Разработка типовой системы языка программирования приложений

А.Е.Недоря

Аннотация. Типовая система является основой любого языка программирования. Полнота, простота, лаконичность и ортогональность типовой системы существенно влияет на продуктивность разработчика, скорость обучения, понимаемость текстов и производительность. В статье кратко описывается инновационная система типов для языка разработки приложений.

Ключевые слова: языки программирования; типовая система; конструкторы типов; программная экосистема.

1. Введение

Типовая система является основой языков программирования, в том числе и динамических языков. Решения, принятые при разработке типовой системы языка, существенно влияют на

- проектирование остальных частей языка
- на трудоемкость реализации компилятора и инструментария

и на важнейшие характеристики языка в целом, такие как:

- продуктивность разработки
- простоту обучения и использования
- выразительность и лаконичность
- эффективность (скорость, память, заряд аккумулятора)
- взаимодействие с другими языками (interop)

Перечисленные пункты достаточно очевидны, возможно, кроме последнего. Современные приложения всегда содержат или используют компоненты (библиотеки, фреймворки, сервисы), написанные на разных языках программирования. Например, для Android-приложений набор используемых языков обычно включает Java/C++ [1, 2], и часто также Kotlin [3], JavaScript [4] и другие языки.

Можно уверенно сказать, что в большинстве программ происходит взаимодействие между компонентами, написанными на разных языках. Эффективность такого взаимодействия во многом зависит от типовой системы. Взаимодействие языков с разными типовыми системами приводит к множеству вопросов, например:

- Нужно ли преобразование данных, если обработка данных выполняется в компоненте, написанной на другом языке?
- Можно ли передать объект класса и использовать его?
- Можно ли передать функциональный объект?
- Как обеспечить корректную работу памяти, если один из взаимодействующих языков использует автоматическое управление памяти, а другой – ручное?

В IT отрасли существует множество частичных решений для поддержки взаимодействия между языками, таких как JNI [5], но общего решения или стандарта не существует. Одной из попыток решить проблему в целом является Common Language Infrastructure (CLI) [6] для .NET, но разработчики пытаются объединить существующие (и очень разные) языки и сталкиваются с множеством противоречий и несовместимостей.

2. Семейство языков

Проблема взаимодействия между языками очень важна для отрасли, и лучшее, на наш взгляд, решение – это **разработка семейства совместимых языков** [7]. Совместимость должна быть обеспечена унификацией:

- системы типов;
- поддержки исполнения (run-time)
- семантики (и, в какой-то мере, синтаксиса)
- распределенной вычислительной моделью для гетерогенных систем

Мы предполагаем, что семейство должно состоять из языков 4-х уровней:

Уровень	**Характеристика**	**Область применения**	**Примеры**
L1	Скриптовый язык высокого уровня	Пользовательские приложения	JavaScript [2], Python [8]
L2	Компилируемый язык высокого уровня	Пользовательские приложения, библиотеки, фреймворки	Kotlin [3], Java [1], Go [9]
L3	Компилируемый язык среднего уровня	Приложения реального времени, сервисы, библиотеки	Swift [10], Go
L4	Язык системного уровня	ОС, базовые библиотеки	C++ [2], Rust [11]

Тот, кто создаст семейство хорошо согласованных языков, получит значительное преимущество в производительности разработчиков, производительности приложений и привлекательности экосистемы. Мы ожидаем, что семейство языков позволит, по крайней мере, удвоить скорость разработки.

В настоящее время мы разрабатываем язык уровня L2, продумывая одновременно языки остальных уровней, в первую очередь – L1. Первая публикация на эту тему [12].

3. Типовая система языка уровня L2

Далее мы будем рассматривать типовую систему L2, но, при этом учитывать, что эта типовая система должна быть частью **унифицированной системы типов семейства**. В более широком контексте (но не в данной статье) мы также должны учитывать пригодность типовой системы для **взаимодействия (interop)** и для **распределенных вычислений**.

В данной статье мы будем рассматривать **«конструкторы» типов**, которые позволяют сконструировать новый тип из уже имеющихся. Базовые (примитивные) типы и обобщенные типы (generic types) мы опустим, так как они не расширяют набор конструкторов.

