прочитай обратно».

$$\text{синтаксис} \xrightarrow{\text{eval}} \text{семантическое значение} \xrightarrow{\text{reify}} \text{каноническая форма}$$

Идея его восходит к семантическому доказательству Мартин-Лёфа о том, что каждый терм его предикативной интуиционистской теории зависимых типов имеет нормальную форму (Martin-Löf, 1975). Бергер и Швихтенберг превратили её в самостоятельный алгоритм под названием «обращение функционала вычисления» (Berger and Schwichtenberg, 1991). Позже Данви переоткрыл ту же схему как type-directed partial evaluation (TDPE) (Danvy, 1996), а Катарина Кокан объяснила её через связь синтаксиса с его моделью (Coquand, 2002).

Каждому типу объектного языка сопоставляется его семантическая интерпретация в метаязыке, в нашем случае в самом Rocq (либо OCaml, если рассматривать экстрагированную в него имплементацию):

$$[\![B]\!]_\Gamma = \mathrm{nf}\ \Gamma\ B \qquad \text{(нормальные формы базового типа } B \text{ в контексте } \Gamma)$$

$$[\![A \to B]\!]_\Gamma = \forall \Delta.\ \mathrm{ope}\ \Delta\ \Gamma \to [\![A]\!]_\Delta \to [\![B]\!]_\Delta \qquad \text{(метаязыковые функции)}$$

$$[\![A \times B]\!]_\Gamma = [\![A]\!]_\Gamma \times [\![B]\!]_\Gamma \qquad \text{(метаязыковые пары)}$$

Окружение $\rho : \mathrm{Env}\ \Delta\ \Gamma$ сопоставляет каждой переменной $x : T$ из контекста $\Gamma$ семантическое значение из $[\![T]\!]_\Delta$. Два индекса играют разные роли: $\Gamma$ представляет исходный контекст терма, переменные которого нужно оценить, а $\Delta$ представляет

контекст, в котором живут полученные семантические значения. В тождественном окружении эти контексты совпадают. Различать их необходимо при переходе в расширенный контекст, например под λ. Вычислитель $\mathrm{eval}$ берёт значения свободных переменных из окружения, а при вычислении тела λ расширяет окружение семантическим значением аргумента.

Важно различать два обозначения: $[\![T]\!]_\Delta$ задаёт семантическую область для *типа* $T$, а $\mathrm{eval}$ интерпретирует *терм* в окружении и возвращает элемент этой области. Формально,

$$\mathrm{eval} : \mathrm{tm}\ \Gamma\ T \to \mathrm{Env}\ \Delta\ \Gamma \to [\![T]\!]_\Delta.$$

Таким образом, $\mathrm{eval}(t)(\rho)$ представляет собой семантическое значение терма $t$ при значениях свободных переменных из $\rho$. Если бы для термов использовалась запись $[\![t]\!]_\rho$, она обозначала бы то же самое; ниже вместо неё везде пишется $\mathrm{eval}(t)(\rho)$.

Вместе эти определения задают денотационную семантику: типы переходят в семантические области, контексты в окружения, а термы в значения, зависящие от окружения. В случае функций значение дополнительно индексировано контекстом и квантифицировано по $\mathrm{ope}\ \Delta\ \Gamma$. OPE (order-preserving embedding) представляет собой конструктивное доказательство того, что Γ вкладывается в Δ с сохранением порядка (другими словами, Δ получен из Γ только добавлением переменных).

Если для наглядности обозначить позиции контекста буквами, вложение может выглядеть так:

$$\begin{array}{rccccc} \Gamma : [\, a & & c & & e\, ] \\ & \downarrow & & \downarrow & & \downarrow \\ \Delta : [\, a & b & c & d & e\, ] \end{array}$$

OPE сохраняет порядок старых позиций $a, c, e$ и пропускает добавленные позиции $b, d$.

Другими словами, мы требуем, чтобы семантическая функция работала и во всех расширениях своего контекста. Это нужно при чтении под λ: там появляется свежая переменная, контекст расширяется и уже вычисленные значения требуется перенести в новый контекст. В терминах теории категорий такие зависящие от контекста значения образуют предпучки на категории контекстов.

