# Язык M Power Query. Система типов. Аспекты

Обращали ли вы внимание, что в редакторе Power Query меню типа данных столбца включает четыре варианта для чисел: *Десятичное число, Валюта, Целое число* и *Процент*? До сих пор в этой серии заметок мы говорили только об одном числовом типе: *type number*. Есть ли какие-то типы, которые мы пропустили?

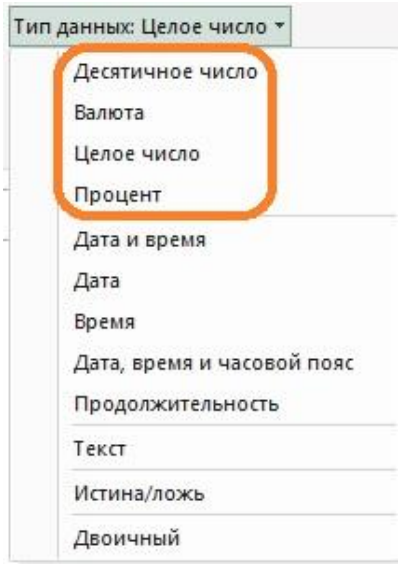



Рис. 1. Аспекты типа *number*

Предыдущая заметка  Следующая заметка

За кулисами меню устанавливает следующие соответствия:

- Десятичное число –> *type number*
- Валюта –> *Currency.Type*
- Целое число –> *Int64.Type*
- Процент –> *Percentage.Type*

Если вы изучите типы, определенные в среде Power Query для чисел, вы также увидите ряд других названий: *Int8.Type, Int16.Type, Int32.Type, Single.Type* и *Double.Type*. Это что ж получается, в предыдущей заметке мы рассмотрели не все типы!? Нет! Это всё подтипы *type number…* так же известные, как *аспекты (англ. facets)*.[1]

## *Аспекты*

Power Query позволяет добавить к типу *аспект*[2] – аннотацию только для информации. Аспект не оказывают существенного влияния на поведение типов или на значения, связанные с ними, ни на уровне языка, ни на уровне движка Power Query. Логика, которую вы используете (включая стандартную библиотеку), может считывать аспекты и реагировать на них, хотя это делается редко (если вообще когда-либо!). Вместо этого аспекты используются почти исключительно в контексте взаимодействия с внешней средой (источники данных, инструменты, хост-среда). Внешние системы часто имеют более сложные системы типов, чем Power Query. Аспекты предоставляют способ передачи дополнительных сведений, связанных с типом, во внешний мир и из него. Аспекты разделяют на простые и… другие.

Представьте, что вы работаете с веб-сервисом, который возвращает таблицу, содержащую два столбца: текстовый и числовой. M знает, что значения в этих столбцах имеют тип *text* и тип *number*. Однако при создании запроса может быть полезным знание того, что текстовый столбец содержит строки длиной до 25 символов, а числовой – целые числа, а не десятичные. Эти сведения, предоставляемые коннектором, не влияют на то, как движок M обрабатывает данные, но они могут быть полезны вам как разработчику.

---

[1] Заметка написана на основе статьи Ben Gribaudo. Power Query M Primer (Part 17): Type System II – Facets. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

[2] Англ. *facet* я перевел как *аспект* в соответствии с документацией MS. В рунете мне не встретились материалы по этой теме, так что сравнить мой вариант перевода просто не с чем.

И наоборот, когда запрос возвращает данные в среду хоста (Microsoft Excel), дополнительная информации о типе значений может повлиять на то, как внешняя среда обработает результат запроса.

Поскольку таблицы являются основной формой ввода и вывода, которыми PQ обменивается с внешним миром, аспекты в первую очередь представляют интерес при работе с таблицами. Однако никакие технические ограничения не препятствуют использованию аспектов и с нетабличными типами.

