Компонентный ассемблер для цифрового пространства

Оглавление

Введение

В последнее время слова «цифровое пространство» звучат постоянно, но, мы, далеко не всегда, осознаем существенную странность нашего движения к цифровому пространству.

Если присмотреться, цифровое пространство с точки зрения оборудования (железа) уже в достаточной мере построено, достаточно обратить внимание на насыщенность мира вычислительными узлами, средствами коммуникации и облачными сервисами.

В то же время, единое программное цифровое пространство не существует. Существует несколько экосистем, слабо связанных друг с другом. Разработка внутри одной экосистемы (iOS, Android, Java, …) существенно проще, чем разработка, затрагивающая несколько экосистем. Замечу, что границы между экосистемами искусственно выстроенные, их могло не быть, так как использованное оборудование, в сущности, универсально.

Так как я осознаю себя, в первую очередь, инструментальщиком (компиляторы, технологии разработки и средства разработки) и архитектором, я вижу странность в движении к цифровому пространству именно в своей области компетенции.

Возьмем простой пример. Язык Go [1], по многим характеристикам, является вполне подходящим для создания частей цифрового пространства. Особенно, если обратить внимание на подход к ООР (интерфейсы, отсутствие наследования и утиная типизация). Но, одновременно, в Go постулируется, что это язык для разработки (монолитных, статически линкованных) программ, что странно: шаг вперед, два шага назад.

Само понятие монолитной программы является достаточно абсурдным в наше время. Сейчас трудно найти приложение, которое является замкнутым – не использует облачные сервисы, микро-сервисы, не обновляется, и т.д. По сути, любое современное приложение – это

распределенная система, состоящая из нескольких (множества) распределенных компонент, работающих на разных устройствах, часто виртуальных.

Если мы хотим разрабатывать распределенные системы, то существенное внимание должно быть уделено взаимодействию между компонентами, а взаимодействие должно быть гибким, надежным и безопасным. Программирование в большом (между компонентами), должно быть таким же надежным, как программирование в малом.

Нужен ли нам вообще язык программирования для поддержки программирования межкомпонентного взаимодействия? Или нужна, скорее, некая надстройка, типа IDL (Interface Definition Language) вместе с обеспечивающими взаимодействие инструментами?

У меня нет определенного ответа на этот вопрос, но есть уверенность в другом – если мы хотим создать современную систему разработки распределенных программ, надо проводить эксперименты, а для того, чтобы проводить эксперименты просто и быстро, нужен экспериментальный язык и экспериментальная среда программирования.

Попытка делать эксперименты на каком-то из существующих языков, к сожалению, как показывает опыт автора, приводит к тому, что большая часть времени тратится на обходы ограничений существующих языков, компиляторов и сред исполнения (RTS).

Если мы примем, в качестве рабочего, предположение, что экспериментальный язык ускорит разработку (а не замедлит её, так как язык/компилятор/RTS тоже надо разрабатывать), то можно перейти к перечислению требований к такому экспериментальному языку.

Требования к экспериментальному языку

Основные и очевидные требования:

1) Язык должен поддерживать **надежное взаимодействие** между **независимо написанными компонентами**
2) **Мультиплатформенность**: Язык должен позволять делать программы, которые работают на всех (большинстве) современных устройствах/операционных системах.
3) **Простота и гибкость**: Язык должен быть предельно простым (as simple as possible, but not simpler) и легко изменяемым.

Добавлю еще требования, которые считаю обязательными, но которые могут быть спорными для других:

4) Разработку языка надо вести с нуля (а не дорабатывать какой-то из существующих языков)
5) Язык не должен использовать сборку мусора (No GC)
6) Компилятор должен порождать нативный код (No VM)
7) Компилятор и RTS должны позволять (достаточно легко) использовать код, написанный на других языках, как минимум, на Си.

Поясню сначала вкратце пункты кроме первого – основного пункта.

Пояснения к требованиям

2) Мультиплатформенность

Без мультиплатформенности нет смысла говорить о распределенных программах. В идеале, мы должны уметь запускать систему, состоящую из множества компонент, на любой конфигурации устройств, в том числе на одном устройстве или на тестовом стенде. С естественным уточнением,

что устройства должны иметь достаточное количество ресурсов и необходимую функциональность.

