ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «МЭИ»

На правах рукописи

Шамаль Павел Николаевич

РАЗРАБОТКА И ИССЛЕДОВАНИЕ МЕТОДОВ И ПРОГРАММНЫХ СРЕДСТВ ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ ФУНКЦИОНАЛЬНЫХ ПРОГРАММ НА МНОГОЯДЕРНЫХ КОМПЬЮТЕРАХ

Специальность: 05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей.

Диссертация на соискание ученой степени кандидата технических наук.

Научный руководитель:
доктор технических наук,
профессор Кутепов В.П.

Москва 2014

Оглавление

ВВЕДЕНИЕ

Увеличение быстродействия компьютера путем повышения тактовой частоты его единственного процессора приблизилось к технологическому пределу [1]. Поэтому ведущие производители процессоров начали наращивать производительность компьютерной системы путем увеличения числа процессоров или ядер – независимых вычислительных блоков внутри процессора. На сегодняшний день данный подход прочно закрепился при производстве процессоров как для персональных компьютеров и серверов, так и для процессоров портативных и мобильных устройств.

Для того чтобы максимально использовать ресурсы многоядерного компьютера при решении на нем различных задач, требуется использование подходов, предусматривающих параллельное выполнение [48]. Широко используемые в промышленном программировании современные объектно-ориентированные языки программирования, такие как Java и C++, основаны на императивной парадигме и не имеют встроенных в язык программирования средств для распараллеливания программ, «прозрачных» для программиста, а лишь предоставляют набор низкоуровневых средств задания параллелизма. Практическое использование этих средств часто приводит к тому, что программа получается плохо масштабируемой и привязанной к конкретной операционной системе, языку программирования или архитектуре вычислительной системы.

Одним из средств решения этой проблемы является применение функциональных языков программирования [2; 49], которые позволяют абстрагироваться от особенностей используемой вычислительной системы и сложной задачи управления параллельными процессами, сконцентрироваться только на решении поставленной задачи и разрабатывать эффективные параллельные программы.

Язык FPTL (*Functional Parallel Typified Language*) [3], реализации которого на многоядерных компьютерах посвящена диссертация, создавался с целью эффективной разработки функциональных программ и последующего их параллельного выполнения на многоядерных компьютерных системах с общей памятью и кластерах. В отличие от известных языков функционального программирования, таких как LISP [4], ML [5], Haskell [6], которые в большой степени основаны на λ-исчислении [7], FPTL основан на использовании традиционной математической формы определения функций путем их композиции с помощью конечного множества операций и общей формы их задания в виде систем функциональных уравнений. Операции композиции функций просты и позволяют описывать параллелизм на семантическом уровне, не используя специальные процессные примитивы, как это делается в других функциональных языках. Однако реализация такого «чисто» функционального языка программирования требует разработки методов и алгоритмов, которые могут обеспечить эффективное параллельное выполнение программ на многоядерных компьютерах. Это создает необходимые предпосылки создания соответствующей системы выполнения языка FPTL на современных многоядерных компьютерных системах.

Цель диссертационной работы. Целью диссертационной работы является разработка и исследование эффективности системы выполнения функциональных параллельных программ на языке FPTL на многоядерных компьютерах. Для достижения этой цели в диссертации решаются следующие задачи:

1. Сравнительный анализ методов распараллеливания вычислений в современных функциональных языках программирования.

2. Разработка и исследование методов и алгоритмов эффективного параллельного выполнения функциональных FPTL-программ на многоядерных компьютерах.
3. Разработка методов контроля типовой корректности FPTL-программ.
4. Создание интегрированной системы параллельного выполнения FPTL-программ на многоядерных компьютерах, которая включает следующие подсистемы:
   - интерпретатор FPTL-программ,
   - управление параллельными процессами выполнения функциональных программ,
   - статический анализ типовой корректности программ.
5. Экспериментальное исследование эффективности созданной системы.

Методы исследования. В диссертационной работе использовались методы и понятия теории графов, теории алгоритмов, теории рекурсивных функций, теории языков и формальных грамматик и операционных систем. Для практической реализации системы использовались методы объектно-ориентированного и параллельного программирования.

Достоверность полученных результатов. Достоверность полученных результатов и выводов подтверждается проведенной серией экспериментов с использованием современных многоядерных компьютеров, а также проведенным сравнением с существующими аналогичными программными системами.

Научная новизна. Научная новизна работы заключается в следующем:

1. Проведено усовершенствование языка функционального параллельного программирования FPTL. Введены новые синтаксические конструкции, упрощающие написание программ.

Расширено множество типов данных, в частности, введены средства для работы с массивами данных.

2. Предложена модель типизации и алгоритм вывода типов в языке FPTL.
3. Разработано внутреннее представление функциональных программ и метод организации работы с данными в процессе вычисления значений функций, увеличивающие эффективность реализации параллелизма.
4. Реализована система выполнения функциональных программ – интерпретатор языка FPTL.
5. Разработан алгоритм эффективного управления параллельным выполнением функциональных программ на многоядерных компьютерах.
6. Проведено исследование эффективности реализованной системы и сравнение с имеющимися аналогами.

Практическая ценность. Созданная система функционального программирования в настоящее время применяется в обучении методам и средствам параллельного программирования студентов и аспирантов кафедры Прикладной Математики московского энергетического института, подготовлена необходимая документация по использованию системы, направлены документы для регистрации программной системы.

Апробация. Положения диссертационной работы докладывались на конференциях: «Параллельные вычисления и задачи управления», Россия, Москва, 2012 г.; XIX ежегодная международная научно-техническая конференция студентов и аспирантов "РАДИОЭЛЕКТРОНИКА, ЭЛЕКТРОТЕХНИКА И ЭНЕРГЕТИКА", Россия, Москва, 2013 г.

Публикации. По теме диссертационной работы также опубликованы три научные работы в изданиях, включенных в перечень ВАК:

1. Реализация языка функционального параллельного программирования FPTL на многоядерных компьютерах. Известия РАН. Теория и Системы Управления, 2014, № 3, с. 46–60.
2. Система типового контроля программ на языке функционального программирования FPTL. Программные продукты и системы, 2014, № 2, с. 11–17
3. Implementation of Functional Parallel Typified Language (FPTL) on Multicore Computers. Journal Of Computer and Systems Sciences International, 2014, vol. 53, № 3, pp. 345–358.

Объем и структура работы. Диссертация состоит из введения, пяти глав и заключения. Список использованной литературы содержит 52 наименования. Текст диссертации содержит 100 страниц машинописного текста, включая 20 рисунков и 5 таблиц.

В **первой главе** проводится обзор методов и средств распараллеливания, предоставляемых функциональными языками программирования.

Во **второй главе** дается описание синтаксиса и семантики функционального языка параллельного программирования FPTL. Также вводится модель параллельного вычисления значений функций языка FPTL и неформальное описание синтаксиса языка FPTL и структуры программы.

В **третьей главе** описывается модель типизации программ на языке FPTL и приводится алгоритм проверки их типовой корректности.

