ISSN 1992-6502 (Print)
2021. Т. 25, № 4 (94). С. 91–100
Вестник УГАТУ
ISSN 2225-2789 (Online)
http://journal.ugatu.ac.ru

УДК 681.3.06:62-503.5 doi 10.54708/19926502_2021_2549491

КОНЦЕПЦИЯ АЛЬТЕРНАТИВНОГО ПРОГРАММИРОВАНИЯ РАСПРЕДЕЛЕННОГО ОБЪЕКТНОГО КОДА НА ОСНОВЕ ПОТОКОВ ДАННЫХ МЕЖДУ УЗЛАМИ КОЛЛЕКТИВА ВЫЧИСЛИТЕЛЕЙ. ИНЖЕНЕРНЫЙ ПОДХОД

В. А. Крюков

VAK_53@mail.ru

АО «Научно-исследовательский институт технологии и организации производства»

Поступила в редакцию 08.11.2021

Аннотация. Существующие распространенные парадигмы программирования, несмотря на прогресс в области разработки средств программирования, интуитивно недоступны специалистам предметных областей, охваченных автоматизацией, особенно в области управления технологическими процессами и механизмами. Налицо усиление проблемы семантического разрыва. Обосновывается и описывается альтернативная концепция распределенного программирования на основе потоков данных между узлами коллектива вычислителей. В предлагаемой парадигме можно успешно описывать алгоритмы на уровне понятий предметной области и решать задачи при помощи распределенного параллельного программирования.

Ключевые слова: польская запись; распределенное параллельное вычисление; функциональное программирование; гонка данных.

Догмы спокойного прошлого не годятся
для бурного настоящего.
А. Линкольн

ВВЕДЕНИЕ

Предлагаемый труд есть плод анализа производственной задачи разработки системы управления экспериментальной технологической установки автоклава, оснащенного дополнительным оборудованием типа мешалок и насосов. Согласно техническому заданию необходимо было обеспечить быструю перенастройку установки для проведения новых экспериментов. Процесс проектирования системы управления привел к неожиданным результатам, которые заставили вернуться к основам программирования и пересмотреть некоторые незыблемые положения.

ПРЯМАЯ И ОБРАТНАЯ ПОЛЬСКАЯ ЗАПИСЬ

Теоретическая основа современных трансляторов для компьютеров базируется на обратной польской записи для создания объектного кода программ, выполняемых в стеке. Но из теории известно, что помимо обратной, или суффиксной польской записи, имеет место быть и прямая польская запись – префиксная [1].

О ее свойствах и применении практически ничего не известно. В русском переводе многотомного «Искусство программирования для ЭВМ» Д. Кнута [2] вскользь указывается на наличие двух форм «польских обозначений» и говорится, что польская запись имеет непосредственное отношение к трем основным порядкам прохождения деревьев. Известно

еще, что Lisp для записи языковых конструкций использует префиксную польскую запись по форме, но для вычисления использует ее свертку на стеке.

Возникают вопросы: можно ли вторую форму польской записи тоже использовать в вычислении? Чем отличается такой вычислитель? Запрос темы в поисковых машинах Internet «использование префиксной польской записи в вычислителе» ничего не выдает.

Проведем сравнительный анализ и синтез вычислителя прямой польской записи самостоятельно.

Напомним суть преобразования выражения в польскую запись. Начнем с простейшего инфиксного выражения 2*2.

Обратная польская запись: 2 2 *. В последовательности вначале идут параметры, которые завершаются знаком операции.

Прямая польская запись: * 2 2. В последовательности вначале идет знак операции, за которым следуют параметры.

Усложним выражение: (7–5)*(1+1)

Обратная польская запись: 7 5 – 1 1 + *

Прямая польская запись: (*(– 7 5) (+ 1 1))

Для удобства восприятия полученной прямой польской записи она дана в синтаксисе языка Lisp с круглыми скобками, которые можно опустить без потери смысла.

В результате преобразования инфиксного выражения в обратную польскую запись получается свертка выражения в объектный код, который в потоке можно передавать в стек для автоматического вычисления выражения. В результате такой свертки мы имеем простые машинные представления и эффективные вычислительные средства, что является несомненным достоинством обратной польской записи.

В свою очередь, анализ прямой польской записи выражения показывает, что она является сверткой дерева решения, которая при чтении записи в потоке (по аналогии с объектным кодом) легко развертывается в иерархическую структуру дерева. Граф развернутого дерева решения рассмотренного выражения представлен на рис. 1.