В спецификации языка M не упоминаются аспекты, хотя они являются частью информации, содержащейся в значениях типов.[3]

### *Простые аспекты*

Простые аспекты используются коннекторами для предоставления дополнительной информации о значениях. Хотя технически аспекты могут быть программно считаны и обработаны, они в первую очередь предназначены для пользователя. Если вы не создаете коннекторы, вы, скорее всего не столкнетесь с необходимостью настройки аспектов. Однако изучение того, как установить аспекты, поможет глубже их понять, так что давайте попробуем.

Чтобы задать значение простого аспекта, используйте Type.ReplaceFacets:

Type.ReplaceFacets(type as type, facets as record) as type

Обратите внимание, что функции можно передать только один *type*, но несколько аспектов. Для любых аспектов, не представляющих интереса, должно быть установлено значение *null* или они должны быть исключены из записи:

```
[
  NumericPrecisionBase = ..., // number
  NumericPrecision = ..., // number
  NumericSale = ..., // number
  DateTimePrecision = ..., // number
  MaxLength = ..., // number
  IsVariableLength = ..., // logical
  NativeTypeName = ..., // text
  NativeDefaultExpression = ..., // text
  NativeExpression = ... // text
]
```

В следующем примере задаются параметры *MaxLength*, *IsVariableLength* и *NativeTypeName* для значения типа *text*.

**Листинг 1**[4]

```
let
    SomeType = type text,
    Facets =
    [
        MaxLength = 25,
        IsVariableLength = true,
        NativeTypeName = "NVARCHAR"
    ],
    TypeWithFacets = Type.ReplaceFacets(SomeType, Facets)
in
    TypeWithFacets
```

*TypeWithFacets* по-прежнему является значением типа, содержащим тип *text*. Установка трех аспектов не привела к созданию нового типа. Присоединение аспектов лишь добавило информацию к существующему типу. Движок M по-прежнему работает с типом *text*.

---

[3] Если вы зайдете на страничку Типы, нажмете Ctrl+F и попробуете найти *int*, вас постигнет неудача.

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

Чтобы узнать настройки аспектов используйте функцию Type.Facets, возвращающую запись:

Type.Facets(type as type) as record

Добавив *Type.Facets* к листингу 1, можно увидеть простые аспекты, которые были ранее определены:

**Листинг 2**

```
let
   SomeType = type text,
   Facets =
   [
      MaxLength = 25,
      IsVariableLength = true,
      NativeTypeName = "NVARCHAR"
   ],
   TypeWithFacets = Type.ReplaceFacets(SomeType, Facets)
in
   Type.Facets(TypeWithFacets)
```

| | |
|---|---|
| NumericPrecisionBase | null |
| NumericPrecision | null |
| NumericScale | null |
| DateTimePrecision | null |
| MaxLength | 25 |
| IsVariableLength | TRUE |
| NativeTypeName | NVARCHAR |
| NativeDefaultExpression | null |
| NativeExpression | null |