Обратное чтение задаётся парой функций, определяемых взаимной рекурсией *по типу*:

- $\mathrm{reify}_T : [\![T]\!]_\Gamma \to \mathrm{nf}\ \Gamma\ T$ превращает семантическое значение в каноническую нормальную форму;
- $\mathrm{reflect}_T : \mathrm{ne}\ \Gamma\ T \to [\![T]\!]_\Gamma$ наделяет нейтральный терм (например, переменную) семантическим значением.

Для базового типа обе функции являются тождественными вложениями. Для функций определяем:

$$\mathrm{reify}_{A\to B}(f) = \lambda x.\, \mathrm{reify}_B(f(\mathrm{reflect}_A(x))), \qquad \mathrm{reflect}_{A\to B}(u)(a) = \mathrm{reflect}_B(u\ (\mathrm{reify}_A(a))).$$

Чтобы прочитать семантическую функцию, мы вводим свежую переменную $x$, *отражаем* её в семантическое значение, применяем функцию и рекурсивно читаем результат. Так алгоритм нормализует под λ и выполняет η-развёртку. Отражение нейтральной функции действует в обратную сторону: её применение

остаётся нейтральным термом, а семантический аргумент сначала читается обратно в синтаксис.

По устройству $\mathrm{eval}$ похож на интерпретатор больших шагов (big-step) с окружениями, но возвращает не обычные операционные значения, а значения семантического домена NbE: функции и пары метаязыка, а для базовых типов нормальные и нейтральные формы. Чтобы получить синтаксический результат, после $\mathrm{eval}$ применяется $\mathrm{reify}$. При вычислении $\lambda$ переходит в метаязыковую функцию, применение в применение, а для итератора $\mathrm{eval}$ сначала вычисляет его натуральный аргумент. Поскольку $[\![\mathbb{N}]\!]_\Gamma = \mathrm{nf}\ \Gamma\ \mathbb{N}$, полученное значение имеет вид $0$, $\mathrm{S}\ n$ либо является нейтральным. В первых двух случаях вычислитель применяет соответствующее уравнение $\mathsf{iterate}$. В последнем он строит нейтральный $\mathsf{iterate}$, у которого шаг и база уже прочитаны обратно.

Нормализатор задаётся композицией reify и eval:

$$\mathrm{norm}(t) = \mathrm{reify}(\mathrm{eval}(t)(\rho_{\mathrm{id}})), \qquad \rho_{\mathrm{id}}(x) = \mathrm{reflect}(x).$$

Терм вычисляется в окружении, где каждая свободная переменная отражена в саму себя. Подстановку и β-редукцию выполняет метаязык, а синтаксис строится заново при обратном чтении. Тотальность конструкции опирается на тотальность System T: с общей рекурсией $\mathrm{eval}$ мог бы зависнуть, в том числе внутри $\mathrm{reify}$ при применении семантической функции к свежей переменной.

Примеры из формализации (`Tests.v`):

- замкнутое вычисление: $\mathrm{norm}(2+2) = 4$;
- η-развёртка: $\mathrm{norm}(f) = \lambda x.\, f\,x$ при свободной $f : \mathbb{N} \to \mathbb{N}$, и $\mathrm{norm}(p) = \langle p_{\mathrm{fst}}, p_{\mathrm{snd}} \rangle$ при свободной $p : A \times B$;
- заблокированная рекурсия: $\mathrm{norm}((1+1)+x) = \mathsf{iterate}(x)(\lambda k\, r.\, \mathrm{S}\, r)\, 2$; известная часть вычислена, нейтральный остаток сохранён.

### Контракт алгоритма

Метатеория формализации доказывает следующие утверждения (мы принимаем их без доказательства; используется техника логических отношений, о которой пойдёт речь в следующих лекциях):

- *здравость*: $\mathrm{norm}(t) \equiv t$, то есть нормализация не меняет программу с точностью до равенства по определению;
- *полнота*: если $t \equiv u$, то $\mathrm{norm}(t)$ и $\mathrm{norm}(u)$ синтаксически совпадают.

