Научная статья

УДК 004.43
DOI 10.25205/1818-7900-2021-19-4-16-35

Абстрактная машина языка программирования учебного назначения СИНХРО

Лидия Васильевна Городняя

Новосибирский государственный университет
Новосибирск, Россия

Институт систем информатики им. А. П. Ершова
Сибирского отделения Российской академии наук
Новосибирск, Россия

lidvas@gmail.com, https://orcid.org/0000-0002-4639-9032

Аннотация

Статья посвящена ряду решений по организации работы с памятью в учебных языках и системах программирования, нацеленных на обучение подготовке многопоточных программ над общей памятью. Рассмотрение выполнено на материале учебного языка программирования СИНХРО, что позволило анализировать варианты таких решений без ограничений, характерных для традиционных производственных инструментов и устройства стандартных систем программирования. В статье дано определение абстрактной машины и расширение ее системы команд, позволяющее определять поведение программы как распределенной системы из ряда потоков, взаимодействующих в терминах доступа к значениям переменных, расположенных в общей памяти. Описано устройство общей памяти и механизмы доступа к ней отдельных процессов, представляющих собой последовательности выполнения команд, часть из которых являются запросами к общей памяти. В центре внимания удобство отладки небольших программ, используемых для ознакомления с проблемами параллелизма в процессе обучения, когда темп понимания проблем обучаемыми важнее достижения эффективности и производительности учебных программ. Решение проблем отладки программ полезно при изучении методов программирования, а также при исследовании истории языков программирования, сравнения парадигм программирования, потенциала используемых схем и моделей, оценки уровня новизны создаваемых языков программирования, создания методики измерения разных характеристик программ на моделях и выборе критериев практичности создаваемых программ.

При расширении системы команд абстрактной машины учтены принципы функционального программирования как популярной парадигмы на этапе подготовки прототипов и моделей многопоточных программ. Из этих принципов выполнен вывод следствий, позволяющих успешно выбрать элементарные команды, поддерживающие работу с памятью в стиле неизменяемости данных, обратимости действий и транзакций при обработке данных. Для функционального программирования, как и для учебных задач параллельного программирования, умение обеспечить правильность и полноту решений важнее эффективности и производительности полученных программ. Это путь к созданию надежного и безопасного программного обеспечения.

Ключевые слова

абстрактная машина, система команд, методы определения языков программирования, функциональное программирование, неизменяемость данных, восстановление данных, освобождение памяти

Для цитирования

Городняя Л. В. Абстрактная машина языка программирования учебного назначения СИНХРО // Вестник НГУ. Серия: Информационные технологии. 2021. Т. 19, № 4. С. 16–35. DOI 10.25205/1818-7900-2021-19-4-16-35

Abstract Machine of the Programming Language for Educational Purposes SYNCHRO

Lydia V. Gorodnyaya

Novosibirsk State University
Novosibirsk, Russian Federation

A. P. Ershov Institute of Informatics Systems
of the Siberian Branch of the Russian Academy of Sciences
Novosibirsk, Russian Federation

lidvas@gmail.com, https://orcid.org/0000-0002-4639-9032

Abstract

The article is devoted to a number of solutions for organizing work with memory in educational languages and programming systems aimed at teaching the preparation of multi-threaded programs over shared memory. The consideration was carried out within the framework of the SYNCHRO programming language for educational purposes, which made it possible to analyze variants of such solutions without the limitations typical for traditional manufacturing tools and devices of standard programming systems. The article gives a clear formulation of an abstract machine and a diagram of a command system that allows you to define the behavior of a program as a distributed system from a number of threads interacting in terms of access to the values of variables located in shared memory. The device of shared memory and mechanisms of access to it by individual processes, which are a sequence of command execution, some of which are requests to shared memory, are described. The focus is on the convenience of debugging small programs used to introduce concurrency problems in the learning process, where the pace of learners' understanding of the problems is more important than achieving program efficiency and performance. The solution to this problem is useful when studying programming methods, as well as studying the history of programming languages, comparing programming paradigms, the potential of the schemes and models used, assessing the level of novelty of the programming languages being created, creating a technique for measuring different characteristics of programs on models and choosing criteria for the practicality of the created programs. When choosing a system of commands for an abstract machine, the principles of functional programming were taken into account as a popular paradigm at the stage of preparing prototypes and models of multi-threaded programs. From these principles, the conclusion is drawn of the consequences that allow you to successfully select elementary instructions that support working with memory in the style of data immutability and their transactional processing. For educational tasks in programming, the ability to ensure the correctness and completeness of solutions is more important than the efficiency and productivity of the received programs. This is the path to building reliable and secure software.

Keywords

abstract machine, instruction set, methods for defining programming languages, functional programming, data immutability, freeing memory

For citation

Gorodnyaya L. V. Abstract Machine of the Programming Language for Educational Purposes SYNCHRO. *Vestnik NSU. Series: Information Technologies*, 2021, vol. 19, no. 4, p. 16–35. (in Russ.) DOI 10.25205/1818-7900-2021-19-4-16-35

Введение

Со времен появления Венской методики определения языков программирования (ЯП) [1; 2] операционная семантика языка программирования обычно базируется на определении абстрактной машины (АМ), допускающей не слишком трудоемкую реализацию на широком классе конкретных машин. Практика создания ЯП накопила уже заметное количество АМ [3–5]. В случае многопроцессорных конфигураций АМ, естественно, становится конструкцией из абстрактных процессоров – абстрактным многопроцессорным комплексом (АПК). С середины 1970-х гг. популярным стал формализм SECD-машины, предложенный П. Лэндиным при создании аппаратной реализации языка Lisp [6], подробно описанный в книге П. Хэндерсона [4]. В конце 1970-х гг. в рамках проекта языка БАРС В. Е. Котов предложил структурировать определение ЯП на четыре основных подъязыка, независимо представляющих изобразительные средства для выражений, структур данных, управления процессами и дисциплины памяти [5]. Именно подъязык дисциплины памяти вызвал заметные трудно-

сти, возможно, связанные с тем, что в те времена было мало опыта работы с многопроцессорными конфигурациями и распределенными системами [7; 8]. Практика реализации систем программирования и компиляторов сосредоточилась на задачах оптимизации распределения памяти, а вопросы разнообразия дисциплины работы с памятью, с ее неоднородностью и внешними устройствами, слабо учитываемые в ЯП, остались без особого развития [9]. С середины 1990-х гг. в сфере параллельных вычислений обрели популярность системы функционального программирования, принципы которого, такие как гибкость ограничений и неизменяемость данных, показали удобство для подготовки многопоточных программ, устроенных из независимых протоков [10]. Проблемы многопоточных программ с взаимодействующими потоками ждут своего решения.

Проблема понимания многопоточности

На уровне первичного элементарного программирования мало заметна принципиальная разница между хранением данных в локальной, общей оперативной или внешней памяти. Особенности работы с общей и внешней памятью много детальнее изучены в практике организации реляционных баз данных и применения языка запросов к базам данных типа SQL, использующих транзакции [11].