Рис. 2. Запись с установленными аспектами

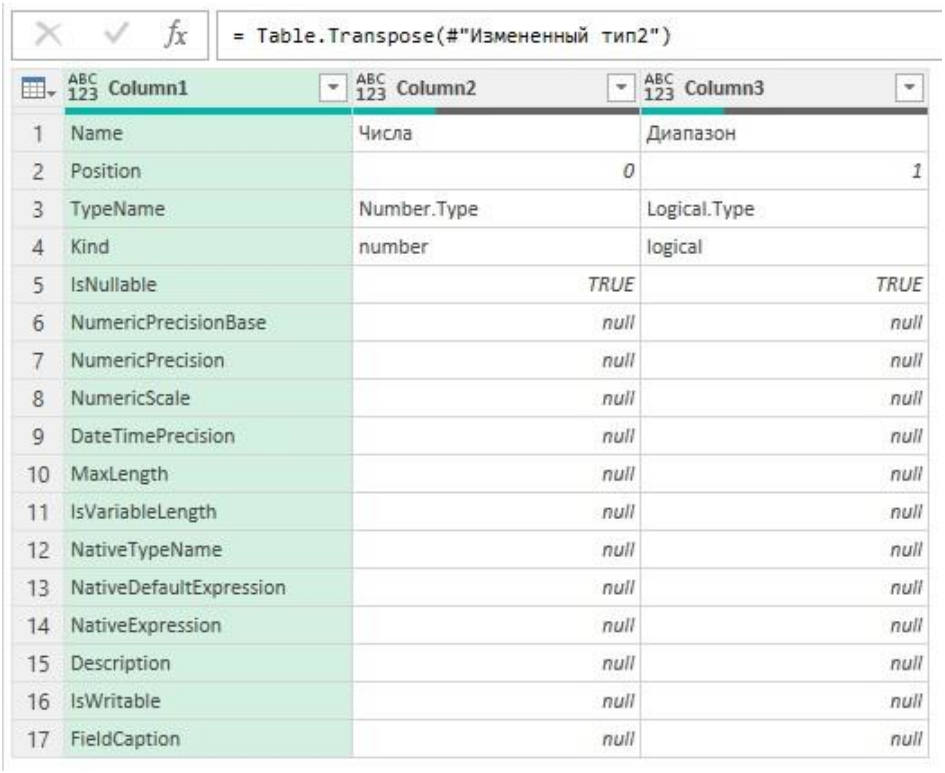
Представьте, что мы не использовали *TypeWithFacets*, а получили данные с помощью коннектора из внешнего источника. На основе записи, представленной на рис. 2 можно сделать вывод, что получены данные из столбца NVARCHAR. Мы понимаем, что данные содержат символы, охватывающие весь диапазон Юникода, а длина строк не превышает 25 символов. При том, что у нас может не быть доступа к удаленной системе, информация в аспектах дает представление о том, как внешняя система думает о данных.

*Type.Facets* позволяет просматривать простые аспекты, связанные с типом, по одному типу за раз. Чтобы увидеть аспекты каждого столбца в таблице, нужно передать тип столбца в *Type.Facets*, или использовать *Table.Schema*. Эта функция позволяет увидеть все аспекты (простые и другие), а также иную информацию сразу для всех столбцов.

**Листинг 3**

```
let
   Источник = Excel.CurrentWorkbook(){[Name="Таблица1"]}[Content],
   #"Измененный тип" = Table.TransformColumnTypes(Источник,{{"Числа", type number}}),
   #"Условный столбец добавлен" = Table.AddColumn(#"Измененный тип", "Диапазон", each if
[Числа] <= 0.5 then 0 else 1),
   #"Измененный тип1" = Table.TransformColumnTypes(#"Условный столбец
добавлен",{{"Диапазон", type logical}}),
   #"Тип" = Table.Schema(#"Измененный тип1"),
   #"Пониженные заголовки" = Table.DemoteHeaders(Тип),
   #"Измененный тип2" = Table.TransformColumnTypes(#"Пониженные заголовки",{{"Column1", type
text}, {"Column2", type any}, {"Column3", type text}, {"Column4", type text}, {"Column5", type any},
{"Column6", type text}, {"Column7", type text}, {"Column8", type text}, {"Column9", type text},
```

```
{"Column10", type text}, {"Column11", type text}, {"Column12", type text}, {"Column13", type text},
{"Column14", type text}, {"Column15", type text}, {"Column16", type text}, {"Column17", type text}}),
  #"Транспонированная таблица" = Table.Transpose(#"Измененный тип2")
in
  #"Транспонированная таблица"
```




| | Column1 | Column2 | Column3 |
|---|---|---|---|
| 1 | Name | Числа | Диапазон |
| 2 | Position | 0 | 1 |
| 3 | TypeName | Number.Type | Logical.Type |
| 4 | Kind | number | logical |
| 5 | IsNullable | TRUE | TRUE |
| 6 | NumericPrecisionBase | null | null |
| 7 | NumericPrecision | null | null |
| 8 | NumericScale | null | null |
| 9 | DateTimePrecision | null | null |
| 10 | MaxLength | null | null |
| 11 | IsVariableLength | null | null |
| 12 | NativeTypeName | null | null |
| 13 | NativeDefaultExpression | null | null |
| 14 | NativeExpression | null | null |
| 15 | Description | null | null |
| 16 | IsWritable | null | null |
| 17 | FieldCaption | null | null |

Рис. 3. Результат работы функции *Table.Schema*

Название аспекта *TypeName* может сбивать с толку. На самом деле базовый тип M для столбца отображается в аспекте *Kind*. *TypeName* – не идентификатор типа в M, а аспект для выбранного типа.