В **четвертой главе** дается описание сетевого представления функциональных программ, алгоритма работы интерпретатора для сетевого представления, алгоритма управления параллельным выполнением функциональных программ, алгоритмов оптимизации выполнения функциональных программ и способов представления данных. Также в главе описаны важные аспекты реализации системы выполнения,

такие как способы представления данных, система управления памятью, организация вызова внешних функций и описание программных средств использованных для реализации системы.

В **пятой главе** представлены результаты экспериментов, направленных на исследование эффективности принятых проектных решений. Приведены описания тестовых программ, графики времени их выполнения и относительного ускорения. Также проведено сравнение полученных результатов с результатами аналогичных экспериментов, проведенных для языка Haskell.

1. ФУНКЦИОНАЛЬНЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ. ЗАДАНИЕ И РЕАЛИЗАЦИЯ ПАРАЛЛЕЛИЗМА

Существующая классификация языков программирования, относящая их к императивным, функциональным, логическим, объектно-ориентированным, процессным, является достаточно произвольной. Она отражает либо их проблемную ориентацию, к примеру, функциональную, логическую или процессную, либо определенный способ построения или поведения программы. Для императивных программ центральным аспектом является такое упорядочение фрагментов программы (операторов, блоков, процедур и др.), чтобы их последовательное выполнение гарантировало получение запланированного результата. Объектно-ориентированное программирование, оставаясь последовательным, как и императивное, претендует на статус большей декларативности в описании программы и выражается в возможности оперирования классами, как обобщенными декларациями либо определенных объектов, либо форм, по которым можно порождать объекты с различными свойствами.

С другой стороны, параллельное и процессное программирование делает основной акцент на описании такого упорядочивания фрагментов программы, при котором определенная их часть может выполняться одновременно. Двойственность программы, как декларативного описания того, что она должна делать, и описания того, как она должна выполняться, зафиксирована в известных двух ее семантиках, а точнее моделях программы: денотационной и операционной. Параллельное программирование актуализировало проблему создания моделей, которые могли бы быть достаточно универсальными, чтобы можно было либо явно отражать в программе, либо реализовывать при ее выполнении возникающие возможности распараллеливания. Сети Петри, модели Милнера [8] и Хоара [9] – известные процессные модели для этого.

В большинстве современных языков программирования, как императивных, так и функциональных, при написании программы программист сопровождает ее на некотором универсальном процессном языке, описывающем ее параллельное выполнение. Для этого могут использоваться специальные операторы языка программирования, аннотации, синтаксические конструкции и процедуры. Естественна и другая постановка задачи: как автоматически определять и реализовывать параллелизм в программе и при этом свести к минимуму участие программиста в этой работе. Для этого, во-первых, денотационная модель и язык программирования должны быть устроены так, чтобы параллелизм в программе мог выражаться средствами этой модели и языка, не прибегая к процессным примитивам для его указания. Во-вторых, должна существовать достаточно универсальная процессная модель, чтобы на ее основе параллелизм в программе можно было эффективно реализовать на практике, используя существующие средства компьютерной системы.

Далее будет рассмотрена реализация поддержки параллелизма в современных функциональных языках программирования. В разделе 1.1 будут рассмотрены статические средства распараллеливания функциональных программ с использованием нитей. В разделе 1.2 проводится анализ динамических средств распараллеливания функциональных программ с применением специальных операторов, позволяющих выделять функции, значения которых следует вычислять параллельно. В разделе 1.3 приводится описание принципиально иного подхода к распараллеливанию функциональных программ, использованного в языке FPTL. Отметим также, что, несмотря на существование в современном мире большого числа функциональных языков программирования, примитивы распараллеливания в них могут быть отнесены к одному из классов, рассмотренных в данной главе. По

этой причине обзор средств распараллеливания будет производиться на примере языков Haskell [6], F# [10] и Linear ML [11].

1.1 Статические средства распараллеливания

Одним из самых распространенных методов программирования многоядерных компьютеров является применение статической многопоточности (static threading) [12]. Она предоставляет программную абстракцию «виртуальных процессоров», или нитей (threads), совместно использующих общую память. Каждая нить поддерживает связанный с ней счетчик команд и может выполнять машинный код независимо от других нитей. Операционная система загружает нить в процессор для выполнения и переключает нити, когда выполнения требует другая нить. Хотя операционная система и позволяет программистам создавать и уничтожать нити, эти операции являются относительно медленными. Таким образом, для большинства приложений нити сохраняются на протяжении всего времени вычислений, почему они и получили название «статические».

Для работы с нитями в языке Haskell в составе стандартной библиотеки имеется функция forkIO [13]. Данная функция создает новую нить, в которой производится вычисление выражения, переданного в качестве входного параметра данной функции. Для реализации взаимодействия между несколькими работающими нитями в языке имеется механизм *общих переменных* и механизм *каналов синхронизации*. Каналы синхронизации позволяют организовывать только одностороннее взаимодействие между нитями посредством обмена сообщений через очереди. Общие переменные в языке Haskell имеют специальный тип MVar [13] и используются для чтения и записи данных из различных нитей. Они отличаются от обычных переменных в языке Haskell, значение которых не может быть изменено после их инициализации. Наличие изменяемых переменных в принципе противоречит основной концепции

функциональных языков программирования – неизменяемости данных (data immutability) [14].

В контексте параллельного программирования это может привести к возникновению несогласованного доступа к данным, или, по-другому, к состоянию гонки [16]. Параллельный алгоритм является детерминированным, если он всегда приводит к одним и тем же результатам при одних и тех же входных данных, независимо от того, в какой последовательности выстраиваются друг относительно друга инструкции, выполняемые разными нитями в многоядерном компьютере. Алгоритм является недетерминированным, если его поведение может меняться от выполнения к выполнению. Параллельные программы, приводящие к состоянию гонки, являются недетерминированными.

Одним из возможных решений проблемы использования изменяемых данных в параллельной среде является использование механизмов статического типового контроля. Такое решение было применено в экспериментальном функциональном языке программирования LinearML [11]. Этот язык создавался на основе языка ML и его ключевой особенностью является так называемая линейная система типов [15]. Такая система типов не позволят создать несколько ссылок на одни и те же данные – каждая переменная ссылается на свой собственный экземпляр данных. Это гарантируется системой типов на этапе компиляции программы.

$$\textbf{let } l_1 = [1; 2; 3]$$
$$\textbf{let } l_2 = \text{List.rev } l_1$$
$$\text{List.}\textbf{irelease } l_2$$

Например, в следующем фрагменте программы на языке LinearML сначала создается список целых чисел l_1, затем он используются для создания списка l_2, представляющего собой реверсированную версию списка l_1. После этого дальнейшее использование в программе переменной

l_1 будет считаться нарушением типизации, и на этапе компиляции будет выдаваться соответствующая ошибка. Кроме того, ошибкой типизации будет считаться, если переменная вообще не была использована, поэтому в конце примера применяется оператор **irelease** для освобождения выделенной памяти.

Такой подход к организации работы с данными при выполнении программы дает следующие преимущества. Во-первых, исключаются все проблемы организации взаимодействия между нитями, связанные с возможным несогласованным изменением состояния (состояния гонки). Во-вторых, исключается необходимость использования автоматической системы управления памятью (все остальные языки семейства ML, а также Haskell, используют для управления памятью автоматическую сборку мусора).