Нормальное завершение многопоточной программы происходит при исчерпании комплекта пассивных потоков, готовых к выполнению, и плановом завершении всех активных потоков. Кроме того, общее завершение работы может происходить преждевременно, при невозможности завершить работу каких-либо потоков из-за взаимоблокировки или исчерпания ресурсов, таких как память. В последнем случае обычно подключается дополнительная память, или привлекаются механизмы ее освобождения. Часть таких проблем алгоритмически не разрешима, поэтому в задачи обучения входит предвидеть опасности и научаться отлаживать программы при их обнаружении. Для простоты учебной модели здесь не рассматривается учет разнообразия категорий систем команд отдельных процессоров и видов используемой памяти с различной дисциплиной функционирования. Механизмы освобождения памяти обычно требуют приостановки основных вычислений, что снижает общую производительность программы. Это особенно заметно при необходимости освобождения общей памяти, требующей приостановки всего многопроцессорного комплекса.

Наполнение многопоточной программы может развиваться независимо от схем управления вычислениями в отдельных потоках, а схемы можно реорганизовывать без дополнительной отладки наполнения. Они играют роль макетов или моделей программ и работают, подобно макросам (открытая подстановка), но с контролем соответствия параметров объявленным синтаксическим типам фрагментов. Кроме того, техника отладки программ может быть обогащена возможностью привлечения протоколов ранее выполненных вычислений и приведения программ к нормальным формам, удобным для сведения к базовым / стандартным моделям параллельных вычислений. В таких случаях более точное описание абстрактного комплекса требует некоторого пересмотра системы команд абстрактной машины для отдельных процессоров – переход к модели абстрактного комплекса.

Абстрактный многопроцессорный комплекс предназначен для определения механизмов выполнения многопоточных программ на многопроцессорных комплексах. АПК обладает конкретной **с**истемой команд, расширенной средствами работы с общей памятью и внешними устройствами. При функционировании комплекса число процессоров может изменяться, что влечет появление динамических запросов к памяти, не заметных при статическом анализе и компиляции программы. Кроме собственно процессоров к выполнению программы можно привлекать дополнительные устройства, которые рассматриваются как специальные процессоры без заранее определенной системы команд, способные выполнять некоторые действия по командам процессора. Системе команд следует поддерживать обработку данных, их размещение и реорганизацию в общей памяти программы или в локальной памяти

процесса, управление ходом выполнения процессов, включая взаимодействие процессоров и устройств, и резервирование данных для их защиты от случайных изменений, подобно средствам операционных систем [7].

Каждый активный процессор выполняет один процесс, порождаемый одной последовательностью команд. Предполагается, что программа – результат компиляции многопоточной программы с предельным распараллеливанием. Программа выполняется в определенном контексте – общей обстановке, данные из которой доступны отдельным процессорам, выполняющим программу. Обстановка, кроме локальной памяти, содержит описания глобальных, возможно изменяемых, общих данных, доступных во всей программе. Возможно, что по умолчанию, имеется один процессор, на котором происходит выполнение основной программы. Такой процессор может не отличаться от остальных по устройству и работать без приоритета. Каждый процессор работает по шагам, соответствующим выполнению одной команды процессора, после каждого шага происходит переход к общему правилу управления многопроцессорной программой. Шаги разных процессоров не синхронизованы, но могут происходить одновременно. Общее правило управления может включать параметр порядка перебора команд процессоров, допускающий и учет приоритетов, и правил тасования, и временной шкалы, и случайного выбора очередного процессора. Если используется временная шкала, то предполагается, что существует дополнительный процессор времени, непрерывно вырабатывающий очередную метку времени, которую могут учитывать остальные процессоры как данное из общей памяти. Для работы общей памяти может существовать отдельный процессор общей памяти.

При сравнении императивного и функционального подходов к программированию П. Лендин (P. J. Landin) предложил специальную абстрактную машину SECD, удобную для спецификации машинно-зависимых аспектов семантики языка Lisp. Подробное описание этой машины можно найти в книге П. Хендерсона по функциональному программированию, где предложен и вариант расширения АМ для поддержки параллельных вычислений. Рассматриваемые здесь процессоры устроены, примерно как абстрактная машина SECD П. Лендина, с добавлением набора команд работы с памятью и внешними устройствами, как у Н. Вирта в Пи-коде [4; 12], команд организации взаимодействия процессоров уровня UNIX [7] и транзакциями, характерными для баз данных [11], восстановления данных, как в системах редактирования [13], с учетом JVM [14] и машины MMIX Д. Кнута [1]. Дальнейшие идеи определения системы команд для многопроцессорных конфигураций сложились при анализе парадигм программирования, включая особенности функционального программирования [9; 15; 16], а также работ по дисциплине программирования [17] и языкам Forth, mpC и СИНХРО [18–20].

Расширение абстрактной машины

Машина SECD работает над четырьмя регистрами: стек для промежуточных результатов, контекст для размещения именованных значений, управляющая вычислениями программа, резервная память (Stack, Environment, Control_list, Dump). Регистры приспособлены для хранения выражений в форме символов или списков. Состояние машины полностью определяется содержимым этих регистров. Поэтому функционирование машины можно описать достаточно точно как переходы изменения содержимого регистров при выполнении команд, что выражается следующим образом:

s e c d → s' e' c' d' – переход от старого состояния к новому.

Вводим расширенно обозначение на случай работы с общей памятью.

M(SECD)+ – многопроцессорный комплекс над общей памятью.

Обозначение **M(SECD)+** символизирует, что **M** – общая память для всех процессоров, **(Процессор)+** – что хотя бы одни процессор обязателен, общее число процессоров произвольно, и не исключено изменение их числа в динамике.

M – общая память, соответствует принципу неизменяемости данных, состоит из регистров произвольного доступа, хранящих счетчик числа потоков, использующих эти регистры, имя переменной, ее текущее значение и протокол произошедших изменений, возможно, устроенный как вектор или список.

Система команд в таком случае представляет собой реализацию базовой семантики языка многопоточных программ, дополненную рядом системных действий по передаче параметров и защите областей действия, подразумеваемых языком, но не имеющих четкого синтаксического представления. Такое определение может быть машинно-независимым и переносимым. Абстрактный многопроцессорный комплекс языка СИНХРО как примера языка многопоточных программ различает следующие категории команд:

1) засылка значений из общей памяти и разных локальных регистров АПК в стек S;

2) вычисления над безымянными операндами из стека S при обработке выражений;

3) пересылка значений из стека S в общую память M или регистры АПК с учетом локализации участков процесса;

4) организация ветвлений, точнее – обходов, что удобно для кодирования программ с параллелизмом и эффективно для архитектур с разрядами управления выполнимостью команд методом прошивания 0/1 в коде команды; (If... Then... без Else);

5) организация циклов, вызовов процедур или функций, определения которых не сложнее одного комплекта потоков, с сохранением контекста в локальной обстановке E и / или в дампе D с возможностью восстановления;

6) организация комплекта процессов с передачей их результатов в общую память M и возможностью обмена сообщениями или ожидания событий при синхронизации.