### *Другие аспекты*

Power Query использует только одну опцию для примитивных типов: все числа имеют тип *number*, все строки – тип *text* и т.д. Внешние системы могут иметь более обширные наборы. Например, вместо универсального числового типа *number* внешняя система может различать целые и десятичные числа. Более того, целые числа могут быть 8-, 16-, 32- и 64-разрядными. Хотя такие нюансы не влияют на выполнения запросов, было бы разумно «протащить» эту информацию сквозь запрос. Т.е., получить нюансы типов из внешней системы, обработать данные в Power Query игнорируя нюансы, и вернуть результат запроса во внешнюю систему с учетом нюансов.

Аспекты предоставляют стандартизированный способ идентификации внешних типов с использованием независимой от системы номенклатуры типов. Коннекторы сопоставляют типы из внешней системы и аспекты в M. Внутри M эта информация может влиять, например, на то, как редактор запросов отражает значки столбцов таблицы.

В отличие от простых аспектов, которые по умолчанию имеют значение *null* (если не заданы), у *других* аспектов определение параметров обязательно. По умолчанию оно соответствует базовому имени типа. Например, аспект *TypeName* типа *text* по умолчанию определяется, как *Text.Type*, а типа *date* – как *Date.Type*. От значений по умолчанию мало прока. Но тот факт, что можно объявить другие имена, делает это интересным.

Имена типа являются предопределенными. Вы не можете указать произвольное значение, как это можно сделать с простыми аспектами. Предопределенные значения определены в стандартной библиотеке. Чтобы использовать их, нужно извлечь предопределенные значения, записанные в аспектах.

### Имена типов

Теперь мам должен быть понятен смысл имен подтипов стандартной библиотеки (*Int8.Type*, *Currency.Type*, *Single.Type* и др.). Но, не позволяйте подтипу в названии сбивать вас с толку. Когда вы создаете что-то в редакторе запросов, вы определяете выражение, которое возвращает значение, и даете типу этого выражения имя. Ваш выбор имени, заканчивающегося на *type*, не создает новый тип на уровне языка, и тот факт, что стандартная библиотека дает подтипам имена, заканчивающиеся на *type*, не приводит к появлению новых типов. Например, *Int8.Type* – это имя, связанное с выражением, которое возвращает значение типа *number* для 8-разрядных целых чисел.

### Визуализация

Подтипы так мало используются за пределами таблиц, что единственный способ увидеть этот аспект – это использовать табличные функции стандартной библиотеки. Однако, поскольку мы изучаем типы в целом и табличные типы в частности, было бы неплохо иметь способ просмотра аспекта, включая нетабличные типы.

Напишем функцию *TypeClaimFacet*, которая будет извлекать аспект *TypeName* из значения типа. Эта функция использует синтаксис, который подробнее рассмотрен в следующем посте.

**Листинг 4**

```
(input as type) as text => Table.SingleRow(
   Table.Schema(
      #table(
         type table [Col1 = input],
         {}
      )
   )
)[TypeName]
```

Если на вход функции подать имя типа, на выходе будет аспект *TypeName*:

**Листинг 5**

```
= TypeClaimFacet(type number)
// "Number.Type", аспект TypeName по умолчанию для типа number
```


Рис. 4. Извлечение аспекта *TypeName* из типа *number*

### *Приписываемые типы*

В Power Query каждому значению приписывается тип. Когда значение создается, Power Query автоматически присваивает ему внутренний тип: числовому значению – тип *number*, строковому – тип text и т.д. Неизменяемость M означает, что вы не можете изменить тип, приписываемый значению. Так что же наличие аспектов бессмысленно? Раз вы не можете связать со значением аспект, отличный от стандартного.