Другим возможным решением проблемы организации конкурентного доступа к общим данным является использование механизма *программной транзакционной памяти* [17]. При данном подходе взаимодействие с данными в оперативной памяти компьютера организуется по принципам, схожим с организацией ACID-транзакций в системах управления базами данных. Использование данного подхода сильно усложняет восприятие программы программистом, незнакомым с механизмами работы программной транзакционной памяти.

Другой проблемой статической многопоточности является сложность обеспечения равномерного распределения работы между нитями. Для любых (кроме самых простейших) программ для сбалансированной загрузки нитей программист должен разрабатывать сложные протоколы взаимодействия между нитями. Такое положение дел привело к созданию параллельных платформ, предоставляющих слой программного обеспечения, который координирует ресурсы параллельных вычислений, планирует их и управляет ими. Тем не менее, нити в

функциональных языках программирования нашли успешное применение при организации асинхронной работы с системой ввода-вывода, например, для работы с сетью или для чтения больших файлов.

1.2 Динамические средства распараллеливания. Параллельные платформы.

Для решения проблем, возникающих при использовании статических средств распараллеливания, была предложена модель динамического распараллеливания. Она позволяет программисту указывать уровень параллелизма в программе, не беспокоясь о коммуникационных протоколах, сбалансированности загрузки и других проблемах, возникающих при использовании нитевого программирования. Параллельная платформа содержит планировщик, который в свою очередь автоматически обеспечивает загрузку нитей, существенно упрощая работу программиста. Применение параллельных платформ позволяет использовать в функциональном языке программирования специальные операторы или синтаксические конструкции, позволяющие выделять выражения, значения которых должны вычисляться параллельно.

В языке Haskell в качестве таких конструкций используются операторы par и $pseq$ [18]. Денотационная семантика этих операторов может быть представлена следующим образом:

$$(par\ f\ g)d = g(d)$$

$$(pseq\ f\ g)d = \begin{cases} \bot, \text{если } f(d) = \bot \\ \text{иначе } g(d) \end{cases}$$

Здесь f и g представляют собой функциональные переменные языка Haskell, d – элемент данных, $\bot$ – неопределенный результат вычисления. Параллельная семантика этих операторов заключается в следующем: оператор par принимает в качестве аргументов два выражения, вычисление значений которых гарантированно производится параллельно.

В качестве результата оператор par возвращает значение второго из указанных выражений. Оператор $pseq$ также принимает на вход два выражения, но, в отличие от оператора par, их вычисление гарантированно производится последовательно. Этот оператор используется в тех случаях, когда необходимо гарантированно дождаться завершения параллельных вычислений, порожденных оператором par. В качестве результата $pseq$ возвращает значение первого выражения, если оно определено, иначе возвращает неопределенное значение.

Далее приводится простой пример применения операторов par и $pseq$ в языке Haskell для вычисления n-го числа Фибоначчи.

```
nfib :: Int -> Int
nfib n | n <= 1 = 1
       | otherwise = par f (pseq g (f + g)) where
                     f = nfib (n − 1)
                     g = nfib (n − 2)
```

В приведенном примере оператор par обеспечивает паралелльное вычисление чисел Фибоначчи для $n-1$ и $n-2$. После получения этих значений, что гарантируется оператором $pseq$, производится их суммирование для получения n-го числа Фибоначчи.

Внутренняя реализация оператора par заключается в том, что при вычислении значения $(par\ x\ y)d$, формируются два специальные структуры данных, называемые *заданиями* [19]. Каждое из заданий содержит описание выражения x и y соответственно и ссылку на выходные данные d. Планировщик параллельной платформы, в свою очередь, производит назначение заданий на выполнение программно-аппаратным ресурсам компьютерной системы. При этом в процессе выполнения задания могут рекурсивно порождаться другие задания.

Приведем другой пример использования параллельных платформ на примере языка F# [10]. Рассмотрим функциональную программу для параллельного вычисления n-го числа Фибоначчи.

```
let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ ->
        let x = Future<int>.Create(fun _ -> fib (n − 2))
        let y = Future<int>.Create(fun _ -> fib (n − 1))
        x.Value + y.Value
```

В рассмотренном примере функция Future.<int>.Create из библиотеки TPL [20] используется для порождения двух параллельных заданий x и y, представляющих соответственно параллельное вычисление значений функций **fun** _ -> $fib\ (n-2)$ и **fun** _ -> $fib\ (n-1)$. В следующей части программы x.Value + y.Value происходит ожидание готовности этих заданий и полученные значения используются для формирования результата. Как видно из примера, вместо специальных операторов распараллеливания используются непосредственные вызовы методов параллельной платформы TPL, а ожидание готовности вычислений производится неявно при обращении к полям заданий.

Применение динамических средств распараллеливания функциональных программ, основанных на использовании параллельных платформ, тем не менее, обладает своими недостатками. Написание корректных параллельных программ на языке Haskell с использованием операторов par и $pseq$ относительно легко, поскольку отсутствие побочных эффектов означает, что программисту не приходится задумываться о решении таких проблем, как состояние гонки или взаимоблокировка, которые могут значительно осложнить написание и отладку параллельных программ с помощью средств нитевого программирования. Однако, написание параллельной программы, которая обладает хорошей масштабируемостью на целом ряде параллельных архитектур, гораздо сложнее. Использование операторов распараллеливания требует от программиста большего внимания к выбору фрагментов программы, требующих распараллеливания. Например, в

языке Haskell, использующем стратегию отложенных вычислений, часто трудно понять, почему «идеальная» программа не выполняется так, как ожидает программист. При некорректном использовании операторов распараллеливания может возникнуть ситуация, при которой одно и то же заданий будет выполнено несколько раз. Пути решения этой проблемы в системе выполнения языка Haskell подробно описаны в [21].

1.3 Язык FPTL

Язык FPTL следует классическому подходу к заданию функций, берущему свое начало из модели вычислимых функций Черча. В этой модели функции стоятся с помощью определенного множества операций путем их композиции и заданных простых базисных функций.

Напомним, что языки LISP, ML, Haskell основаны на λ-нотации задания функций, которая создавалась с целью явного различения в математических контекстах связанных и свободных переменных. Ее функциональная семантика (однозначность редукции λ-выражений) была результатом доказательства получения однозначного результата редуцирования λ-выражений, если этот результат существует.

В языке FPTL существует четыре простых операции композиции функций, три из которых обладают параллельной семантикой. Общая форма задания функций – это система функциональных уравнений. Этого инструмента, как доказано, достаточно чтобы можно было определить любую функцию вычислимую над определенным в языке множеством данных.

В общем случае, программа на языке FPTL задается в виде тройки $\langle S, I, M \rangle$, где S – схема программы (система определенных в ней функций), I – интерпретация, позволяющая сопоставлять схеме различные функции путем переопределения в схеме конкретных базисных функций, M – модель вычисления значений функций. Первые две компоненты позволяют

существенно расширить в одном определении множество интересующих программиста функций (меняя интерпретацию схемы можно получить различные функции). Модель вычисления значений функций, в частности параллельного вычисления, формально извлекается исходя из свойств операций композиции функций [3].