Суммарно комплект команд включает в себя следующие действия (выделены добавленные к SECD команды):

LD – ввод данного из локального контекста в стек;

LDC – ввод константы из программы в стек;

LDF – ввод определения функции в стек;

LDM – ввод данного из общей памяти в стек;

COMPSUB – компиляция «на лету» фрагмента программы, заданного в виде текста;

AP – применение функции, определение которой уже в стеке;

RTN – возврат из определения функции к вызвавшей ее программе;

RAP – применение **рекурсивной** функции;

DUM – резервирование памяти для хранения **поколений** аргументов рекурсивной функции или цикла;

SEL – **обход** участка или ветвление в зависимости от активного (верхнего) значения стека;

JOIN – переход к общей точке после обхода;

CRCL – цикл – повтор;

BR – принудительное завершение выполнения цикла, комплекта, потока или функции;

PAIR – CONS – формирование **узла** по двум верхним значениям стека для списков;

HEAD – CAR – первый элемент узла из активного значения стека;

TAIL – CDR – остаток узла без первого элемента активного значения стека;

ARR – формирование **вектора** из заданного числа элементов, расположенных в стеке;

ELM – выбор элемента из вектора по заданному индексу;

ATOM – **неделимость** (атомарность) верхнего элемента стека;

NUMBER – проверка на число;

LIST – проверка на список;

ARRAY – проверка на вектор;

TEXT – проверка на текст или строку;

EQ – **равенство** двух верхних значений стека с учетом произвольного значения;
STOP – **останов;**
SET – **запись** в общую память комплекса из стека;
LET – сохранение локальных значений в **контексте;**
MLL – пересылка головного элемента из списка в список головного элемента;
MLV – пересылка из списка головного элемента в указанный элемент вектора;
MVL – пересылка из указанного элемента вектора в головной элемент списка;
MVV – пересылка из указанного элемента вектора в указанный элемент другого вектора;
CHNG – обмен данными в общей памяти комплекса;
DELETE – удаление верхнего элемента стека;
WRT – вывод данных из стека в процесс внешнего периферийного устройства с сигналом успеха-провала;
RD – ввод данных от процесса внешнего периферийного устройства в стек с сигналом успеха-провала;
ANY – размещение в стеке выравнивающего данного, если данное с устройства не введено;
KIT – **комплекс** из процессоров или процессов выполнения действий-директив, мощность комплекса известна;
ROW – **процесс для** выполнения ряда действий-директив на одном процессоре, длина ряда известна в каждый момент;
REZ – размещение заданного числа результатов процесса в свой стек;
WAIT – **ожидание** приостановки указанного процесса (завершения или сообщения);
SEND – передача **сообщения** указанному процессу;
NEXT – ожидание завершения предыдущего действия;
INC – **увеличение** числа на 1;
DEC – **уменьшение** числа на 1;
SUB – **вычитание** из верхнего элемента стека;
ADD – **прибавление** к верхнему элементу стека;
MUL – **произведение** двух чисел, первое в стеке;
DIV – **частное** от деления верхнего элемента стека на второй элемент;
LT – выяснение**,** верхнее значение в стеке **меньше** ли, чем второе;
ERR – реакция на ошибку, аппаратную или программируемую, если в коде программы она встроена.

Для начала абстрактная машина считается идеальной, программу для нее компилятор строит без учета возможных аварийных ситуаций, считает, что все данные на устройстве ввода расположены в соответствии с запросами программы, именно те, что нужны. Состояния стеков к моменту выполнения команд сформированы вполне корректно. Регистры общей памяти подчинены принципу неизменяемости данных. Новые данные размещаются на верху стека или в первом элементе списка, а к прежним значениям, хранимым в общей памяти, можно вернуться при необходимости, например при отладке посмотреть историю изменения данных. При выполнении обратимых действий допускается и обратимость воздействий на общую память. Могут быть и другие категории команд, реализуемые как не консервативное расширение абстрактного комплекса ради эффективности или решения других задач.

Принцип неизменяемости данных

Для языка СИНХРО предлагается усилить абстрактную машину добавлением регистра, используемого при определении механизма работы с общей памятью для семейства процессоров в рамках принципа неизменяемости данных, характерного для ФП. Таким образом можно от машины SECD перейти к абстрактному комплексу вида M[SECD]+, где явно отмечена многопроцессорность и существование общей памяти. Такая работа с памятью должна

быть поддержана в транзакционном стиле, подобно обработке записей в базах данных, т. е. каждое воздействие на память или выполняется полностью, или восстанавливается состояние до начала выполнения не завершившейся команды. Кроме того, каждый элемент памяти в любой момент времени обрабатывается только в одном процессе программы, т. е. подобно захвату-освобождению файла или устройства. Хранение данных в общей памяти сопровождается протоколом изменений, чтобы при отладке можно было видеть, какой процесс внес изменения, и при необходимости вернуть утраченное значение.

Таким образом, абстрактный комплекс определяется над функционирующими как стеки или списки неограниченного размера пятью регистрами: M – общая память, S – стек для окончательных и промежуточных результатов, E – контекст для размещения локальных именованных значений, C – управляющая вычислениями программа, D – резервная память (Memory, Stack, Environment, Control_list, Dump). В определенном смысле состояние общей памяти M можно рассматривать как непременное дополнение к формальному результату работы программы. Ниже приведено более детальное определение работы команд как переходов от одного состояния машины к другому:

m s e c d → m' s' e' c' d' – переход от старого состояния к новому,

где

m – Memory = общая память, выглядит как внешнее устройство, доступное всем процессам;

s – Stack = стек для окончательных и промежуточных результатов;

e – Environment = контекст для размещения локальных значений переменных;

c – Control_list = управляющая вычислениями программа;

d – Dump = резервная память для обеспечения надежности вычислений.

Стек устроен традиционно по схеме «первый пришел, последний ушел». Размер стека не ограничен. Каждая команда абстрактной машины «знает» число используемых при ее работе элементов стека S и их форматы, элементы она удаляет из стека S и вместо них размещает единственный выработанный результат для обычных команд или ряд результатов для многопроцессорных команд, размещая число процессоров в верхнем элементе стека. Всегда известно число текущих результатов в стеке S, которые можно явно свернуть в единственный результат специальной, редуцирующей командой. Такие команды выясняет компилятор и размещает в регистре C для абстрактной машины.

Фактически исполняются команды по очереди, последовательно, начиная с первых в регистре C управляющей программы, хотя формально может считаться, что они могут исполняться и в произвольном порядке. Отдельная машина прекращает работу при выполнении команды «останов» – STOP, которая формально характеризуется отсутствием изменений в состоянии машины:

m s e (**STOP**) d → m s e (STOP) d

Следуя П. Хендерсону, для четкого отделения обрабатываемых элементов АПМ от остальной части списка будем использовать следующие обозначения:

(x . L) – это значит, что первый, головной элемент списка – x, а остальные, хвостовые, находятся в списке L;

(x y . L) – первый элемент списка – x, второй элемент списка – y, остальные находятся в списке L и т. д.

Теперь мы можем методично описать эффекты всех перечисленных выше команд.

m s e (**LDC** q . c)d → m (q . s) e c d	% константы из программы в стек
m (a b . s) e (**PAIR** . c)d → m ((a . b) . s) e c d	% узел структуры из стека
m ((a . b) . s) e (**HEAD** . c) → m (a . s) e c d	% головной элемент списка из стека

m ((a . b) . s) e (**TAIL** . c) → m (b . s) e c d % остальные элементы списка без головного

ARR – формирование вектора из заданного числа элементов, предварительно размещенных в стеке.

m (a b . s) e (**ARR** 2 . c)d → m (@[a, b] . s) e c d % вектор, содержащий 2 элемента

@ – **адрес** данного

ELM – выбор элемента из вектора по заданному индексу.

m (i @X . s) e (**ELM** . c) → m (X[i] . s) e c d % элемент вектора

m (2 @[a, b] . s) e (ELM . c) → m (b . s) e c d % 2-й элемент вектора

Для предиката оговариваем, в каких случаях вырабатывается значение «истина»:

m ((a . b) . s) e (**ATOM** . c) d → m (NIL . s) e c d % ложь

m (a . s) e (ATOM . c) d → m (a . s) e c d % истина

m (Nil . s) e (ATOM . c) d → m (T . s) e c d % истина

Для составных, неатомарных значений – NIL, т. е. ложь, истина «T» – для атомов:

m (a a . s) e (**EQ** . c) d → m (T . s) e c d % два верхних элемента совпали

m (a b . s) e (**EQ** . c) d → m (NIL . s) e c d % два верхних элемента различны

Истина «T» – для совпадающих указателей, для несовпадающих – NIL, т. е. ложь.

