Чистая компиляция как парадигма программирования

А.В. Столяров <avst@cs.msu.ru>
О.Г. Французов <franoleg@intelib.org>
А.С. Аникина <a.anikina1993@gmail.com>
Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, вл. 1

Аннотация. В научной литературе широко представлено исследование возможностей интерпретируемого выполнения компьютерных программ. При этом противоположный подход, основанный на элиминации любых проявлений интерпретируемости, насколько можно судить, никем всерьез не исследовался. В настоящей статье рассматривается подход, предполагающий отделение всего происходящего во время исполнения программы от инструментов, использующихся программистом при ее написании, и намечается путь к построению универсального (равно пригодного как для системных, так и для прикладных задач) языка программирования.

Ключевые слова: парадигма программирования; компиляция; интерпретация; рефлексия.

DOI: 10.15514/ISPRAS-2018-30(2)-1

Для цитирования: Столяров А.В., Французов О.Г., Аникина А.С. Чистая компиляция как парадигма программирования. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 7-24. DOI: 10.15514/ISPRAS-2018-30(2)-1

1. Введение

Граница между интерпретируемым и компилируемым исполнением в наше время утратила четкость; элементы интерпретации проникают в компилируемое окружение и наоборот. Многие популярные языки программирования, такие как C# и Java, традиционно компилируются, но результатом компиляции становится не машинный код, а некоторое промежуточное представление, предназначенное для выполнения (т. е. интерпретации) виртуальной машиной. С другой стороны, современные интерпретаторы в большинстве случаев сначала переводят исходный текст программы в некоторое внутреннее представление, которое затем исполняют в режиме интерпретации. Отличия между таким компилятором и таким интерпретатором оказываются довольно эфемерными: в первом случае промежуточное представление сохраняется во внешнем файле, а виртуальная машина реализована отдельно от транслятора, во втором случае виртуальная

машина интегрирована с транслятором, что делает файл промежуточного представления необязательным.

Можно сформулировать достаточно очевидный критерий для отнесения модели исполнения к компиляции или интерпретации: интерпретатор во время исполнения программы должен сам находиться в памяти компьютера, что и отличает его от компилятора, который (сам по себе) во время исполнения не нужен. К сожалению, этот критерий не отличается высокой содержательностью. Для языков программирования, содержащих примитив EVAL и вследствие этого заведомо относящихся к интерпретируемым, таких как Lisp и (особенно) Scheme, существуют компиляторы, удовлетворяющие рассматриваемому критерию, что дает основания их сторонникам отрицать интерпретируемую сущность таких языков. То обстоятельство, что в итоговый исполняемый файл (напрямую, либо в виде динамически связываемой библиотеки) включается практически полный код интерпретатора, сторонников этой точки зрения не смущает. Более того, если рассмотреть одну из наиболее популярных в настоящее время реализаций Common Lisp – SBCL, обнаруживается, что этот транслятор работает в режиме интерпретации, а создать исполняемый файл хотя и позволяет, но крайне неочевидным путем, и размер этого файла делает использование такого режима практически бессмысленным; тем не менее, авторы SBCL в документации прямо заявляют, что их реализация является компилируемой, и предлагают «игнорировать» всех, кто говорит об интерпретируемой сущности языка Lisp.

Еще более запутывает картину применение так называемой JIT-компиляции (just-in-time compilation), при которой интерпретатор во время исполнения программы переводит некоторые ее части непосредственно в машинный код и затем исполняет его.

С другой стороны, даже в программах на таком традиционно компилируемом языке, как C, можно обнаружить элементы интерпретации: форматные строки функций семейств scanf и printf анализируются во время исполнения программы, причем анализу во время исполнения подвергается именно строка текста в том виде, в котором она написана программистом в исходном тексте программы. Конечно, форматные строки не обладают тьюринг-полнотой, их возможности примитивны, но это не исключает факта анализа фрагмента исходного текста во время исполнения. Некоторые авторы относят к элементам интерпретации также позднее связывание, реализуемое (например, в языке C++) механизмом виртуальных функций.

Больше того, можно столкнуться с утверждением, что любое исполнение является интерпретирующим, поскольку даже исполнение машинного кода центральным процессором есть не что иное, как интерпретация, причем, если вспомнить о существовании в процессорах микрокода, такое утверждение оказывается не столь далеким от истины.

В литературе, в том числе в научных работах, подробно обсуждаются возможности, получаемые при добавлении все новых и новых элементов

интерпретации. Эти возможности известны под общим термином «рефлексия». Обзор этого направления можно найти в работе [1]. Как ни странно, противоположный подход, который должен был бы основываться на элиминации интерпретирующего исполнения, в литературе не освещается вовсе.

2. Рефлексия

Намек на присутствие рефлексии можно заметить во многих современных языках программирования. Такие языки предоставляют возможность получения некоторой информации об исходном коде программы во время ее исполнения. Например, одни языки позволяют динамически определять тип переменной или объекта (операторы dynamic_cast и typeid в языке C++), а другие – получить информацию об именах переменных, функций, классов и т. д. (в таких языках как C# и Python существуют способы получения имени класса, придуманного программистом). Следует признать, что такие возможности удобны, в частности, при отладке программ.