Конкретизируем требование:

- Компилятор должен строить код на любые процессоры.
- Код компоненты должен быть существенно независим (изолирован) от ОС устройства. Иначе мы будем сильно ограничены в выборе целевых устройств для компоненты. Требование очень важно (и не просто в реализации), но говорить о нем нужно отдельно.

3) Простота и гибкость

Очевидно, что нужно быстро/легко добавлять возможности в язык/компилятор/RTS и легко отбрасывать то, что не прошло проверку на практике.

Менее очевидно, что мы должны предельно ограничивать себя в добавлении новых возможностей или новой функциональности: все, без чего можно обойтись, должно быть вынесено за границы экспериментального языка.

Замечу, для упрощения экспериментов, желательно разрабатывать компоненты на том же языке, который задает взаимодействие между компонентами, просто чтобы уменьшить число сущностей, о которых надо думать и о средствах выражения мыслей.

Итого, для экспериментов нужен язык **минимального достаточного уровня**, обеспечивающий возможности:

1) Разработки компонент (достаточно легко и надежно)
2) Конструирования распределенной системы из компонент (достаточно легко и надежно)

Назовем языки такого класса «**компонентными ассемблерами**».

Замечу, что в слове «ассемблер» есть игра смыслов:

- Явный смысл: ассемблер – сборщик, монтажник. В нашем случае, инструмент для сборки (конструирования) системы из компонент.
- Неявный и более привычный смысл: язык программирования низкого уровня.

4) Разработку языка надо вести с нуля

Это требование больше философско-психологическое, чем технологическое. Если у нас уже что-то есть, то выбросить из него даже то, что считаешь лишним, достаточно трудно. Проще объяснить себе, что выбрасывать не надо. Леность мышления…. А вот если надо что-то добавить, то тут лень может играть позитивную роль, подталкивая к поиску минимально достаточных решений.

Впрочем, технологический аспект тут тоже есть. Большинство программ в нашем мире плохо продуманы и плохо реализованы, выкинуть что-то из такой программы может быть сложнее, чем добавить.

Замечу, что «с нуля» относится только к программному коду, но никоим образом не к идеям!

5) Язык не должен использовать сборку мусора (No GC)

Сборка мусора, для большинства решаемых задач, очень нужный механизм, упрощающий программирование и повышающий надежность. Но не для всех, для задач системного уровня сборка мусора может быть непригодна. Если мы делаем экспериментальный язык для программирования компонент, наличие GC ограничит спектр решаемых задач.

В идеале надо говорить о трех языках – **триаде языков** или, скорее о трех группах языков:

- Языки системного уровня (нативный код, без сборки мусора)
- Языки прикладного уровня (нативный код, сборка мусора)
- Языки для притирки (склейки) компонент (скриптовые, сборка мусора).

Язык системного уровня задает основу, остальные языки должны быть «совместимы». О языке прикладного уровня и о скриптовом языке есть смысл думать тогда, когда язык системного уровня будет более-менее определен.

Желательно, чтобы отказ от сборки мусора, в нашем экспериментальном языке не приводил к усложнению программирования и снижению надежности. Тут мы можем опереться на решения, используемые в Rust [2], Cyclone [3] и подход, уже используемый автором в Вир [4]. Говоря коротко, решение заключается во введении понятий: владелец указателя и владелец выделенной памяти, и использовании region-based и thread-based memory management (см. [5]). Подробно об этом надо говорить отдельно.

6) Компилятор должен порождать нативный код (No VM)

Это требование обусловлено двумя соображениями:

1) Упрощение взаимодействия – VM добавляет барьер, осложняющий взаимодействие
2) Производительность

На мой взгляд, при современном состоянии технологии, использование VM (с интерпретацией кода) является ошибкой. Безусловно JVM и другие VM дали нам очень много с точки зрения развития технологий (очевидный пример – JIT компиляция), но сейчас мы уже умеем достигать те же результаты без использования VM и можем убрать лишний уровень сложностей.

Добавлю, если бы те усилия, которые вложены в Java, JVM и сопутствующие технологии были вложены в нативное мультиплатформенное программирование, мы были бы гораздо ближе к полету к звездам.

7) Использование «внешнего» кода

«Нельзя объять необъятное» и не нужно. Надо построить объединяющую экосистему, а не систему, заменяющую то, что есть. Нужна эволюция, а не революция.

Упоминание Си тоже очевидно, Си используется в настоящее время как практически единственный мультиплатформенный язык описания интерфейса библиотек (не говоря уже о том, что сам Си - это переносимый ассемблер).