Для доступа к значениям, расположенным в **контексте**, можно определить специальную функцию N-th, выделяющую из списка элемент с заданным номером N в предположении, что длина списка превосходит заданный адрес:

m s e (**LD** n . c) d → m (x . s) e c d

где x – это значение (N-th n e), т. е. значение по указанному номеру в контексте.

При реализации обхода управляющая программа соответствует шаблону:

(... SEL (... JOIN))

Необязательный участок размещен в подсписке с завершителем JOIN, после следует общая часть вычислений. Для большей надежности на время выполнения обхода общая часть сохраняется в дампе – **резервной памяти**, а по завершении **восстанавливается** в регистре управляющей программы:

m (NIL . s) e (**SEL** cc . c) d → m s e c d % обход участка

m (T . s) e (**SEL** cc. c) d → m s e cc (c . d) % выполняется участок

m s e (**JOIN**) (c . d) → m s e c d % восстановление при выходе из участка

Получается нечто вроде оператора If ... Then .. без Else, что позволяет делать исполнимый код без ветвлений и тем самым уменьшать потребность в динамическом порождении новых процессов.

Организация вызова функций или процедур требует защиты контекста E от локальных изменений, происходящих при интерпретации тела определения. Для этого при вводе определения функции в стек S создается специальная структура, содержащая код определения функции и **копию текущего состояния контекста E.** До этого в стеке должны быть размещены фактические параметры. Завершается функция командой RTN, восстанавливающей регистры, заранее сохраненные в регистре D, и размещающей в стеке S полученный при выполнении тела функции результат:

m s e (**LDF** f . c) d → m ((f . e) . s) e c d % замыкание функции в стек
m ((f . ef) vf . s) e (**AP** . c) d → m NIL (vf . ef) f (s e c . d) % вызов функции,
% ранее размещенной в стеке, и пополнение контекста значениями связанных переменных
m (x) e (**RTN**) (s e c . d) → m (x . s) e c d % выход их функции

где f – тело определения,
ef – контекст в момент вызова функции,
vf – фактические параметры для вызова функции,
x – результат функции.

Рекурсивные вызовы функций дополнительно требуют резервирования памяти для уникальных ссылок на разные поколения фактических аргументов, что достигается путем размещения фиктивного объекта «**ω**», который структуро-разрушающая, деструктивная функция rplaca, в порядке исключения, замещает на реальные данные:

m s e (**DUM** . c) d → m s (**ω** . e) c d

% **ω** – элемент памяти для размещения очередного поколения фактических параметров,
m ((f . ef) vf . s) e (**RAP** . c) d → m NIL (rplaca vf ef) f (s e c . d)
% размещение очередного поколения параметров рекурсивной функции

Получается нечто вроде многоголового списка. На каждом витке рекурсии порождается узел, становящийся головой хвоста списка локальных параметров E, а прежний список запоминается в дампе D. При выходе из рекурсии восстанавливаются прежние состояния локального контекста E.

Для M(SECD)+, как и для SECD, реализационное замыкание ядра включает в себя структуроразрушающие, деструктивные функции **rplaca** и **rplacd**, размещающие свой результат непосредственно в памяти второго аргумента (табл. 1). Это требует соответствующих ветвей в определениях синтаксиса и интерпретатора, что можно рассматривать как шаг раскрутки СП.

Таблица 1

Расширение системы команд леструктивными операциями

Table 1

Expansion of the command system by destructive operations

	Новая конструкция	Значение	Примечание
Семантика	(RPLACA x (y . z))	(x . z)	**Новое** значение размещается в **старой** памяти, замещая прежнее значение
	(RPLACD x (y . z))	(y . x)	

Система команд АПМ может быть расширена простым дополнением правил для новых команд. Так, можно ее механизм распространить на другие типы данных, например на целые числа (табл. 2).

Контроль типов данных на уровне системы команд отсутствует. Считается, что компилятор создал программу после проверки соответствия типов данных, и при исполнении кода она уже не нужна.

Таблица 2

Расширение абстракной машины на работу с числами

Table 2

Extension of an abstract machine to work with numbers

m (a b . s) e (**ADD** . c) d	→ m ((a + b) . s) e c d
m (a b . s) e (**SUB** . c) d	→ m ((a – b) . s) e c d
m (a b . s) e (**MUL** . c) d	→ m ((a * b) . s) e c d
m (a b . s) e (**DIV** . c) d	→ m ((a / b) . s) e c d
m (a . s) e (**INC** . c) d	→ m ((a + 1) . s) e c d
m (a . s) e (**DEC** . c) d	→ m ((a – 1) . s) e c d
m (a . s) e (**NUMBER** . c) d	→ m (t . s) e c d % число или нет
m (a b . s) e (**LT** . c) d	→ m (t . s) e c d % верхнее **меньше**, чем второе

Примечание: t – **истинностное** значение NIL или T.

Общая память

Для производственного применения ЯП требуется расширение семантики ЯП средствами обмена данными с общей памятью и периферийными устройствами, позволяющими вводить программы и видеть результаты ее выполнения. В SECD и то, и другое происходит чудом. Машина SECD уже достаточна для обеспечения полноты вычислений по Тьюрингу, а для соответствия архитектуре Фон Неймана нужна определенность средств работы с общей памятью и ввода-вывода данных, что дает машина M(SECD)+.

Подобные средства Н. Вирт включил в описание Пи-кода для языков Pascal и Oberon, команды которой можно рассматривать как дополнение и расширение SECD. Разница между SECD и M(SECD)+ в основном сводится к операциям над данными. Кроме того, введены дополнительные команды по непосредственной работе с общей памятью для реализации присваиваний, передаче управления для реализации итераторов и обмена с внешними устройствами. Обе абстрактные машины, как SECD, так и M(SECD)+, содержат кроме образа базовых средств ЯП дополнительные команды, обеспечивающие пересылки данных между регистрами, реализационные средства, поддерживающие эффективность реализации ЯП (rplaca, метки и др.) и профилактику некорректного нарушения непрерывности действий при обмене данными.

Используем дополнительные обозначения:

[x] – содержимое памяти по адресу x;

e[n] – содержимое n-го элемента контекста;

C(Pr) – номер процесса

@c – адрес позиции «c» в программе или векторе;

_ – Произвольное значение (_ подчерк).

Регистр M является списком списков, или стеков, каждый элемент которого начинается с имени глобальной переменной, вслед за которым расположено ее текущее значение, а дальше следует протокол изменения значений, выглядящий как последовательность идентификаторов процесса с последующим установленным этим процессом значением.

Если M содержит элемент X вида M = [. . . (**X xt** At xt A1 x1 A2 x2 . . .)], то после изменения процессом Ak значения переменной X на xk этот элемент приобретет вид

M' = [. . . (**X xk Ak xk** At xt A1 x1 A2 x2 . . .)].

Возникнет побочный эффект присваивания, допускающий при необходимости восстановление прежних значений. Такая реализация позволяет поддержать транзакции и обратимость обработки памяти.