Следовательно, $t \equiv u \iff \mathrm{norm}(t) = \mathrm{norm}(u)$: равенство по определению разрешимо, нормальные формы единственны, а каждый замкнутый терм базового типа равен конструкторному значению.

Схема «вычислить в семантике, затем прочитать обратно» применяется и за пределами этого нормализатора. Вслед за перечнем Линдли эти применения можно сгруппировать в две области (Lindley, 2016). В программировании это встраиваемые DSL, частичные вычисления и эффективная β-редукция с помощью замыканий метаязыка. В исследованиях по основаниям логики и языков это исполнимая денотационная семантика, нормализация доказательств и проверка равенства по определению в теориях типов. Реализации NbE используются в тайпчекерах зависимо типизированных языков и в Isabelle/HOL, где команда `value [nbe]` вычисляет термы методом NbE. Ещё один пример даёт конфигурационный язык Dhall. Работы Киселёва с соавторами показывают, как частичное вычисление и схема, вдохновлённая NbE, применяются для оптимизаций встраиваемых DSL (Carette et al., 2009; Kiselyov, 2014).

## 3.4. Реализуемость и интерпретация «Диалектика»

Исторически Диалектика возникла из программы Гильберта. Она требовала доказать непротиворечивость классической арифметики Пеано (PA) «финитными» средствами: элементарными рассуждениями о числах, формулах и доказательствах как конечных последовательностях символов, без обращения к завершённой бесконечности. Надо сказать, точной формальной границы этого понятия Гильберт не задал.

После теорем Гёделя о неполноте эту финитную точку зрения пришлось расширить. Гёдель предложил считать конструктивными объектами не только числовые вычисления, но и *примитивно-рекурсивные функционалы конечных типов*, образующие System T. Здесь «конечный» описывает построение типа из $\mathbb{N}$ с помощью стрелок, а не число его значений: $\mathbb{N} \to \mathbb{N}$ тоже является конечным типом. В 1941 году Гёдель изложил функциональную интерпретацию арифметики Гейтинга (HA), а в 1958 году опубликовал её в журнале *Dialectica* (Gödel, 1958) (откуда и название). HA имеет тот же язык, арифметические аксиомы и индукцию, что и PA, но использует интуиционистскую, а не классическую логику; добавление закона исключённого третьего превращает HA в PA. Поэтому классические доказательства PA сначала переводятся в HA с помощью перевода двойного отрицания,[17] а затем интерпретируются Диалектикой. Извлечение программ из доказательств представляет собой более позднее прочтение той же конструкции; современный обзор дан в Avigad and Feferman (1998).

Ниже рассматривается упрощённый вариант интерпретации, использованный для формализации. Формулы в нём строятся из связок $\wedge$, $\vee$, $\supset$, кванторов по натуральным числам и разрешимых атомов. Каждый атом уже задан булевой программой System T; так можно закодировать равенство, порядок и другие примитивно-рекурсивные предикаты. Это позволяет сосредоточиться на том, как Диалектика преобразует логическую структуру формулы.

В более буквальной формализации HA можно было бы "честно" задать синтаксис арифметических термов и атомарных формул, а затем дополнительно компилировать их в булевы программы System T. Такой вариант добавил бы рутинный слой трансляции и технические леммы о подстановке и переводе выводов, но не изменил бы самой сути компилятора Диалектики.

### Свидетели против контрпримеров

Каждой формуле $A$ интерпретация сопоставляет два типа System T и бесквантерную *матрицу*:

$$A \;\rightsquigarrow\; \exists w : W(A).\ \forall c : C(A).\ |A|(w, c),$$

где $|A|(w, c)$ представляет собой разрешимый тест. Удобно читать это как игру: $W(A)$ задаёт тип *свидетелей* (witness, ходов защитника), $C(A)$ задаёт тип *контрпримеров* (counter, ходов оппонента), и свидетель $w$ «побеждает» контрпример $c$, если $|A|(w, c)$ истинно. Типы определяются рекурсией по формуле:

| $A$ | $W(A)$ | $C(A)$ |
|---|---|---|
| атом | $\mathbb{1}$ | $\mathbb{1}$ |
| $X \wedge Y$ | $W(X) \times W(Y)$ | $C(X) \times C(Y)$ |
| $X \vee Y$ | $\mathbb{B} \times (W(X) \times W(Y))$ | $C(X) \times C(Y)$ |
| $X \supset Y$ | $(W(X) \to W(Y)) \times (W(X) \to C(Y) \to C(X))$ | $W(X) \times C(Y)$ |
| $\forall x.\, X$ | $\mathbb{N} \to W(X)$ | $\mathbb{N} \times C(X)$ |
| $\exists x.\, X$ | $\mathbb{N} \times W(X)$ | $C(X)$ |

