Язык Арс. Взаимодействие

А. Е. Недоря

Введение

Около года назад я показывал первые наброски языков Арс и Арвиль на семинаре STEPS-2024 и обсуждал их с коллегами. Напомню, что

- Арс — это более-менее декларативный язык описания архитектурных схем, то есть язык программирования уровня программных систем или приложений
- Арвиль — язык для разработки компонент (которые я называю арками — архитектурными компонентами). Арки — это атомарные единицы из которых собираются компоненты более высоких уровней, вплоть до программной системы.

Последний год был очень плотным по другим проектам, но между этими другими проектами я продолжал думать об Арсе. Думал далеко не так плотно, как при работе над Тривилем, и раздумья эти были и есть принципиально сложнее. Если при работе над Тривилем я воплощал уже продуманные за несколько десятков лет решения, то здесь приходилось думать заново, и проверять экспериментально все задумки.

За год я много раз менял терминологию, синтаксис языка Арс, переделывал прототип компилятора, и проверял сделанное на примерах. Основной задачей, которую я ставил, была найти терминологию (простую и русскоязычную), которая достаточно хорошо соответствует основным понятиям и их определениям. И выработать синтаксис, который прост и более-менее понятен интуитивно. Сейчас я пришел к решению, которое, на мой взгляд, является достаточно хорошим. Впрочем, это не значит, что поиск окончен…

В этой статье я опишу терминологию и, на примере, основные конструкции языка Арс. Часть из них реализована в прототипе компилятора, а часть еще предстоит сделать.

Но сначала несколько замечаний.

Параллелизм

Поддержка параллелизма необходима. Я склонен к том, что он будет поддержан в виде акторов/активных объектов, впрочем, не отрицая вероятность использования корутин. Я пока не пытался глубоко войти в тему, просто потому что сначала нужно продумать основу. Впрочем, время думать про parallelism and/or concurrency явно приближается.

Управление памятью и изоляция

Архитектурная схема требует тщательного продумывания управления памятью. Например, с точки зрения взаимодействия компонент с разным управлением памятью (Rust, Swift, Go, …) и с точки зрения изоляции, то есть предотвращения влияния через память компонент друг на друга тем или иным образом. В рамках уже упомянутых мной других проектах, я попытался сделать широкий обзор современных подходов, например, рассматривая языки Pony, Argentum и Project Verona и другие. Впрочем, так же как с параллелизмом, предметная работа в Арсе/Арвиле с памятью еще впереди.

Замечание о динамике и статике

Я уже писал о том, что, в общем случае, схема динамическая, в том смысле, что при компиляции схемы могут быть неизвестны интерфейсы частей, из которых она состоит. При подключении части к схеме во время исполнения происходит проверка интерфейсов, я называю ее “динамическим знакомством”. Это не динамическая типизация в смысле динамических языков (Javascript, …), так как динамическая проверка происходит один раз, а дальше все типы известны. Для динамического знакомства схема должна содержать требования к интерфейсу части.

Динамическая природа схемы позволяет упростить прототипирование. Архитектор указывает требования, но может не думать сначала о выборе или о разработке нужной части.

Если подключаемая часть известна во время компиляции схемы, то проверка может быть выполнена статистически - “статическое знакомство”. Кроме того, может быть выполнена оптимизация для статически известных частей, или для всей программной системы.

По аналогии с gradual typing, такой подход можно назвать gradual transformation to static.

В данный момент компилятор Арс не использует информацию об частях программы, то есть все части подключаются динамически с использованием динамического знакомства.

Замечание о ленивости

Архитектурная схема будет большой для больших проектов, поэтому она изначально должна быть “ленивой”, то есть инициализация частей схемы и построение run-time структур должна выполняться по необходимости.

Терминология

Сначала неформальные (и краткие) определения, в следующем разделе пояснения на примере. Возможно, что начинать читать лучше с примера

Схема: Описание (статической) структуры программной системы и взаимодействия частей. При исполнении схема может изменяться (новые связи) и достраиваться. Схема состоит из **Узлов**.

В схеме используются описания типов, констант, функций и деталей. В каком-то смысле Схема — это аналог main, что там написано, то и будет работать, собранное из описанных сущностей.

Узел: структурная и именованная часть схемы. Схема состоит из корневого Узла и под-узлов. Узел — это настроенный (сконфигурированный) экземпляр **Детали**. Узел может быть атомарным или сколько угодно сложным, включающим множество под-узлов.

Деталь: это тип (чертеж) компоненты. Деталь может быть описана на месте (прямо в схеме) или описана отдельно и подключена в определенную точку схемы, образуя Узел. В любом случае, в месте подключения деталь может быть сконфигурирована. В схеме может быть произвольное число экземпляров одной детали. Аналог: есть UI компонента “Кнопка”, которая ставится на окно (страницу) и настраивается. Компонента “Кнопка” — это Деталь, а поставленная и настроенная кнопка — это Узел.

Деталь может быть построена на основе программной части, написанной на обычном языке программирования, например, на основе Арке, написанной на языке Арвиль или собрана из под-узлов.