Ввод данного X из общей памяти в стек. В памяти имеется элемент вида (**X xt .** Px), где

X – имя **глобальной** переменной,

xt – текущее значение переменной,

Px – протокол изменения значений переменной.

m=[. . . (**X xt .** Px)] s e (**LDM** X . c) d → m (xt . s) e c d

Сохранение значения из стека в глобальной памяти;

m=[. . . (**X x0 .** Px) . . .] (xt . s) e (**SET** X . c) d → m'==[. . . (**X xt Ct xt .** Px)] s e c d

m=[. . .] (xt . s) e (**SET** X . c) d → m'==[. . . (**X xt Ct xt**)] s e c d % новая переменная,

где Ct – номер текущего процесса, задавшего изменение значения глобальной переменной. Имена данных в общей памяти все различны.

LET – сохранение локальных значений после однократных присваиваний в локальном контексте;

m (ssa . s) e (**LET** x . c) d → m s (e[x]=ssa) c m d

В стеке список результатов однократных присваиваний. К верхнему элементу контекста прицепляется этот список в хвост, вслед за аргументами. Предполагается, что имена формальных параметров и локальных данных различны.

m (ssa . s) (et . e) (**LET** . c) d → m (s) ((**APPEND** et ssa) . e) c d

APPEND – сцепление структур

ssa – участок однократных присваиваний локальным переменным, имена которых уникальны, отличаются от имен формальных параметров. Для глобальных переменных такую работу выполняет SET.

Пересылки и обмен данными нужны для профилактики возникновения временных интервалов между парами взаимосвязанных присваиваний в общей памяти. Без такой профилактики могут возникать так называемая «фантомная» память или доступ к формально уже удаленным данным, что иногда обнаруживается при использовании JVM, стандарт на которую был утвержден более 20-ти лет назад.

MLL – пересылка из списка в список головного элемента

MLV– пересылка из списка головного элемента al в указанный элемент i вектора v

MVL– пересылка из указанного элемента i вектора v в головной элемент списка al

MVV – пересылка из указанного элемента вектора в указанный элемент другого вектора

m ((al . s) e (**MLL** bl . c) d) → m ((PAIR (HEAD bl) (TAIL al)) (PAIR (HEAD al) (TAIL bl)) . s) e c d)

В стеке 2 результата: пара из головы второго списка с хвостом первого и пара из головы первого списка с хвостом второго.

m ((i v . s) e (**MLV** al . c) d) → m (@(v [i] = (HEAD al)) (TAIL al) . s) e c d)

В стеке 2 результата: адрес вектора, в котором i-й элемент заменен на головной элемент списка al, и хвост списка al.

m ((i v . s) e (**MVL** al . c) d) → m ((PAIR v [i] al) @v . s) e c d)

В стеке 2 результата: удлинненный список и адрес вектора. *Здесь вектор не изменился, но возможна реализация, в которой элемент будет объявлен неопределенным.*

m ((i1 v1 . s) e (**MVV** i2 v2 . c) d) → m (@(v1 [i1] = v2 [i2]) @(v2 [i2] = v1 [i1]) . s) e c d)

В стеке 2 результата: адреса векторов, элементы которых изменены.

m ((a . s) e (**DEL** . c) d) → m (s e c d)
CHNG – обмен данными в глобальной памяти

m=[. . . (**X xt** . Px) . . . (Y **yt** . Py) . . .] (Y . s) e (**CHNG** X . c) d)
→ m'=[. . . (**X yt Ct xt** . Px) . . . (Y **xt Ct yt** . Py) . . .] s e c d)
В стеке имя одной переменной, в программе – другой.

Внешние устройства

В результате АПМ способна выполнять вычисления несколько более широкого класса, чем задано базовой семантикой данного ЯП. Это приводит к идее определения реализационного замыкания ЯП в соответствии с фактическими возможностями АМ, включая работу с внешними устройствами. Такое замыкание выполняет роль ядра СП.

m ((**x0 .** s) e (WRT **OUT** . c) d) → m ((+ **x0 .** s) e c d) – OK
m ((**x0 .** s) e (WRT **OUT** . c) d) → m ((- **x0 .** s) e c d) – ESC

OUT' = (x0 . OUT) – на устройстве OUT появляется выведенное данное. В стеке S оно остается, чтобы вывод можно было вставлять в любой позиции выражений программы.

Вывод значения из стека S в процесс над внешним устройством OUT.

Реально при работе с устройствами через конечное время вырабатывается сигнал, говорящий или об успешном выполнении действия, или о причине отказа в его завершении. Здесь для простоты команды определены как абстрактное понимание идеального выполнения действий, без аварийных ситуаций. Выработка сигнала об отказе внешнего устройства приводит к запоминанию в протоколе отказов информации об ущербно выработанных результатах, что можно учесть при отладке, и выполнению остаточной программы в стиле смешанных вычислений.

m (s e (RD **IN** . c) d) → m ((+ **xt (IN) .** s) e c d) – OK
m (s e (RD **IN** . c) d) → m ((- ESC **(IN) .** s) e c d) – ESC

Ввод данного из процесса над устройством IN в стек S.

В зависимости от специфики устройства введенное данное может исчезнуть из него или сохраниться.

(xt . IN) → IN
(xt . IN) → (xt . IN)

Считается, что на внешнем устройстве расположено именно то данное, которое требуется по программе из стека управления «C».

m ((ESC . s) e (**ANY** . c) d) → m ((**xany .** s) e c d)
xany – выравнивающее данное, если был **ESC** – безуспешный ввод.

Параллелизм

Для поддержки параллельных вычислений требуется еще несколько команд по организации комплексов (неупорядоченных комплексов процессов **KIT**) и процессов (последовательности действий – **ROW**), передаче их результатов, локализации воздействий на транзакционную общую память и явную обработку ошибок.

AMn – обозначение отдельного процессора АМ с номером n. Каждый АМ знает свой номер.

AM0 = m (2 . s) e (**KIT** c1 c2 . c) d → AM0 = m s e c d
| AM1 = @m s e c1 d | AM2 = @m s e c2 d

Число процессов в комплексе заранее известно, по их числу порождаются независимые автоматы для асинхронного выполнения процессов: AM1, AM2

Общая память для порожденных АМ доступна через ссылку=**адрес @m .**

AM0 = m (2 . s) e (**ROW** c1 c2 . c) d → AM0 = m (1 . s) e (**ROW** c2 . c) d
| AM1 = @m s e c1 . c) d

Число элементов процесса заранее известно, по их числу последовательно порождаются независимые автоматы для последовательного запуска элементов процессов: AM1, AM2

AM0 = m (**1** . s) e (**ROW** c2 . c) d → AM0 = m s e c d
| AM2 = @m s e (c2 . c) d

Остался 1 элемент – выход из рекурсии порождения процессов.

NEXT – поддержка строго императивного запуска элементов процесса. Без этого второй элемент процесса работает, не дожидаясь завершения предшественника.
NEXT = WAIT (CNP-1) – ожидание завершения заданного процесса.

REZ – РЕЗУЛЬТАТ – запись в стек числа результатов комплекса или процесса

AM0 = m s e c d % c2 уже запущен, только **ждет** c1
| AM1 = @m (a b s) e (**REZ** 2 ass . c) d % получены результаты c1
| AM2 = @m s e (**WAIT** (AMP-1) c2 . c) d % c2 дождался завершения c1
→ AM0 = m (2 b a . s) e (ass . c) d
%AM0 забирает результаты c1 и может выполнить их свертку
| AM2 = @m s e c2 d % c2 становитя исполнимым фрагментом,

где
AMn – номер текущего процесса,
ass – представление функции, выполняющей свертку указанного числа элементов в стеке в одно данное. Например: ***список, структура, сумма, произведение, макс, мин, последний*** и т. п.

