УДК: 004.43, 004.4'4

1.3. Разработка языка Тривиль. Часть 3. Баланс

А. Е. Недоря, г. Санкт-Петербург, Россия

Статья является третьей из серии статей, в которых описывается разработка языка программирования Тривиль. В предыдущих статьях были определены цели языка, требования к языку и описаны основные языковые решения. Эта статья рассматривает сложные языковые конструкции, в разработке которых основное внимание уделялось балансу между полнотой языковой конструкции и сложностью языка и компилятора. Как и вся серия, статья нацелена, большей частью, не на программиста, который использует язык, а на разработчика языков программирования.

Введение

В первых двух статьях серии [1, 2] были определены цели языка, основные требования к языку и описаны основные конструкции языка Тривиль, при этом особое внимание было уделено тому, как выбор конструкций вытекал из целей и требований. В данной статье будет рассмотрено несколько сложных конструкций, в разработке которых приходилось искать баланс между полнотой конструкции и сложностью ее реализации.

Содержание этой статьи можно кратко передать фразой: **“Лучше день потерять, а потом за пять минут долететь” или нет?**

Фраза содержит цитату из известного советского мультфильма[1], в котором гриф пытался научить страуса летать, чтобы добраться до вкусного за 5 минут полета вместо долгого бега.

В разработке Тривиля я неоднократно сталкивался с выбором такого же типа:

- если в язык добавить некоторую конструкцию, то она упростит дальнейшую разработку и **ускорит** переход к разработке следующих языков семейства (лучше день потерять);
- в то же время разработка языковой конструкции и реализация её в компиляторе может потребовать много усилий, а это **замедлит** переход к разработке следующих языков.

Может показаться, что это редкая ситуация. Увы, разработчики программных систем постоянно вынуждены делать выборы примерно такого типа. Даже если они понимают, как сделать правильно, хорошо и надолго (что далеко не всегда), обстоятельства часто вынуждают использовать *моток синей изоленты* вместо хорошего решения и надеяться, что когда-нибудь это место будет переделано с надлежащим качеством. Для такой времянки даже есть специальный термин: *технический долг*.

Любопытно, что технический долг — это специфика программной инженерии. Почему? Потому что недооценка времени и трудозатрат, задержка времени выполнения проектов и прочие неприятные вещи являются обязательными спутниками (почти) всех программных проектов. В одной из следующих статей мы подробно поговорим о том, почему это происходит, кто виноват и что делать. Впрочем, ответ на вопрос *кто виноват?* можно дать прямо сейчас — виноваты мы, а именно: “товарищи ученые, доценты с кандидатами[2]”.

Вернусь к Тривилю. Часть конструкций языка я считаю обязательными, независимо от затрат на их реализацию, например, те, что реализуют безопасность ссылок (null safety). Часть - такими, без которых можно обойтись, хоть это может быть в чем-то неудобно. Для части конструкций (полное решение которых трудозатратно) приходилось искать компромиссные решения, которые покрывают существенную часть необходимой функциональности, но, в то же время, могут быть сделаны за небольшое время.

В следующих разделах мы рассмотрим такие вопросы:

- операторы цикла;
- инициализация объектов;
- реализация обобщенных хеш-таблиц.

Циклы всякие нужны

В предыдущей статье в списке операторов был указан единственный оператор цикла — цикл пока. И это понятно: так вместе с оператором прервать (break) этот цикл позволяет выразить любое повторяющееся действие. Зачем же в языки программирования добавляют другие операторы цикла? Для увеличения выразительности и для уменьшения числа ошибок (что равно энергосбережению). Например, если обработку элемента массива делать циклом пока, то можно ошибиться в проверке условия и изменении счетчика, да и запись становится сложнее.

Рассмотрим, какие циклы есть в других языках на примере Go и затем попробуем найти баланс между усложнением языка и компилятора и упрощением программирования.

1 Мультфильм “Крылья, ноги и хвосты”, 1987 г.

2 В. Высоцкий, “Товарищи ученые”

1	бесконечный цикл	`for { … break … }`
2	цикл с предусловием	`for a < b { … }`
3	арифметический цикл	`for i := 0; i < 10; i++ { … }`
4	цикл перебора элементов	`for i, val := range arr { … }`

Бесконечный цикл легко выражается через цикл с предусловием, арифметический цикл не имеет смысла в компиляторах, а вот перебор элементов массива или контейнера — это нужное действие. Цикл перебора элементов часто называется foreach и есть во всех современных языках программирования, см., например, Kotlin, Swift, Rust.

В общем виде цикл позволяет перебирать элементы любого контейнера и требует, чтобы язык поддерживал:

- итераторы, чтобы разработчик библиотеки контейнера мог обеспечить возможность использования цикла;
- возможность возвращать из итератора несколько значений, например, через поддержку кортежей (Swift) или функций, возвращающих несколько значений (Go).

И то, и другое существенно влияет и увеличивает сложность языка и компилятора и в общем виде не подходит для Тривиля.

Что именно нужно перебирать в компиляторе? Элементы массива и элементы словаря (hash-map). Увы, перебор элементов словаря придется отложить на следующий не-совсем-тривиль язык, и как мы дальше увидим, это вполне оправдано.