Таким образом, программист в общем случае не должен, как это имеет место в рассмотренных языках Haskell и F#, явно определять и указывать те фрагменты программы, которые будут выполняться параллельно. В FPTL это реализует интерпретатор соответствующей модели параллельного вычисления значений функций. Как следствие, программист, разрабатывая программу, имеет дело только с заданием данных и необходимых функций над ними. Более того, путем эквивалентных преобразований схемы программы, программист может регулировать степень параллелизма в ней [22].

1.4 Заключение

Статические средства распараллеливания основаны на использовании нитей – низкоуровневых процессных средств операционной системы. Они берут свое начало из императивных языков программирования (C, C++, Java и т.д.) и требуют при написании параллельных программ использования средств взаимодействия между параллельными процессами, которые сложны в написании и приводят к многочисленным трудно обнаруживаемым ошибкам.

Динамические средства распараллеливания, т.н. параллельные платформы, берут на себя всю работу по организации и планированию параллельных вычислений. Это позволяет избавить программиста от реализации протоколов взаимодействия между нитями. С другой стороны, применяемые в них средства распараллеливания с помощью специальных операторов, доступные в большинстве параллельных платформ для

функциональных языков программирования, требуют от программиста явного выделения в программе параллельных частей.

Язык FPTL в свою очередь не требует от программиста использования специальных операторов распараллеливания. Параллельная семантика задается в нем неявно с помощью использования операций композиции функций. В то же время, модель параллельного вычисления значений функций в языке FPTL может быть реализована как с использованием статических средств распараллеливания, так и с использованием концепций параллельных платформ.

2. ЯЗЫК FPTL

В настоящей главе будет дано описание языка FPTL. В разделе 2.1 будет рассмотрена теоретическая база языка FPTL. В ней будет приведено описание операций композиции, представления функций и данных, а также будет рассмотрен один из вариантов реализации модели вычислений значений функций – модель преобразования деревьев. Далее в разделе 2.2 будет дано неформальное описание синтаксиса языка и примеры синтаксических конструкций, описание функций ввода-вывода и работы с массивами.

2.1 Теоретические основы языка FPTL

Теоретическую базу языка FPTL составляют исследования по функциональной схематологии и функциональным системам [23; 51], которые обобщены в теории направленных отношений, объединяющей функциональный и логический стили программирования [21; 24].

Язык FPTL имеет две семантики: денотационную и параллельную операционную семантику, которые будут рассмотрены в следующих разделах.

2.1.1 Представление функций

Функции в FPTL рассматриваются, как типизированные $t_1' \times t_2' \times \ldots \times t_m' \to t_1'' \times t_2'' \times \ldots \times t_n''$ (m,n)-арные соответствия $(m \geq 0, n \geq 0)$ между кортежами данных, где $t_i', i = \overline{1,m}$ и $t_j'', j = \overline{1,n}$ - типы элементов входного и выходного кортежей, m - длина кортежа на входе функции, n - на выходе. Функции арности $(0,1)$ рассматриваются в FPTL как константы. Для m и n равных 0 имеем кортеж нулевой длины, обозначаемый λ, со свойствами $\lambda\alpha = \alpha\lambda = \alpha\lambda$, где α - произвольный кортеж. Кортеж данных в языке представляется в виде последовательной записи его элементов.

В FPTL в отличие от общепринятой формы задания функций с явным указанием ее аргументов (так называемая форма задания общего значения функции) строго различается собственно функция как отображение одного множества в другое и ее аппликация к конкретным данным. Роль переменных в задании функций в FPTL выполняют функции выбора необходимого элемента из кортежа данных. Формально функция выбора аргумента, обозначаемая $I(i,m)$ (в языке обозначается сокращенно как $[i]$), при применении к кортежу произвольного типа данных длины m ($m > 0$) выбирает его i-й элемент, $i = \overline{0,m},\ m > 0$. Для $i = 0$ выбираемое значение есть λ. Функции в FPTL являются в общем случае частичными, причем неопределенное значение функции может быть выражено либо как неограниченный процесс вычисления ее значения, либо как специальное вычисленное неопределенное значение, обозначаемое ω, со свойствами ωα = αω = ω для любого кортежа α.

Формально функции определяются как системы функциональных уравнений $F_i = \tau_i,\ i = \overline{1,n}$, где τ_i – функциональные термы, построенные из заданных (базисных) функций и функциональных переменных F_i путем применения четырех бинарных операций композиции функций: →, +, *, •. Для функций и функциональных переменных задана их арность, а для базисных функций – также их тип. Тип функциональных переменных совпадает с типом определяющих их термов и однозначно определяется из задания типов базисных функций и правил вывода типов для функций, построенных путем применения операций композиции.

Пусть $f^{(m,n)}$ – (m,n)-арная функция, $f(\alpha)$ – результат ее применения к кортежу α, f_1, f_2 – заданные функции. Синтаксис и семантика операций композиции определяются следующим образом.

1. Последовательная композиция (•):

   $f^{(m,n)} \stackrel{\text{def}}{=} f_1{}^{(m,k)} \cdot f_2{}^{(k,n)}$;

   $f(\alpha) \stackrel{\text{def}}{=} f_2\big(f_1(\alpha)\big)$,

где $\alpha = \alpha_1\ \alpha_2\ \alpha_3\ \ldots\ \alpha_m$ – кортеж данных типа $t_1 \times t_2 \times \ldots \times t_m$; t_i – тип данных i-го аргумента функции. Здесь и далее сначала задается синтаксис операции композиции, а затем ее семантика, определяемая через применение функций к кортежу данных. Предполагается, что типы кортежа значений функции f_1 и кортежа аргументов функции f_2 одинаковы.

Заметим, что в FPTL используется префиксная форма записи операции последовательной композиции, задающая последовательный характер вычисления значений функций f_1 и f_2 и эквивалентная последовательному характеру задания следования операторов при выполнении последовательных программ.

2. Операция конкатенации кортежей значений функций (*):

$$f^{(m,n_1+n_2)} \stackrel{\text{def}}{=} {f_1}^{(m,n_1)} * {f_2}^{(m,n_2)};$$

$$f(\alpha) \stackrel{\text{def}}{=} f_1(\alpha) f_2(\alpha).$$

Предполагается, что типы аргументов функций f_1 и f_2 одинаковы.

3. Операция условной композиции[1]:

$$f^{(m,n)} \stackrel{\text{def}}{=} {f_1}^{(m,k)} \to {f_2}^{(m,n)};$$

$$f(\alpha) \stackrel{\text{def}}{=} \begin{cases} f_2(\alpha), \text{если } f_1(\alpha) \text{ определено и отлично от значения «ложь»}, \\ \omega \text{ иначе.} \end{cases}$$

Предполагается, что типы кортежей аргументов функций f_1 и f_2 одинаковы.