К счастью, вместо изменения типа, связанного со значением (что недопустимо), вы можете заменить его с помощью функции *Value.ReplaceType*. Она возвращает значение, которое является тем же, что и раньше, за исключением одного: значение связано с новым типом.

**Листинг 6**

```
// приписывает тип, имеющий аспект TypeName, отличный от типа по умолчанию
let
   StartingValue = 1, // мы знаем, что это тип number
```

```
    Result = Value.ReplaceType(StartingValue, Int64.Type)
in
    Result // значение 1 типа number, с TypeName = Int64.Type
```

**Листинг 7**

```
// приписывает тип, имеющий простой аспект
let
    StartingValue = 1, // мы знаем, что это тип number
    OriginalType = Value.Type(StartingValue),
    Faceted = Type.ReplaceFacets(OriginalType, [NativeTypeName = "INT"]),
    Result = Value.ReplaceType(StartingValue, Faceted)
in
    Result // значение 1 типа number, украшенное аспектом NativeTypeName = "INT"
```

**Листинг 8**

```
// приписывает тип, имеющий как утверждение о типе, так и простой аспект
let
    StartingValue = 1, // мы знаем, что это тип number
    Faceted = Type.ReplaceFacets(Int64.Type, [NativeTypeName = "INT"]),
    Result = Value.ReplaceType(StartingValue, Faceted)
in
    Result /* значение 1 типа number, с TypeName = Int64.Type
    и украшенное аспектом NativeTypeName = "INT"*/
```

В этих примерах базовый тип значения не менялся с *number* на какой-либо другой базовый тип. Вместо этого тип *number* с именем по умолчанию (*TypeName = Number.Type*) украшался (или можно сказать *дополнялся*) одним или несколькими аспектами.

Приписываемый тип всегда должен быть структурно совместим со значением, которому он приписывается. Таким образом, числовые значения могут быть приписаны только значениям числового типа, текстовые значения могут быть связаны только со значениями текстового типа и т.д. Вы не можете использовать *Value.ReplaceType* для приведения числового значения к типу *text* или наоборот.

То, как работает приписывание типов, когда задействованы пользовательские типы, является сложным и, возможно, нелогичным, но все же соответствует правилу, согласно которому приписываемый тип должен быть совместим со значением (подробнее в следующей заметке).

### *Утверждение типов против преобразования типов*

Повторю ключевой момент: аспекты типов не оказывают влияния, ни на движок M, ни на поведение на уровне языка. В частности, аспекты не определяют и не создают новые типы или подтипы — ничего подобного. Это просто информационные аннотации.

Значение типа *number* является числом и будет вести себя как любое другое число, независимо от того, связан ли тип значения с утверждением, что это *Int16.Type* или *Percentage.Type*. 1 + 2 всегда будет равно 3, независимо от того, какие аспекты могут быть связаны с каждым из этих значений.

Приведенное ниже значение не является 64-разрядным целым числом, поскольку оно находится за пределами диапазона значений целых чисел, которые могут быть представлены в 4 байтах. Однако никаких проблем не возникает, если ему приписывается аспект утверждения типа *Int16.Type* — потому что аспекты являются строго информационными. Они не проверяют факты.

```
Value.ReplaceType(9223372036854775808.01, Int64.Type)
```

С другой стороны, в редакторе запросов, если вы используете команду *Изменить тип* для столбца таблицы, чтобы преобразовать числа в целые (что соответствует *Int64.Type*), значение 9223372036854775808.01 вызовет ошибку:

**Листинг 9**

```
let
    Data = #table({"Col1"}, {{9223372036854775808.01}}),
    #"Changed Type" = Table.TransformColumnTypes(Data,{{"Col1", Int64.Type}})
```

in
  #"Changed Type"