В некоторых языках программирования присутствуют средства, позволяющие во время исполнения программы заменить одну реализацию метода класса на другую (так, в Objective-C это делается с помощью функции class_replaceMethod). Такие возможности тоже считаются рефлексией.

Само понятие рефлексии было введено Брайаном Смитом в работе [2]. Смит описывает вычисление программы в виде связей между тремя *доменами*: синтаксическим доменом (исходный код), доменом внутреннего представления программы и доменом «реального мира». Связи между доменами представлены процессами *интернализации* (internalization), *нормализации* (normalization) и *денотации* (denotation), которые соответствуют переходу от объектов исходного кода программы к объектам внутреннего представления, преобразованиям этих объектов (попросту говоря, вычислению результата) и интерпретации полученных результатов, то есть отображению объектов внутреннего представления на объекты предметной области.

При обсуждении рефлексии часто используется понятие *реификации* (reification), исходно пришедшее из философии. В данном случае под реификацией того или иного аспекта устройства программы или ее выполнения подразумевается, что к этому аспекту программе предоставляется доступ наравне с обычными данными или другими объектами, с которыми она может работать. В качестве широко распространенного примера реификации можно привести средства анализа объектов программы во время выполнения, которые обычно относят к рефлексии: к примеру, функция dir в языке Python и метод Type.GetProperties() стандартной библиотеки платформы .NET позволяют получить список атрибутов объекта. В качестве важного случая реификации следует упомянуть доступ к именам, введенным программистом, во время выполнения программы. Так, платформа .NET позволяет получить имя класса, метода, свойства и т. п. через функции стандартной библиотеки из пространства имен System.Reflection. Аналогом этого для языка Lisp будут встроенные

примитивы symbol-name и intern, которые позволяют из атома получить строку и наоборот; Prolog предусматривает для той же цели встроенный предикат name.

Брайан Смит не включал в понятие рефлексии этот аспект – по-видимому, потому, что Lisp для его работы служил отправной точкой.

Любопытно заметить, что язык C позволяет преобразовывать адреса переменных и функций в целые числа и обратно без потери информации, что также можно считать проявлением реификации.

В терминах, введенных Смитом, язык программирования поддерживает *структурную рефлексию* (structural reflection), если он позволяет анализировать или изменять объекты внутреннего домена. Если же язык позволяет вмешаться в процесс вычисления (*нормализации* по Смиту), т. е. предоставляет прямой доступ к вычислительному контексту программы, речь идет о *поведенческой рефлексии* (behavioral reflection). Смит отмечает, что для ее обеспечения от языка требуется поддержка реификации не только объектов программы, но и самих языка и вычислителя. Иначе говоря, программе через те или иные примитивы должна быть предоставлена возможность управлять интерпретатором, который ее выполняет. Для компилируемого языка должны быть полностью реифицированы как среда времени выполнения, так и транслятор, что в определенном смысле превращает компиляцию в интерпретацию, ведь транслятор теперь не может отсутствовать во время выполнения.

Поведенческая рефлексия изучена гораздо хуже, чем структурная, хотя отдельные ее элементы уже давно встречаются в языках программирования; в качестве примера можно назвать *продолжения* (continuations) языка Scheme. Для платформы .NET относительно недавно появился набор инструментов под общим названием Roslyn, включающий в себя компиляторы C# и Visual Basic, управляемые через библиотечные функции [3].

Имея доступ к управлению собственным исполнителем, программа, написанная на языке с поддержкой поведенческой рефлексии, может во время вычисления изменять структуры, которые вычисляют ее саму. Это может привести к несовместимости изменений, внесенных рефлективным кодом, и изменений, внесенных интерпретатором. Такой феномен называется *интроспективным наложением* (introspective overlap). Для борьбы с этим феноменом Смит в своей работе ввел понятие *рефлективной башни* (reflective tower), которая представляет собой стек из интерпретаторов, где первый интерпретатор исполняет исходную программу, второй интерпретатор исполняет первый и т. д. Таким образом, для поддержания возможностей поведенческой рефлексии требуется наличие не только интерпретатора программы в памяти, а их потенциально бесконечное количество.

Интересно отметить, что Смит для обозначения составляющих рефлективной башни ввел термин *рефлективная обрабатывающая программа* (reflective processor program); это более общее понятие, нежели «интерпретатор», но, так или иначе, это некоторый программно реализованный исполнитель, который должен присутствовать в памяти на момент вычисления.

3. Недостатки интерпретируемого исполнения

Когда речь идет о недопустимости или нежелательности интерпретации в тех или иных случаях, чаще всего в качестве *единственного* недостатка интерпретации почему-то (иногда неявно) рассматривают потери в эффективности; сторонники интерпретируемого исполнения тратят много сил и красноречия на обоснование возможности эффективной интерпретации, то есть такой, которая либо вообще не уступает компилируемому исполнению, либо уступает настолько незначительно, чтобы потерями в эффективности (в основном по времени исполнения, реже припоминают также эффективность использования оперативной памяти) можно было пренебречь.

