Программирование как вид человеческой деятельности

Введение

В качестве введения мне хотелось бы начать разговор с истории и цитат. История эта – о физике Людвиге Больцмане, который хотел достичь своих результатов путем громоздких вычислений. Кто-то однажды пожаловался на то, что его методы ужасны, на что Больцман заявил, что «об элегантности должны заботиться портные и сапожники», дав тем самым понять, что его самого это никоим образом не беспокоит.

В противоположность ему, я хотел бы процитировать другого известного ученого XIX века, Джорджа Буля. В своей книге «Исследование законов мышления», в главе «Условия совершенного метода», он писал: «Я говорю здесь не только о том совершенстве, которое состоит в могуществе, но и о том, которое основывается на концепции изящества и красоты. Вполне возможно, что тщательное изучение этого вопроса приведет нас к выводу, что совершенный метод должен быть не только эффективным по отношению к объектам, для которых он разработан, но и демонстрировать определенное единство и гармонию всех своих частей и процессов». Вряд ли кто-то не заметит коренного различия в этих подходах.

Мы подсознательно ассоциируем элегантность с роскошью. Возможно, это одна из причин того, что для нас само собой разумеется, что элегантность должна дорого обходиться. Одна из моих основных целей – показать, что элегантность может быть также выгодна. Это даст нам ясное понимание истинной природы качества программ и пути, которым оно может быть достигнуто, а именно – языка программирования. Поняв это, мы попытаемся вывести некоторые ключевые моменты, например, какие особенности языков программирования являются наиболее предпочтительными. Наконец, мы надеемся убедить вас в том, что различные цели конфликтуют друг с другом меньше, чем это кажется на первый взгляд.

О качестве результатов

Даже полагая, что машины работают безупречно, мы должны задать себе вопрос: «Когда компьютер выдает результаты, почему мы должны им доверять, если только мы им действительно доверяем?», а затем: «Какие меры мы можем предпринять, чтобы повысить степень нашей уверенности в том, что выданные результаты – это то, что нам нужно на самом деле?».

Насколько важен первый вопрос, можно проиллюстрировать на простом, даже несколько упрощенном примере. Предположим, что математик, работающий в области теории чисел, имеет в своем распоряжении машину с программой факторизации чисел. Этот процесс может завершиться двумя способами: либо он выдает факторизацию данного числа, либо отвечает, что заданное число является простым. Предположим теперь, что наш математик хочет подставить в этот процесс, скажем, число с 20-ю десятичными знаками, для которого у него есть веские причины полагать, что оно простое. Если машина подтверждает это ожидание, он будет счастлив; если она находит факторизацию, математик будет разочарован, так как интуиция снова подвела его, но, если он сомневается, он может взять счетную машинку и перемножить полученные множители, чтобы проверить, получится ли в результате исходное число. Если же, наоборот, машина выдает ответ, что данное число, согласно его ожиданиям и горячему желанию, является простым, с чего бы ему верить этому?

Наш пример демонстрирует, что даже в полностью абстрактных задачах получение результата не является четко определенным процессом, четко определенным в том смысле, что можно сказать: «Я сделал это», не беспокоясь за убедительность результата, то есть его «качество».

Ситуация, в которой находятся программисты, очень похожа на ту, в которой находятся чистые математики, которые разрабатывают теорию и доказывают результаты. Долгое время чистые математики думали – а некоторые из них до сих пор думают – что теорема может быть доказана полностью, что вопрос о том, является ли предложенное доказательство теоремы адекватным или нет, допускает абсолютный ответ «да» или «нет». Но это иллюзия, потому что как только кто-то начинает думать, что доказал что-то, он должен доказать, что его доказательство безукоризненно, и так далее, до бесконечности! Никто не может гарантировать, что доказательство корректно, в лучшем случае он может сказать: «Я не нашел ни одной ошибки». Порой мы льстим себе мыслью о неопровержимом доказательстве, но на самом деле все, что мы делаем, это лишь придаем правдоподобный вид своим заключениям.

