Язык M Power Query. Система типов. Основы

Система типов Power Query помогает понять значения, с которыми мы работаем, описать типы данных, ожидаемых от пользовательских функций, дает доступ к документации (которую может отображать IntelliSense) и предоставляет механизм информирования пользователей о структуре данных, которые могут отображаться в хост-среде, например, для настройки типов столбцов.[1]

Подход M к типам на макроуровне заключается в следующем:

- Каждое значение имеет свой тип.
- Тип сам по себе является значением.
- Типы используются для классификации значений.

Предыдущая заметка Следующая заметка

Основы системы типов

Каждое значение имеет свой тип

1 // вычисляется значение типа number

"Hello World!" // вычисляется значение типа text

true // вычисляется логическое значение типа logical

null // вычисляет значение null

Text.From(123) // возвращает значение "123", которое имеет тип text

Number.From("123") // возвращает значение 123, которое имеет тип number

Тип значения возвращается функцией *Value.Type*:

= Value.Type(1) // возвращает "number"

Листинг 1[2]

```
let
   Data = #date(2022, 1, 31),
   Type = Value.Type(Data)
in
   Type // возвращает "date"
```

Тип сам по себе является значением

Результат функции *Value.Type* является значением – значением типа. Значение типа аналогично другим значениям: оно может быть сохранено в переменной, обработано функциями, к нему можно применить операторы. Такой тип называется *type*.

Value.Type("abc") // возвращает значение типа *text*

Value.Type(Value.Type("abc")) // возвращает значение типа *type*

Ряд библиотечных функций возвращают значения типа. Эти значения также могут быть созданы с использованием синтаксиса языка. Хотя мы узнаем больше об этом синтаксисе позже, для начала приведем несколько примеров:

type number // вычисляет значение типа *type* (значение типа), которое содержит тип *number*

type null // вычисляет значение типа *type*, содержащее тип *null*

Листинг 2

```
let
   NumberType = type number
in
   NumberType // создает значение типа, содержащее тип number
```

Если значение типа отображается в интерфейсе, вы увидите краткое описание типа:

---

[1] Заметка написана на основе статьи Ben Gribaudo. Power Query M Primer (Part 16): Type System I – Basics. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

[2] Номер листинга соответствует номеру запроса в приложенном Excel файле.