4. Операция объединения (графиков) ортогональных функций:

$$f^{(m,n)} \stackrel{\text{def}}{=} {f_1}^{(m,n)} + {f_2}^{(m,n)};$$

$$f(\alpha) \stackrel{\text{def}}{=} \begin{cases} f_1(\alpha), \text{если значение } f_1(\alpha) \text{ определено}, \\ f_2(\alpha), \text{если значение } f_2(\alpha) \text{ определено}, \\ \text{иначе не определено.} \end{cases}$$

---

[1] Для удобства программирования, в язык также введена тернарная операция условной композиции, которая является аналогом условного оператора *if-then-else* и представляется в FPTL как $(p \to f_1, f_2)(x)$, где p – предикат, а f_1 и f_2 функции, значение одной из которых будет использовано в зависимости от того, истинно или ложно значение $p(x)$. В приведенных далее в главе примерах программ на языке FPTL будет использован именно этот вариант условной композиции.

Предполагается, что типы аргументов и значений функций f_1 и f_2 одинаковы. Напомним, что функции f_1 и f_2считаются ортогональными, если для всякого кортежа данных α определена не более чем одна из них. Операция объединения ортогональных функций была введена с целью представления параллельных функций (известный пример – функция голосования в телефонии) [22].

Все операции композиции являются ассоциативными, а операция + выступает как коммутативная. Приоритет операций композиции определяется следующем образом (в порядке возрастания): +, →, *, •.

Как уже было отмечено, термы в задании функций в виде систем функциональных уравнений представляют собой композиции, построенные из базисных функций и функциональных переменных путем применения операций композиции. Предполагается, что арности и типы терма и определяемой им функциональной переменной в системе функциональных уравнений одинаковы. Функциональные переменные выполняют двойную роль при задании функции (построении функциональной программы): одни из них появляются как необходимые элементы при задании рекурсивных функций, другие определяются далее (в следующем уравнении уточняемой функции), позволяя просто реализовать пошаговую разработку функциональной программы по технологии проектирования «сверху-вниз».

2.1.2 Представление данных

В FPTL можно представлять любой структурный тип данных, называемый *абстрактным типом данных* (АТД), определяя его по аналогии с определением функций в общем случае через систему рекурсивных уравнений. При определении абстрактных типов данных используются функции-конструкторы и обратные к ним функции-деструкторы, которые вместе с функциями выбора аргумента образуют полный набор базисных функций в том смысле, что любая вычислимая

функция над данным рассматриваемого типа может быть выражена средствами языка FPTL [3; 25; 26].

При определении абстрактных типов данных в качестве исходных можно также использовать встроенные типы: *bool*, *real*, *int*, *string* и др. вместе с определенными в конкретной компьютерной системе операциями над ними. Для определения типов данных применяются те же операции композиции, которые применяются для задания функций, за исключением того факта, что операция + трактуется как операция объединения двух множеств данных (в языке обозначается как ++). Заметим, что ω можно использовать, как специальное обозначение «неопределенного» значения функции, которое определяется конструктивно. Из этого следует, что базисные функции, извлекаемые из определения абстрактного типа данных, могут рассматриваться как всюду определенные структуры.

Приведем пример определения в FPTL абстрактного типа данных (списка натуральных чисел):

$$Nat = c\_null \mathbin{++} Nat \bullet c\_succ;$$

$$ListOfNat = c\_nil \mathbin{++} (Nat * ListOfNat) \bullet c\_cons;$$

Здесь функции-конструкторы *c_null*, *c_succ*, *c_nil* и *c_cons* имеют арности (0,1), (1,1), (0,1) и (2,1) соответственно и следующие типы: {λ}→Nat, Nat → Nat, {λ} → *ListOfNat*, *Nat * ListOfNat → ListOfNat*.

Обратные к ним функции (деструкторы), обозначаемые как *~c_null*, *~c_succ*, *~c_nil*, *~c_cons*, автоматически извлекаются из определения типа и имеют следующую интерпретацию:

$$\sim c\_null(x) = \begin{cases} \lambda, \text{если } x = 0, \\ \omega, \text{в противном случае;} \end{cases}$$

$$\sim c\_succ(x) = \begin{cases} y, \text{если } x = c\_succ(y), \\ \omega, \text{в противном случае;} \end{cases}$$

$$\sim c\_nil(x) = \begin{cases} \lambda, \text{если } x = c\_nil, \\ \omega, \text{в противном случае;} \end{cases}$$

$$\sim c\_cons(x) = \begin{cases} y, \text{если } x = c\_cons(y), \\ \omega, \text{в противном случае.} \end{cases}$$

Приведем пример функции-предиката, которая проверяет принадлежность кортежа типу данных *ListOfNat*:

IsListOfNat = ~c_nil + ~c_cons •([1] •*IsNat* * [2] •*IsListOfNat)*;

IsNat = ~c_null + ~c_succ.IsNat.

Функция *IsListOfNat* определена на любом кортеже данных, принадлежащих *ListOfNat*, и ее значением является λ, что может трактоваться как «истина». Для других отличных от *ListOfNat* данных в качестве результата применения *IsListOfNat* будет неопределенное значение ω, которое может также трактоваться как «ложь». Заметим, что любое значение функции-предиката, отличное от ω и «ложь» может трактоваться как «истина».

2.1.3 Модель параллельного вычисления значений функций

В этом параграфе будет рассмотрена одна из моделей параллельного вычисления значений функций, представляющая собой процесс преобразования деревьев [27].

На каждом шаге состояние вычисления представляется бинарным размеченным деревом, таким, что:

- листья дерева помечены символами элементов $D \cup \{\omega\}$, где D – универсум данных, ω – вычислимое неопределенное значение;
- внутренние вершины дерева помечены либо символами операций •, *, →, +, либо функциональными термами.

Не ограничивая общности, допустим, что интересующая нас функция X_1 описывается системой рекурсивных функциональных уравнений типа $X_i = \tau_i$, $i = \overline{1,n}$.

Вычисление значения функции $X_1^{(\min)}$, являющегося первой координатой наименьшего решения для X_1 системы функциональных уравнений для аргумента – кортежа d, представляется в виде конечной или неограниченной последовательности состояний, начальный элемент которой – дерево с двумя вершинами, причем корневая вершина помечена символом функциональной переменной X_1, а листовая – аргументом d. Эта последовательность содержит единственное (если процесс заканчивается успешно) конечное состояние. Если это состояние есть дерево из одной вершины, помеченное некоторым данным $d' \in D$, то говорим о результативном окончании вычисления значения функции с d'. Если это конечное состояние есть ω или процесс вычислений неограничен, то делается вывод о безрезультатном завершении вычисления. Переходы из состояния в состояние при поиске значения функции определяются правилами преобразования деревьев, разделенными на два подмножества: правила развертывания и свертывания деревьев.

Правила преобразования деревьев задаются в виде схем изменения состояний и обозначаются $u' \Longrightarrow u''$, где u' и u'' – схемы состояния. Схема состояния отличается от конкретного состояния тем, что в ней для разметки вершин дерева состояния могут использоваться следующие метапеременные: t – произвольный функциональный терм, u – произвольное дерево-состояние, d – произвольный элемент $D \cup \{\omega\}$, f – базисная функция, X_i – функциональная переменная.

Результат применения правила $u' \Longrightarrow u''$ к состоянию u есть дерево, полученное путем замены в u некоторого его поддерева u' на дерево u''.