Рассмотрим подробней перебор элементов массива. Есть три варианта того, что используется в теле цикла:

- значение элемента;
- индекс элемента;
- индекс и значение.

Go поддерживает все три варианта за счет использования символа игнорирования:

- нужно только значение: `for _, val := range arr {…};`
- нужен только индекс `for i, _ := range arr {…}` или `for i := range arr {…}`.

Вот последняя запись меня удивляет: на мой взгляд, значение “важнее” и чаще используется, почему же запись с одним индексом короче? Когда я пишу на Go, я каждый раз теряю время/силы на то, чтобы вспомнить, как надо писать, пусть совсем немного, но теряю. Поэтому я приложил существенные усилия, чтобы найти синтаксис, который более очевиден и, следовательно, сберегает энергию. Вот что получилось:

Что перебирается	**Go**	**Тривиль**	**Исп. (*)**
массив: только значение	`for _, val:= range arr {…}`	**цикл** эл **среди** arr {…}	137
массив: индекс, значение	`for i, val:= range arr {…}`	**цикл** [№]эл **среди** arr {…}	16
массив: только индекс	`for i:= range arr {…}`	**цикл** [№] **среди** arr {…}	0
словарь: ключ, значение	`for key, val:= range hm {…}`	нет поддерживается	47

В последнем столбце записано число использований конструкции в компиляторе Тривиля. Как видно, перебор значений — это самая используемая конструкция и она самая короткая в Тривиле. Если в цикле нужен индекс, используется дополнительный синтаксис, указывающий, что это именно индекс, то есть то, что используется для индексации. А вот перебор только индексов, то есть то, что в Go записывается короче всего, вообще не используется в компиляторе. Перебор элементов словаря в Тривиле делается отдельным образом, без использования такого цикла.

Важно то, что перебор элементов массива — это ¾ всех случаев перебора элементов в компиляторе (153 из 200). Выбранное частное решение хорошо тем, что

- покрывает существенную часть случаев,
- не требует добавления в язык сложных конструкций,
- и является простым в реализации.

Инициализация объектов

Одно из сложных мест в любом языке, который каким-либо образом поддерживает OOP — это инициализация объектов. Это сущностно сложная конструкция, поэтому особые усилия должны быть направлены на ее упрощение в языке.

Инициализация объекта должна привести объект в *корректное* состояние, или состояние готовности к работе, после чего любые *корректные* действия над объектом должны сохранять корректное состояние (другими словами, сохранять инвариант класса или объекта[3]).

Некоторые языки, такие как Swift, явно выделяют две фазы инициализации объекта [3], задача которых:

1. задать определенные значения полям объекта, так чтобы избежать неопределенного поведения (undefined behaviour);
2. перевести объект в семантически *корректное* состояние или прервать выполнение, если это невозможно.

На мой взгляд, выделение двух фаз — это верный подход, так как язык не может гарантировать корректность семантики (это задача разработчика), даже если инвариант явно задан[4] (см. Eiffel). А вот гарантировать, что все поля проинициализированы, язык и компилятор могут и обязаны делать в языке, который заявлен как безопасный[5].

Далее в статье будут рассмотрены подходы к инициализации в Swift, Kotlin и Go, потом в Тривиле и, далее рассмотрены ограничения Тривиля: баланс сложности (простоты) и полноты.

Для сравнения сначала определим критерии или требования:

1. безопасность: инициализация должна выполняться безопасно, то есть язык и компилятор должны гарантировать отсутствие ошибок в первой фазе. Вторая фаза — в зоне ответственности разработчика;
2. простота: инициализация должна записываться просто, насколько возможно, но, обязательно безопасно, см. п.1;
3. полнота: все “нормальные” (безопасные) случаи должны быть поддержаны.

Инициализация в Swift

Рассмотрим пример, в котором есть класс, каждый объект которого содержит поле с порядковым номером объекта (в порядке создания):

```
var counter = 0 // счетчик объектов

class Counted {
    var ord: Int
    init() {
      // фаза 1:
      self.ord = 0
      // фаза 2:
      counter += 1
      self.ord = counter
    }
}
var first = Counted()
   print(first.ord)
```

Выполнение примера выводит 1 на консоль.

3 Для языков, в которых нет классов: OOP, да не CLOP.

4 Это никак не относится к языкам, в которых неопределенное состояние является штатным.

5 CLOP язык — язык поддерживающий class-oriented programming.

Как мы видим, на фазе 1 происходит инициализация всех полей класса (одного, в данном случае), и она заканчивается, когда все поля проинициализированы. Фаза 2 переводит объект в семантически корректное состояние. В нашем примере, это состояние, когда значение поля ord — это порядковый номер объекта.