Несмотря на все свои недостатки, математические рассуждения представляют замечательную модель того, как можно охватить чрезвычайно сложные структуры посредством мозга, возможности которого ограничены. Кажется, стоит разобраться, до какой степени эти проверенные методы могут быть перенесены в искусство использования компьютеров. В разработке языков программирования можно позволить себе руководствоваться в первую очередь тем, «что машина может делать». Впрочем, учитывая, что язык программирования – это мост между пользователем и машиной и фактически может рассматриваться как его инструмент, кажется столь же важным рассматривать, «что Человек может придумать». Именно в этом русле мы продолжим наше исследование.

О структуре убедительных программ

Техника управления сложностью известна с древних времен: “Divide et impera” (разделяй и властвуй). Аналогия между построением доказательства и построением программы, пожалуй, просто поразительна. В обоих случаях даны отправные точки (аксиомы и существующая теория против примитивов и доступных библиотечных программ), в обоих случаях задана цель (доказанная теорема против желаемых результатов), в обоих случаях сложность преодолевается делением на части (леммы против подпрограмм и процедур).

Я полагаю, что гениальность программиста соответствует сложности решаемой задачи, а также полагаю, что он сумел добиться подходящего разделения задачи. Затем он продолжает действовать следующим образом:

1. Он разрабатывает полные спецификации отдельных частей.

2. Он убеждается, что проблема в целом решается при наличии в его распоряжении частей программы, удовлетворяющих этим спецификациям.

3. Он разрабатывает отдельные части в соответствии со спецификациями, но при этом делает их максимально независимыми друг от друга и от окружения, в котором они будут использоваться.

Очевидно, что построение каждой такой отдельной части может снова оказаться сложной задачей, так что для данной части задачи потребуется дальнейшее разбиение.

Некоторые могут счесть описанный метод разбиения на части недостаточно прямолинейным и слишком извилистым путем достижения конечной цели. Мое собственное мнение я лучше всего могу выразить так: я твердо уверен, что «царских дорог в математике нет», или, другими словами, что у меня очень маленькая голова, и я вынужден обходиться ей. Поэтому я рассматриваю технику разбиения на части как базовый прием человеческого мышления и считаю, что стоит попробовать создать условия, в которых она может быть наиболее плодотворно применена.

Предположение о том, что программист сделал подходящее разбиение, находит подтверждение в том, что становится возможным выполнить первые два этапа: спецификацию частей и проверку, что они совместно решают задачу. Здесь элегантность, точность, ясность и тщательное понимание задачи являются необходимыми предпосылками. Но в целом техника разбиения основывается на том, что обсуждалось значительно меньше, а именно на том, я назвал бы «принципом невмешательства». На втором этапе подразумевается, что правильная работа целого может быть установлена путем рассмотрения только внешних спецификаций частей, без деталей их внутреннего строения. На третьем этапе принцип невмешательства всплывает снова; здесь подразумевается, что отдельные части могут быть поняты и построены независимо друг от друга.

Возможно, здесь уместно заметить, что если я правильно понял нынешнее отношение к проблеме определения языка, при несколько более формальном подходе состоятельность техники разбиения подвергается сомнению. Те, кто выдвигает возражения, аргументируют свою точку зрения следующим образом. Когда вы используете механизм, подобный описанному двухэтапному, во-первых, должны быть созданы спецификации, а во-вторых, описано, как все это работает. При этом вы вынуждены в лучшем случае сказать дважды одно и то же, но вероятнее всего, вы придете к противоречию. С точки зрения статистики, как ни грустно мне об этом говорить, последнее замечание достаточно серьезно. Единственный ясный путь к определению языка, возражают они, это просто определение механизмов, потому что все, что они будут делать, следует из этого. Мой вопрос: «А как оно следует?» мудро оставляют без ответа, и я боюсь, что это тонкое, но порой значительное различие между понятиями «определено» и «известно» сделают их работу интеллектуальным упражнением, которое ведет в тупик. После этого отступления вернемся к собственно программированию. Каждый, кто знаком с ALGOL 60, согласится, что его концепция процедуры в высокой степени удовлетворяет нашей концепции невмешательства, как по своим статическим (например, в свободном выборе локальных идентификаторов), так и по динамическим свойствам (например, возможность вызова процедуры, прямо или косвенно, из себя самой).