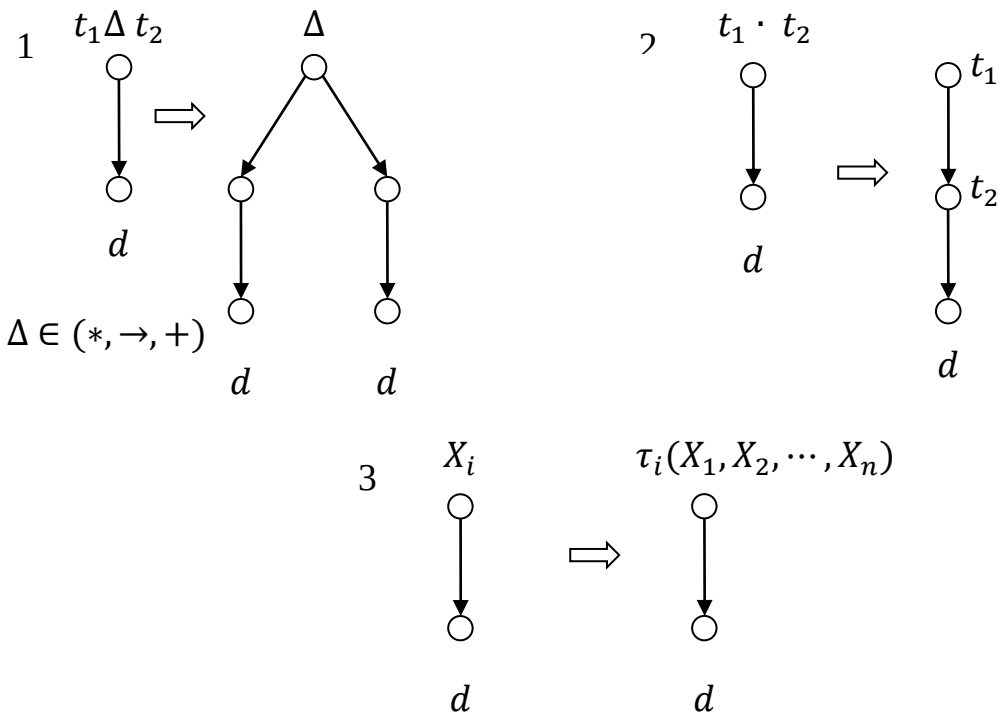

Рис 2.1. Правила развертывания деревьев.

На рис. 2.1 показаны правила развертывания, на рис. 2.2 – свертывания деревьев. Справедливость правил 6-9 вытекает из того, что операция + применяется только к ортогональным или совместным функциям.

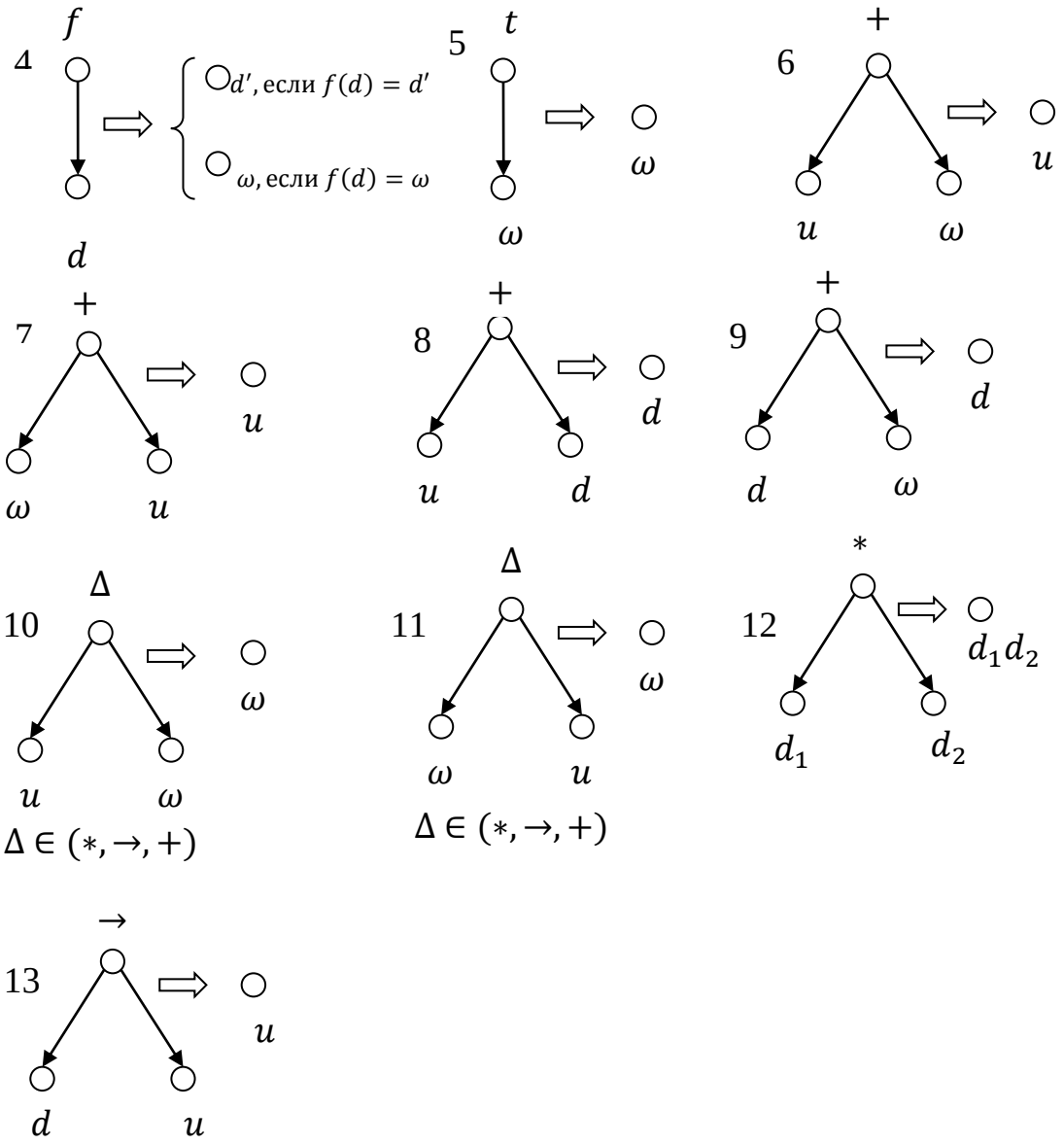

Рис. 2.2. Правила свертывания деревьев.

Данная модель является недетерминированной, поскольку в общем случае возможно применение нескольких правил к дереву-состоянию вычислений, и поэтому в зависимости от используемого правила будут получаться различные последовательности вычислений. Модель также параллельна, так как возможно одновременное применение нескольких правил к несвязным кустам дерева состояния. Источник параллелизма – свойства операций $*$, $\rightarrow$, $+$. Практически это означает, что процесс

вычислений развивается независимо по различным ветвям дерева-состояния, что дало право назвать модель вычислений асинхронной [27].