Интерфейс и **Контакт**: У узла есть Интерфейс и набор Контактов. Они могут быть определены в Детали, над которой построен узел) или непосредственно в Узле.

Интерфейс — это набор методов (управляющих воздействий), которые предоставляет Узел (Деталь).

Контакт (контактная площадка) — это то, что можно было бы назвать частью **экстра**фейса, то есть тем, что должно быть подключено снаружи, чтобы узел работал.

Например, у двигателя в интерфейсе есть метод “добавить газу”, а для работы двигателя к его контактной площадке должна быть присоединена подача бензина (или электричества).

Не уверен, что эти определения можно понять, просто прочитав, поэтому перейду к примеру.

Пример: Расширенный “Привет, мир!”

Это программа берет имя из какого-то источника данных, собирает приветственное сообщение с этим именем, и передает его для вывода.

Итогом работы является вывод на консоль: “Привет, Вася!”

Суть примера в том, что основная часть не связана статически с источником данных и направлением вывода.

Пример состоит из 3-х узлов верхнего уровня:

- корневой узел схемы, который содержит два узла и исполняемый “вход” (main)
  - Узел “ввод-вывод” (строки 2-14): часть обеспечивающая работу
  - Узел “мир” (строки 15-19): основная часть

Полный текст схемы:

```
1  арс {
2    ввод-вывод: {
3      контакт напечатать строку: уведомление (с: Строка)
```

```
 4        контакт генератор имен: запрос (): мб Строка
 5
 6        вывод: арка(адрес:  "a2::примеры/стд-вывод") {
 7           интерфейс { фн напечатать строку (с: Строка)  }
 8           связь (ввод-вывод:>напечатать строку) = вывод::напечатать строку
 9        }
10        имена: арка(адрес:  "a2::примеры/имена") {
11           интерфейс { фн дай имя(): мб Строка }
12           связь (ввод-вывод:>генератор имен) = имена::дай имя
13        }
14     }
15     мир: арка(адрес:  "a2::примеры/пример4") {
16        интерфейс {  фн привет()  }
17        связь Имя = ввод-вывод:>генератор имен
18        связь Сообщить = ввод-вывод:>напечатать строку
19     }
20     вход мир::привет()
21  }
```

Рассмотрим Узел с именем “мир”, он построен из Детали, определенной на месте и построенной на базе арки с указанным адресом. У Детали задан **интерфейс** из одного метода и две **связи**, задающие подключения к **контактам** арки.

Приведу этот фрагмент схемы вместе с исходным текстом арки:

```
15     мир: арка(адрес:  "a2::примеры/пример4") {
16        интерфейс {  фн привет()  }
17        связь Имя = ввод-вывод:>генератор имен
18        связь Сообщить =
19           ввод-вывод:>напечатать строку
20     }
```

```
 1  арка пример4
 2  импорт "стд::строки"
 3
 4  контакт {
 5     Имя: запрос (): мб Строка
 6     Сообщить: уведомление (имя: Строка)
 7  }
 8
 9  фн (а) привет*() {
10     а.Сообщить(строки.ф("Привет, $;!\n",
11           а.Имя()))
12  }
```

Как видно, у арки (справа) определены два **контакта**, которые должны быть соединены с чем-то (нужного типа) для её работы. Эти соединения задаются в схеме (слева) через конструкцию **связь**:

- контакт (запрос) “Имя” подключается к контакту “генератор имен” в узле “ввод-вывод”
- контакт (уведомление) “Сообщить” подключается к контакту “напечатать строку” в узле “ввод-вывод”

Это означает, что при обращении к контакту, например вызове “а.Имя()” в экземпляре арки, будет вызвано действие, подключенное к контакту “генератор имен”. А если оно не подключено, то будет ошибка времени исполнения.

В схеме (см. выше) подключение к контакту “генератор имен” выполняется в строке 12:

```
связь (ввод-вывод:>генератор имен) = имена::дай имя
```

Тем самым создаются такие связи:

Имя => “генератор имен” (узел “ввод-вывод”) => “дай имя” (узел “имена”)

Действие может быть связано напрямую с интерфейсным методом арки, например, в строке 17 схемы могло быть написано:

связь Имя = ввод-вывод\имена::дай имя

Но при этом будет утеряна гибкость, которая есть в примере: замена под-узлов узла “ввод-вывод” не потребует изменений узла “мир”. Понятия “контакт” и “связь” позволяют развязывать связи в исходного коде по импорту/экспорту, и заменять их на более гибкие связи через схему.

Я не хочу описывать сейчас синтаксис конструкций, они будут описаны в описании языка Apc, мне кажется, что синтаксис достаточно прост для интуитивного понимания.

Что дальше?

Ближние планы:

- доработка компилятора
- доработка библиотечного слоя
- подготовка описания языка Apc

С дальними планами сложнее, мне хочется сделать подключение графики (SDL2 ?) и начать делать простенькие программы с GUI, что должно привести в итоге к разработке IDE.

И по ходу дорабатывать то, о чем я писал в замечаниях в начале статьи. Но, возможно, придется сделать еще какие-то промежуточные и вспомогательные действия.