[17] Он систематически добавляет двойные отрицания и тем самым переводит классические доказательства в интуиционистские.

Единичные типы в строке для атома не означают, что атом автоматически истинен. Они означают только, что для него не извлекаются отдельные свидетель и контрпример: вся содержательная проверка уже находится в самом булевом атоме. Если атом задан программой $p : \mathbb{B}$, то

$$|p|((),()) = p.$$

Для конъюнкции $X \wedge Y$ свидетель $\langle w_X, w_Y \rangle$ содержит свидетелей обеих частей, а контрпример $\langle c_X, c_Y \rangle$ содержит по вызову для каждой из них. Матрица требует, чтобы $w_X$ победил $c_X$, а $w_Y$ победил $c_Y$.

Для дизъюнкции $X \vee Y$ свидетель имеет вид $\langle b, \langle w_X, w_Y \rangle \rangle$. Булев флаг $b$ выбирает защищаемую часть: $\mathtt{tt}$ выбирает $X$, а $\mathtt{ff}$ выбирает $Y$. Тип хранит кандидатов для обеих частей, потому что в этой кодировке нет типа суммы; неиспользуемый компонент можно заполнить стандартным значением соответствующего типа. Каждый тип рассматриваемого варианта System T обитаем, поэтому такой заполнитель всегда существует. Контрпример $\langle c_X, c_Y \rangle$ предлагает вызов для каждой части, а матрица по флагу выбирает только $c_X$ или только $c_Y$.

Свидетель импликации $X \supset Y$ представляет собой *пару функций* $\langle f, f^{\leftarrow} \rangle$. Обе функции входят в свидетель: прямая компонента $f$ превращает свидетель $X$ в свидетель $Y$, а обратная компонента $f^{\leftarrow}$ превращает контрпример к $Y$ в контрпример к $X$, используя свидетель $X$. Сама обратная компонента контрпримером не является. Контрпример к импликации имеет вид $\langle w_X, c_Y \rangle$: он предъявляет свидетель посылки и вызов заключению, который $f^{\leftarrow}$ переносит назад к посылке. Та же структура прямого и обратного прохода встречается в обратном дифференцировании и связана с линейной логикой.

Для читателя, знакомого с линзами из функционального программирования, эта пара имеет тип *биморфной линзы* из $(W(X), C(X))$ в $(W(Y), C(Y))$. Линза объединяет прямое отображение $f : W(X) \to W(Y)$ с обратным отображением $f^{\leftarrow} : W(X) \to C(Y) \to C(X)$ (Hedges, 2018). Правило композиции импликаций компилируется в композицию таких линз:

$$\langle g, g^{\leftarrow} \rangle \circ \langle f, f^{\leftarrow} \rangle = \langle \lambda w.\, g(f(w)),\ \lambda w\, c.\, f^{\leftarrow}(w)(g^{\leftarrow}(f(w))(c)) \rangle.$$

Здесь используется только тип и композиция линз; обычные законы линз не утверждаются.

Типы кванторов также заранее задают форму программы. Например,

$$W(\forall x.\, \exists y.\, y = \mathrm{S}\, x) \ = \ \mathbb{N} \to (\mathbb{N} \times \mathbb{1})$$

Поэтому свидетель этой теоремы обязан быть программой, вычисляющей по $x$ значение $y$; тип уже предсказывает форму извлечённой программы.