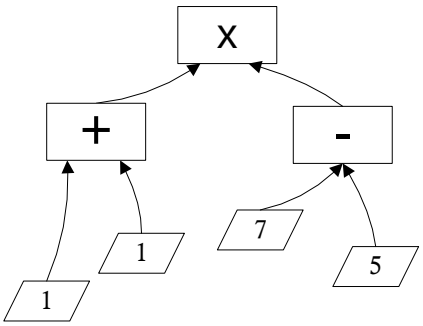

Рис. 1. Граф дерева решения, развернутого при однопроходном чтении прямой польской записи

Направление стрелок графа задает направление потока данных вычисления решения. Граф дерева решения является обратно ориентированным деревом, в смысле теория графов: дуги направлены от листьев через узлы к вершине. Узлы дерева решения содержат знаки операций, листья – числовые аргументы операций. Количество входящих дуг узла показывает арность соответствующей операции.

В отличие от объектного кода обратной польской записи, приспособленного для вычислений на стеке[1], код прямой польской записи подготовлен к транслированию в древовидную структуру вычислителей, принимающих потоки данных. Заметим, что подчиненные ветки дерева решения (поддеревья вычисления) «–» и «+» могут быть в процессе развертки программы[2] в структуру дерева переданы на другие распределенные узлы коллектива вычислителей.

---

[1] *Конечно, программа устроена более сложно, но процесс вычисления сводится именно к обработке на стеке.*

[2] *Будем понимать термин программа в традиционном смысле этого слова, как заданную последовательность действий, хотя, весьма вероятно, будут отличия.*

Выбор способа реализации узла остается за разработчиком программы. Узел может рассматриваться как черный ящик с очерченным функционалом и программным интерфейсом. Предпочтение разработчика может быть отдано отлаженным быстрым узлам на традиционных языках программирования, как например, C и Python. Здесь мы в описании абстракции дерева решений забежали немного вперед, но необходимо было подчеркнуть, что «чистое» дерево решений легко трансформируется в гетерогенное дерево, отдельные узлы которого представляют собой свертки обратной польской записи.

Для сохранения ясности и понимания важно следовать общей модели прямой польской записи, при которой сохраняются понятные человеку представления на машинном уровне. При обратной польской записи это не удается сделать: появляется семантический разрыв между программной сверткой алгоритма и прикладными знаниями, который не удается устранить никакими надстройками над базисом типа объектно-ориентированного программирования [3].

Коренное отличие прямой польской записи от обратной – обратная польская запись является моделью потока операции (и функций) над полем данных, а прямая польская запись представляет поток данных над полем операций (и функций). Причем цели обеих моделей одинаковы – это формализация решения задач пользователя.

РЕШЕНИЕ КВАДРАТНОГО УРАВНЕНИЯ С ПОМОЩЬЮ ДЕРЕВА РЕШЕНИЯ

В качестве примера отображения алгоритма на дерево решения рассмотрим обычное квадратное уравнение:

$$ax^2 + bx + c = 0.$$

В этом примере можно найти:

– элементы параллельного программирования, причем, потоки управления как распараллеливаются, так и сливаются в точке (узле);

– проблему гонки данных.

Традиционное описание алгоритма решения данного уравнения следующее:

1. Вычисление дискриминанта D по формуле

$$D = b^2 - 4ac. \quad (1)$$

2. Если $D < 0$, то квадратное уравнение не имеет действительных корней: цепочка вычислений прерывается.

3. Если $D \geq 0$, то квадратное уравнение имеет два или один корень:

$$x_1 = \frac{-b - \sqrt{D}}{2a} \quad (2)$$

и

$$x_2 = \frac{-b + \sqrt{D}}{2a}. \quad (3)$$

Представим алгоритм в виде дерева решений в нотации, близкой к стандартной по ГОСТ 19.701–90.

Схема на рис. 2 требует небольших комментариев:

– схема не может быть получена трансляцией формул (1)–(3) в программу на современном языке программирования: схема по сути сама является программой;

– потоковые данные *a, b, c, d, sqrt* и т.д. обозначены рамкой в форме косого четырехугольника, у конечных данных результата *server_x1* и *server_x2* над четырехугольником проведена черта, символизирующая, что в рамках данного алгоритма данные уже не передаются в другие узлы дерева решения; эти стационарные данные будем называть *memo*-данными;

– узлы дерева решения являются исполняющимися блоками или функциями (об этом можно судить по надписям), в которые входят аргументы и выходит результат;

– в дереве решения узлы не развернуты до уровня элементарных операций; несущественный для потока управления функционал можно скрыть; например, узел *sqrt* показан без извлечения корня и без проверки дискриминанта на отрицательность. Да и вообще, узлы следует рассматривать как «черные ящики».