Забрали в основной стек результаты выполненного процесса, и можно запустить следующий:

AM0 = m s e (**REZ** 2 . c) d | AM1 = @m (a b s) e (**REZ** 2 . c) d
→ AM0 = m (2 b a . s) e c d

% забрали в основной стек результаты выполненного процесса AM1, и его можно отключить.
% передача результата синхронизована с готовностью их приема.

SEND – передача **сообщения** указанному процессу

AMk = @m (message . s) e (**SEND** AMn ck . c) d % подготовлено сообщение для AMn
| AMn = @m s e (**WAIT** AMk cn . c) d % AMn дожидается сообщения от AMk
→ AMk = m s e (ck . c) d %AMk отправил сообщение для AMn
| AMn = @m (message . s) e cn d % c2 сообщение получено

Реализация подобна рандеву в языке ADA – обмен происходит между активными процессами, и каждый знает, что и кому он передает или от кого сообщение получает.

WAIT – ожидание события, сообщения или завершения заданного процесса.

Реализация таких команд существенно зависит от распределения памяти и независимости ее частей, что приводит к необходимости контроля границ памяти при индексировании векторов и доступе по указателю. Информация для такого контроля во многих ЯП представляется в форме предписания типа данных (ТД) переменным. Таким образом, реализационное замыкание ЯП над M[SECD]+ включает в себя представление ТД для всех переменных, в том числе параметры процедур и их расширение для удобства лаконичных представлений, синтаксического конструирования.

Считаем, что при компиляции создана программа для АМК, пригодная для безаварийного выполнения при условии бесперебойного функционирования всех устройств. Все внешние данные соответствуют требованиям программы, и команды устройств срабатывают неделимо.

Дальнейшее расширение ЯП может быть сведено к подключению ТД и присоединению вычислительных средств методом раскрутки или пошаговой разработки. Цели раскрутки:

- снижение трудоемкости начального этапа программирования;
- увеличение потенциала СП, т. е. числа типовых задач, решение которых обладает приемлемой для практики трудоемкостью.

Такая идеальная конфигурация работает как бы на бесконечной общей памяти, не решает проблемы ее исчерпания и необходимости освобождения от ставших ненужными данных. Проблема освобождения памяти на отдельных локальных потоках имеет стандартные, не слишком обременительные решения. Для общей памяти ее решение резко снижает производительность из-за приостановки всех процессов. Не исключено, что освобождение общей памяти потребует решений, подобных опробованным в практике операционных систем при решении вопросов управления распределенными системами с совмещением работы независимых программ. Каждая отдельная программа для совмещения их работы сопровождалась паспортом, в котором представлялась информация о требуемом объеме памяти, ожидаемом времени счета и внешних устройствах и других атрибутах.

Паспорт переменной

Рассмотрим для примера определение абстрактного комплекса, допускающее управление доступом к общей памяти с использованием подобных паспортов, хранящих вектор необходимых регистров общей памяти. Каждый локальный процесс при «сборке мусора» формирует новое состояние такого вектора, доступного процессору внешней памяти. Конфигурация комплекса расширяется выделением отдельного процессора для работы с общей памятью. Остальные процессоры вместо полного доступа к общей памяти обладают лишь вектором V для доступа к определенным регистрам.

v s e c d → v' s' e' c' d' – переход от старого состояния к новому,

где

v – вектор доступа к отдельным регистрам общей памяти, доступный процессам;
s – Stack = стек для окончательных и промежуточных результатов;
e – Environment = контекст для размещения локальных значений переменных;
c – Control_list = управляющая вычислениями программа;
d – Dump = резервная память для обеспечения надежности вычислений.

Если команды процессора общей памяти можно особо не пересматривать, локальные процессоры несколько изменяют формат взаимодействия с общей памятью. Она уже не принадлежит им равноправно, а доступна через вектор указателей на регистры общей памяти. Это можно видеть на примере команды MLL – пересылка из списка в список головного элемента, доступные через указатели @al и @b.

V ((@al . s) e (**MLL** @bl . c) d) → V’ ((PAIR (HEAD @bl) (TAIL @al)) (PAIR (HEAD @al) (TAIL@ bl)) . s) e c d)

В стеке разместится 2 результата: пара из головы второго списка с хвостом первого и пара из головы первого списка с хвостом второго.

При освобождении общей памяти появляется возможность частичного освобождения памяти, не задействованной в объединении векторов от локальных процессоров, без приостановки всех процессоров. В частности, при отказе отдельного процессора процессор общей памяти может по шкале глобальных переменных выполнить уменьшение счетчиков доступа к этим переменным, влияющих на решения по освобождению или восстановлению памяти.

Восстановление состояний общей памяти

Несколько проще выглядит модель комплекса со специальным процессором общей памяти, работающим как пассивный процесс. Такой АПК состоит из ряда обычных процессоров и одного процессора общей памяти, с которым могут взаимодействовать обычные процессоры. Между собой процессоры взаимодействуют только через общую память. Каждый процессор выполняет одну последовательность команд, среди которых встречаются команды запросов к процессору общей памяти. Это **активные команды**, выполняемые по ходу процесса без особенностей. Команды запроса к общей памяти **имеют двойников** среди команд процессора общей памяти. Это **пассивные команды**, типа ленивых вычислений, возбуждаемые при выполнении запросов из активных процессов. Пассивные команды процессора общей памяти структурированы в ряд очередей, каждая из которых соответствует обычному процессору и содержит двойников запросов в том же порядке, что и в активном процессе. Пары «запрос и его двойник» выполняются неразрывно в стиле рандеву языка Ada. Механизм рандеву обеспечивает исключение случайного вмешательства сторонних процессов. При этом происходит обмен данными между локальной памятью активного процесса и общей памятью (табл. 3). В случае одновременного поступления запросов от разных процессоров они упорядочиваются случайным образом и выполняются по одному. При отладке фактический порядок можно установить по протоколам изменения значений переменных, сопровождаемых номерами процессоров. Кроме того, процессор общей памяти может обладать небольшим набором своих специальных команд, выполняемых по запросам из активных процессов или отдельного процесса управления отладкой многопроцессорной программой.

Команды обычных процессоров, на которых выполняются активные процессы:

GIVE – запрос регистра для переменной с номером H;
SAFE – запрос на хранение данного в переменной по регистру H;
READ – запрос на чтение данного из переменной по регистру H;
UNDO – **з**апрос на получение предыдущего данного из переменной по адресу H;
FREE – объявление, что не нужна больше переменная по регистру H.

Команды-двойники процессора общей памяти, пассивно ждущего запросов от активных процессов или отладчика:

TAKE – выбор регистра для переменной с указанным номером H, счетчик кратности доступа к нему его надо увеличить на 1;

WRITE – размещение данного в регистре H. Если данное – указатель на вектор или список, то они копируются полностью в общую память. В протокол вносится копия указателя на данного и номер активного процесса;

VAR – чтение данного из регистра H для передачи обычному процессору – состояние регистра H не меняется;

UNDO – чтение предыдущего данного из регистра H – состояние регистра H не меняется;

FREE – освобождение памяти, доступной через регистр H, – счетчик кратности доступа уменьшается на 1.