Переходим к главному

Вернемся к главному пункту: экспериментальный язык - **компонентный ассемблер** должен поддерживать **надежное взаимодействие** между **независимо написанными компонентами**.

Рассмотрим несколько примеров, часть из которых упоминались в [6].

Пример 1: Человек в музее подходит к экспонату со смартфоном, и, программа «Гид» на смартфоне позволяет ему получить информацию об экспонате (авторство, историю, легенды, стоимость, …). При этом, в соответствии с обобщенным принципом единственной ответственности (SRP), «Гид» ничего не знает ни об экспонате, ни о музее (ни даже о их существовании), а компонента «виртуализации экспоната» ничего не знает о программе «Гид».

Какие возможности должно предоставить цифровое пространство, чтобы «Гид» и «Виртуальный образ экспоната» могли взаимодействовать?

1) Должна быть возможность поиска локально доступных компонент, с разным определением «локальности». В данном случае «локально» – относится к 3-х мерному физическому пространству.
2) Должна быть возможность выбора компоненты с нужными интерфейсами или запрос к компоненте «Что ты умеешь?», чтобы узнать об имеющихся интерфейсах (USB like).
3) Должна быть возможность выбрать один из интерфейсов и подключится к нему, чтобы перейти собственно к взаимодействию, как последовательности команд (запросов) и ответов.

Пример 2: В том же музее к тому же экспонату подъезжает робот-уборщик и чистит экспонат так, как это позволено для этого конкретного экспоната. Естественно, что информация о том, что позволено, доступна только через виртуальный образ. Пример показывает естественную необходимость множества интерфейсов.

Пример 3: В систему управления предприятием добавлен плагин (компонента). В процессе установки новая компонента «обнюхала» остальные части системы, сообщила о том, к каким подсистемам она может подключиться, запросила права доступа и подключилась к тем подсистемам, к которым ей разрешили подключиться, увеличив тем самым возможности всей системы. Компонента может быть написана сторонним разработчиком, в другой среде разработки.

Пример 4: Пользователь установил плагин для браузера, и получил новую возможность. Вот только плагин написан для любого браузера, или для более широкого класса программ, поддерживающий нужный плагину интерфейс (или набор интерфейсов).

Примеры показывают спектр применения: в примерах 1, 2 взаимодействующие компоненты находятся на разных устройствах, связанных сетью. В примере 3 компоненты работают в рамках одного сервера или кластера. В примере 4 – на одном устройстве.

Общим в этих примерах являются шаги перехода к конкретному взаимодействию. Пусть есть активная (устанавливаемая компонента), для того, чтобы перейти к взаимодействию, она

1) Выбирает компоненту (из находящихся «рядом» в каком-то смысле), с которой взаимодействие возможно, назовем это действие: **Выбор компоненты**.
2) Проверяет наличие подходящего интерфейса и «подключается» к нему: **Подключение к компоненте**. В общем случае, сначала выбирает подходящий интерфейс из имеющихся, потом выполняется проверка и подключение.

В примере 4, по сути, есть только две компоненты – браузер и плагин, поэтому Выбор компоненты прост, а вот Подключение необходимо.

Пример 4 позволяет плавно перейти к примерам 5 и 6, иллюстрирующие другие случаи с простым **Выбором**.

Пример 5: Распределенная программная система собрана из компонент по некой схеме (топологии). В этом случае понятие «рядом» определяется схемой, а Выбор компоненты явно задан, так как связи компонент задаются схемой. **Подключение** необходимо.

Пример 6: то же самое, в случае «большой» компоненты, которая собрана из (меньших) компонент с использованием явной схемы, например, как в [4]. В этом случае тоже нет Выбора, а есть Подключение.

В этой статье мы сосредоточимся только на Подключении компоненты. Выбор компоненты может существенно отличатся для разных определений понятия «рядом», и реализация выбора в меньшей степени связана с языком программирования.

Подключение компоненты для использования

Дано:

- Компонента А, которой для работы надо использовать компоненту Б
- Компонента Б, которая предположительно реализует функциональность, необходимую А

Замечу, что если взаимодействие двухстороннее, то есть компонента Б так же использует компоненту А, то мы рассматриваем эту ситуацию, как два подключения: Б к А и А к Б.

Рассмотрим случай, когда компонента А уже имеет доступ к компоненте Б (см. примеры 4, 5, 6). Или, переходя на уровень исполнения, компонента А каким-то образом получила «указатель» на компоненту Б.