Другой впечатляющий пример улучшения ясности посредством невмешательства, гарантированного структурой, представлен всеми языками программирования, в которых допустимы алгебраические выражения. Вычисление этих выражений последовательной машиной, имеющей арифметическое устройство ограниченной сложности, подразумевает использование временного хранилища для промежуточных результатов. Их анонимность в исходном языке гарантирует невозможность того, что один из них будет нечаянно разрушен до использования, что было бы возможно при программировании в кодах машины Фон Неймана.

Сравнение некоторых альтернатив

Развернутое сравнение кода машины Фон-неймановского типа – хорошо известное отсутствием ясности – и различных типов алгоритмических языков было бы не лишним. Выполнение программы всегда состоит в периодическом взаимодействии двух информационных потоков, одного постоянного во времени («программы»), другого изменяющегося («данные»). Много лет одним из основных достоинств кода Фон-неймановского типа считалась возможность программы изменять свои собственные инструкции. Со временем мы обнаружили, что именно эта возможность в немалой степени ответственна за отсутствие ясности в программах на машинном коде. Тут же была поставлена под вопрос необходимость этого: все алгебраические компиляторы, которые я знаю, производят объектные программы, остающиеся неизменными все время выполнения.

Это наблюдение приводит нас к рассмотрению статуса переменной информации. Давайте ограничимся рассмотрением языков программирования без операторов присваивания и перехода. При условии, что набор доступных функций достаточно широк и понятие условного выражения входит в число примитивов, можно описать вывод любой программы как значение большой (рекурсивной) функции. Для последовательной машины она может быть транслирована в постоянную объектную программу, в которой во время выполнения стек используется для отслеживания текущей иерархии вызовов и значений фактических параметров, передаваемых этим вызовам.

Несмотря на элегантность подобного языка программирования, имеется серьезный довод против него. Информация в стеке может рассматриваться как объекты с вложенными временами жизни и постоянными значениями в течение всего времени жизни. Нигде (за исключением явного наращивания счетчика команд, отражающего ход времени) значение уже существующего именованного объекта не заменяется другой величиной. В результате единственный способ сохранить только что полученный результат – выложить его на верхушку стека; у нас нет способа выразить, что прежнее значение устарело, и время его жизни будет продолжаться, хотя безо всякого интереса для нас. Второй довод против, возможно, является следствием первого: такую программу после некоторого, довольно быстро достижимого, уровня вложенности будет ужасно трудно читать.

Обычное средство от этого – совместное введение операторов присвоения и goto. Оператор goto позволяет нам посредством перехода назад повторить часть программы, тогда как оператор присвоения может создать необходимое различие в состоянии между последовательными повторениями.

Но у меня есть причины спросить: будучи примененным в качестве спасительного средства, оператор goto не хуже ли сам по себе, чем тот дефект, который он призван исправить? Например, два менеджера программных отделов из разных стран и разной квалификации – один в основном ученый, другой в основном коммерсант – сообщили мне независимо друг от друга и по собственной инициативе о своем наблюдении, что квалификация их программистов обратно пропорциональна частоте появления оператора goto в их программах. Это было стимулом к тому, чтобы попытаться обойтись без оператора goto.

Идея состоит в следующем: то, то нам известно как «передача управления», т.е. подмена счетчика инструкций, - это операция, обычно подразумеваемая как часть более содержательного действия. Я упомяну переход на следующий оператор, вызов процедуры и возврат из нее, условные конструкции и оператор for. Это еще вопрос, не собьет ли программиста с толку наличие отдельного средства управления передачей управления. Я поставил несколько программных экспериментов и сравнил текст на ALGOL с текстом, полученным мной на модифицированной версии ALGOL 60, в котором оператор goto был упразднен, а оператор for – будучи слишком помпезным и сложным – заменен более простой

конструкцией повтора. Последние версии было потруднее создавать: мы так хорошо знакомы с командой перехода, что требовались некоторые усилия, чтобы забыть его! Во всех попытках, впрочем, программы без goto оказались короче и понятнее. Начальный шаг в повышении ясности достаточно понятен. Хорошо известно, что не существует алгоритма для определения, завершится данная программа или нет. Другими словами: каждый программист, который хочет создать безукоризненную программу, должен хотя бы убедиться сам, проверив, действительно ли завершается его программа. В программе, где goto применен в неограниченных количествах, этот анализ может оказаться очень трудным, учитывая большое разнообразие путей, по которым программа может зациклиться. После упразднения goto остаются только два способа, которыми программа может зациклиться: либо бесконечная рекурсия – т.е. через механизм процедур – либо конструкция цикла. Это весьма упрощает проверку.