Арифметические термы внутри формул уже являются термами System T, поэтому трансляция меняет логическую структуру, но не арифметику. Матрица $|A|$ компилируется структурной рекурсией в программу System T

$$\mathtt{diaT}\ A : W(A) \to C(A) \to \mathbb{B},$$

например $|X \wedge Y|(\langle w_X, w_Y \rangle, \langle c_X, c_Y \rangle) = |X|(w_X, c_X)\ \&\&\ |Y|(w_Y, c_Y)$, а $|\exists x.\, X|(\langle n, w \rangle, c) = |X[n/x]|(w, c)$. Матрица исполняется во время вычислений: извлечённые реализаторы вызывают её для выбора контрпримеров.

**Компиляция доказательств**

До сих пор мы описывали только трансляцию формул: формуле $A$ сопоставлялись типы $W(A)$ и $C(A)$ и матрица $|A|$. Чтобы извлечь программу, нужно также перевести доказательство этой формулы.

В описанной раньше натуральной дедукции суждение $\Gamma \vdash A$ содержит явный список логических предпосылок $\Gamma$, которыми правила оперируют непосредственно. В гильбертовском исчислении списка предпосылок нет: доказательство собирается из схем аксиом с помощью modus ponens и нескольких других правил. Поэтому в типе `proof Γ A`, кодирующем гильбертов вывод, тот же символ $\Gamma$ обозначает только контекст свободных числовых переменных, а не логические гипотезы.

Для программиста гильбертовское доказательство похоже на программу для стекового языка: схема аксиом кладёт готовое доказательство на стек, а modus ponens забирает доказательства $A \supset B$ и $A$ и возвращает доказательство $B$. Натуральная дедукция, напротив, ближе к λ-исчислению: предпосылки играют роль переменных, а введение импликации соответствует λ-абстракции.

Оба представления могут задавать одну и ту же логику. Гильбертовский вариант выбран здесь ради более простой формализации компилятора: каждой схеме аксиом и каждому правилу соответствует отдельный случай трансляции, а для списков гипотез не приходится доказывать технические леммы о перестановке предпосылок и подстановке одного вывода в другой.

Интерпретация доказательств задаётся компилятором

$$\texttt{wit} : \mathrm{proof}\ \Gamma\ A \to \mathrm{tm}\ \Gamma\ (W(A))$$

Компилятор `wit` строит только свидетель типа $W(A)$. Контрпример типа $C(A)$ он не извлекает: тот поступает извне как вызов, с которым свидетель проверяется матрицей $|A|$. В частности, обратные компоненты для импликаций входят в построенный свидетель и преобразуют контрпримеры, но не являются отдельным результатом `wit`.

Компилятор определяется рекурсией по выводу гильбертовского исчисления HA с индукцией. Внутренняя типизация гарантирует, что результат `wit` имеет тип $W(A)$. Например:

- *modus ponens*: из свидетеля $u$ импликации берётся его прямая компонента и применяется к свидетелю посылки, $\texttt{wit} = u_{\mathrm{fst}}(a)$;
- *введение* $\exists$: терм $t$, использованный в правиле, упаковывается в пару $\langle t, w\rangle$. Свидетель существования не «синтезируется» интерпретацией, а извлекается из доказательства, где он уже содержался;
- *сжатие* (contraction) $P \supset P \wedge P$: прямая компонента свидетеля импликации дублирует свидетель $P$, а обратная компонента получает *два* контрпримера и должна вернуть один. Извлечённая программа запускает матрицу и выбирает тот контрпример, который действительно опровергает:

  $$\langle\, \lambda w.\, \langle w, w\rangle,\quad \lambda w\, \langle c_1, c_2\rangle.\, \mathsf{if}(|P|(w, c_1))\, c_2\, c_1\, \rangle.$$

  Так логическая структура превращается в поток управления времени исполнения; выбор возможен потому, что матрица разрешима.