Важно отметить, что не всякий порядок применения правил преобразования состояний приводит к корректному вычислению значения функции. Например, при вычислении $(t_1 + t_2)(d)$, если сначала делается попытка вычисления $(t_1)(d)$, которое не определено и процесс продолжается бесконечно, а значение $(t_2)(d)$ определено, нужный результат не будет получен. Таким образом, условием корректности реализации модели является параллельное (или квазипараллельное) вычисление значений функций, соединяемых операцией +.

Не менее существенным является умение прерывать ненужные вычисления. Как видно из правил 6-11, при получении на одной из ветвей значения d, отличного от ω, или значения ω вычисления, связанные с другой ветвью, необходимо прервать.

Кроме этого, модель предоставляет возможность повышения эффективности вычислений за счет условных конструкций с упреждением. Если имеется достаточное количество ресурсов, то при определении значения $(t_1 \to t_2)(d)$ можно значение $(t_1)(d)$вычислять одновременно с $(t_2)(d)$, стремясь максимально распараллелить процесс.

Эти особенности модели являются принципиальными при ее реализации на вычислительных системах и при разработке эффективных алгоритмов планирования параллельного выполнения функциональных программ.

Другая модель организации процесса параллельного вычисления значений функций создавалась специально для эффективной реализации на многоядерных процессорах и будет детально рассмотрена в главе 4.

2.2 Описание языка FPTL

В этом параграфе будет дано неформальное описание основных конструкций языка и показано их использование на простых примерах. Сначала будет приведена общая структура функциональной программы на языке FPTL, после чего будет рассмотрена структура отдельных ее элементов (блоков).

Структурно FPTL-программа состоит из следующих блоков:

- блок описания импорта внешних функций;
- блок описания данных;
- блок описания функций, заданных в виде функциональных уравнений;
- блок интерпретации функциональных уравнений и задания статических параметров функциональной программы.

Обязательной частью программы является только блок описания функциональных уравнений, остальные блоки могут отсутствовать.

Далее будет описан синтаксис каждого из перечисленных блоков. При описании синтаксиса будут использоваться следующие условные обозначения.

- Курсивом будут обозначаться идентификаторы, задаваемые программистом.
- Полужирным шрифтом будут обозначаться ключевые слова языка FPTL.
- Запись < *E* > обозначает, что конструкция *E* может быть опущена, либо может повторяться 1, 2 и более раз.
- Запись << *E* >> обозначает, что конструкция *E* может повторяться не менее 1, 2 и более раз.

2.2.1 Блок импорта внешних функций

Данная конструкция предназначена для описания функций, реализация которых находится во внешних библиотеках. Такие функции

могут быть реализованы на различных языках программирования (например, C или C++). Описание внешних функций представляет собой список следующего вида:

< **import** *имя_внешней_функции* **from** “*имя_файла_библиотеки*”; >

Пример 1:

import *some_function* **from** “*some_lib.dll*”;

В данном примере импортируется функция *some_function* из файла библиотеки с именем *some_lib.dll*. Реализация механизма импорта библиотек и вызова внешних функций будет описана в 4.8.

2.2.2 Блок описания данных

Данный блок предназначен для описания абстрактных типов данных и их конструкторов и состоит из списка конструкций вида:

data *имя_АТД*
{
<< *имя_АТД* = конструктор < ++ конструктор > ; >>
}

Синтаксис описания конструкторов следующий:

имя_типа <* *имя_типа*> . *имя_конструктора*

Здесь имя типа может являться как именем встроенного типа данных (int, real, boolean, string и т.д), так и именем абстрактно типа данных (включая и определяемый АТД). Возможно также задание конструктора, принимающего на вход пустой кортеж данных. В этом случае описание конструктора состоит только из его имени.

Пример 2:

data *Nat*
{
Nat = c_nil ++ Nat . c_succ;
}

В данном примере задается абстрактный тип данных – натуральное число, содержащий конструкторы: *c_nil* и *c_succ*. Конструктор *c_nil*

принимает на вход пустой кортеж, в то время, как конструктор *c_succ* принимает кортеж из одного элемента типа *Nat*.

Пример 3:

```
data ListOfNat
{
        ListOfNat = c_empty ++ Nat * ListOfNat . c_cons;
}
```

В этом примере задается АТД – список натуральных чисел, тип данных которых описан в примере 2. Конструктор c_empty принимает на вход пустой кортеж и соответствует варианту пустого списка. Конструктор *c_cons* соответствует списку, содержащему один элемент данных типа *Nat* и хвост списка типа *ListOfNat*.

Пример 4:

```
data ComplexNumber
{
        ComplexNumber = real * real . c_complex;
}
```

В этом примере определяемый АТД *ComplexNumber* определяет тип данных – комплексное число. Он содержит единственный конструктор *c_complex*, принимающий на вход кортеж из двух вещественных чисел: вещественной и мнимой частей комплексного числа.

В FPTL-программах можно также определять параметризованные типы данных. Эти определения имеют следующий синтаксис:

```
data имя_АТД [‘ имя_параметра <, ‘ имя_параметра >]
{
        << имя_АТД  = конструктор < ++ конструктор > ; >>
}
```

Типовой параметр представляет собой идентификатор, начинающийся с символа «‘». Приведем пример задания параметризированного АТД.

Пример 5:

```
data List[‘t]
{
```

*List = c_empty ++ ‘t * List*[‘*t*] . *c_cons*;
}

В этом примере задается АТД *List*['*t*] с единственным типовым параметром '*t*. Присвоив параметру ‘*t* значение *Nat* (тип *Nat*), мы получаем ранее определенный абстрактный тип *ListOfNat*.

Подобным образом можно представить и комплексное число:

Пример 6:

data *Pair*[‘*s*, ‘*t*]
{
*Pair = ‘s * ‘t . c_pair*;
}

Присваивая параметру '*s* и параметру '*t* значения real мы получаем представление комплексного числа из примера 4.

2.2.3 Блок описания функций

Базисные функции в языке FPTL имеют зарезервированные имена. Их полный список приведен в приложении 1. Операции композиции ∗, · и → определяются в языке через символы «*», «.» и «->» соответственно. Для удобства чтения, в дальнейших примерах операция условной композиции по-прежнему будет обозначаться через символ →.

Блок описания функций является основной частью FPTL-программы и называется *схемой*. В нем описываются непосредственно функциональные уравнения. Синтаксис описания системы функциональных уравнений следующий:

scheme *имя_схемы*
{
<< *имя_функциональной_перем.* = функциональный терм; >>
}

Одно из функциональных уравнений, описанных в схеме, называется *главным* функциональным уравнением. Имя определяемой им функциональной переменной должно совпадать с именем схемы. Функциональные термы строятся на основе базисных функций, функций,

импортируемых из внешних библиотек, функциональных переменных, конструктов и деструкторов по правилам, описанным ранее в разделе 2.1.1. Следует отметить, что описанные в разделе 2.1.1 специальные значения λ и ω не могут быть явно использованы в программе, а могут быть получены только в результате вычислений.