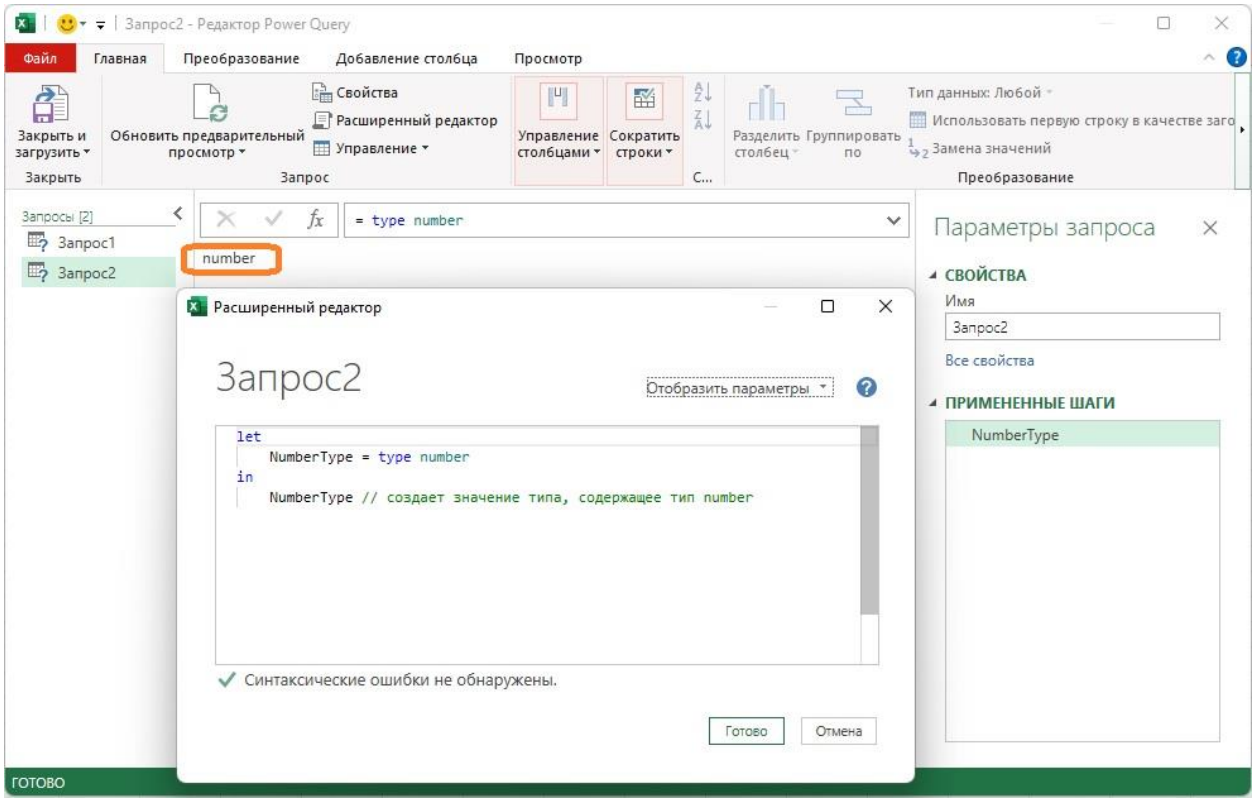

Рис. 1. Результат – значение с типом *type*

Это лишь текстовое представление результата. Значения типа не является строкой. Интерфейс не гарантирует, что будут отображаться все сведения о типе. Для доступа к полному набору информации о значении типа необходимо использовать библиотечные функции.

Типы используются для классификации значений

Классификация означает описание. Например, использование типа таблицы предполагает получение информации о содержащихся в ней столбцах. Классификация также подразумевает группировку по категориям. Например, механизм запроса классифицирует значение, переданное в качестве параметра функции, как приемлемое или неприемлемое, в зависимости от того, соответствует ли тип значения ожидаемому типу параметра. Если тип значения является приемлемым – значение разрешено передавать. Если тип значения является неприемлемым – выдается сообщение об ошибке.

В следующей функции определяются как тип аргумента, так и тип возвращаемого значения. Движок M будет обеспечивать выполнение этих ожиданий, требуя, чтобы передаваемые и возвращаемые значения соответствовали указанным типам.

Листинг 3

```
(ages as list) as text => "Combined years of life: " & Text.From(List.Sum(ages))
```

На первый взгляд может показаться, что утверждения *as list* и *as text* предполагают, что значение параметра должно иметь тип *list*, а возвращаемое значение должно иметь тип *text*. Не совсем. Эти утверждения требуют, чтобы значения были *совместимы* с указанными типами. Существует разница между значением определенного типа и совместимостью с определенным типом. Проверки совместимости выполняются на уровне *обнуляемого примитивного типа* (nullable primitive type). Вскоре мы обсудим эти проверки, но прежде разберемся с семейством типов.

Обнуляемые примитивные типы

До сих пор в этой серии заметок мы говорили о следующих типах: *binary, date, datetime, datetimezone, duration, function, list, logical, null, number, record, table, text, time, type*. Все это обнуляемые примитивные типы, но они не составляют полный набор обнуляемых примитивных типов. Давайте познакомимся с остальными членами семьи.

Тип any

Все типы совместимы с типом *any*. Все значения совместимы с типом *any*. Вернемся к функции из листинга 3. Если вы определите параметр без указания типа, параметр по умолчанию будет иметь тип *any*. Аналогично, если вы опустите указание типа функции, он по умолчанию будет *any*. Следующие два кода эквивалентны:

(somevalue) => ...

(somevalue as any) as any => ...