Код инициализации в примере намеренно сделан избыточным, чтобы показать разделение на фазы. Его можно переписать так, чтобы вся инициализация проходила на фазе 1:

```
    init() {
      // фаза 1
      counter += 1
      self.ord = counter
      // начало фазы 2
        }
```

Во время фазы 1 нельзя обратиться к полям или методам объекта, например, нельзя обратиться к полю до его инициализации:

```
    init() {
      // фаза 1
      print(self.ord) // error - used before being initialized
      counter += 1
      self.ord = counter
      // начало фазы 2
        }
```

Компилятор покажет ошибку. Отсутствие инициализации тоже приведет к сообщению об ошибке:

```
class Recursive {
    var next: Recursive // error - not initialized
   }
```

А такой код скомпилируется, но приведет к бесконечной рекурсии во время выполнения, что более-менее приемлемо, так как соответствует поведению для обычных рекурсивных функций:

```
class Recursive {
    var next: Recursive
    init() {
      next = Recursive()
    }
}
```

У внимательного читателя, незнакомого с Swift, может возникнуть вопрос, как описывать рекурсивные структуры, например, дерево. Ответом является использование опционального типа:

```
class Tree {
    var l: Tree? = Optional.none
    var r: Tree? = Optional.none
}
```

В этих примерах показана малая часть того, что есть в инициализации объектов в Swift, размер раздела инициализации в спецификации языка — это около 30 страниц текста. Но нам пока достаточно, чтобы сравнить с другими языками.

Замечу, что часть кода в примерах можно было написать короче, например, опустить self или писать nil вместо Optional.none. Я намеренно использую более полный и ясный для читателя вариант кода.

Краткий вывод для Тривиля: очень сложно, не надо так.

Инициализация в Kotlin

Повторим пример с порядковым номером объекта:

```
var counter = 0

class Counted {
   var ord: Int
   init {
     counter += 1
     this.ord = counter
   }
}

fun main() {
  val first = Counted()
  print(first.ord)
}
```

Код выглядит очень похоже на код на Swift и ведет себя также. В примере с печатью неинициализированного поля компилятор также сообщает об ошибке, а пример с рекурсивным типом компилируется и приводит к бесконечной рекурсии при выполнении. В примере с деревом также надо использовать опциональный тип.

А вот следующий пример показывает принципиальную разницу между двумя языками:

	Kotlin		Swift
1	`class A { val f = 1 }`	1	`class A { var f = 1 }`
2		2	
3	`class C {`	3	`class C {`
4	`    val a: A`	4	`    var a: A`
5	`    init {`	5	`    init() {`
6	`        fn()`	6	`        fn() // error`
7	`        a = A()`	7	`        a = A()`
8	`    }`	8	`    }`
9	`    public fun fn() {`	9	`    public func fn() { }`
10	`        println(a.f)`	10	`}`
11	`    }`		
12	`}`		
13	`fun main() {`		
14	`    val c = C()`		
15	`    c.fn()`		
16	`}`		

Пример на Kotlin успешно компилируется, а при **выполнении** выдает ошибку в строке 10: Cannot invoke "A.getF()" because "this.a" is null.

А в примере на Swift ошибку выдает **компилятор** в строке 6: вызов метода объекта до инициализации объекта.

Разница в том, что Kotlin заявлен как язык с безопасными ссылками (null safe), а в примере не проверяется доступ к неинициализированной переменной, что недопустимо в таком языке. Подробнее об этом можно прочитать в [4].

Краткий вывод для Тривиля: сложно, да еще и небезопасно. Не надо так.

Инициализация в Go

Инициализация объектов в Go существенно отличается от инициализации в Swift и Kotlin, так как

- язык не поддерживает безопасность ссылок (not null safe);
- нет классов, классы эмулируются указателями на структуры;
- нет инициализирующих методов;
- для создания объектов используются композитные литералы (composite literals).

Пример с порядковым номером объекта:

```
 1  package main
 2  import "fmt"
 3
 4  var counter = 0
 5
 6  type Counted = struct {
 7     ord int
 8  }
 9
10  func New() *Counted {
11    counter++
12    return &Counted{ord: counter}
13  }
14
15  func main() {
16    var first = New()
17    fmt.Println(first.ord)
18  }
```

Текст выглядит непривычно для тех, кто привык писать на CLOP[6] языках, поэтому несколько пояснений:

1. первая фаза инициализации в Go выполняется автоматически, так как для переменной любого типа есть значение по умолчанию (0 для int, nil для указателей и т.д.). Кажется, что это хорошо, но это “хорошее” свойство является следствием небезопасных решений, а именно:
   - в языке есть явные указатели;
   - любой указатель может быть “пустым”, то есть иметь значение null[7].
2. конструкторов в языке нет, поэтому вторую фазу инициализации приходится делать через отдельную функцию (строки 10-13);
3. в строке 12 записан композитный литерал: Counted{ord: counter}, который задает значения полей структуры, затем к литералу применяется префиксный оператор &, который должен выдать адрес структуры. При этом срабатывает алгоритм escape-анализа [5], который копирует структуру в кучу, выдавая адрес выделенной в куче памяти.

Рассматривать другие примеры нет смысла, они не добавляют качества.

Краткий вывод для Тривиля: сложно, использует неявные алгоритмы, нельзя напрямую использовать, так как язык существенно другой.

---

6 См. Tony Hoar's billion-dollar mistake.

7 Я стараюсь не использовать транслитерацию английских терминов, но в данном случае, я не знаю лучшего решения. Термин “обобщенные типы и функции” слишком длинный и не очень удачный. Так что следом за другими переводами, я использую термин “дженерик”.