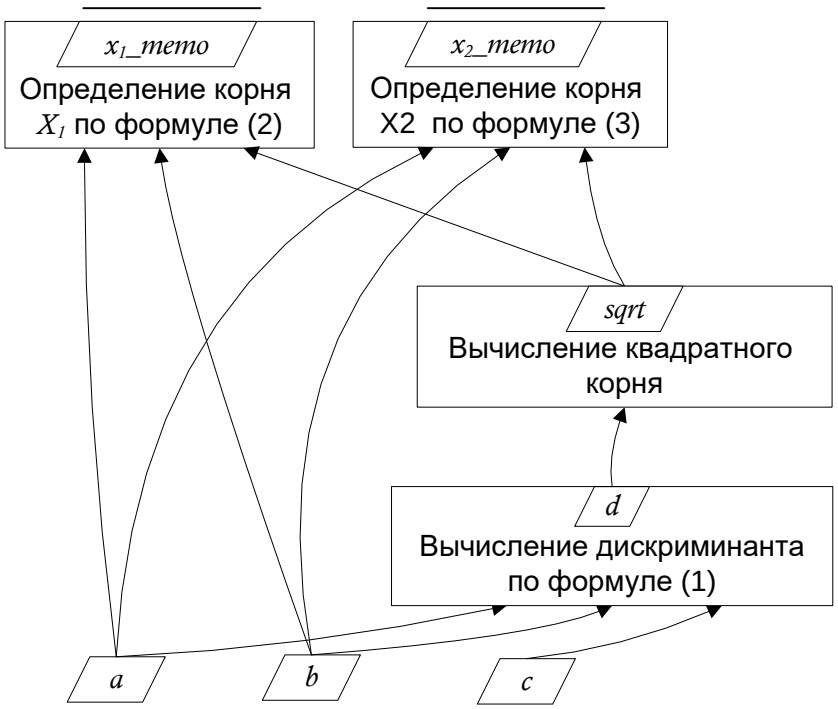

Рис. 2. Дерево решения квадратного уравнения

Из схемы можно сделать несколько следующих выводов:

1. Для данного уровня абстракции неважно, откуда берутся первичные данные: они могут вводиться с клавиатуры, поступать из компьютерной сети, считываться с датчиков в режиме реального времени. Важно, что первичные данные поступают в дерево решения асинхронно. Фактически узлы *a, b* и *c* являются независимыми параллельными процессами генерации данных.
2. Восходящие параллельные потоки данных могут замыкаться, как, например, в узле *d*.
3. Исполняющие узлы могут поставлять результат, например, *sqrt*, параллельно в несколько адресов.
4. В примере вычислительный процесс делится на два независимых процесса в узлах *x*1 и *x*2. В программировании параллельных потоков это называется «распараллеливание по данным». В результате, дерево решений имеет две вершины. В общем случае их может быть еще больше и в конечном итоге мы можем иметь дело с большим количеством решений.
5. Узлы вычисления могут начать выполняться только по мере готовности их входных аргументов. Должна быть обеспечена синхронизация поступления данных на вход вычислителя. Если актуальность данных со временем может «испариться», возникает временной фактор, учет которого подводит к системам реального времени.
6. Узлы дерева решения последовательного потока управления можно агрегировать в более крупные блоки, которые скрывают несущественные мелкие детали вычислений. Их внутреннее устройство можно рассматривать как «черный ящик», о чем мы говорили ранее: блоки вычислений могут быть реализованы как угодно, в том числе стековым образом, т.к. программы на стеках показывают высокую производительность и вычислительную эффективность. Поэтому лучше иметь в виду именно гетерогенную архитектуру вычисления.

7. Завершение решения произойдет при получении данных в корнях дерева решения или при вычислении квадратного корня из отрицательного дискриминанта с выводом соответствующего предупреждения на экран терминала оператора, но цепочка вычисления корней будет постоянно готова к повторению вычисления при получении данных *a, b* и *c*.

Представленный шаблон может легко модифицироваться, адаптироваться и расширяться. Например, по этой диаграмме можно описать выполнение процессов PID-регулятора.

ПОТОКОВЫЕ ДАННЫЕ И MEMO–ДАННЫЕ

Из рассмотренной схемы можно получить еще одно интересное заключение.

Данные на схеме на рис. 2 мы разделили на потоковые, которые передаются от узла к узлу, и хранимые, над которыми проведена горизонтальная черта.

Потоковые данные не находятся стационарно в ячейках памяти, а находятся в движении «как ртуть». Все узлы дерева решения на рис. 2 связаны единым потоком данных в ориентированный связный граф, поэтому справедливо назвать его деревом потока данных.