Конечно, практика показывает, что эффективность интерпретируемого исполнения, несмотря на все усилия его сторонников, в подавляющем большинстве случаев оставляет желать много лучшего, но дело, как ни странно, не в этом. Сравнительно низкая эффективность по времени исполнения представляет собой существенный, но отнюдь не единственный и даже, возможно, не главный недостаток интерпретации; мы попытаемся провести краткий обзор других ее недостатков.

Одно из основных преимуществ интерпретируемой парадигмы – гибкость во время исполнения программы. Возможность в любой момент воспользоваться рефлексией для того, чтобы доинтерпретировать какой-то фрагмент кода или поработать со структурами данных, создает определенный потенциал, которого языки, тяготеющие к компилируемости и статичности, лишены; но любая возможность имеет свою цену. Повышенная гибкость программы во время исполнения приводит к увеличению **хрупкости кода**. Чем больше нетривиальной логики относится на этап выполнения программы, тем труднее становится тестирование. Компилируемые языки в чем-то ограничивают программиста, но эти ограничения служат страховкой хотя бы от части ошибок – по крайней мере, синтаксические ошибки будут обнаружены на этапе компиляции. В то же время получить синтаксическую ошибку на этапе выполнения программы, написанной на интерпретируемом языке – явление обычное.

Известен метод борьбы с этой проблемой – стопроцентное покрытие модульными тестами (что само по себе – прекрасная практика), но если в дело идут механизмы рефлексии, то нельзя быть уверенным, что даже при стопроцентном покрытии нет граничного случая, когда логика работы кода окажется нарушена. Интерпретируемая парадигма при этом не позволяет исключить даже ошибок синтаксиса или ситуаций отсутствия поля или метода в объекте.

Механизмы рефлексии позволяют расширить интерфейс объекта во время выполнения, «по месту» – такая техника носит не очень уважительное название *«латания по-обезьяньи»* «monkey patching» [4]. Ее применение также служит

источником хрупкости, однако среди программистов на Ruby, тем не менее, эта техника считается нормальной и допустимой.

Логичным продолжением стремления обнаружить как можно больше ошибок как можно раньше (например, на этапе трансляции) можно считать инструменты статического анализа кода. Таких инструментов существует множество, они могут, например, указывать на возможные проблемы (lint) или, интегрируясь с текстовым редактором, упрощать процесс написания кода, предоставляя средства навигации, автодополнения и т. д. Гибкость интерпретируемых языков ограничивает возможности такого класса инструментов, увеличивая **сложность статического анализа** вплоть до полной невозможности его выполнения. Задача выдачи списка атрибутов объекта для компилируемых языков решается относительно просто на базе статического анализа типа. В рамках интерпретируемой парадигмы потребовалось бы полностью выполнить программу до указанной точки, чтобы точно ответить на вопрос, какие атрибуты в итоге у объекта окажутся.

Одним из препятствий к использованию интерпретируемых языков программирования оказывается необходимость наличия интерпретатора на компьютере конечного пользователя. Интерпретаторы сами по себе относятся к достаточно сложным программным инструментам, а их авторы зачастую совершенно не стремятся к упрощению процедуры развертывания их продуктов, предполагая, что их пользователь – профессиональный программист. Это приводит к появлению **языковых экосистем**. Конечный пользователь, который уже по тем или иным причинам работает с программными средствами на некотором интерпретируемом языке, при выборе новых инструментов отдает предпочтение тем, которые написаны на уже «освоенном» им языке, поскольку при их внедрении ему не придется тратить время на освоение еще одной языковой экосистемы.

Многие интерпретируемые языки программирования (в качестве примеров можно привести Perl, Python и Ruby) имеют собственные пакетные решения для установки как прикладных программ и средств, так и библиотек (зависимостей). Молчаливо предполагается, что конечный пользователь должен уметь пользоваться «экосистемным» пакетным менеджером и владеть некоторым базовым набором знаний и умений для работы внутри экосистемы. Так, краткое руководство по установке и настройке популярного генератора статических сайтов Jekyll обещает, что начать работу с инструментом можно будет «за считанные секунды», при этом первой командой, которую предлагается выполнить в руководстве, оказывается `gem install jekyll bundler`. Эта команда требует наличия настроенного разработческого окружения Ruby, но в руководстве Jekyll вопрос того, как, для начала, добиться работоспособности этой команды, остается за кадром.

Дело усугубляется тем, что «экосистемные» пакетные менеджеры конфликтуют с пакетными менеджерами операционной системы, при этом для конечного пользователя часто оказывается невозможно избежать использования

«экосистемного» средства, даже если он задастся такой целью. Документация на Jekyll, к примеру, заявляет в качестве системных требований наличие не только Ruby «со всеми заголовочными файлами», но и компилятор GCC с системой сборки make [5]. Таким образом, граница между средой времени выполнения, очевидно необходимой интерпретируемым языкам, и окружением разработчика оказывается фактически стертой.