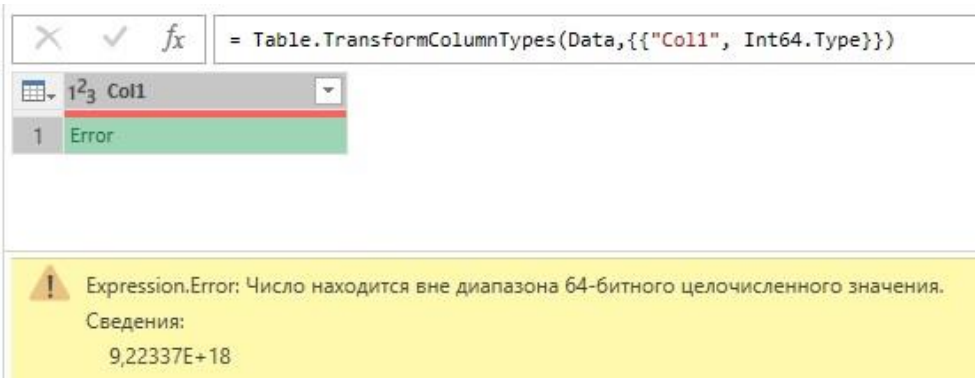



Рис. 5. Ошибка преобразования типа

Не противоречит ли это утверждению *никакого влияния на поведение*?

Функция *Table.TransformColumnTypes* выполняет две вещи: устанавливает для столбца тип, украшенный утверждением о типе (аспектом), и пытается преобразовать каждое значение в столбце в соответствующий тип, попутно проверяя, соответствуют ли значения указанному утверждению о типе. *Table.TransformColumnTypes* использует набор функций, имена которых заканчиваются на *.From*, для выполнения преобразований значений. Вы можете напрямую поиграть с этими функциями, чтобы увидеть, как работает процесс преобразования.

Например, в случае *Int64.From* сначала входное значение преобразуется в тип *number* (если оно еще не является числом), а любая десятичная составляющая округляется. Затем результирующее целое число проверяется, находится ли оно в диапазоне допустимых значений для 64-разрядного целого числа. Если это так, то возвращается число. Если нет, то возникает сообщение об ошибке.

**Листинг 10**
```
= Int64.From(9223372036854775808.01) /* возвращает Expression.Error:
   Число находится вне диапазона 64-битного целочисленного значения.*/
```

**Листинг 11**
```
= Int64.From(9223372036854775807.01) /* возвращает 9223372036854775807,
   число на единицу меньше предыдущего, произошло округление до целого*/
```

Процесс преобразования – это то, что вызывает ошибки, связанные со значениями, выходящими за пределы ожидаемого диапазона. Процесс преобразования не приписывает тип. Например, результат, возвращаемый *Int64.From* не связан с аспектом *Int64.Type* утверждения типа.

**Листинг 12**
```
let
   Источник = Int64.From(9223372036854775807.01),
   #"Преобразовано в таблицу" = #table(1, {{Источник}}),
   Result = Table.Schema(#"Преобразовано в таблицу")[[Name], [TypeName]]
in
   Result
```

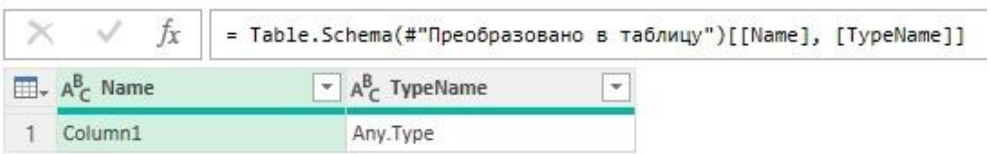



Рис. 6. Функция *Int64.From* преобразует значения, но не приписывает тип

В дополнение к преобразованию значений, *Table.TransformColumnTypes* задает соответствующий тип столбца. Мы можем наблюдать это, обратившись к функции *Table.Schema*.