Хранимые данные мы обозначили суффиксом *memo*, чтобы подчеркнуть их стационарность. Это общие данные, которые необходимо хранить для «внешнего потребления», например, для транспортировки на OPC-сервер.

Далее из примера на *Erlang* будет ясно, что по сути *memo*-данные отличаются от потоковых данных наличием механизма принудительной транспортировки данных по запросу *get*. Это специфичное взаимодействие с чужим процессом показано на рис. 3. Для обобщения можно сказать, что фактически все потоковые данные транспортируются по запросу типа *put*.

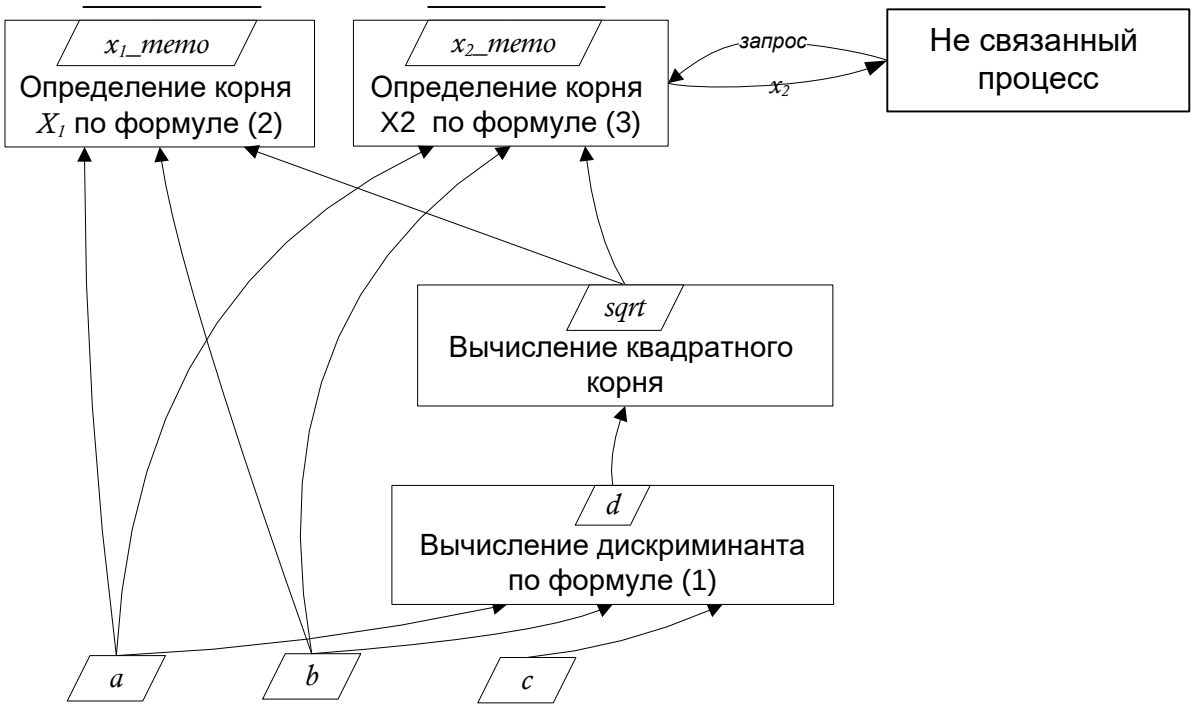

Рис. 3. Дерево потока данных и взаимодействие с «чужим» потоком через *memo*-данные

Наш поток данных и «чужой» поток являются несвязанными или слабосвязанными процессами вычисления.

Мы подробно рассмотрели вопрос о существовании *memo*-данных, потому что они являются, по сути, глобальными объектами параллельного программирования. В программировании на традиционных языках они размещаются в общей памяти, доступ к которой необходимо защищать (блокировать), например, мьютексами. Но это уже другая тема.

ВЫБОР ЭКСПЕРИМЕНТАЛЬНОГО ДВИЖКА ДЛЯ ВЫЧИСЛЕНИЯ ДЕРЕВА РЕШЕНИЯ

Попробуем реализовать дерево потока данных решения квадратного уравнения на какомлибо подходящем языке, чтобы наши рассуждения подкрепить фактологическим материалом. Не вдаваясь в подробности предварительных изысканий, сразу скажу, что мой выбор остановился на языке *Erlang*, хотя сам я специалистом по функциональному программированию не являюсь.

Перечислим полезные характеристики *Erlang*:

– *Изначальная параллельность процессов.* Вместо использования потоков вычислений, которые обмениваются данными с помощью разделяемой памяти, в *Erlang* для каждого процесса выделяется отдельная область памяти. Процессы взаимодействуют друг с другом через передачу сообщений. Сообщение может содержать любой терм значения в *Erlang*. Сообщения передаются асинхронно.