Инициализация в Тривиле

Наш небольшой экскурс в сравнительное изучение инициализации объектов привел нас к тому, что выбирать не из чего, точнее, все варианты плохие:

- подход Swift: обеспечивает безопасность, но очень сложный;
- подход Kotlin: не обеспечивает безопасность и очень сложный;
- подход Go: не обеспечивает безопасность, основан на неявной семантике.

Придется делать свое. Вспомним требования к инициализации, которые были выписаны в начале раздела:

- безопасность,
- простота,
- полнота.

Начнем с простоты, потому что здесь возможны компромиссы, а в обеспечении безопасности — нет. Подход Go существенно проще. Возможно, что это не видно из статьи, так как я не рассматриваю сложные случаи, но хорошо заметно, если сравнить длину раздела в спецификациях языков Go и Swift.

За счет чего сделано упрощение?

За счет использования декларативного языка[8] для создания объекта вместо императивного. В Go инициализация объекта — это всего лишь задание значений полей. В Swift (и во многих других языках) для инициализации используется специальная функция (конструктор или инициализатор). При этом в Go, пока объект не создан, его нельзя использовать, просто потому что нет никакого *магического* идентификатора, типа self или this. Соответственно, нет лишних языковых конструкций и нет стоящей за ними сложной семантики.

Попробуем применить подход к Тривилю (английские идентификаторы оставлены намеренно для упрощения сравнения):

	Тривиль		**Go**
1	`модуль main`	1	`package main`
2	`импорт "стд::вывод"`	2	`import "fmt"`
3		3	
4	`пусть counter := 0`	4	`var counter = 0`
5		5	
6	`тип Counted = класс {`	6	`type Counted = struct {`
7	`  ord := 0`	7	`   ord int`
8	`}`	8	`}`
9		9	
10	`фн New(): Counted {`	10	`func New() *Counted {`
11	`  counter++`	11	`  counter++`
12	`  вернуть Counted{ord: counter}`	12	`  return &Counted{ord: counter}`
13	`}`	13	`}`
14		14	
15	`вход {`	15	`func main() {`
16	`  пусть first = New()`	16	`  var first = New()`
17	`  вывод.ф("$;\n", first.ord)`	17	`  fmt.Println(first.ord)`
18	`}`	18	`}`

Текст очень похож, хотя есть два существенных отличия, как синтаксических, так и семантических:

1. Тривиль обеспечивает выполнение первой фазы инициализации за счет обязательной явной инициализации полей (строка 7), а не за счет инициализации по умолчанию. Инициализация поля может быть отложена, но она всегда явная.
2. Конструктор класса в Тривиле (строка 12) сразу же создает объект класса, здесь нет неявной семантики (нет escape-анализа).

Обеспечивается ли в Тривиле безопасность? Да, обеспечивается, покажем на следующих фрагментах кода.

8 Make it simple as possible, but not simpler — слова А.Эйнштейна, использованные Никлаусом Виртом как лого языка Oberon.

Пример 1. Отсутствие инициализации поля.

В Тривиле инициализация полей обязательна, но ее можно отложить, используя ключевое слово **позже**, но только до конструктора класса. Если в конструкторе класса поле не задано, то компилятор выдаст ошибку:

```
тип К = класс {
   номер: Цел64 = позже
}
пусть к = К{} // ошибка: значение поле ‘номер’ должно быть задано
```

Замечу, что семантическая проверка в данном случае тривиальна, в отличие от того, что приходится делать в компиляторах Swift и Kotlin.

Пример 2. Рекурсивный тип. В отличие от Swift и Kotlin, где программа падает по бесконечной рекурсии, компилятор Тривиля выдает ошибку:

```
тип К = класс {
   номер: Цел64 = позже
}
пусть к = К{} // ошибка: значение поле ‘номер’ должно быть задано }
```

Эту ошибку ловит общий алгоритм, отслеживающий рекурсивные определения, например, рекурсивное определение констант:

```
конст А = Б // ошибка: рекурсивное определение ‘А’
конст Б = А
```

В итоге существенно более простой компилятор оказывается мощнее семантически.

Пример 3. Пример, который обнаруживает дыру в безопасности Kotlin, в Тривиле нельзя написать. Приведу текст на Swift (находит ошибку) и Тривиле (нельзя сделать ошибку):

	Swift		**Тривиль**
1	`class A { var f = 1 }`	1	`тип А = класс { ф := 1 }`
2		2	
3	`class C {`	3	`тип К = класс {`
4	`  var a: A`	4	`  а: А = позже`
5	`  init() {`	5	`}`
6	`  self.fn()  // error`	6	
7	`  a = A()`	7	`фн (к: К) Ф() {}`
8	`  }`	8	
9	`  public func fn() { }`	9	`пусть к = К{а: А{}}`
10	`}`	10	`к.Ф()`

В Тривиле нет возможности обратиться к недостроенному объекту (до завершения конструктора класса), поэтому нет необходимости в семантической проверке, которая совсем не тривиальна. Идеальный способ решения проблемы: нет проблемы, не надо решать.