Для того, чтобы ситуация не выглядела тривиальной, вспомним, что мы говорим о взаимодействии между независимо написанными (и независимо скомпилированными) компонентами. В общем случае, компоненты могут быть написаны на разных языках, и в коде могут использоваться разные бинарные соглашения (application binary interface, соглашения о вызовах, метаданные и т.д.).

Нам надо выделить **минимальный уровень совместимости**, которые позволит компоненте А использовать компоненту Б.

Для использования:

- Компонента А должна каким-то образом получить адрес функции компоненты Б;
- Причем, только в том случае, если сигнатура функции компоненты Б (список параметров, тип результата, соглашение о вызове) совпадает с сигнатурой, ожидаемой компонентой А.

Если это сделано, то далее А может вызвать функцию из Б с нужными параметрами.

Вариант 1. Реализация через интерфейсы (Go like)

На Go это могло бы выглядеть так:

В компоненте А задан тип интерфейса:

```
type ИнтерфейсБ interface {
        НужнаяКоманда (….)
}
```

И далее вызов:

```
указательБ.( ИнтерфейсБ).НужнаяКоманда(…)
```

Для выполнения конструкции:

```
        указательБ.( ИнтерфейсБ)
```

должна быть:

- выполнена проверка на то, что динамический тип указательБ реализует интерфейс (все команды интерфейса)
- построена виртуальная таблица методов, через которую потом будет делаться вызов.

Это может быть сделано только динамически на основе метаданных. При этом, должны использоваться

- метаданные компоненты А – сигнатура метода интерфейса
- метаданные компоненты Б – наличие и сигнатура функции

И мы сталкиваемся с проблемой: метаданные компоненты А и Б должны быть совместимы. Замечу, что компоненты А и Б могут быть скомпилированы в разное время, разными компиляторами и использовать разные версии RTS, что приводит к невозможности корректно выполнить проверку и построение виртуальной таблицы.

Go избегает таких проблем, ограничиваясь статической сборкой исполняемой программы. Простой, хотя и несколько страусовый способ…

Очевидно, что решение с метаданными в нашем случае не подходит, так как требует высокого уровня совместимости компонент.

Вариант 2. Минимизация уровня совместимости

Необходимо решение, которое инкапсулирует необходимые действия внутри компонент, так чтобы стандартизованный интерфейс компоненты был минимальным, и, одновременно, позволял выполнять любую (совместимую) команду любой компоненты.

Для этого достаточно, зафиксировать бинарное представление компоненты.

Определим Компоненту, как указатель на структуру, в которой первым полем является указатель на функцию «Запросить команду».

Используем близкую к Go нотацию:

```
type Компонента * {
        «Запросить команду» func (имя, сигнатура) Команда
}
```

Каждая компонента должна включать реализацию функции «Запросить команду», которая
- по имени находит функцию компоненты,
- проверяет совпадение сигнатур
- возвращает найденную функцию

О том, что такое сигнатура поговорим позже.

Другим вариантом бинарного представления может быть команда, возвращающая интерфейс:

```
type Компонента * {
        «Запросить интерфейс» func (имя, сигнатура-интерфейса) Интерфейс
}
```

В этом случае «Запросить интерфейс» возвращает указатель с виртуальной таблицей для всех методов интерфейса. Но этот вариант сложнее, так как при его использовании возникают дополнительные проблемы:
- где-то должно быть однозначное определение интерфейса и связь его с именем, так чтобы компонента А при обращении Б.«Запросить интерфейс» («Нужный интерфейс», …) получала именно то, что запросила.
- вместо сигнатуры функции в случае «Запросить команду» нужно передавать сигнатуру интерфейса (множества команд), так как одинаковое имя интерфейса не гарантирует одинаковость сигнатуры функций и даже одинаковость набора функций.

Из-за этих проблем вариант с запросом интерфейса выглядит слишком сложным по сравнению с запросом команды. Впрочем, вариант с «Запросить команду» не отменяет возможности построить интерфейс в виде объекта с виртуальной таблицей методов через последовательность вызовов «Запросить команду». Такой объект будет существовать только в рамках компоненты А и соответственно, не должен быть бинарно совместим с компонентой Б.