На другом полюсе спектра решений ─ подход, когда программа, написанная на интерпретируемом языке, вместе со всеми своими зависимостями и интерпретатором упаковывается в дистрибутивный комплект, обеспечивающий возможность установки конечным пользователем. Такой подход часто применяется для массовых продуктов, которые требуется распространять среди пользователей ОС Windows; например, именно так организован дистрибутив системы контроля версий Mercurial, несмотря на то, что этот инструмент очевидным образом предназначен для профессиональных программистов.

Этот подход решает часть проблем, но и он не лишен недостатков: помимо неоправданно большого размера дистрибутивного комплекта, распространяемая таким образом программа оказывается развернута в своем собственном экземпляре экосистемы и может, например, не иметь доступа к пакетам, установленным стандартными «экосистемными» средствами.

Когда у одного из авторов статьи возникла необходимость развернуть на сервере с ОС Windows систему контроля версий Mercurial и систему отслеживания замечаний Trac, воспользоваться штатными дистрибутивными комплектами каждой из систем не удалось. Обе системы написаны на языке Python, но при установке из дистрибутивных комплектов модули Mercurial, предоставляющие его API, оказались не доступны системе Trac из-за того, что каждая из двух систем использовала свой отдельный интерпретатор языка со своим набором модулей-пакетов. Единственным возможным вариантом оказалась установка всех пакетов вручную из исходного кода, по модели, более пригодной для окружения разработчика, чем конечного пользователя; при этом пришлось пожертвовать производительностью, отказавшись от собственных расширений системы Mercurial, так как для их сборки потребовалось бы разворачивать на активно эксплуатируемой серверной машине еще и систему программирования на базе языка C.

Для программ, имеющих существенный объем исходного кода, интерпретирующее исполнение может оказаться непригодным из-за **длительного времени запуска программы**, ведь фактически трансляцию программы приходится при каждом запуске производить заново. Естественно, современные интерпретаторы стараются эту проблему так или иначе решить, переводя программу во внутреннее представление по частям по мере надобности, либо сохраняя сгенерированное внутреннее представление (полностью или частично) для использования во время последующих запусков; факта существования проблемы все это не отменяет.

4. Библиотечная поддержка времени исполнения

Ситуация резко осложняется, когда написанная программа предназначена к непосредственной работе с аппаратурой и не может полагаться на операционную систему: либо эта программа сама представляет собой часть операционной системы (например, драйвер), либо операционная система отсутствует, как в случае прошивок для микроконтроллеров. В таких условиях оказываются неприемлемы многие особенности трансляторов, на первый взгляд не имеющие отношения к интерпретирующему исполнению, такие как необходимость библиотечной поддержки времени исполнения.

Так, авторы стандартов языка C, волюнтаристски включив в спецификацию языка огромный пласт библиотечных возможностей (так называемые *стандартные библиотеки*), были, тем не менее, вынуждены признать, что все это не может быть обязательной частью языка, и определили два возможных окружения – обычное, предполагающее наличие операционной системы и библиотечной поддержки (hosted), и *самостоятельное* (freestanding), для которого стандарт требует поддержки только семи заголовочных файлов, ни один из которых не вводит библиотечных функций. Следует заметить, что признание допустимости самостоятельного окружения было вынужденным шагом. Стандартная библиотека не используется и не может быть использована в ядре операционной системы. Потребовав на уровне стандарта языка, чтобы стандартная библиотека предоставлялась всегда и везде, пришлось бы считать, что для написания ядер операционных систем используется некий «нестандартный диалект» языка C; между тем, этот язык был изначально создан специально для написания ядра ОС Unix.

Появившийся несколько лет назад язык Rust изначально включал ряд высокоуровневых возможностей вплоть до сборки мусора (путем подсчета ссылок [6]); создатели Rust вовремя осознали, что такой подход существенно ограничивает область применения языка, и к моменту выпуска версии 1.0 предусмотрели аналог «самостоятельного» окружения языка C, для чего спектр возможностей языка пришлось урезать.

В контексте нашей статьи здесь интересен тот факт, что транслятор, ориентированный на компилируемое исполнение, может оказаться непригоден в конкретной задаче из-за особенностей библиотек, на наличие которых он полагается. Чаще всего внутреннее устройство компиляторов не подвергается документированию такого уровня, который бы позволил пользователю заменить версии библиотек времени исполнения на более подходящие; как следствие, в определенных случаях неподходящим оказывается весь компилятор целиком. При внимательном рассмотрении вопрос *допустимости требования о наличии во время исполнения программы компонентов транслятора* (включая библиотеку времени исполнения) оказывается обобщением вопроса о допустимости тех или иных элементов интерпретации.

5. Компилируемое выполнение как парадигма

При проектировании программы с расчетом на интерпретируемое исполнение практически отсутствуют ограничения на средства, поддерживающие выполнение программы. Во время исполнения оказываются при желании доступны средства манипуляции состоянием программы как объектом («структурная рефлексия»); в памяти присутствует интерпретатор, так что можно средствами выполняющейся программы формировать новые фрагменты для нее же самой и сразу их выполнять (примитив EVAL; в частности, конфигурационные файлы могут быть написаны на том же языке, что и сама программа), имеется информация об особенностях исходного текста программы – например, о введенных в нем именах, даже если сам исходный текст как таковой почему-то недоступен; как уже отмечалось, сторонники рефлексивного окружения на этом не останавливаются, требуя, чтобы язык допускал средства манипуляции программой, то есть чтобы программным образом во время исполнения можно было вносить в программу изменения так же, как во время написания программы их вносит программист. Можно заметить, что при этом вообще стирается грань между временем написания и временем исполнения программы.