Итак, из 3-х требований к инициализации, два (безопасность и простота) в Тривиле выполняются. А вот с полнотой есть проблема.

Проблема полноты инициализации в Тривиле

Рассмотрим пример с наследованием (в терминологии ООР):

- в модуле base определяется базовый класс А со счетчиком,
- в модуле main определяется класс В, расширяющий А, а счетчик из базового класса инициализируется начальным значением.

Напишем сначала код на Go:

```
1  package base
2
3  type A struct {
4    count int
5  }
6
7  func (a *A) Get() int {
8    return a.count
9  }
10
11 func Init(x int) A {
12   return A{count: x}
13 }
```

```
1  package main
2  import "fmt"
3  import "base"
4
5  type B struct { base.A }
6
7  func New(x int) *B {
8    return &B{A: base.Init(x)}
9  }
10
11 func main() {
12   fmt.Println(New(3).Get())
13 }
```

В Go для того, чтобы имя могло быть использовано за пределами модуля (другими словами — экспортировано), оно должно начинаться с заглавной буквы. Имена A, Get, Init экспортируются из пакета base, а count (имя поля) не экспортируется. Счетчик скрыт (инкапсулирован) в пакете base. Для того, чтобы значение счетчика можно было задать, добавлена функция Init.

В пакете main описан тип B, содержащий тип A. Это, в некотором смысле, замена наследования, так как методы типа A могут быть применены к объекту типа B. (см. строку main:12). Функция main создает объект типа *B (указатель на структуру B).

Нас интересует строка main:8, где конструируется объект типа B, а значение под-объекта A задается функцией base.Init.

Попробуем теперь написать тот же пример на Тривиле:

```
1  модуль base
2
3  тип A* = класс {
4   count: Цел64 := позже
5  }
6
7  фн (a: A) Get*(): Цел64 {
8   вернуть a.count
9  }
10
11 фн Init*(x: Цел64): A {
12  вернуть A{count: x}
13 }
```

```
1  модуль main
2  импорт "стд::вывод"
3  импорт "base"
4
5  тип B = класс (base.A) {}
6
7  фн New(x: Цел64): B {
8    вернуть B{count: x}
9  }
10
11 вход {
12   вывод.ф("$;\n", New(3).Get())
13 }
```

В отличие от Go, в Тривиле экспорт явный (а не по заглавной букве), экспортируются имена, отмеченные символом '*'. То есть тип A экспортирован (и доступен из других модулей), а поле count — нет. И это приводит к проблемам.

Первая проблема видна уже в модуле base: функция Init возвращает указатель на объект (reference type), а не структуру (value type), так как в Тривиле нет структур, а это дополнительное выделение памяти[9]. Но основная проблема в модуле main, строка 8 — создать объект B невозможно, так как:

- поле count не экспортировано,
- для типа B использовано наследование, а не композиция, как в Go.

Наследование, как и в других языках, приводит к "плоской" модели, доступ к полю базового класса A не отличается от доступа к полю класса B. Кроме того, нет языковых средств, которые позволяют работать с полями базового класса, как с одним целым.

Как можно разрешить эту проблему?

- на уровне примера: экспортировать поле count, тогда код в строке main:8 станет корректным, но будет потеряна инкапсуляция;
- на уровне языка: добавить некоторый языковый механизм, который решит эту проблему. Например, ввести понятие инициализатора, как в Swift (и сделать семантику языка существенно более сложной) или сделать что-то иное (что нетривиально, и, неизвестно, сколько займет времени и усилий).

9 Я рассматриваю очевидное решение, не претендуя на то, что оно лучшее и единственное.

На самом деле у меня есть наметки того, как решить эту задачу, не отказываясь от декларативного подхода, но я не собираюсь добавлять эту возможность в Тривиль. Просто потому, что в компиляторе не нужна возможность скрывать поле в базовом классе от наследников. Об инкапсуляции и о ложном её понимании мы подробнее поговорим в следующей статье. Пока же, решение простое — ничего не делать на уровне языка. В целом, это ограничение, но не мешающее двигаться дальше.

Взаимодействие конструкций

Задача инициализации объектов является хорошим примером, на котором можно показать одну из существенных сложностей в разработке языков, которая называется взаимодействие конструкций (feature interaction). Я упоминал о ней в первой статье, теперь есть материал для немного более подробного рассмотрения. Проблема эта широко известна в очень узком кругу разработчиков языков.

Вернемся к сравнительному анализу и перечислим конструкции, которые используются для инициализации объекта:

- Swift: функция инициализатор (initializer),
- Go: конструктор значения структуры (composite literal),
- Тривиль: конструктор значения объекта (ссылки).

Зададим вот такой вопрос — какие конструкции языка нам пришлось учитывать, думая про инициализацию объектов?

Конструкция	**Swift**	**Go**	**Тривиль**
Безопасность ссылок	есть	нет	есть
Наследование	есть	нет, композиция	есть
Значения полей по умолчанию	нет	есть	нет
Обязательная инициализация полей	нет	нет	есть
Видимость в других модулях	public, protected, private	экспорт	экспорт

Как показывает таблица, на инициализацию влияют, по крайней мере, 5 других языковых конструкций. Это обычная проблема в разработке языка, которая хорошо описывается поговоркой: хвост вытащил, нос увяз.