Таблица 3

Дуплеты – парные команды – срабатывают только неразрывно вместе

Table 3

Duplets – paired commands – work only inextricably together

Запросы из активных процессоров	Двойники в процессоре общей памяти	Неразделимая пара
GIVE	**TAKE**	**Дай-Бери**
SAFE	**WRITE**	**Храни-Пишу**
READ	**VAR**	**Читаю-Доступ**
UNDO	**UNDO**	**UNDO-UNDO**
FREE	**FREE**	**Свободен-Свободен**

Сохранение значения из стека в глобальной памяти:

T – **номер текущего процесса**, задавшего запрос на использование или изменение значения глобальной переменной, совпадающий с номером очереди ленивых партнеров-двойников для него;

H – имя переменной для данного в общей памяти. Все имена различны, они известны на всех процессорах, определены при компиляции;

@H – адрес текущего значения переменной H в регистре, хранящем и протокол изменения значений – историю изменения значений;

Ц – **кратность доступа** к переменной в текущий момент;

[@H …] – шкала свободных регистров, содержащая адрес @H.

Дай-Бери

% на стеке локального процесса номер переменной замещается на адрес ее значения
% имя переменной заносится в паспорт процесса
V= [...] (H . s) e (**GIVE** . c) d → V= [H …] (@H . s) e c d
% @H адрес переменной в общей памяти исчезает из шкалы свободных регистров
m=[[@H …] …] s e (**TAKE** . c) d → m'=[[…] … (1 H)] s e c d % новая переменная
m=[[@H …] … (Ц H . Px) …] s e (**TAKE** . c) d → m'=[[…] … ((Ц+1) H . Px)] s e c d
% переменная уже была

Храни-Пишу

% переменная уже выделена, xt – данное для хранения в переменной H
% T – номер потока, известный в очереди с номером, совпадающим с его номером
% значение переменной остается на стеке после записи его в регистр переменной
% присваивание возможно лишь переменным из паспорта процесса
V= [H …] (xt . s) e (**SAFE** H . c) d → V= [H …] (xt . s) e c d
m=[… (Ц H . Px) …] (H . s) e (**WRITE** . c) d → m'= [… (Ц H **xt T xt** . Px)] s e c d
m=[… (Ц H)] (H . s) e (**WRITE** . c) d → m'= [… (Ц H **xt T xt**)] s e c d
% чистая переменная, в ней еще не было значений

Читаю-Доступ

% имя переменной на стеке замещается значением переменной из регистра в общей памяти

% чтение возможно лишь из переменных из паспорта процесса

V= [H …] (H . s) e (**READ** . c) d → V= [H …] (xт . s) e c d

m=[… (Ц H xт T xт . Px) …] (H . s) e (**VAR** . c) d → m s e c d % память не изменилась

UNDO-UNDO

% на стеке появляется предыдущее значение из регистра в общей памяти

% восстановление возможно лишь переменным из паспорта процесса

V= [H …] (H . s) e (**UNDO** X . c) d → V= [H …] (xтт . S) e c d

m=[… (Ц H xт T xт Тт xтт . Px) …] (H . s) e (**UNDO** X . c) d → m s e c d % память не изменилась

Свободен-Свободен

% имя переменной исчезает из стека, а в общей памяти уменьшается счетчик доступа

% удалить можно лишь переменную из паспорта процесса, она оттуда исчезает

V= [H …] (H . s) e (**FREE** . c) d → V= [...] s e c d

m=[… (Ц H xт T xт . Px) …] (H . s) e (**FREE** . c) d → m'==[… ((Ц-1) H xt T xt . Px)] s e c d

% если счетчик должен стать нулем, то сразу адрес регистра объявляется свободным

m=[… (1 H xт T xт . Px) …] (H . s) e (**FREE** . c) d → m'==[[@H …] …] s e c d

% важно, что это происходит неразрывно, без опасности вмешательства других команд

% адрес удаленной переменной размещается в шкале свободных регистров общей памяти

Специальные команды могут выполняться время от времени, можно их запускать после каждой команды «**FREE**». Если нашли счетчик ноль, то возбуждается команда «**Отключить**», и соответствующий адрес переходит в свободные адреса. Такой запрос может произойти при освобождении памяти при сборке мусора в активном процессе. Явного запроса в активном процессе нет, а при анализе выясняется, что некоторые переменные больше не нужны.

m=[… (0 H . Px) …] s e c d → m'==[… (0 H . Px)] s e (**Отключить** . c) d

m=[[…] … (0 H . Px) …] s e (**Отключить** . c) d → m'==[[@H …] …] s e c d

Таким образом, процессор общей памяти может функционировать как элемент распределенной системы, включающийся по мере поступления запросов из активных процессов и не требующий приостановки всех процессов на время решения проблемы освобождения памяти. Техника такой работы характерна для работы с памятью во многих системах программирования. Здесь она дополнена хранением протоколов изменения значений.

При рассмотрении традиционного варианта с восстановлением общей памяти по методике «Stop&Copy» оказалось, что он более сложен и не решает проблему приостановки активных процессоров на время освобождения общей памяти. Представленный здесь метод состоит из ряда процессоров со своей локальной памятью и процессора общей памяти. Локальные процессоры работаю автономно, по мере необходимости обращаясь к общей памяти и время от времени освобождая локальную память. В каждом процессоре хранится шкала регистров используемой общей памяти, которая может то расширяться, то сужаться по мере инструкций из программы или сужаться в результате сборки мусора. Общая память состоит из двух частей – регистров прямого доступа и кучи для хранения протоколов, а также структур данных и старых значений после присваивания на случай необходимости восстановления состояний памяти. При необходимости можно восстанавливать отдельные старые значения. Время от времени при исчерпании той или иной области общей памяти или просто по графику запускается алгоритм, похожий на «Stop&Copy» в системах функционального программирования.

Заключение

Представленное построение показывает, что на этапе ознакомления с проблемами параллелизма можно строить и отлаживать небольшие многопоточные программы взаимодействия процессов, выполняемых на модели произвольной многопроцессорной конфигурации над общей памятью. Опыт такой работы поможет обучаемым видеть и понимать разные явления, происходящие при взаимодействии потоков, предвидеть проблемы и накапливать рецепты решения проблем при подготовке и отладке программ. Для этих целей предлагается принцип неизменяемости данных, характерный для функционального программирования, распространить на работу с переменными в общей памяти и преобразовать его в принцип восстановимости данных, используя протоколы изменения значений переменных. Важную роль играют профилактика «фантомной» памяти и защита данных от некорректного стороннего доступа. Проблема освобождения общей памяти может быть решена как композиция методов «сборки мусора» и учета кратности доступа к переменным из потоков, взаимодействующих через использование данных, хранящихся в общей памяти. До полноты картины нужны еще команды управления процессами, но это уже не входит в тему данной статьи.