**Листинг 13**
```
let
```

```
    Data = #table({"Col1"}, {{9223372036854775808.01}}),
    #"Changed Type" = Table.TransformColumnTypes(Data,{{"Col1", Int64.Type}})
in
    Table.Schema(#"Changed Type")[[Name], [TypeName]]
    // возвращает таблицу с одной строкой [Name = "Col1", TypeName = "Int64.Type" ]
```

Обратите внимание, что *Col1* после преобразования имеет утверждение типа *Int64.Тип*, отражаемое в столбце *TypeName*:


Рис. 7. Функция *Table.TransformColumnTypes* преобразует значения и приписывает тип

### Преобразовать или приписать?

Если вы *думаете* (или надеетесь), что значения столбцов будут соответствовать некому типу, используйте *Table.TransformColumTypes*, чтобы установить аспект. Функция преобразует значения, а когда это будет невозможно, вернет ошибку, предупреждая о проблеме. В дальнейшем вы сможете обработать ошибки, например, использовав *Table.ReplaceErrorValues*.

С другой стороны, если вы *уверены*, что значения будут соответствовать некому типу (например, при создании коннектора, который может определить это с помощью метаданных внешней системы), припишите соответствующие аспекты типа столбцам таблицы без преобразования. Нет необходимости тратить ресурсы на избыточную обработку. Присвоение аспектов типам столбцов включает в себя пользовательские типы, о которых мы поговорим в следующей заметке.

### Внешние системы и аспект утверждения типа

Почему аспект утверждения типа имеет значение, если он не оказывают воздействия на движок? М сам по себе **не** проверяет соответствие типам, но другие системы могут это делать. Чаще всего это происходит, когда данные передаются во внешние системы. Предположим, что в редакторе Power Query вы создаете в таблице столбец, содержащий числа, и присваиваете столбцу тип *Int64.Type*. Power Query не заботится о том, действительно ли значения столбца соответствуют этому утверждению. Однако, когда таблица передается в среду хоста, это утверждение поможет Excel установить тип хранилища, который он использует для столбца. Далее, из-за того что используется типа хранилища для целых чисел, импорт строк содержащих десятичные числа приведет к ошибке.

Проверяйте, что любые аспекты утверждения типа, которые выводятся вашими запросами, верны. Если вы не уверены, лучше отказаться от утверждения типа, даже если внешней среде, которую вы используете, в данный момент, похоже, все равно.

### *Область действия*

Если у вас есть значение, тип которого включает аспекты, сохраняются ли они при выполнении операций и вызовах функций? Допустим, вы применяете оператор для значений, которые связаны с аспектом утверждения типа, отличным от стандартного. Будет ли этот аспект перенесен на результат?

**Листинг 14**

```
let
    ValueA = Value.ReplaceType(1, Int64.Type),
    ValueB = Value.ReplaceType(2, Int64.Type),
    Result = ValueA + ValueB
in
    Result
```

Листинг 14 очевидно возвращает 3, но связан ли тип этого числа с *Int64.Type*? Нет, не связан, и так не должно быть. Аспекты не влияют на язык M и поведение движка. Сложение двух чисел приводит к получению нового числа, которому будет присвоен тип по умолчанию – *number*.

С другой стороны, аспекты могут быть сохранены при вызове функции. Как правило, вы будете наблюдать это при использовании табличных функций. Некоторые функции списка также сохраняют аспекты. Функции использует исходные типы столбцов (или элементов списка) для определения типа результата. Поскольку типы сохраняются, все связанные с ними аспекты также остаются. Например, если вы применяете *Table.Sort* типы столбцов сохраняются вместе с аспектами.

### *Заключение*

Помните, что аспекты типов в первую очередь важны при взаимодействии с внешним миром (источниками данных, средой хоста и инструментами). Поскольку таблицы являются основным средством обмена данными с внешним миром, аспекты используются почти исключительно с таблицами. Аспекты не влияют на поведение, ни на уровне языка, ни на уровне движка. Применение аспектов и присвоение имен, заканчивающихся на *.Type*, не создают новых типов.

### *В следующей заметке*

Таблицы, функции, записи и списки – все они могут принимать пользовательские типы для описания своих особенностей. В следующий раз мы узнаем, как определять пользовательские типы и работать с ними. Приготовьтесь к сюрпризу: приписывание типов и проверка совместимости могут работать не так, как вы ожидаете.