И еще одно замечание, как видно из таблицы, нет ни одной строки, в которой все три языка используют одинаковое решение. Решение не устоялось, идет активный поиск.

Что нам стоит хеш-таблицу построить?

Я уже упоминал в [2] о том, что для разработки компиляторов нужен тип hash-map, точнее, нужен контейнер, который я называю Словарь, который хранит пары (ключ, значение) и обеспечивает быстрый доступ по ключу. Называть этот контейнер hash-map не совсем корректно, так как это название способа реализации, который, теоретически, может быть разным.

Впрочем, это традиционное название: в Rust и Kotlin такой контейнер называется HashMap, в Go — map, а вот Swift пошел другим путем и назвал его — Dictionary.

С точки зрения реализации, есть существенная разница между Go и остальными языками, в Go map — это встроенный тип, определенный в языке, в остальных языках он реализован в библиотеке. Об этом мы поговорим немного позже.

Важность типа Словарь для компилятора подтверждает статистика: В компиляторе Тривиля тип Словарь используется 23 раза, причем с разными комбинациями типов (ключ, значение). Число комбинаций слишком велико, чтобы писать реализацию каждой комбинации отдельно. Чтобы избежать дублирования кода, есть два пути:

- путь Go: встроить тип в язык;
- путь остальных языков: использовать обобщенные типы или дженерики[10] (generics).

Замечу, что в Go дженерики появились в 2022 году (версия 1.18), через 10 лет после выхода первой версии языка. Полагаю, что, если бы они были добавлены в Go изначально, тип map тоже был бы библиотечным.

Дженерики позволяют писать полиморфный код, то есть код, который работает с множеством наборов типов, а это экономия на разработке, отладке и сопровождении. Дженерик — это языковая конструкция, обычно тип или функция, параметризованная типами (типовыми параметрами). Хороший обзор дженериков сделан в статье [6].

10 Ошибка, например, может быть в коде проверки инварианта.

Замечу, что дженерики — это далеко не единственный способ писать полиморфный код, и это далеко не *серебряная пуля*, но разговор об этом не входит в тему статьи.

С использованием дженериков в библиотеке можно написать (на некотором языке) тип Map<K, V> который настраивается на конкретные типы, например, Map<string, int>.

Вернемся к Тривилю. Что выбрать из двух вариантов: добавление встроенного типа или добавление дженериков? Увы, это выбор между сложным и частным (встроенный тип) и более универсальным и очень сложным (дженерики).

Идти по пути Go мне представляется неверным, добавление еще одного конструируемого типа к трем уже имеющимся увеличивает сложность языка и компилятора, а реализацию этого типа придется писать в коде поддержки исполнения (runtime support) на другом языке программирования, что увеличивает сложность разработки и отладки. Так что этот путь я отбросил сразу.

Подумаем о добавлении дженериков в язык. На опыте разработки и реализации других языков я знаю, что добавление в язык дженериков — это очень непростая задача, как с точки зрения самого языка (синтаксиса и, особенно, семантики), так и реализации в компиляторе.

Так как путь Тривиля — это путь упрощения (make it as simple as possible), то подумаем, можем ли мы предложить некое упрощение дженериков, уменьшающее объем работы, но все же позволяющее написать Словарь (but not simpler[11]). Для того, чтобы понять, что можно упростить, надо определить, в чем сложность дженериков. Не претендуя на полноту, перечислю три проблемы:

- сложность семантики,
- необходимость введения в язык ограничений на типовые параметры (constraints),
- баланс между производительностью и размером кода.

Проблема 1. Сложность семантики

Не вдаваясь в детали, скажу лишь о том, что дженерик-типы добавляют в язык три новых категории типов со своим семантическими правилами:

- обобщенные типы, например, Map<K, V>,
- конкретные типы, например, Map<string, Person>,
- частично-конкретные типы, например, Map<string, V>.

При этом семантические правила для каждого из этих типов обычно весьма нетривиальны. Любопытно, что только конкретные типы существуют во время исполнения программы, обобщенные и частично-конкретные типы — это абстракции, существующие только во время компиляции.

Проблема 2. Нужны ограничения на типовые параметры

Рассмотрим простой пример. Мы хотим написать функцию сортировки sort<T>, строящую упорядоченный список элементов контейнера. Для сортировки содержимого надо[12], чтобы

- для контейнера была определена операция получения числа элементов,
- и операция извлечения элемента, например, по порядковому номеру,
- для элементов была определена операция сравнения.

Очевидно, что любой тип не может быть использован вместо типа T. Для того, чтобы компилятор мог проверить семантическую корректность настройки функции, язык должен позволить задать ограничения (constraints) типового параметра, как в списке выше.

Ограничение может быть задано неким интерфейсом (Go interface, Swift protocol), назовем его Sortable. Часть проблем мы закрыли, но, например, мы хотим использовать функцию также для сортировки встроенных массивов, для которых интерфейс нельзя задать. Придется делать ограничения более сложными, и это может привести к необходимости переопределения операций (operator overloading), определения методов для любых типов языка (в том числе предопределенных) и к прочим веселым последствиям (например, unified type system), которые существенно усложняют язык и компилятор.