Правило индукции представлено аксиомой

$$(Q(0) \wedge \forall k.\, (Q(k) \supset Q(\mathrm{S}\, k))) \supset \forall n.\, Q(n).$$

Её свидетель состоит из пары примитивных рекурсий. Прямая компонента строит свидетеля $Q(n)$ снизу вверх: $w_0$ служит базовым свидетелем, $s_k$ служит

свидетелем шага $Q(k) \supset Q(\mathrm{S}\,k)$, а $w_{k+1} = (s_k)_{\mathrm{fst}}(w_k)$. Обратная компонента того же свидетеля преобразует контрпримеры, выполняя гёделевский *поиск контрпримера*: получив вызов против $Q(n)$, она протаскивает его назад сквозь свидетелей шагов, на каждой ступени запуская матрицу $|Q(k)|$, и возвращает первую ступень, которую можно «обвинить», то есть контрпример базе или контрпример конкретному шагу. Каждая компонента использует по одному iterate, поэтому индукция и при исполнении остаётся примитивной рекурсией.

Заметим, что интерпретация дополнительно реализует два интуиционистски *не выводимых* принципа. Здесь $\neg A$ является сокращением для $A \supset \mathsf{False}$, где False есть атом, заданный булевой программой `ff`. Бесквантворный принцип Маркова и независимость посылки для $\forall$-посылок имеют вид

$$\begin{aligned} &\mathrm{MP}: \quad (\neg\forall x.\,\neg P(x)) \supset \exists x.\,P(x),\\ &\mathrm{IP}_{\forall}: \quad ((\forall x.\,P(x)) \supset \exists y.\,Q(y)) \supset \exists y.\,((\forall x.\,P(x)) \supset Q(y)). \end{aligned}$$

В MP предикат $P$ берётся из предусмотренного формализацией разрешимого бесквантворного фрагмента. В формализации оба принципа добавлены как аксиомы с реализаторами, так что Диалектика «конструктивизирует» и эти расширения интуиционистской логики.

Чтобы сформулировать валидность извлечённого свидетеля, мы строим более простую модель System T, чем модель NbE. Она интерпретирует $\mathbb{N}$, $\mathbb{B}$ и $\mathbb{1}$ обычными типами значений метаязыка, а функции и пары метаязыковыми функциями и парами. Здесь не нужны нейтральные значения, OPE и обратное чтение, поскольку модель не строит нормальную форму, а только вычисляет значение терма.

Окружение $\rho : \mathtt{cxtden}\ \Gamma$ этой модели сопоставляет свободным переменным числа, булевы значения, функции и пары, а $\mathtt{tmden}(t, \rho)$ вычисляет значение терма $t$. Относительно этой модели предикат $\mathtt{valid}(A, t)$ означает, что программа $t : W(A)$ при любых значениях свободных переменных побеждает любой контрпример. Функция `dia` вычисляет матрицу, поэтому содержательно определение имеет вид

$$\mathtt{valid}(A, t) \quad \Longleftrightarrow \quad \forall \rho.\ \forall c.\ \mathtt{dia}(A, \rho, \mathtt{tmden}(t, \rho), c) = \mathtt{tt}.$$

Общая теорема здравости soundness утверждает, что для всякого вывода $d$ формулы $A$ реализатор $\mathtt{wit}(d)$ побеждает любой контрпример в смысле предиката `valid`.

Полнота здесь не утверждается: из существования программы $t : W(A)$, для которой выполнено $\mathtt{valid}(A, t)$, в общем случае нельзя восстановить доказательство формулы $A$ в НА. Диалектика служит интерпретацией доказательств и методом извлечения программ, а не критерием доказуемости.

### Канонические реализаторы

Два алгоритма (NbE и Диалектика) образуют цепочку:

$$\text{вывод } d \text{ формулы } A \xrightarrow{\ \mathrm{wit}\ } \text{программа } \mathrm{wit}(d) : W(A) \xrightarrow{\ \mathrm{norm}\ } \text{каноническая программа}$$

Первая стрелка реализует Гёделевскую идею: программу $\mathrm{wit}(d)$ уже можно исполнять. Отдельный этап $\mathrm{norm}$ приводит её к канонической форме. Алгоритмы нормализации программ и трансляции доказательств возникли из разных задач, но здесь работают с одним синтаксисом System T.

Композиция двух алгоритмов записывается тривиально:

```
Definition realizer {Γ} {A : prp Γ} (d : proof Γ A) : nf Γ (W A) :=
  norm (wit d).
```

Заметим также, что для замкнутой формулы окружение в определении `valid` отсутствует, а здравость можно сформулировать синтаксически: $\mathrm{norm}(\texttt{diaT}\ A \cdot \texttt{wit}(d) \cdot c) = \texttt{tt}$ для любого замкнутого $c$.