– *Распределенные вычисления.* Средства распределенного программирования встроены в синтаксис и семантику языка. По умолчанию узлы (средства выполнения *Erlang*) обмениваются данными по протоколу TCP/IP. Они могут быть соединены в однородную сеть, в которой каждый из узлов не зависит от операционной системы, на которой он запущен.

– *Применение в системах реального времени.* Несмотря на то что *Erlang* – высокоуровневый язык, его можно использовать в системах псевдореального времени (*soft real-time system*).

– *Надежность.* Процессы в *Erlang* могут быть соединены (*link*) так, что если один из них падает, то второй узнает об этом и либо исправит проблему, либо сам завершится. Обобщенные процессы соединяются с процессом-наблюдателем (*supervisor*). Единственная обязанность процесса-наблюдателя заключается в наблюдении за процессами и обработке завершения процессов.

– *Интеграция с другими программными компонентами*, написанными на традиционных языках программирования, например, *C* и *Python* (здесь уместно вспомнить о гетерогенной природе дерева решений). Это может быть использовано для получения интерфейсов, например, к протоколу Modbus через стороннюю библиотеку.

Импонировала возможность включения охранных выражений в выражение приема и обработки сообщений. Было понятно, что в *Erlang* легко организовать:

– структуру узлов процессов для моделирования дерева решений;

– работу потоков данных в дереве решения при помощи посылок сообщений между узлами.

Вспомним, что в дереве потока данных операция на вычислителе выполняется по мере готовности его входных аргументов, например, операция $x1$ может быть выполнена только после получения аргументов входных процессов a и b и результата от нижележащего вычислителя *sqrt*. Но в *Erlang* нет готового механизма синхронизации гонки данных. Его придется изобретать и реализовывать самостоятельно.

СИНХРОНИЗАЦИЯ ДАННЫХ НА ВХОДЕ ВЫЧИСЛИТЕЛЬНОГО БЛОКА В ДЕРЕВЕ РЕШЕНИЯ. РЕШЕНИЕ ПРОБЛЕМЫ «ГОНКИ ДАННЫХ»

Как мы уже отметили, все вычислители необходимо заставить работать слаженно, и на вход вычислительных узлов поступление данных в качестве аргументов должно происходить синхронно. Необходимость в синхронизации потока данных приводит к учету временного фактора в движке вычисления. Этот вопрос нуждается в дальнейшем исследовании, но уже на этапе понятно, что он чрезвычайно актуален в системах управления реального времени.

Можно предложить несколько решений проблемы синхронизации данных:

1. «Ленивая» синхронизация методом ожидания прихода всех аргументов.

Здесь стратегия синхронизации простая – вычислительный блок ждет прихода своих аргументов, чтобы потом произвести вычисления. Этот способ используется в движке решения задачи, приведенной в статье.

2. Контроль поступления данных в заданном интервале временной разбежки.

Данные двигаются снизу непрерывным потоком, что характерно для данных, поступающих от датчиков в системе реального времени. Для управления этим процессом предлагается использовать метод задания и контроля временной разбежки прихода данных к вычислителю.

Проведем некоторую аналогию. В электротехнике известна задача «гонки сигналов», т.е. *n* входных сигналов должны по возможности одновременно прийти от входов устройства по цепочке элементов к выходному элементу устройства, чтобы произвести правильный выходной сигнал. При этом в переходный период возможно появление на выходах устройства некоторых промежуточных значений сигналов, не соответствующих заданному состоянию устройства. Наиболее часто используемый способ борьбы с гонками сигналов сводится к учету времени задержки распространения сигнала в устройстве.

Метод временной разбежки данных не стремится рассчитать задержку гонки данных, а направлен на контроль временного диапазона прихода данных. Если временной диапазон разбежки получения очередного порции определенных данных превысил допустимый установленный порог, то процесс может решить, что набор полученных данных устарел и «обнулить» все данные, чтобы начать сеанс приема сначала. Или решить прекратить сеанс ожидания и подать сигнал тревоги о внештатной ситуации. А может увеличить время ожидания. Все зависит от стратегии, заложенной в логике контроля гонки данных.

3. Теоретически допустимая «принудительная» синхронизация методом «рекурсивного спуска» по дереву решения вниз и инициализации нижележащих блоков.

При такой синхронизации все данные будут гарантированно вычислены, а программа выполнена. Однако здесь есть два возражения:

– рекурсивный спуск сродни структуре традиционного стека;

– мы ограничились рассмотрением только восходящих потоков данных.