Для оценки типовой системы и для сравнения с типовыми системами других языков мы будем использовать следующие критерии:

- **полнота** или выразительность, то есть возможность сконструировать любой «полезный» тип данных,
- **ортогональность** или, в каком-то смысле, принцип единственной ответственности (single responsibility) каждого конструктора,
- **лаконичность** (краткость нотации, минимизация церемоний)
- **простота понимания и обучения** (readability, learning curve)
- **типобезопасность** (type safety)

Эти критерии, увы, не количественные, кроме, разве что, «лаконичности», но мы будем их использовать, ввиду того, что это лучшее, что мы сейчас имеем. Важно еще отметить, что улучшение оценки по одному критерию может ухудшить оценку по другому критерию, например, сокращение нотации записи может приводить к осложнению понимания.

Попробуем использовать критерий «полноты», как инструмент определения набора конструкторов типов. Возьмем несколько современных и популярных языков: Swift [10], Kotlin [3] и Go [9], и выпишем типы, которые в них есть:

1. класс
2. структура
3. тип функции
4. тип-произведение (кортеж)
5. тип-сумма (enum, enum class)
6. опциональный тип (optional or nullable)
7. интерфейсный тип
8. массив (фиксированного размера)
9. динамический массив
10. тип указателя (Go pointer)
11. отображение (Go map)

Этот список нужно почистить перед употреблением:

1. вычеркнем тип указателя, который должен быть в языках уровня L3 и L4, но является низкоуровневым для L2.

2. вычеркнем тип map, который, очевидно, появился ввиду отсутствие обобщенных типов в Go (см. предложение для Go 2.0 [13])
3. кажется, что также надо убрать динамический массив, так как он тоже может быть заменен библиотечным обобщенным типом. Но простые конструкторы должны быть реализованными простыми способами (забивать гвозди микроскопом можно, но неудобно и дорого).

Далее заметим, что

- enum (Swift) и enum class (Kotlin) – это не чистый алгебраический тип-сумма [14], а смесь двух существенно разных на наш взгляд понятий: тип-перечисление и тип-сумма. На наш взгляд, разработчики этих языков совершили принципиальную ошибку. К сожалению, формат статьи не позволяет углубиться в эту тему. Существенно важным для типовой системы является тип-сумма, или вариантный тип, именно его мы включим в обязательный набор конструкторов.
- тип interface (Go) и типы protocol (Swift), interface (Kotlin), несмотря на внешнюю схожесть, обеспечивают разные варианты полиморфизма, а именно:
    - множественное наследование интерфейсов – Swift, Kotlin
    - утиная типизация – Go [15]

  На наш взгляд, оба варианта полиморфизма нужны в L2 языке.

Если учесть все эти соображения, мы придем к обязательному набору из 10 конструкторов типов. Сравним типовые системы рассматриваемых языков и языка L2 (последняя колонка):

#	**Тип**	**Go**	**Swift**	**Kotlin**	**L2**
1	class type	- *(partly as *struct)*	class	class	class
2	struct type	struct	struct	- *(partly data class)*	struct
3	array (fixed size)	[4]int	-	-	[4]i32
4	dynamic array	[]int	- via library (as class)	- via library (as class)	[*]i32
5	function type	+	+	+	+
6	tuple (algebraic	±	+	-	{T1,

	product)	(partly: function result)		via library (as set of classes)	T2, …}
7	variant (algebraic sum)	-	enum	enum class	(T1 | T2 …)
8	optional/nullable type	-	T?	T?	(T | none)
9	interface (inheritance)	-	protocol	interface	spec
10	interface (duck typing)	interface	-	-	spec
	Число конструкций: языковые + библиотечные	6	7 + 1 = 8	5+(1+2+N) > 8	8

Существенные отличия L2 от типовых систем рассматриваемых языков:

- выделение чистого типа-сумма или вариантного типа (строка 7)
- использование вариантного типа для обеспечения null safety [16, 17, 18] – отсутствие отдельного конструктора для optional/nullable (строка 8)
- использование типа спецификации (spec) для поддержки как утиной типизации, так и множественного наследования интерфейсов (строки 9, 10)

Кратко пройдем по вышеперечисленным критериям оценки типовых систем.

Полнота	8 конструкторов L2 реализуют весь обязательный набор конструкторов типов. Как видно из таблицы, типовые системы других языков не полны, при этом число конструкторов в Swift и Kotlin не меньше, чем в L2.
Ортогональность	Есть. Использование одной конструкции для вариантных и опциональных типов естественно, оно не приводит к дополнительной семантике. Так же естественно использование одной конструкции для поддержки двух сторон полиморфизма.
Лаконичность	По приведенным фрагментам трудно судить о лаконичности, оставим оценку до выхода языка и компилятора в Open Source.
Простота понимания и обучения	По сути, ни одна из конструкций не придумана нами, и все они достаточно привычны с точки зрения и синтаксиса, и семантики. Более того, на наш взгляд типовые системы всех трех используемых для сравнения языков существенно сложнее для понимания. Но это тема для отдельного разговора.