Тип *any* – это абстрактный тип. Никакое значение никогда не может иметь абстрактный тип. Вы можете определить выражение, как имеющее абстрактный тип (например, TypeValue = type any), но вы никогда не увидите значение, содержимое которого имеет абстрактный тип. Ни одно выражение не может создать значение, тип которого равен *any*. Несмотря на то, что значение не может быть абстрактного типа, абстрактные типы все равно полезны, поскольку они предоставляют способ классификации значений. Тип *any* имеет свое место. Например, хорошее значение по умолчанию, указывающее, что разрешены значения всех типов.

nullable – тип, допускающий значение null

Что делать, если вы создаете функцию, которая должна обрабатывать значения определенного типа и *null*? Допустим, вы обрабатываете числа из столбца базы данных, которые могут иметь значение *null*. Функции типа…

(amount as number) => ...

…отлично работает для чисел из столбца, но не будет совместима с *null*. Вы могли бы вернуться к…

(amount as any) => ...

…которая допускает как числа, так и *null*, но также, к сожалению, допускает любые другие типы, чего бы вам хотелось избежать.

На выручку приходит тип, допускающий значение *null* – *nullable*. Возьмите любой тип, дополните его типом *nullable*, и будет возвращен тип, совместимый как с данным типом, так и с *null*.

type nullable text // эквивалентно определению: тип *text* или тип *null*

```
let
  TypeValue = type number
in
  type nullable TypeValue // возвращает тип nullable number
```

nullable можно добавить к определению типа параметров:

(amount as nullable number) => … // параметр принимает *number* и *null*

А вот саму функцию нельзя определить, как *nullable* (но иногда это можно обойти).

Обычно тип, возвращаемый *nullable*, представляет собой абстрактный тип *nullable T*, где *T* – это тип, с которого вы начали. Однако, если вы применяете *nullable* к типу, который уже совместим с *null*, наличие *nullable* игнорируется.

type nullable null // возвращает тип *null* (поскольку он уже совместим с *null*), а не тип *nullable null*

type nullable any // возвращает тип *any* (поскольку он уже совместим с *null*), а не тип *nullable any*

Чтобы превратить *nullable* тип в его ненулевой аналог, используйте библиотечную функцию:

Type.NonNullable(type nullable text) // возвращает тип *text*

Если указанный тип уже не допускает нулей, функция вернет тот же тип, что и в аргументе.

Type.NonNullable(type text) // возвращает тип *text*

Идея перехода от *nullable* типа к его ненулевому аналогу приводит нас к двум специальным типам. Выше мы ввели *nullable* вариации для обычных типов. Теперь обсудим типы, не допускающие нулевых значений.

Тип anynonnull

Тип *any* совместим со всеми типами, включая тип *null*. Каков его ненулевой аналог? *anynonnull*. Этот абстрактный тип работает так же, как следует из его названия: он совместим со всеми типами, за исключением тех, которые совместимы с *null*; любое значение, кроме *null*, совместимо с этим типом.

```
Type.NonNullable(type any) // возвращает тип anynonnull
// движение в другом направлении:
type nullable anynonnull // возвращает тип any
```

В следующем примере параметр функции может иметь любое значение, кроме *null*.

```
(input as anynonnull) => ... // функция, аргумент которой принимает любое значение, кроме null
```

Тип none

Что такое ненулевой аналог типа *null*? Представляем странный случай абстрактного типа *none*! Никакое значение никогда не может иметь тип *none* или быть совместимым с ним.

```
Type.NonNullable(type null) // возвращает тип none
// движение в другом направлении:
type nullable none // возвращает тип null
```

Сопряжение типов *null* и *none* является единственным исключением из правила *nullable*, в котором говорилось, что применение *nullable* возвращает тип, совместимый с данным и *null*. *nullable none* возвращает тип *null*, который несовместим с типом *none*. Тип *none* совместим только с самим собой, типом *any* (потому что все типы совместимы с типом *any*) и типом *anynonnull* (потому что все типы, не допускающие *null*, совместимы с *anynonnull*).

Если типы используются для описания и классификации значений, и ни одно значение никогда не может иметь тип *none*, в чем его смысл? Вспомните, что каждое выражение M должно, либо возвращать значение, либо вызывать ошибку. Выражение, которое всегда выдает ошибку, можно описать как возвращающее тип *none*, потому что оно никогда ничего не возвращает.