Проблема 3. Как найти баланс между производительностью и размером кода

Речь идет о том, сколько экземпляров кода будет создано для типов с разными типовыми аргументами, например: Map<string, int>, Map<int, string> и Map<string, Person>.

Два крайних подхода используют C++ и Java:

- C++ делает реплику кода для каждой настройки дженерика, что дает возможность увеличить производительность за счет оптимизаций кода. Например, если в коде дженерик-функции используется знак “+”, то в настройке этой функции на целый тип будет целое сложение, что, в свою очередь, позволит компилятору сделать дополнительные оптимизации. При этом размер кода будет увеличиваться.
- Java использует один и тот же код для всех настроек и тем самым минимизирует размер кода, но ухудшает производительность, так как во всех случаях будет работать один, неоптимизированный код.

11 Под *декларативным* я (здесь) понимаю язык, которые описывает **что** делать, а не **как** делать.

12 Как я уже писал в [1]: производительность не нужна только в том случае, если она есть. А чтобы она была, “лишнего” ухудшения производительности надо избегать.

Другие языки, например, C#, используют смешанную технологию, когда часть настроек делается через репликацию кода, а часть нет. Очевидно, что смешанный подход усложняет и так непростую реализацию.

Тривиль: Дженерики? Нет, обобщенные модули

Пока я не начал писать этот раздел статьи, я считал, что в Тривиле сделаны дженерики, необычным образом, но все же дженерики. Если подходить строго, дженерики подразумевают использование типового параметра, вместо которого могут быть подставлены разные типовые аргументы. В Тривиле сделано не так, поэтому строго говоря, термин дженерик использовать нельзя, хотя это, безусловно, обобщенные конструкции. Второе существенное отличие, это то, что единицей обобщения является модуль.

Начну с примера на Тривиле. Рассмотрим простейший контейнер: обобщенный стек (см. стд/контейнеры/стек.tri [7]):

```
модуль стек

тип Элементы = []Элемент

тип Стек* = класс {
   элементы = Элементы[]
   верх: Цел64 := 0 // индекс свободной ячейки
}

фн (с: Стек) положить*(э: Элемент) {
   если с.верх = длина(с.элементы) {
      с.элементы.добавить(э)
   } иначе {
      с.элементы[с.верх] := э
   }
   с.верх++
}
// другие методы
```

Этот модуль выглядит обычным образом, без каких-либо дополнительных конструкций, за одним исключением — тип Элемент в нем не описан, он просто отсутствует. Сам по себе этот модуль недоопределен и не может быть использован, при попытке компиляции будет выдана ошибка.

Для того, чтобы его использовать, надо добавить то, что было не определено. Для этого есть специальная синтаксическая конструкция (см. стд/контейнеры/стек/стек-стр.tri в [7]):

```
настройка "стд::контейнеры/стек"
модуль стек-стр

тип Элемент = Строка
```

Настройка должна добавлять то, что отсутствует в исходном модуле. Настроенный модуль можно импортировать обычным образом и использовать:

```
модуль пример

импорт "стд::контейнеры/стек/стек-стр"

вход {
   пусть с = стек-стр.Стек{}
   с.положить("Вася")
}
```

Конкретный или настроенный модуль собирается из двух частей. Сборка двух частей — это простое действие, и оно уже реализовано в компиляторе, так как любой модуль в Тривиле может состоять из нескольких исходных файлов. При компиляции любого модуля для каждого файла модуля строится АСД, в вершине которого узел для модуля, а потом верхние узлы сливаются в один. Для слияния двух узлов модуля надо соединить импорты и описания верхнего уровня.

После слияния следующие этапы компиляции, а именно семантическая проверка и генерация кода - выполняется над модулем целиком. В случае обобщенного модуля, если настройка неверна, например, задан неподходящий тип, то компилятор выдаст ошибку.

Вспомним теперь проблемы дженериков, которые мы рассматривали выше.

- сложности семантики.
  - Тривиль: сложности нет. Есть только конкретные типы, обработка которых не отличается от обычных типов, собственно, они и есть обычные типы;
- ограничения на типовые параметры (constraints).
  - Тривиль: отдельная синтаксическая конструкция для ограничений не нужна, собранный модуль или скомпилируется (настройка верная), или нет (настройка неверная).
- баланс между производительностью и размером кода.
  - Тривиль: проблемы нет, выбор сделан в сторону производительности. Теоретически проблема с размером кода может быть (как и в C++), но для Тривиля это несущественно.

Как видим, этот подход, вместо решения проблем, их выкидывает: нет проблемы, не надо решать.

Трудоемкость решения минимальна, в компиляторе пришлось написать 120 строк (сто двадцать строк кода) для реализации обобщенных модулей, при том, что реализация дженериков в более-менее полном виде потребовала бы написать несколько тысяч строк кода.

Любопытно, что есть еще, по крайней мере, один язык, в котором используется обобщение на уровне модулей. Это язык C3 [8], о котором я узнал уже после того, как язык и компилятор Тривиля были сделаны. К сожалению, я не нашел в документации по этому языку обоснования использования обобщенных модулей.