Понятие цикла, столь фундаментальное в программировании, имеет еще один аспект. Нет ничего необычного в том, что среди последовательности повторяющихся операторов появляется одно или более подвыражений, которые не изменяют значение в ходе цикла. Если такая последовательность должна повторяться много раз, было бы досадной потерей машинного времени перевычислять одни и те же значения снова и снова. Один из способов избежать этого – переложить на оптимизирующий транслятор выявление подобных константных подвыражений, чтобы он мог вынести вычисление их значений за пределы цикла. При отсутствии оптимизирующего транслятора очевидное решение – предложить программисту расписать их явно путем введения стольких дополнительных переменных, сколько константных подвыражений имеется в цикле, и присвоения им значений перед входом в цикл. Я должен подчеркнуть, что оба способа написания программ в равной степени сбивают с толку. В первом случае транслятор сталкивается с ненужной головоломкой по выявлению константности, во втором – мы вводим переменную, единственное назначение которой – обозначать константное значение. Последнее рассмотрение указывает выход из этого затруднения: помимо переменных, в распоряжении программиста должны быть «локальные константы», т.е. идентифицируемые величины с ограниченным временем жизни, в течение которого они сохраняют постоянное значение, заданное в момент появления величины. Эти величины не новы: формальные параметры процедур уже демонстрируют это свойство. Вышеупомянутое – это призыв осознать, что концепция «локальной константы» имеет право на существование. Если меня правильно информировали, это уже было заложено в CPL, язык программирования, разработанный совместными усилиями в математической лаборатории Кембриджского университета в Англии.

Двойная выгода от ясности

Я подробно рассуждал о том, что убедительность результата сильно зависит от ясности программы, от степени, в которой она отражает структуру выполняемого процесса. Для тех, кто в первую очередь озабочен эффективностью, измеряемой сомнительной мерой в виде занимаемой памяти и машинного времени, я хотел бы указать, что увеличение эффективности всегда ведет к использованию структуры. Для них я хотел бы особо подчеркнуть, что все упомянутые структурные свойства могут быть использованы для повышения эффективности реализации. Следовало бы снова пересмотреть их вкратце.

Аспекты, связанные со временем жизни, разрешаемые при помощи локальных величин процедуры, позволяют нам разместить их в стеке, тем самым используя доступную память весьма эффективно; анонимность промежуточных результатов позволяет нам динамически минимизировать обращения к памяти при помощи автоматически управляемого набора аккумуляторов; постоянство текста программы во время выполнения очень полезно

на машинах с разными уровнями памяти и значительно уменьшает сложность управления; оператор цикла облегчает динамическое выявление бесконечного зацикливания, и, наконец, локальные константы – подходящие кандидаты для хранения в памяти с медленной записью и быстрым чтением, где она имеется.

Позвольте мне подвести итог. Когда я познакомился с понятием алгоритмических языков, я никогда не оспаривал господствующего тогда мнения, что проблемы разработки и реализации языка – в основном вопрос компромиссов: каждое новое удобство для пользователя должно быть оплачено при реализации, либо в виде дополнительных проблем при трансляции, либо при выполнении, либо при том и другом. Что ж, мы определенно живем не в раю, и я не собираюсь отрицать возможность конфликта между удобством и эффективностью, но теперь я возражаю, когда этот конфликт подается как закономерный итог ситуации. Я придерживаюсь мнения, что стоило бы разобраться, до какой степени интересы Человека и Машины совпадают, и посмотреть, какие технологии мы можем изобрести на пользу всем нам. Я верю, что это исследование принесет плоды, и если этот рассказ пробудит в вас такую же надежду, он достиг своей цели.

Prof. Dr. E.W.Dijkstra
Department of Mathematics
Technological University
P.O.Box 513
EINDHOVEN
The Netherlands