Возможна модификация варианта с «Запросить команду», в которой вместо «Запросить команду» стоит функция более общего действия:

```
type Компонента * {
        «Выполнить» func (имя-действия, доп-параметры) Результат
}
```

Функция «Выполнить», в случае если имя-действия есть строка «Запросить команду», выдает Команду, а в других случаях, выполняет другие, стандартизованные по имени, действия. Причем, в случае «Запросить команду» параметрами должны быть имя команды и сигнатура, как раньше.

Впрочем, как показывает опыт системы «Вир» единственной функции «Запросить команду» вполне достаточно. Все остальное может быть сделано через неё, например, через описание набора стандартных имен команд, которые могут быть реализованы компонентой, например: «Запросить имя», «Включить отладочный режим» и т.д.

Поэтому, в качестве основного варианта, используем вариант с «Запросить команду».

Перейдем к следующей важной части, к определению сигнатуры и способам проверки.

Проверка сигнатуры команды

Так как мы говорим о надежном взаимодействии, необходима проверка того, что сигнатура функции компоненты Б точно соответствует «ожиданиям» компоненты А. Вызов команды компоненты Б (точнее любой подключенной компоненты) в коде А скомпилирован статически. Но статическая проверка сигнатуры (проверка типов параметров, типа результата и соглашения о вызовах) не может быть выполнена, так как мы уже постулировали, что компоненты разработаны и скомпилированы независимо. Мы должны рассматривать любые случаи, например, что компонента Б скомпилирована задолго (за десятки лет[1]) до компоненты А.

Тем не менее сравнение сигнатур на равенство должно быть сделано, и сделано предельно простым и экономичным способом.

Предположим для начала рассуждения, что в сигнатуре используются только стандартные (встроенные) типы, например, целые, вещественные, логические, и для каждого задан определенный размер (например, int32).

Тогда мы можем построить строку, состоящую из имен этих типов (и способов передачи). Имена типов должны быть «нормализованы», в том смысле, что мы должны использовать некоторые «канонические» имена, а не имена какого-то конкретного языка программирования. Причем,

---

[1] На мой взгляд, нам пора переходить от временных решений, примотанных синей изолентой, к решениям постоянным. Нет смысла в том, чтобы десятки тысяч раз писать одно и то же на разных языках программирования, для разных платформ (кроме обогащения корпораций). Пора переходить к промышленному производству, накапливать и переиспользовать лучшие решения.

Для этого (помимо много чего другого), нужно технологическое пространство, в котором слова time to market считаются неприличными. Естественно, это нужно только в рамках общества, которое стремится к звездам, а не к покупке модного гаджета.

набор канонических типов должен быть скорее объединением, чем пересечением набора типов нескольких языков.

Например, в наборе наряду с int8, int16, int32, int64 должны быть типы uint8, uint16, uint32, uint64, хотя не все эти типы могут быть использованы во всех языках программирования.

С другой стороны, в наборе не должно быть типов, специфичных для конкретных платформ, например, float80 (x86), так как мы изначально ориентируемся на мультиплатформенные распределенные системы.

Если набор канонических типов задан, то мы можем задать сигнатуру, например, в LLVM-like нотации, используя канонические имена типов, вместо типов LLVM.

Например:

```
int32(float64) // для Go функции: func (float64) int32
```

Для завершения сигнатуры добавим соглашение о вызове, тоже из LLVM IR: [ccc]int32(float64). Другой вариант: зафиксировать одно соглашение о вызове для всех экспортированных из компоненты функций.

Остается только

- в коде компоненты А – передать строку сигнатуры в вызов «Запросить команду»
- при компиляции Б сохранить строку сигнатуры для каждой «экспортированной» команды
- и в реализации «Запросить команду» сравнить сохраненную строку сигнатуры с переданной.

Далее, естественно расширить возможность сигнатур за счет конструкторов типов, например, типов массивов и структур. Здесь так же можно использовать LLVM нотацию или любую другую нотацию, которая должна быть стандартизирована.

Например: int32(*[]uint8)

Замечание 1: в сигнатуре должно быть использовано **«бинарное» представление типа**, а не языковое. Неважно, как устроен тип string в языке программирования, но если значение этого типа передается как указатель на открытый массив байтов, то это *[]unit8. Впрочем, для этого должен быть определен бинарный стандарт на «указатель на открытый массив».

Замечание 2: при передаче указателя должно передаваться еще указание на ограничения использования, например, можно ли явно возвращать память или она должна быть возвращена сборщиком мусора, а поэтому указатель должен быть «учтен» сборщиком мусора тем или иным способом.