Список литературы

1. **Кнут Д. Э.** Искусство программирования. М.: Вильямс, 2007. Т. 1, вып. 1: MMIX – RISC-компьютеры нового тысячелетия. 160 с. ISBN 978-5-8459-1163-6
2. **Вирт Н.** Построение компиляторов. М.: ДМК Пресс, 2010. ISBN 978-5-94074-585-3, 0-201-40353-6
3. **Айлиф Дж.** Принципы построения базовой машины. М.: Мир, 1973. 119 с.
4. **Хендерсон П.** Функциональное программирование. М.: Мир, 1983. 349 с.
5. **Котов В. Е.** МАРС: архитектуры и языки для реализации параллелизма // Системная информатика. Новосибирск: Наука, 1991. Вып. 1: Проблемы современного программирования. С. 174–194.
6. **Иртегов Д. В.** Введение в операционные системы. СПб.: БХВ-Петербург, 2008. 1040 с.: ил. ISBN 978-5-94157-695-1
7. **Пеппер П., Экснер Ю., Зюдхольд М.** Функциональный поход к разработке программ с развитым параллелизмом // Системная информатика. Новосибирск: Наука, 1995. Вып. 4: Методы теоретического и системного программирования. С. 334–360.
8. **Грабер М.** Введение в SQL. М.: Лори, 1996. 377 с.
9. **Лавров С. С., Городняя Л. В.** Функциональное программирование. Интерпретатор языка Лисп // Компьютерные инструменты в образовании. 2002. № 5. С. 49–58.
10. **Эванс Б., Гоф Дж., Ньюленд К.** Java: оптимизация программ. Практические методы повышения производительности приложений в JVM. М.: Диалектика, 2019. 448 с. ISBN 978-5-907114-84-5
11. **Gorodnyaya L.** Method of paradigmatic analysis of programming languages and systems. In: CEUR Workshop Proceedings, 2020, no. 2543, pp. 149–158.
12. **Gorodnyaya L.** On the presentation of the results of the analysis of programming languages and systems. In: CEUR Workshop Proceedings, 2018, no. 2260, pp. 152–166.
13. **Cann D. C.** SISAL 1.2: A Brief Introduction and tutorial. Preprint UCRL-MA-110620. Lawrence Livermore National Lab. Livermore, California, 1992, May, 128 p.
14. **Backus J.** Can programming be liberated from the von Neumann style? A functional stile and its algebra of programs. In: Commun. ACM 21, 1978, no. 8, pp. 613–641.
15. **Филд А., Харрисон П.** Функциональное программирование / Пер. под ред. В. А. Горбатова. М. Мир, 1993. 638 с.
16. **Городняя Л. В.** Основы функционального программирования. М.: Интернет-университет информационных технологий, 2004. 272 с.

17. **Дейкстра Э.** Дисциплина программирования. М.: Мир, 1978. 275 с.
18. **Городняя Л. В.** Язык параллельного программирования СИНХРО, предназначеный для обучения. Новосибирск, 2016. Препринт ИСИ СО РАН № 180. 30 с.
19. **Баранов С. Н., Колодин М. Ю.** Феномен Форта // Системная информатика. Новосибирск: Наука, 1995. Вып. 4: Методы теоретического и системного программирования. С. 193–271.
20. **Ластовецкий А. Л.** Предварительное сообщение о языке программирования mpC // Вопросы кибернетики: Приложения системного программирования / Научный совет по комплексной проблеме «Кибернетика» РАН. М., 1995. С. 20–39.

References

1. **Knut D. E.** Iskusstvo programmirovaniya [The Art of Computer Programming]. Moscow, Vilyams Publ., 2007, vol. 1, iss 1: MMIX – A RISC Computer for the New Millennium, 160 p. (in Russ.) ISBN 978-5-8459-1163-6
2. **Virt N.** Postroyeniye kompilyatorov. Moscow, DMK Press, 2010. (in Russ.) ISBN 978-5-94074-585-3, 0-201-40353-6
3. **Aylif J.** Printsipy postroyeniya bazovoy mashiny. Moscow, Mir, 1973, 119 p. (in Russ.)
4. **Henderson P.** Funktsional'noye programmirovaniye. Moscow, Mir, 1983, 349 p. (in Russ.)
5. **Kotov V. E.** MARS: arkhitektury i yazyki dlya realizatsii parallelizma. In: Sistemnaya informatika. Novosibirsk, Nauka, 1991, iss. 1, pp. 174–194. (in Russ.)
6. **Irtegov D. V.** Vvedeniye v operatsionnyye sistemy. St. Petersburg, BKHV-Peterburg, 2008, 1040 p.: il. (in Russ.) ISBN 978-5-94157-695-1
7. **Pepper P., Eksner Yu., Zyudkhold M.** Funktsional'nyy pokhod k razrabotke programm s razvitym parallelizmom. On: Sistemnaya informatika. Novosibirsk, Nauka, 1995, iss. 4, pp. 334–360. (in Russ.)
8. **Graber M.** Vvedeniye v SQL. Moscow, Lori, 1996, 377 p. (in Russ.)
9. **Lavrov S. S., Gorodnyaya L. V.** Funktsional'noye programmirovaniye. Interpretator yazyka Lisp. *Komp'yuternyye instrumenty v obrazovanii*, 2002, no. 5, pp. 49–58. (in Russ.)
10. **Evans B., Gof Dzh, Nyulend K.** Java: optimizatsiya programm. Prakticheskiye metody povysheniya proizvoditel'nosti prilozheniy v JVM. Moscow, Dialektika Publ., 2019, 448 p. (in Russ.) ISBN 978-5-907114-84-5
11. **Gorodnyaya L.** Method of paradigmatic analysis of programming languages and systems. In: CEUR Workshop Proceedings, 2020, no. 2543, pp. 149–158.
12. On the presentation of the results of the analysis of programming languages and systems. In: CEUR Workshop Proceedings, 2018, no. 2260, pp. 152–166.
13. **Cann D. C.** SISAL 1.2: A Brief Introduction and tutorial. Preprint UCRL-MA-110620. Lawrence Livermore National Lab. Livermore, California, 1992, May, 128 p.
14. **Backus J.** Can programming be liberated from the von Neumann style? A functional stile and its algebra of programs. In: Commun. ACM 21, 1978, no. 8, pp. 613–641.
15. **Fild A., Harrison P.** Funktsional'noye programmirovaniye. Trans., ed. by V. A. Gorbatov. Moscow, Mir, 1993, 638 p. (in Russ.)
16. **Gorodnyaya L. V.** Osnovy funktsional'nogo programmirovaniya. Moscow, Internet-Universitet Informatsionnykh tekhnologiy, 2004, 272 p. (in Russ.)
17. **Deykstra E.** Distsiplina programmirovaniya. Moscow, Mir, 1978, 275 p. (in Russ.)
18. **Gorodnyaya L. V.** Yazyk parallel'nogo programmirovaniya SINKHRO, prednaznachenyy dlya obucheniya. Novosibirsk, 2016, preprint ISI SO RAN № 180, 30 p. (in Russ.)
19. **Baranov S. N., Kolodin M. Yu.** Fenomen Forta. In: Sistemnaya informatika. Novosibirsk, Nauka, 1995, iss. 4, pp. 193–271. (in Russ.)

20. **Lastovetsky A. L.** Predvaritel'noye soobshcheniye o yazyke programmirovaniya mpC. In: Voprosy kibernetiki: Prilozheniya sistemnogo programmirovaniya. Moscow, 1995, pp. 20–39. (in Russ.)

Информация об авторе

Лидия Васильевна Городняя, кандидат физико-математических наук, доцент

Information about the Author

Lydia V. Gorodnyaya, Candidate of Sciences (Physics and Mathematics), Associate Professor

Статья поступила в редакцию 10.10.2021;
одобрена после рецензирования 01.12.2021; *принята к публикации 01.12.2021*
The article was submitted 10.10.2021;
approved after reviewing 01.12.2021; *accepted for publication 01.12.2021*