Очевидно, что мысль о применении средств такого рода может возникнуть лишь в рамках определенного стиля мышления, что позволяет говорить о *парадигме* интерпретируемого исполнения. Аналогичным образом, по-видимому, будет корректно рассмотреть *парадигму компилируемого исполнения*; как ни странно, авторам статьи не удалось найти ни одной работы, в которой бы вводились такие термины. Между тем, определенные элементы мышления, которое можно было бы обозначить как парадигму компилируемого исполнения, встречаются на практике, и достаточно часто. Для примера можно рассмотреть ситуацию интерпретатора, встроенного в программу, написанную на компилируемом языке. К такому решению часто прибегают в ситуации, когда для настройки программы оказывается недостаточно простых конфигурационных файлов и требуется давать ей указания на тьюринг-полном языке; в качестве встраиваемых часто используют такие языки, как Lua, Tcl, разнообразные диалекты Лиспа и т. п. Многие программисты на интуитивном уровне предполагают, что такой интерпретатор, встроенный в основную программу, может выполнять программы, предоставленные конечным пользователем и предназначенные для целей настройки основной программы, но при этом встроенный интерпретируемый язык не следует применять для написания частей *самой программы*; все ее возможности следует реализовать на основном языке проекта, даже если такая реализация окажется более трудоемкой. Интересно, что в большинстве случаев не удается найти никаких технических аргументов в пользу реализации той или иной особенности именно на основном, а не встроенном языке, речь в лучшем случае идет о некой «концептуальной целостности». Не следует, однако, недооценивать концептуальную целостность. Программирование, как известно, представляет собой чрезвычайно тяжелый вид

интеллектуальной деятельности, и культура мышления способна оказать существенную помощь в борьбе со сложностью осмысления компьютерных программ.

Аналогично тому, как сторонники парадигмы рефлексии определяют несколько уровней возможностей, в контексте парадигмы компилируемого исполнения можно указать различные уровни строгости соответствия программы этой парадигме. Если рефлексия предполагает наличие и доступность инструментов программиста во время исполнения программы, то компиляция, по-видимому, должна предполагать их отсутствие; если продвинуться еще дальше, следует потребовать, чтобы язык программирования и инструментарий программиста допускал четкое разделение между особенностями программы, видимыми пользователю, и решениями, принимаемыми программистом в процессе ее реализации; иначе говоря, должна быть обеспечена как минимум *возможность* того, чтобы выбираемые программистом способы и методы достижения целей никак не влияли на все, что видит конечный пользователь программы.

В качестве первого и наиболее очевидного уровня соответствия используемых инструментов парадигме компилируемого исполнения можно предложить требование отсутствия транслятора в памяти компьютера во время исполнения программы; транслятор должен позволять создание исполняемого файла, запуск которого возможен в отсутствие транслятора. На следующем уровне можно потребовать отсутствия в памяти не только транслятора, но и отдельных его элементов, таких как лексический и синтаксический анализатор, генератор кода и прочее. Широко известен подход к синтезу исполняемого файла путем соединения значительной части компонентов интерпретатора с исходным текстом программы или тем или иным его внутреннем представлением; такой подход удовлетворяет первому уровню требований, но уже не удовлетворяет второму.

Заметив, что библиотеки очевидным образом представляют собой часть системы программирования, причем многие из них (не только «стандартные») прилагаются к транслятору, можно довести два предыдущих уровня требований до определенного логического завершения, потребовав для начала, чтобы генерируемый исполняемый файл не зависел (или по крайней мере *мог* не зависеть) от наличия каких бы то ни было компонентов системы программирования, в том числе динамически подгружаемых библиотек; переходя к следующему уровню, придется потребовать, чтобы и в сам исполняемый файл не включались никакие библиотеки кроме тех, которые в явном виде затребовал программист, а минимально допустимое множество таких библиотек было пустым. Удовлетворить такому требованию может лишь язык программирования, из которого исключены (вытеснены в библиотеку) любые возможности, требующие сколько-нибудь нетривиальной реализации на уровне машинного кода.

Отдельно следует упомянуть имена (идентификаторы), вводимые программистом в исходном тексте программы. В ранних диалектах языков,

ориентированных на обработку символьной (слабоструктурированной) информации, таких как Lisp, Prolog, Planner и т. п., отсутствовали строковые константы как отдельная сущность; в роли слов естественного языка (элементов текста) выступали обычные идентификаторы (атомы). Впоследствии от такого использования атомов отказались; все современные диалекты Лиспа включают строковые константы как отдельную сущность, есть они и в наиболее современных реализациях языка Prolog, таких как SWI-Prolog. Можно сделать достаточно логичное предположение о причинах отказа от атомов в роли строк. В эпоху, когда языки рассматриваемой категории только возникли, практически не существовало *конечных пользователей*; любой человек, работающий с компьютером, был программистом. Появление конечных пользователей позволило осознать, что имена, используемые программистом в программе, и строки, видимые пользователю, предназначены для принципиально разных целей, относятся, если угодно, к разным предметным областям и не должны смешиваться – хотя бы из соображений концептуальной чистоты.