До запуска NbE компилятор `wit` выдаёт обычные исполнимые термы System T. Нормализация устраняет в них промежуточные конструкции и строит каноническую βη-длинную форму. Для свидетелей высших типов такая форма показывает функциональную структуру, которую нельзя свести к одному числовому ответу. Атомарные равенства в типах свидетелей стираются до единичного типа $\mathbb{1}$ и представлены значением $()$; само равенство при этом проверяется булевой матрицей. Примеры из формализации (`Examples.v`; через $r_R$ обозначен фиксированный реализатор правила $R$ гильбертовского исчисления):

- $\texttt{realizer}(\vdash \exists y.\, y = 2\,) = \langle 2, () \rangle$;
- для доказательства `ex_succ` формулы $\vdash \forall x.\, \exists y.\, y = \mathrm{S}\, x$ компилятор сначала выдаёт

  $$\begin{aligned}\texttt{wit}(\texttt{ex\_succ}) = r_{\mathrm{mp}}(r_{\forall I}(r_{\mathrm{mp}}(r_{\mathrm{cur}}(r_{\wedge E_1}),\\ r_{\mathrm{mp}}(r_{\exists I}(\mathrm{S}\, x), ()))),\\ ()).\end{aligned}$$

  Композиция правил оставляет здесь вспомогательные применения λ-функций. После NbE получается $\texttt{realizer}(\texttt{ex\_succ}) = \lambda x.\, \langle \mathrm{S}\, x, () \rangle$: по $x$ программа вычисляет свидетеля $y = \mathrm{S}\, x$;
- для аксиомы сжатия $P \supset P \wedge P$ стратегия заметна уже в результате компиляции:

  $$\begin{aligned}\texttt{wit}(\texttt{ax\_and\_contr}) &= r_{\wedge C}\\ &= \big\langle \lambda w.\, \langle w, w \rangle,\\ &\quad \lambda w\, \langle c_1, c_2 \rangle.\\ &\qquad \mathsf{if}(\texttt{diaT}\ P \cdot w \cdot c_1)\, c_2\, c_1 \big\rangle.\end{aligned}$$

  Здесь NbE не меняет содержательную структуру программы, а лишь раскрывает скомпилированную матрицу $\texttt{diaT}\ P$ и приводит терм к канонической паре

  $$\langle\, \lambda w.\, \langle w, w \rangle,\ \lambda w\, \langle c_1, c_2 \rangle.\, \mathsf{if}(|P|(w, c_1))\, c_2\, c_1\, \rangle,$$

  где первая функция дублирует свидетель, а вторая запускает матрицу $|P|$ и направляет назад тот контрпример, который может опровергнуть $P$;
- доказательство $\vdash \forall x.\, 0 + x = x$ индукцией даёт $\lambda n.\, \mathsf{iterate}(n)(\lambda\_\_.\, ())\, ()$, скелет индукции со стёртым до единиц содержанием. При определении сложения с рекурсией по второму аргументу это не равенство по определению. Поскольку равенство является атомом, извлечённый свидетель возвращает только $()$. Арифметическое условие $0 + x = x$ остаётся в булевой матрице, а `iterate` сохраняет структуру доказательства индукцией;
- доказательство $\vdash \exists y.\, y = 0$ *через принцип Маркова* (нарочито неконструктивное: из $\neg\forall y.\, \neg(y = 0)$) всё равно нормализуется в $\langle 0, () \rangle$. Нормализация прогоняет контрпримерную машинерию негативного доказательства и добывает конкретного свидетеля.

Итак, System T служит тотальным целевым языком, Диалектика компилирует в него логические выводы, а NbE при необходимости приводит полученные программы к каноническому виду.

У трансляции Гёделя есть варианты Диллера–Нама, Крайзеля и Стейна, а также монотонная интерпретация Коленбаха. Последняя лежит в основе proof

mining, то есть систематического извлечения количественных оценок из неконструктивных доказательств в математическом анализе (Kohlenbach, 2008).