Неожиданным следствием подхода Тривиля является то, что модуль может быть настроен не только типом, но и константой, функцией и даже методом. Например, для настройки контейнера Словарь надо указать два типа (ключ, значение) и хеш-функцию.

Возникает вопрос: действительно ли этот подход является хорошим? Не совсем, он часто неудобен в использовании (необходимо писать отдельную настройку), есть и другие недостатки. Подход скорее остроумный, чем хороший. Впрочем, он позволил с минимальными усилиями реализовать необходимый контейнер и дал возможность идти дальше.

Я вполне допускаю, что его можно довести до хорошего, но тут надо думать.

Недостатки или точки роста

В статье рассмотрены языковые конструкции, для которых удалось найти простые, ограниченные, но достаточные решения, что позволило минимизировать трудозатраты. В следующей статье будет рассмотрен график разработки, и результат этих усилий будет наглядно виден.

Замечу еще, что внесенные ограничения в языковые конструкции можно считать недостатком языка. С другой стороны, любое ограничение создает **точку роста**. Все что сделано в Тривиле хорошо, может быть использовано в следующих языках как есть (впрочем, это не значит, что обязательно будет использовано как есть), а вот ограничения, такие как отсутствие итераторов, ограничение инициализации объектов, недостатки обобщенных модулей, — это то, на чем стоит сосредоточить внимание и улучшить в следующих языках.

Следующая статья будет посвящена реализации языка. В ней будет рассмотрена и обоснована архитектура компилятора и, как уже было сказано, график разработки и тоже с обоснованиями..

Литература

1. Недоря А. Е. Разработка языка Тривиль. Первые шаги к семейству языков. Часть 1, 2024. http://digital-economy.ru/stati/razrabotka-yazyka-trivil-pervye-shagi-k-semeystvu-yazykov-chast-1
2. Недоря А. Е. Разработка языка Тривиль. Часть 2, 2024. http://digital-economy.ru/stati/razrabotka-yazyka-trivil-chast-2
3. The Swift Programming Language. Two-Phase-Initialization. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/initialization/#Two-Phase-Initialization
4. Kogtenkov A.V. Null safety benchmarks for object initialization. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). 2017;29(6):135-150. https://doi.org/10.15514/ISPRAS-2017-29(6)-7
5. Language Mechanics On Escape Analysis, https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-escape-analysis.html
6. T. Hume, Models of Generics and Metaprogramming: Go, Rust, Swift, D and More. https://thume.ca/2019/07/14/a-tour-of-metaprogramming-models-for-generics/
7. Тривиль: публичный репозиторий. https://gitflic.ru/project/alekseinedoria/trivil-0

8. C3. Generics. https://c3-lang.org/generic-programming/generics/

Спецификации языков программирования

[Eiffel] ECMA International: Standard ECMA-367 – Eiffel: Analysis, Design and Programming Language 2nd edition (June 2006). https://ecma-international.org/publications-and-standards/standards/ecma-367

[Go] The Go Programming Language Specification. https://go.dev/ref/spec

[Kotlin] Kotlin language specification. https://kotlinlang.org/spec/kotlin-spec.html

[Oberon] The Programming Language Oberon. https://people.inf.ethz.ch/wirth/Oberon/Oberon.Report.pdf

[Rust] The Rust Reference. https://doc.rust-lang.org/reference/

[Swift] The Swift Programming Language.
https://docs.swift.org/swift-book/documentation/the-swift-programming-language/

References in Cyrillics

1. Nedorya A. E. Razrabotka yazy`ka Trivil`. Pervy`e shagi k semejstvu yazy`kov. Chast` 1, 2024. http://digital-economy.ru/stati/разработка-языка-тривиль-первые-шаги-к-семейству-языков-часть-1
2. Nedorya A. E. Razrabotka yazy`ka Trivil`. Chast` 2, 2024. http://digital-economy.ru/stati/разработка-языка-тривиль-часть-2
3. Trivil: publichny`j repozitorij. https://gitflic.ru/project/alekseinedoria/trivil-0

Ключевые слова

язык программирования, семейство языков программирования, разработка языков программирования, компилятор, прототипирование компиляторов, энергосбережение разработчика

Недоря Алексей Евгеньевич, к.ф.-м.н.
ORCID 0000-0001-8998-7072
aleksei.nedoria@yandex.ru
Телеграмм канал: t.me/vorchalki_o_prog

Aleksei Nedoria, Development of programming language Trivil. Part 3. Balance.

Keywords

programming language, family of programming languages, programming language development, compiler, compiler prototyping, developer energy saving

DOI: 10.34706/DE-2025-01-03
JELclassification

Abstract

The article is the third in a series of articles that describe the development of the Trivil programming language. In previous articles, the goals of the language, the requirements for the language were defined and the main language solutions were described. This article examines complex language constructs, the development of which focused on the balance between the completeness of the construct and the complexity of the language and compiler. Like the entire series, the article is aimed, for the most part, not at the programmer who uses the language, but at the developer of programming languages.