Из сказанного вытекает очередное требование к компилируемому стилю исполнения: работа программы не должна никак зависеть от выбранного программистом набора имен (идентификаторов). Можно сказать, что программа, полностью соответствующая парадигме компилируемого исполне-ния, должна быть *устойчива к альфа-преобразованию*: при любом переименовании идентификаторов в исходном тексте программы, если такое переименование не нарушает очевидного требования о сохранении уникальности идентификаторов, результаты работы программы не должны никак измениться.

Отметим, что всем перечисленным требованиям удовлетворяют в наше время разве что языки ассемблеров. В частности, компиляторы языка C обычно зависят от пусть и небольшого, но далеко не пустого множества библиотечных функций; так, компилятор gcc имеет специальный режим для создания «самостоятельных» исполняемых файлов (флаг -ffreestanding), но даже в этом режиме на этапе компоновки может потребоваться подключение библиотеки libgcc: например, перемножение 64-битных целых чисел на 32-битных процессорах этот компилятор реализует через вызов библиотечной функции. Кроме того, в документации к компилятору перечислены несколько функций, на наличие которых он полагается, несмотря на режим создания «самостоятельных» исполняемых файлов (например, memcpy). Более того, язык C, строго говоря, не обладает устойчивостью к альфа-преобразованиям за счет наличия в макропроцессоре возможности превратить идентификатор в его имя (строковую константу). Например, в программе можно объявить такой макрос:

```
#define GETIDNAME(x) (#x)
```

и затем использовать следующий вызов функции printf:

```
int myvar;
  /* ... */
printf("%s\n", GETIDNAME(myvar));
```

Такой вызов выведет строку «myvar»; если переменную myvar в программе переименовать, то, очевидно, изменится и выдаваемая программой строка.

Впрочем, даже достижение уровня требований, которым удовлетворяют только языки ассемблеров, не обязывает нас остановиться. Следующее требование оказывается очевидным как с точки зрения эффективности исполнения, так и с точки зрения самодисциплины программиста, но, тем не менее, соблюдать его в современных условиях практически невозможно. Итак: *любые вычисления и преобразования, которые могут быть выполнены во время компиляции, не должны переноситься на время исполнения программы*. Нарушением этого принципа будет, в частности, интерпретация форматных строк в функциях printf/scanf. К сожалению, язык С не обладает изобразительными средствами, которые позволили бы адекватно (без чрезмерных трудностей для программиста) создать какую-то структуру данных, содержащую все необходимые сведения о формате ввода или вывода в таком виде, который не требовал бы лишнего анализа (например, перевода чисел из строкового представления, при том что само число уже известно во время компиляции).

В терминах содержания исполняемого файла можно сформулировать еще два требования к компиляции. Во-первых, система программирования должна позволять создать такой исполняемый файл, чтобы никакими средствами анализа нельзя было определить, какие конкретно инструменты были задействованы для его создания. Это требование позволяет надеяться, что в исполняемом файле *действительно не останется ничего лишнего*, обусловленного только выбранными инструментами и методами решения задачи и не имеющего отношения к решенной задаче как таковой. Во-вторых, можно потребовать, чтобы система программирования позволяла сформировать *любой* исполняемый файл, корректный с точки зрения целевой платформы. Среди существующих инструментов такими свойствами обладают только ассемблеры. Исполняемый файл, созданный компилятором языка С, позволяет определить, какой был использован компилятор, причем в большинстве случаев – с точностью до версии.

Модель трансляции программ, удовлетворяющую всем перечисленным требованиям, мы назовем *чистой компиляцией* или *чисто компилируемым исполнением*. Очевидно, что для практического применения этой модели потребуется новый язык программирования, поскольку существующие языки высокого уровня (даже язык С) введенным требованиям заведомо не удовлетворяют, а предложение о более активном использовании ассемблеров можно всерьез не рассматривать; впрочем, как мы видели, для применения противоположной по смыслу *полной рефлексии* тоже не годится ни один существующий язык программирования.

Отметим, что «интерпретация» машинного кода центральным процессором в принятых терминах никак не противоречит введенной модели, поскольку ни сам центральный процессор, ни его микрокод не являются инструментами программиста.

6. Заключение

На протяжении всего времени существования программирования как дисциплины наблюдалась тенденция повышения уровня абстракций в языках программирования, обусловленная снижением трудоемкости создания программ, во всяком случае, на стадиях кодирования и отладки. Использование неких программно реализованных посредников между программой и вычислительной машиной – интерпретаторов – оказалось наиболее простым и действенным способом перехода от концепций, навязываемых машиной, к абстрактным вычислительным моделям высокого уровня.