Спуск от верхней точки дерева решений может быть применен при развертывании (загрузки) дерева решений на вычислительные средства, но этот вопрос нуждается в отдельном исследовании.

ПРОТОТИП ДВИЖКА УПРАВЛЕНИЯ ПОТОКАМИ

Программный код на языке *Erlang* решения квадратного уравнения как прототипа движка управления потоками размещен на сайте *https://github.com/VAK-53/anti-turing*.

Язык *Erlang* заставляет программиста следовать наработанным канонам, в чем я убедился на личном опыте, приступая к реализации движка дерева решений. Приведу выписку рекомендации из [4]:

«Существует общий шаблон поведения процессов, который не зависит от их назначения. Процесс должен быть запущен и зарегистрирован. Первым действием инициализируются данные цикла обработки процесса. Данные цикла – это обычно аргументы, передаваемые в *spawn*[3], их мы будем называть состоянием процесса (*process state*). Далее состояние передается функции выполнения цикла, которая принимает сообщение, обрабатывает его, обновляет состояние и рекурсивно вызывает себя с этим состоянием. Если в функцию поступает сообщение *stop*, процесс освобождает ресурсы и завершается. Эту схему построения процесса мы будем называть шаблоном проектирования (*design pattern*). Она будет часто встречаться вне зависимости от выполняемой процессом задачи. Схема шаблона процесса изображена» на рис. 4.

[3] *Spawn – встроенная в Erlang функция запуска новых процессов.*

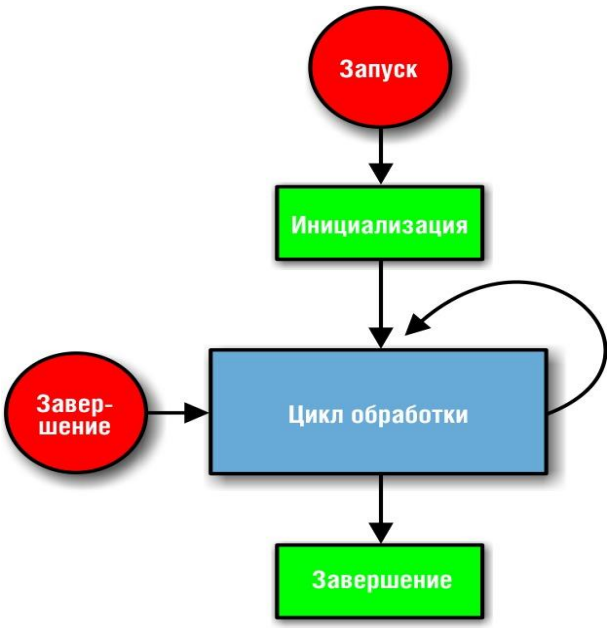

Рис. 4. Схема шаблона процесса

Движок был реализован в точности с рекомендациями построения процессов в языке *Erlang*, причем с рекомендациями я познакомился уже после реализации движка. И заслуга в этом принадлежит разумной и выверенной конструкции языка.

Внешние функции модуля движка называются *start_a/0, start_b/0* и *start_c/0* в соответствии с названиями входных параметров *a, b* и *c*. В них же производится инициализация данных *a, b* и *c* атомом (начального значения) *undefined.* И все потоковые данные инициализируются значением атома *undefined.*

В начале программного модуля при запуске строится дерево потока данных в соответствии с деревом решения на рис. 3.

Входные параметры *a, b* и *c* считываются в терминалах асинхронно циклически.

Содержательная часть программы, реализующая алгоритм потока решения, соответствует дереву потока данных схемы на рис. 3. Работа алгоритма реализована через механизм посылки сообщений от узла к узлу, а доступ к *memo*-данным *x*1-*memo* и *x*2-*memo* канонически скрыт оберткой в виде экспортируемых функций *get().*

Гонка входных аргументов на вход процесса решается простой проверкой отсутствия среди входных данных экземпляра со значением *undefined*. После срабатывания процесса в узле всем входным аргументам снова присваиваются значение *undefined* для подготовки выполнения новой итерации расчетов.

Memo-данные *x*1-*memo* и *x*2-*memo* хранятся в виде переменных состояния в процессах с хвостовой рекурсией (собственно говоря, все процессы в программе являются процессами с хвостовой рекурсией). В примере внешний несвязанный процесс запускается в отдельном узле и выводит полученное значение на экране. При достаточной степени абстрагирования процесс можно уподобить некоторому получению данных с OPC-сервера с целью построения отчета для оператора.

При испытании программного моделя *Анти-Тьюринг* запускался в четырех терминальных окнах. Результат испытаний программы показан на рис. 5.