Типобезопасность	Мы считаем это требование обязательным для выполнения, но детальный разговор о нем выходит за рамки статьи.

4. Заключение

В статье кратко представлена типовая система экспериментального L2 языка. Типовая система L2 является вполне инновационной, но её новизна заключается скорее в тщательном подборе и балансе конструкций, нежели в новизне самих конструкций. Многие важные аспекты пришлось оставить за рамками статьи, например, базовые типы, обобщенные типы, типобезопасность, поддержка распределенных систем, поддержка параллелизма, особенности типовых систем других языков семейства.

Описанная типовая система в настоящее время проверяется с использованием экспериментального компилятора.

Заметим, что в процессе разработке языка рабочей группой под управлением автора была проделана огромная исследовательская работа, включая сравнительный анализ языков (Java, Kotlin, Swift, Dart [19], Go, C# [20], Rust, JS, …), подготовку и обсуждение предложений, оценку их и экспериментальную реализацию (около 500 документов, из которых 2/3 затрагивают те или иные аспекты типовой системы)

Еще одну тенденцию мы считаем необходимым отметить, пока на уровне предположения: типовые системы современных языков приближаются к «идеальной типовой системе», которая может стать основой для унифицированной, многоязыковой среды разработки с близким к идеальному взаимодействию между языками. На наш взгляд, типовая система L2 делает еще один шаг к «идеальной».

Список литературы

[1]. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, The Java® Language Specification. Available at: https://docs.oracle.com/javase/specs/jls/se8/html/index.html, Version of 2015-02-13.

[2]. C++ language. Available at: https://en.cppreference.com/w/cpp/language, accessed 18.10.2021.

[3]. Kotlin language specification. Available at: https://kotlinlang.org/spec/introduction.html, accessed 18.10.2021.

[4]. JavaScript reference. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference, accessed 18.10.2021.

[5]. Java Native Interface 6.0 Specification. Available at: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html, accessed 18.10.2021.

[6]. ECMA-335. Common Language Infrastructure (CLI). Available at: https://www.ecma-international.org/publications-and-standards/standards/ecma-335/, Version of June 2012.

[7]. Недоря А.Е.. Триада языков программирования. Available at: http://xn--80aicaaxfgwmwf3q.xn--p1ai/?p=298, published 20.09.2018

[8]. The Python Language Reference. Available at: https://docs.python.org/3/reference/, accessed 18.10.2021.

[9]. The Go Programming Language Specification. Available at: https://golang.org/ref/spec, Version of July 26, 2021.

[10]. The Swift Programming Language. Language Reference. Available at: https://docs.swift.org/swift-book/ReferenceManual/AboutTheLanguageReference.html, accessed 18.10.2021.

[11]. Klabnik S., Nichols C.. The Rust Programming Language. Available at: https://doc.rust-lang.org/book/title-page.html, accessed 18.10.2021.
[12]. Недоря А.Е.. Разработка языка: OOP or not OOP or better OOP. Труды ИСП РАН, том 1, вып. 2, 2019 г., стр. 15–19.
[13]. Ian Lance Taylor, Generics proposal. Available at: https://go.dev/blog/generics-proposal, Version of 12 January 2021.
[14]. Algebraic data type. Available at: https://en.wikipedia.org/wiki/Algebraic_data_type, accessed 18.10.2021.
[15]. Interface in Go. Available at: https://golangbyexample.com/interface-in-golang/, Version of July 18, 2020.
[16]. Void Safety (null safety). Available at: https://en.wikipedia.org/wiki/Void_safety, accessed 18.10.2021.
[17]. Sound null safety (Dart). Available at: https://dart.dev/null-safety, accessed 18.10.2021.
[18]. Null Safety (Kotlin). Available at: https://kotlinlang.org/docs/null-safety.html, accessed 18.10.2021.
[19]. Dart Programming Language Specification. Available at: https://dart.dev/guides/language/specifications/DartLangSpec-v2.10.pdf, Version 2.10 of April 9, 2021.
[20]. C# 6.0 draft specification. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction, accessed 18.10.2021.