На сегодняшний день понятия *высокоуровневости* и *интерпретируемости* языков программирования во многих случаях едва ли не отождествляются. Большинство современных языков программирования высокого уровня так или иначе вбирает в себя свойства, присущие парадигме интерпретации; это могут быть как чисто интерпретируемые языки (Python, Ruby, JavaScript), так и компилируемые, но обладающие развитыми возможностями рефлексии, которые обеспечиваются интерпретацией промежуточного представления (Java, C#).

Оплотом компилируемого исполнения остаются сравнительно немногочисленные области, в которых интерпретация или полностью неприменима, или по тем или иным причинам очевидно нежелательна. В современных условиях чаще всего для таких областей применяется язык C, причем в некоторых случаях единственной альтернативой ему оказывается программирование на языке ассемблера, которое в силу его чрезвычайно высокой трудоемкости можно всерьез не рассматривать.

Трудоемкость программирования на C ниже, чем для языков ассемблера, но все еще такова, что для использования C нужны серьезные причины. Этот язык применяют, в частности, когда во время исполнения заведомо невозможно присутствие какой бы то ни было программной инфраструктуры, как в случае ядер операционных систем и прошивок микроконтроллеров. Язык C также выбирают при наличии повышенных требований к безопасности, поскольку это едва ли не единственный язык, допускающий осмысленный ручной аудит исходных текстов. Кроме того, C позволяет создавать программы практически настолько же эффективными по времени и памяти, насколько это вообще возможно на конкретной используемой аппаратуре. Наконец, сколь бы странно это ни выглядело, именно на C удается добиться наибольшей степени переносимости программ (на уровне исходных текстов) между разнообразными аппаратными платформами и операционными системами.

В подавляющем большинстве случаев применения языка C такой выбор оказывается вынужденным; о разнообразных недостатках этого языка написано огромное количество текстов, но определяющей особенностью здесь можно считать чрезвычайную выразительную бедность этого языка, требующую большого количества ручной работы, от которой невозможно избавиться никакими ухищрениями. Вызывает недоумение тот факт, что среди сотен новых

языков программирования, появившихся позже языка C, ему до сих пор не нашлось адекватной замены.

В этом плане показателен язык Go, в котором его создатели, в том числе Кен Томпсон и Роб Пайк, по их собственному утверждению, пытались исправить ошибки, допущенные при создании C. Этот язык использует сборку мусора как основную модель управления памятью, что полностью исключает применение Go в большинстве ситуаций, в которых сейчас используется C. На примере языка Go можно понять важный факт: низкоуровневость языка программирования, или, если говорить в терминах, введенных выше, его особенности, продиктованные стремлением к чисто компилируемому исполнению, следует рассматривать не как «ошибки» дизайна языка, но, напротив, как обязательное свойство для языка программирования, который смог бы, наконец, заменить неуклюжий и очевидным образом морально устаревший язык C.

Обилие «высокоуровневых» (читай – интерпретируемых до той или иной степени) языков программирования и всего одного языка, широко используемого для «системных» задач (чистого C) подспудно формирует массовое убеждение, что при «системность» языка неизбежно влечет трудоемкость работы на нем, а снижение трудоемкости программирования возможно только внедрением в язык программирования возможностей, далеких от базового вычислителя.

Авторы появившегося не так давно языка Rust попытались (возможно, непреднамеренно) опровергнуть это утверждение, создав язык, пригодный как для низкоуровневого, так и для высокоуровневого программирования, не пожертвовав сформулированными выше свойствами. В частности, компилятор Rust позволяет создавать программы, работающие в самостоятельном окружении, а развитые возможности автоматического управления памятью не требуют поддержки времени выполнения. Цена этого – высокая сложность языка Rust и, как следствие, компилятора. Тем не менее, Rust демонстрирует, что высокоуровневый и выразительный язык программирования может оставаться в рамках парадигмы чисто компилируемого исполнения. Сможет ли Rust со временем заменить C – вопрос открытый; часть его возможностей не допускает тривиальной реализации, и это может стать препятствием для его использования в таких ситуациях, когда программисту требуется предсказуемость порождаемого машинного кода.

Достаточно любопытен в этом плане язык C++, который на ранних стадиях своего развития, оставаясь низкоуровневым, позволял благодаря поддержке абстрактных типов данных и перегрузки операций создавать абстракции произвольного уровня сложности. После введения в язык таких возможностей, как обработка исключений и идентификация типов во время исполнения (RTTI), C++ оказался безнадежно далек от соответствия принципам чистой компиляции; кроме того, даже на начальном этапе становления C++ не мог считаться чисто компилируемым (в смысле, введенном в нашей статье), поскольку, во-первых, обладал практически всеми недостатками чистого C (кроме разве что

интерпретации форматных строк printf/scanf), и, во-вторых, включал достаточно сложную реализацию виртуальных функций.

В результате последовательного принятия все более и более сложных «стандартов» C++ этот язык как явление к настоящему времени может рассматриваться только как язык высокого уровня, заведомо непригодный для задач системного программирования, несмотря на то, что среди всех популярных ныне высокоуровневых языков C++ остается единственным *компилируемым в машинный код*; все остальные либо интерпретируются, либо транслируются в код промежуточной виртуальной машины. В контексте нашей статьи здесь скорее важен тот факт, что, оставаясь компилируемым, C++ позволяет исключить рутинную ручную работу, хорошо известную программистам, пишущим на чистом C. Делается это за счет библиотечных расширений, опирающихся на поддержку в C++ полноценных абстрактных типов данных, которые позволяют контролировать не только выполнение обычных операций над объектами таких типов, но и их присваивание, копирование при передаче через параметры и при возврате из функций и т. п.