А зачем вам выражение, которое всегда выдает ошибки? Как насчет вспомогательной функции, которая выдает сообщения об ошибках?

```
(problem as text) as none => error Error.Record("Business Rule Violation", problem)
```

Функция определена с типом *none*. Если функция попытается вернуть значение (скажем, кто-то изменит код, но оставит *as none*), движок M выдаст ошибку, поскольку возвращаемое значение будет несовместимо с типом *none*.

Есть еще один случай, когда выражение можно рассматривать как возвращающее тип *none*. Что, если выражению требуется много времени, чтобы вернуть значение... очень много времени... бесконечно много времени? Технически выражение, которое никогда не возвращает значение, также может быть описано как возвращающее тип *none*. Зачем кому-то писать такое бессмысленное выражение – это другой вопрос ... но теоретически это можно сделать.

Внутренний секретный тип action

Наконец, существует специальный секретный (он же внутренний Microsoft) тип *action*, который нигде не упоминается в официальной спецификации языка и который практически не может быть использован с помощью инструментов Power Query, ориентированных на потребителя. Его описание выходит за рамки темы. Если вам интересно узнать о типе *action*, рекомендую M Mysteries: The Mysterious Type Action—An M-Internal Means to Write Data Modifications to External Systems.

Вся семья

Итак, полное семейство обнуляемых примитивных типов Power Query: *action, any, anynonnull, binary, date, datetime, datetimezone, duration, function, list, logical, none, null, number, record, table, text, time, type*. А еще конструкции *nullable + T* с возможностью обнуления каждого *T* (за исключением тех типом, которые само по себе совместимы с *null*). Из этого семейства типы *any, anynonnull, none* и *nullable T* являются абстрактными, так же как типы *table, record* и *function*.

Последние три, будучи абстрактными, могут вызвать удивление. Все таблицы совместимы с типом *table*, все записи с типом *record* и все функции с типом *function*. Но типы *table, record* и *function* не полностью описывают конкретную таблицу, запись или функцию. Для полного описания требуется пользовательский тип – производный тип, который содержит сведения о столбцах таблицы, полях записи или параметрах функции. Пользовательские типы иногда относятся и к спискам, хотя тип *list* не является абстрактным.

Теперь, когда вы познакомились со всем семейством обнуляемых примитивных типов, обсудим проверку совместимости.

Проверка совместимости

Если вы хотите определить, совместимо ли значение с типом, вы можете использовать *оператор совместимости типов* – ключевое слово *is*. Этот оператор возвращает значение *true* или *false*.

1 is number // true

1 is nullable number // true, потому что значение совместимо с типом *number*, либо *null*

1 is anynonnull // true, потому что значение не равно *null*

1 is any // true, "is any" возвращает значение true для любого значения

1 is none // false, всегда возвращает false потому что ни одно значение не совместимо с *none*

1 is null // false

#table({"Col1"}, {}) is table // true

#table({"Col1"}, {}) is list // false

#table({"Col1"}, {}) is date // false

SomeValue is date // ответ зависит от того, что содержит *someValue*

SomeValue is any /* true, потому что someValue должно содержать
какое-то значение, а все значения совместимы с any */

SomeValue is anynonnull // ответ зависит от того, содержит ли *someValue* значение *null*

SomeValue is none /* false, потому что переменные всегда
содержат значения, а никакое значение не совместимо с типом *none* */

Предположим, вы используете правила форматирования для отображения значений различных типов в тексте. Создайте пользовательскую функцию на основе *Text.From*, которая будет принимать значение и возвращать строку, отформатированную с использованием правила, выбранного на основе типа значения.

```
(input) =>
if input is date then Date.ToText(input, "MMMM d")
else if input is time then Time.ToText(input, "\h:h \m:m \s:s")
else Text.From(input)
```

В языке M также есть *оператор утверждения типа* – ключевое слово *as.* Проверяет, совместимо ли значение с заданным обнуляемым примитивным типом, и возвращает само значение, или ошибку.

2 as number // возвращает 2

3 as any // возвращает 3

1 as text // Expression.Error: Не удается преобразовать значение 1 в тип text

Утверждения типа в коде функции также определяются с использованием синтаксиса *as*, хотя это *as* немного отличается от оператора утверждения типа *as*. Значение проверяется на совместимость с типом примитива. Если совместимо, возвращается само значение (в случае оператора совместимости типов) или разрешается (в случае определения параметра функции). Если не совместимо, выдается сообщение об ошибке.