Все, что связано с управлением памятью, это отдельная и непростая тема, которая не рассматривается в этой статье.

Замечание 3: Как только мы «развернули» «бинарное» представление типа (string, в нашем примере), мы неявно перешли к структурной совместимости, и это неизбежно, так как именная совместимость не будет работать для компонент, написанных на разных языках. Естественно, тут же возникают вопросы:

- до какого уровня должно идти разворачивание
- что делать с рекурсивными типами

У меня нет окончательного ответа на эти вопросы, тут нужен набор материала и эксперименты. В качестве начальных предположений:

1) разворачивание должно идти до уровня первой структуры, за исключение структур, состоящих только из простых типов, например: Point (float32, float32)
2) Альтернативой разворачивания может быть сериализация (десериализация), которая безусловно необходима (например, для взаимодействия компонент, работающих на разных устройствах) и должна делаться предельно автоматически за счет метаданных времени исполнения.
3) Так же может быть полезна стандартная языково-нейтральная реализация часто используемых типов, я имею в виду: списки, деревья, хеш-таблицы, и т.п. В этом случае можно использовать именную совместимость на уровне «стандартных» имен, но только в рамках общей памяти одного устройства. В остальных случаях все равно нужна сериализация.
4) Для некоторых «сложных» типов можно использовать именную совместимость. Для этого типы и их имена должны быть зарегистрированы на каком-то из сервисов, доступных для компонент.

Замечание 4: Вместо «нормализованной» строки можно использовать хеши от «нормализованной» строки. В использовании хешей есть свои плюсы и минусы, разговор о которых стоит пока отложить.

Замечание 5: В сигнатурах не могут использоваться платформенно-зависимые типы, например, int, если на разных платформах значение его занимает разное число байт.

Краткие выводы

Кратко опишем подход к подключению компоненты, к которому мы пришли, и который сейчас реализуется в Вир/а1:

1) Каждая компонента во время исполнения представлена указателем на запись, в которой по фиксированному смещению размещен адрес функции:

   «Запросить команду» (имя-команды, сигнатура-команды: *uint8): Команда

   Функция возвращает Команду, которая определена как

   Команда = * { указатель: Компонента, адрес-функции, …. }

   Функция возвращает null, если команды с таким именем нет, или сигнатура не совпадает

   Вызов Команды аналогичен вызову метода.

2) В Вир/а1 добавлен оператор, обеспечивающий, внутри компоненты получение команды со статическим контролем типов, с уточнением, что динамическая проверка происходит в компоненте Б.

3) Для каждой «экспортируемой команды» компилятор а1 строит сигнатуру команды. И для каждой компоненты добавляет (явно указанный) код, реализующий функцию «Запросить команду». Реализация может выбираться, например, в зависимости от числа команд, например: список команд, упорядоченная таблица для бинарного поиска и т.п.

4) Сигнатура команды строится на основе
   - Канонических типов
   - Стандартных конструкторов типов
   - Другие описанные выше способы будут добавляться по итогам экспериментов

По сути, такой способ подключения задает описание компоненты, как черного ящика с единственной ручкой управления. Впрочем, через эту ручку можно получить доступ к любой другой разрешенной ручке управления.

Замечание: В статье практически не упоминается о взаимодействии компонент, работающих на разных устройствах в основном для того, чтобы ограничить размер статьи. На мой взгляд, взаимодействие между устройствами должно обеспечиваться на уровне «соединяющих» компонент, реализующих очереди, протоколы, потоки, топологические схемы, роутинг и т.п. (см. например, Apache Kafka и подобные системы). Например, это могут быть «прокси» компоненты, которые делают RPC для «Запросить команду».

Предыдущий опыт

Мысль о том, что надо разрабатывать (повторно используемые) компоненты и собирать из них программы, безусловно, не нова. Начиная с понятия библиотеки (Maurice V. Wilkes, премия Тьюринга за 1967 г.) попытки собирать «программы» из компонент делались много раз.

Для нас они практически не представляют интереса, так как или были реализованы в рамках одной экосистемы (DCOM, JavaBeans) или были дико переусложнены (CORBA). Впрочем, переусложнены были все. Хороший обзор конца прошлого века можно прочитать в [7]. В это время еще оставалась надежда на то, что какое-то из решений найдет себе заметное место на рынке.