Если учесть как негативный, так и положительный опыт языка C++, можно создать новый язык, допускающий, с одной стороны, чистую компиляцию в терминах, введенных выше, но при этом, с другой стороны, позволяющий ввести сколь угодно сложные абстракции. Такой язык мог бы одинаково хорошо (при условии адекватного выбора используемых библиотек, различного в каждом конкретном случае) подходить для решения как «системных», так и для прикладных задач, то есть быть в полном смысле *универсальным*.

Список литературы

[1]. Malenfant J., Jacques M., Demers F.-N. A Tutorial on Behavioral Reflection and its Implementation. Proceedings of Reflection 96, 1997, pp. 1-20. Доступно по ссылке: http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf

[2]. Smith B. C. Procedural Reflection in Programming Languages. Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at the Massachusetts Institute of Technology, February 1982, 762 p. Доступно по ссылке: http://repository.readscheme.org/ftp/papers/bcsmith-thesis.pdf

[3]. McAllister N. Microsoft's Roslyn: Reinventing the compiler as we know it. IDG News Service. Дата обращения 20.10.2011 (online). Доступно по ссылке: https://www.infoworld.com/article/2621132/microsoft-net/microsoft-s-roslyn--reinventing-the-compiler-as-we-know-it.html

[4]. Limi A., Hathaway S. Monkey patch. Plone Foundation. Дата обращения 03.07.2008 (online). Доступно по ссылке: http://web.archive.org/web/20080604220320/http://plone.org/documentation/glossary/monkeypatch

[5]. Installation – Jekyll. Дата обращения 01.09.2017 (online). Доступно по ссылке: https://jekyllrb.com/docs/installation/

[6]. The Rust Language Tutorial (version 0.4). Дата обращения 30.04.2017 (online). Доступно по ссылке: https://static.rust-lang.org/doc/0.4/tutorial.html

Pure Compiled Execution as a Programming Paradigm

A.V. Stolyarov <avst@cs.msu.ru>
O.G. Frantsuzov <franoleg@intelib.org>
A.S. Anikina <a.anikina1993@gmail.com>
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. Interpreted execution of computer programs, its capabilities and advantages is well-covered in the computer science literature. Its key feature is reflection: the ability to access and modify the source code at run time. At the same time, interpreted execution has its shortcomings: lower performance; higher code fragility caused by the possibility to change code during run time; more complicated static analysis; runtime-tied ecosystems. And in some cases like embedded systems, runtimes and interpreted code are impractical or impossible, and compiled code with zero dependencies is the only option. Pure compiled execution can be treated as a paradigm directly opposite to reflection-powered interpretation. If the primary trait of interpreted execution is reflection, then pure compilation should cleanly separate development time and run time. This implies no part of translator being available during run time, no requirements for runtime libraries availability, and, finally, no dependence on the implementation details like variable names. While interpretation is wildly popular, compiled execution can be a conscious choice not only for low-level applications, but other cases as well. The dichotomy between low-level languages and expressive reflection-enabled language is a false one. It should be possible to create an expressive purely compiled programming language, and such a language might be equally suitable both for system programming and application development.

Keywords: programming paradigm; compilation; interpretation; reflection

DOI: 10.15514/ISPRAS-2018-30(2)-1

For citation: Stolyarov A.V., Frantsuzov O.G., Anikina A.S. Pure Compilation as a Programming Paradigm. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 2, 2018, pp. 7-24 (in Russian). DOI: 10.15514/ISPRAS-2018-30(2)-1

References

[1]. Malenfant J., Jacques M., Demers F.-N. A Tutorial on Behavioral Reflection and its Implementation. Proceedings of Reflection 96, 1997, pp. 1-20. Available at: http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf

[2]. Smith B. C. Procedural Reflection in Programming Languages. Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at the Massachusetts Institute of Technology, February 1982, 762 p. Available at: http://repository.readscheme.org/ftp/papers/bcsmith-thesis.pdf

[3]. McAllister N. Microsoft's Roslyn: Reinventing the compiler as we know it. IDG News Service, October 20, 2011 (online). Available at: https://www.infoworld.com/article/2621132/microsoft-net/microsoft-s-roslyn--reinventing-the-compiler-as-we-know-it.html

[4]. Limi A., Hathaway S. Monkey patch. Plone Foundation, Retrieved 2008-07-03 (online). Available at: http://web.archive.org/web/20080604220320/http://plone.org/documentation/glossary/monkeypatch

[5]. Installation – Jekyll. Retrieved 2017-09-01 (online). Available at: https://jekyllrb.com/docs/installation/

[6]. The Rust Language Tutorial (version 0.4). Retrieved 2017-04-30 (online). Available at: https://static.rust-lang.org/doc/0.4/tutorial.html