Как *оператор совместимости типов*, так и *оператор утверждения типа* требуют, чтобы имя типа было жестко запрограммировано. Если вы хотите использовать динамическое имя типа, воспользуйтесь функциями Value.Is и Value.As.

```
let
  TestType = type number
in
  Value.Is(1, TestType) // true
  //Value.Is("abc", TestType) // false
```

```
let
  TestType = type number
in
  Value.As(1, TestType) // возвращает 1
  /*Value.As("abc", TestType) // Expression.Error: Не удается
преобразовать значение "abc" в тип Number. */
```

Вторым аргументом в функциях Value.Is и Value.As допускается только обнуляемый примитивный тип.

Проверку совместимости значения с типом также можно выполнить с помощью функции *Type.Is*. Функция проверяет, *всегда ли* тип, указанный в качестве первого аргумента, совместим со вторым аргументом, который должен быть обнуляемым примитивным типом.

```
Type.Is(type number, type any) // true, number всегда совместимо с any

Type.Is(type any, type number) // false, any не всегда совместим с number

Type.Is(type logical, type nullable logical) /* true, logical совместим,
либо с logical, либо с null, что фактически означает тип nullable logical */

Type.Is(type null, type nullable logical) // true

Type.Is(type none, type any) /* true, доказательство того, что none совместим
с типом any, даже если никогда не может быть значения типа none */

Type.Is(type none, type anynonnull) /* true, доказательство
того, что none совместим с типом anynonnull */
```

Динамическое типизирование

Power Query типизируется динамически. Проверки, совместимо ли значение с заданным типом, выполняются путем просмотра типа значения во время выполнения. Нет понятия переменной, имеющей тип, и приведение типов также не имеет значения – важен тип фактического значения.

```
let
   Inner = () as any => ...,
   Result = Inner()
   …
```

Здесь указано, что *Inner as any.* Но это не влияет на тип значения, хранящегося в *Result*. Указание *as any* не изменяет тип значения на *any*. Это просто гарантирует, что все, что возвращается из *Inner*, совместимо с *any*. Тип значения в *Result* будет типом того, что было возвращено *Inner* (*text*, *number*, *null*, ...). Этот тип может меняться каждый раз при вызове выражения в зависимости от того, что возвращает *Inner*.

Переменная *Result* сама по себе не имеет типа. Это просто переменная, которая содержит значение – и это значение имеет тип. Чтобы подчеркнуть тот факт, что тип значения – это тоже значение, рассмотрим код:

```
let
   Inner = () as any => ...,
   Outer = (input as anynonnull) => ...
in
   Outer (Inner())
```

Как и прежде, *Inner* имеет такое же утверждение *as any*. *Outer* ожидает значение с типом *anynonnull*. Тип *any* несовместим с *anynonnull*, но это выражение не приводит к проверке двух типов, используемых в утверждениях, на совместимость. *Inner as any* просто гарантирует, что функция вернет что-то совместимое с *any*. *Outer as anynonnull* гарантирует, что переданное значение совместимо с *anynonnull*. Возможно иметь значение, совместимое с обоими утверждениями, даже если сами типы утверждений несовместимы.

Во время выполнения, если значение, возвращаемое *Inner()*, не равно *null*, это выражение будет работать нормально; если это не так, будет выдана ошибка.

Итак, тип значения – это то, что имеет значение. Утверждения типа просто проверяют во время выполнения, что передаваемое через них значение совместимо с указанным типом. Сами переменные не имеют типа.

В следующей заметке

Пока всё было относительно просто. Но прежде чем перейти к пользовательским типам, мы узнаем об украшении типов аннотациями. Они не изменяют тип, а служат для информации. Эти аннотации обычно называют *фасетами*. В редакторе запросов вы, наверное, видели, что существует четыре типа чисел: десятичные, валюта, целые и проценты. На самом деле это просто тип *number*, дифференцированный по граням. Подробнее об этом в следующей заметке.