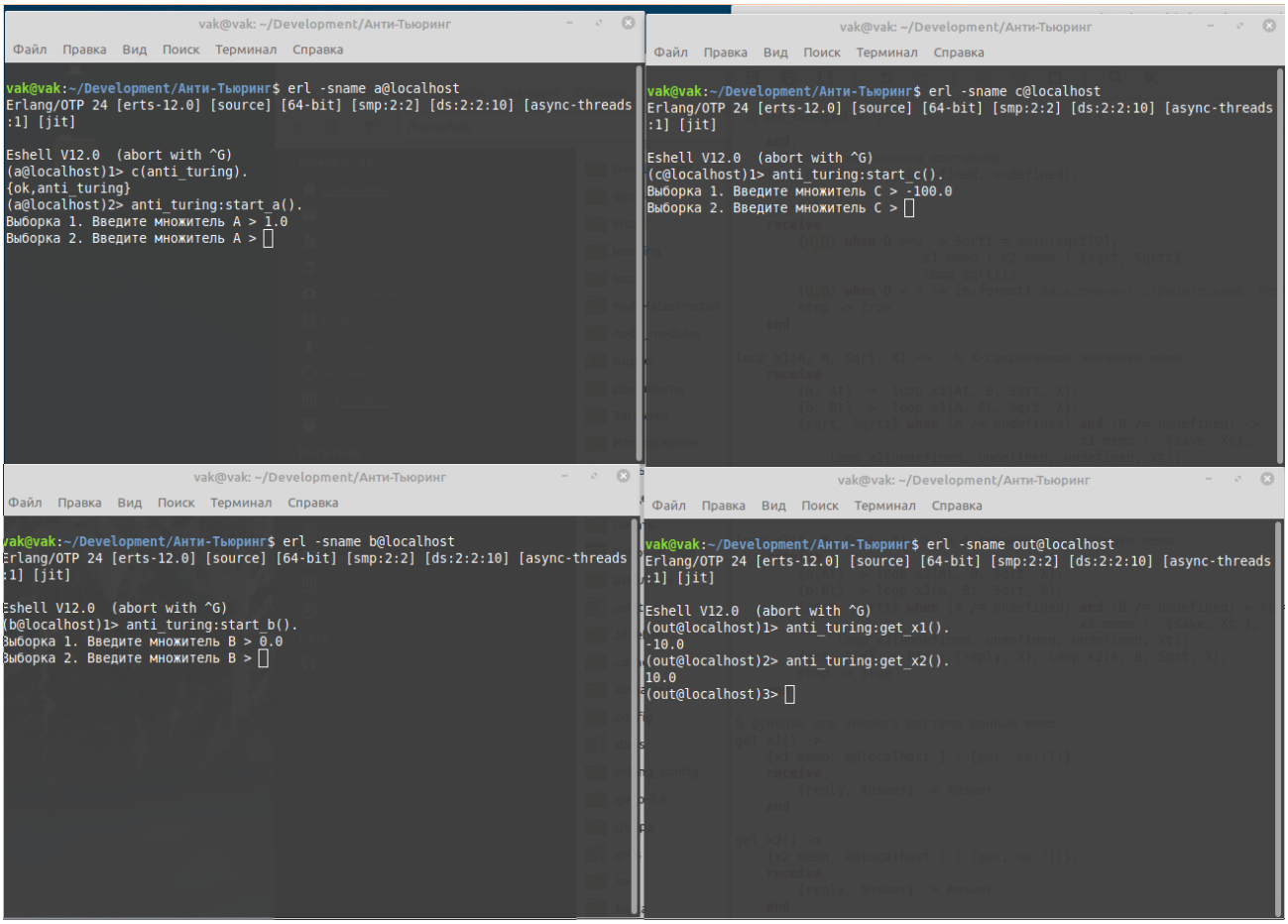

Рис. 5. Результат работы программы Анти-Тьюринг

Асинхронные процессы задания коэффициентов *a*, *b* и *c* запускались в узлах самостоятельных вычислителей. В четвертом окне был запущен несвязанный процесс для запроса результатов вычисления.

Испытания программы показали работоспособность предлагаемой парадигмы дерева потока. Работу парадигмы можно описать как «вычисление через переписку по почте» (ВЧП). Думаю, что это может вызвать в памяти ряд ассоциативных примеров.

ЗАКЛЮЧЕНИЕ

1. В предлагаемой альтернативной парадигме можно успешно описывать алгоритмы и решать задачи. На базе новой парадигмы возможно создание языковых средств программирования, ориентированных на программирование потоков данных. Предполагаемый движок (коллектив вычислителей) позволит программировать в стиле связывания функциональных узлов и выполнения вычисления через почту.