Принципиальной ошибкой всех этих подходов, на мой взгляд, было то, что они все делались, как надстройка на существующим кодом и это всегда была смесь компонентного и других подходов. В итоге, каждое решение было вынуждено иметь дело с огромным количеством предыдущих решений, ошибок и проблем. Отсюда и требование – разработка с нуля для компонентного ассемблера.

Единственным известным мне «чистым» решением является среда разработки «Вир» [4]. Как можно было ожидать, «Вир» существенно проще и логичней.

Первая версия экспериментальной среды программирования «Вир» появилась в 2008 году и сразу была использована для разработки нескольких достаточно крупных проектов.

Основная особенность Вира - это использования явной схемы программы и репозитория стандартных компонент **разных уровней**. Репозиторий компонент пополнялся по ходу разработки и сейчас в нем около 6000 компонент.

Вся разработка в среде Вир велась на языке a0 – это компонентный ассемблер низкого уровня, который позволил выполнить множество экспериментов, ввиду своей предельной простоты и гибкости. Но для перехода к более широкому использованию Вира a0 должен быть заменен на язык более высокого уровня. В первую очередь, для повышения надежности, так как в a0 статическая типизация отсутствует, а динамическая делается вручную.

Основным принципом при разработке языка a0 и программировании на a0 являются принципы DIR (Do It Right) и KISS. Некоторые называют первый принцип DIRS (вместо DIR), добавляя ту же S, что и в аббревиатуре KISS. На мой взгляд, это не совсем уместно.

При переходе на a1 автор ставил себе задачу заменить основной принцип с DIR на SAP (Simple as Possible).

При проектировании a1 автором были рассмотрены языки со свежими подходами к OOP (в первую очередь, Go, Rust, частично Lua), управлению памятью (Rust, Cyclone) и увеличению надежности (частично Котлин).

Естественно, что языки, ориентированные на устаревший подход к ООР, а именно на Class-Oriented Programming (CLOP), такие как C++ были сразу отброшены, хотя идеи из не ООР части рассматривались.

Впрочем, ценность идеи не зависит от языка, в котором она используется, поэтому все, что может упростить программирование и повысить надежность, было, по возможности (то есть без нарушения SAP), использовано.

Заключение

В статье приведены рассуждения, которые привели к одной из существенных особенностей экспериментального компонентного ассемблера Вир/a1, а именно, механизма, обеспечивающего надежное взаимодействие независимо разработанных и скомпилированных компонент.

За пределами статьи оставлен: синтаксис и семантика конструкций языка, подход к управлению памятью (включая владение памятью, локальное для компоненты и локальное для нитки (thread) управление памятью, и многое другое). Продолжение следует.

Часть важных тем, в том числе использование явных схем программ, описаны в [4] и в блоге автора [8].

В настоящее время разработка компилятора Вир/a1 идет в среде разработки Вир (на языке Вир/a0). Мультиплатформенность достигается за счет использования LLVM [9].

В ближайшее время планируется переписать компилятор a1 на Вир/a1, в качестве одного из теста, проверяющего годность языка к достаточно серьезным разработкам.

Разработка Вир/a1 идет в рамках исследовательского проекта, поэтому автор открыт к конструктивной критике и предложениям совместной работы.

Список литературы

[1] The Go Programming Language Specification // https://golang.org/ref/spec

[2] The Rust Programming Language // https://doc.rust-lang.org/stable/book/second-edition/

[3] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, Trevor Jim, “Safe Manual Memory Management in Cyclone” // https://dl.acm.org/citation.cfm?id=1163774

[4] Недоря А.Е. “Технология разработки мультиплатформенных программ на основе явных схем программ” // http://digital-economy.ru/stati/tekhnologiya-razrabotki-multiplatformennykh-programm-na-osnove-yavnykh-skhem-programm, 2018

[5] SapMM: New Open Source Multi-Thread ready Memory Manager // https://github.com/alan008/sapmm

[6] Недоря А.Е., Буняк В.В. “Интернет — в поиске чистого воздуха” // http://digital-economy.ru/stati/internet-v-poiske-chistogo-vozdukha, 2017

[7] Clemens Szyperski, “Component Software: Beyond Object-Oriented Programming”, Addison-wesle, 1998

[8] Недоря А.Е. «Вир» // заметки в блоге http://алексейнедоря.рф/?cat=13

[9] The LLVM Compiler Infrastructure, http://llvm.org/, 2017

PS: И, последнее, так как Карфаген уже разрушен, мы должны сосредоточиться на полете к звездам.