2. В альтернативной парадигме вычислений отсутствует понятие общая память коллективного пользования, вместо нее – поток данных внутри коллектива распределенных вычислителей, организованных в дерево решений. Вычислители операций являются событийными устройствами (реагирующими на сообщение) и находятся в постоянном ожидании готовности своих аргументов.

Важно отметить, что данные в памяти как набор 0 и 1 теряют свой семантический смысл и приобретают его в совокупности с вычислителем с соответствующим алгоритмом. В отличие от объектно-ориентированного программирования данные не инкапсулируют функции, а наоборот «обволакиваются» функциональными узлами и разработчики функциональных языков похоже восприняли эту идиому для реализации в виде абстракции структур [5].

3. Выявлена грань соприкосновения между традиционным программированием и программированием потоков данных. Оптимальная система программирования является открытой и гетерогенной; исполнительные блоки могут быть написаны на традиционных современных языках программирования, которые высокоэффективно исполняются на стеках.

4. С самого начала данной работы ставилась цель достижения цели активного участия в работе узкопрофильного специалиста предметной области на стадии конфигурирования системы управления гипотетическим устройством. Задача решается наработкой пула общих понятийных идиом потоковой обработки данных в качестве мостика через семантический разрыв между пользователем и программистом.

5. Существует предпосылка для разработки управляющих систем устройствами и технологическими установками на новых базовых принципах.

6. Априори в дереве решения отсутствует гонка и конкуренция между потоками процессов. Но имеется гонка данных. Рассмотрены способы синхронизации потоков данных на параллельных асинхронных потоках.

СПИСОК ЛИТЕРАТУРЫ

1. **Трамбле Ж., Моренсон П.** Введение в структуры данных. М.: Машиностроение, 1982. 770 с. **[** J. Tremblay, P. Morenson, *Introduction to data structures,* (in Russian). Moscow: Mashinostroenie, 1982. **]**

2. **Кнут Д.** Искусство программирования. Т. 1. М.: Мир, 1976. 734 с. **[** D. E. Knuth, *The art of programming,* (in Russian). Vol. 1. Moscow: Mir, 1976. **]**

3. **Суздальницкий И.** Мнение: объектно–ориентированное программирование — катастрофа на триллион долларов. [Электронный ресурс]. URL: https://tproger.ru/translations/oop-the-trillion-dollar-disaster/ (дата обращения 20.10.2021). **[** I. Suzdalnitsky (2021, Oct. 20). “Opinion: Object-Oriented Programming is a Trillion-Dollar Disaster”, [Online], (in Russian). Available: https://tproger.ru/translations/oop-the-trillion-dollar-disaster/ **]**

4. **Томпсон С., Чезарини Ф.** Программирование в Erlang. М.: ДМК Пресс, 2010. 486 с. **[** S. Thompson, F. Cesarini, *Programming in Erlang,* (in Russian). Moscow: DMK Press, 2010. **]**

5. **Юрич С.** Elixir в действии. М.: ДМК Пресс, 2020. 376 с. **[** Yurich S., *Elixir in action*, (in Russian). Moscow: DMK Press, 2020. **]**

6. **Федотов И.** Модели параллельного программирования. М.: Солон-Пресс, 2012. 384 с.: ил. **[** I. Fedotov, *Parallel programming models*, (in Russian). Moscow: Solon-Press, 2012. **]**

ОБ АВТОРЕ

КРЮКОВ Вячеслав Анатольевич, вед. программист АО «Научно–исследовательский институт технологии и организации производства»**.**

METADATA

Title: The concept of alternative programming distributed object code based on the data flows between the nodes of the collective of computers. Engineering approach.

Author: V. A. Kryukov

Affiliation: JSC Research Institute of Technology and Organization of Production (NIIT), Russia.

Email: VAK_53@mail.ru

Language: Russian.

Source: Vestnik UGATU (scientific journal of Ufa State Aviation Technical University), vol. 25, no. 4 (94), pp. 91-100, 2021. ISSN 2225-2789 (Online), ISSN 1992-6502 (Print).

Abstract: Existing common programming paradigms, despite advances in the development of programming tools, are intuitively inaccessible to specialists in the subject areas covered by automation, especially in the area of process and mechanism control. There is an intensification of the problem of the semantic gap. An alternative concept of distributed programming based on data flows between nodes of the collective of computers is justified and described. The proposed paradigm can successfully describe algorithms at the level of domain concepts and solve problems using distributed parallel programming.

Key words: polish notation; distributed parallel computing; functional programming; data race.

About author:

KRYUKOV, Vyacheslav Anatolievich, Lead Programmer, JSC Research Institute of Technology and Organization of Production.