Пример 7:

```
scheme Factorial
{
    Factorial = ([1] * 0) . equal → 1, (([1] * 1) . sub . Factorial * [1]) .
mul;
}
```

Для разграничения лексического контекста в языке FPTL существует еще один вариант конструкций – конструкции **fun**. Синтаксис их описания аналогичен синтаксису описания схемы.

```
fun имя_функции
{
    << имя_функциональной_перем = функциональный терм; >>
}
```

Каждая конструкция **fun** имеет свой лексический контекст. Другими словами внутри нее могут быть переопределены имена функциональных переменных описанных в схеме. Конструкции **fun** могут быть вложенными.

Пример 8:

```
scheme Integral
{
    Sqr = ([1]*[1]).mul;
    Integral = (0.0 * 2.0).Calc;

    fun Calc
    {
        Fs = ([1].Sqr * [2].Sqr).add;
        Dx = ([2] * [1]).sub;
        Trp = ((Fs * Dx).mul * 0.5).mul;
    }
}
```

В данном примере приводится простая функциональная программа вычисления значения интеграла функции $f(x) = x^2$ на отрезке [0, 2] методом трапеций.

Другим вариантом использования конструкции **fun** является задание функционалов. Оно имеет в FPTL следующий синтаксис:

fun *имя_функционала*[*имя_параметра* <, *имя_параметра* >]
{
<< *имя_функциональной_перем* = функциональный терм; >>
}

Синтаксис использования функционала в правой части функционального уравнения следующий:

имя_функционала (параметр <, параметр>)

В качестве фактических параметров могут выступать функциональные переменные, базисные функции, константы, конструкторы и деструкторы. Использование выражений с операциями композиции не допускается.

Пример 9:

```
scheme Functional
{
    Sqr = ([1]*[1]).mul;
    Functional = Trp(Sqr);

    fun Trp[F]
    {
        Fs = ([1].F * [2].F).add;
        Dx = ([2] * [1]).sub;
        Trp = ((Fs * Dx).mul * 0.5).mul;
    }
}
```

В примере 9 приведена программа, аналогичная описанной в примере 8, но написанная с использованием функционала.

2.2.4 Блоки интерпретации и применения схемы

В общем случае схема, подобно конструкции **fun**, задает функционал. Интерпретация позволяет получить из этого функционала конкретную функцию путем подстановки конкретной функциональной переменной вместо функционального параметра. Сохраняя общую структуру программы, описываемую схемой, но применяя к ней различные интерпретации, можно получать различные конечные программы. Синтаксис описания блока интерпретации следующий:

interpretation
<< имя_интерпретации
{
<< *имя_функционального_параметра_схемы* = значение ; >>
}
>>

В качестве значений функциональных параметров могут выступать базисные функции, константы, конструкторы и деструкторы. Использование выражений с операциями композиции не допускается. Имя конкретной интерпретации, которая будет использована при выполнении функциональной программы, задается программистом при запуске выполнения программы через параметры командной строки.

Для передачи схеме начального кортежа данных в FPTL-программе может быть использован блок задания констант. Он имеет следующий синтаксис:

application
< *имя_константы* = значение_константы ; >
% имя_схемы(имя_констранты <, имя_константы>)

В данном блоке описываются константы, которые будут использованы в качестве кортежа данных, поступающего на вход схемы. Приведем пример FPTL-программы с использованием в нем блоков **application** и **interpretation**.

Пример 10:

data *ListOfComplex*
{
Complex = (real * real).*c_complex*;

```
        ListOfComplex = c_empty ++ (ListOfComplex * Complex).c_list;
    }
    scheme Accumulate {
        Accumulate = ~c_empty → 0.0, ~c_list.([1].Accumulate * [2].Part).add;
    }
    interpretation
        Real { Part = ~c_complex.[1]; }
        Img  { Part = ~c_complex.[2]; }
    application
    stack = (((c_empty * ((0.1*0.1).c_complex)).c_head)*
((0.2*0.2).c_complex)).c_head;
    %Accumulate(stack)
```

В данном примере приводится функциональная программа вычисляющая сумму мнимых или вещественных частей списка комплексных чисел. В секции **scheme** описывается рекурсивное функциональное уравнение *Accumulate*, секция **interpretation** задает 2 интерпретации с именами *Real* и *Img* (которые определяют функцию *Part* в схеме), извлекающие значение вещественной или мнимой части соответственно.

2.3 Организация ввода-вывода в языке FPTL

Для осуществления операций ввода-вывода в языке FPTL имеются следующие функции, описанные в таблице 2.1. Эти функции относятся к базисным.

Имя функции	Входные данные		Выходные данные		Описание
fread	string	Имя файла.	string	Считанная строка.	Производит чтение данных из текстового файла.
fwrite	string	Имя файла.			Производит чтение структуры данных из двоичного файла.
	string	Данные для записи.			
print	string	Данные для записи			Производит вывод кортежа данных на дисплей.

Таб. 2.1. Базисные функции ввода-вывода.

Более сложные операции ввода-вывода могут быть при необходимости реализованы на других языках программирования с помощью механизма вызова внешних процедур. Также следует отметить, что средства ввода-вывода не защищены от побочных эффектов и ответственность за корректную работу с ним несет разработчик функциональной программы.

2.4 Работа с массивами

Для представления массивов в язык FPTL был введен специальный тип данных – Array['*t*] и следующие процедуры для работы с ним.

Имя функции	Входные данные		Выходные данные		Описание
arrayCreate	int	Длина массива.	Array['*t*]	Созданный массив.	Создает массив заданной длины и заполняет все его элементы начальным значением.
	't	Начальное значение.			
arrayAssign	Array['*t*]	Массив.	Array['*t*]	Измененный массив.	Присваивает элементу массива с заданным индексом заданное значение.
	int	Индекс элемента.			
	't	Значение.			
arrayGet	Array['*t*]	Массив.	*'t*	Элемент массива.	Возвращает элемент массива с заданным индексом.
	int	Интдекс элемента.			

Таб. 2.2. Базисные функции для работы с массивами.

В качестве типового параметра '*t* массива могут выступать встроенные типы данных, АТД и массивы других типов (допускается задание вложенных массивов). Заметим, что, подобно функциям ввода-вывода, работа с массивами в языке FPTL не защищена от возможных побочных эффектов. Корректность работы с массивами лежит на

разработчике функциональной программы. Приведем пример работы с массивами в языке FPTL.

Пример 12.

```
scheme FillArray
{
        FillArray = ([1] * ([1] * 0) . arrayCreate) . Fill;
        F = ([2] * [1] * rand) . arrayAssign;

        fun Fill
        {
                N = [1];
                Arr = [2];
                Fill = (N * Arr * 0) . Recurse . [1];
                Recurse = ([3] * N) . equal  -> Arr * Arr, (N * ([3] * Arr) . F
* ([3] * 1) . add) . Recurse;
        }
}
```

В данном примере приводится функция заполнения массива случайными числами. Здесь функциональное уравнение *F* отвечает непосредственно за присваивание одному элементу массива значения, сгенерированное базисной функцией rand. Рекурсивное функциональное уравнение *Fill* используется для того, чтобы последовательно произвести заполнение всего массива.

2.3 Заключение

В настоящей главе было дано описание синтаксиса и семантики языка FPTL, в том виде, в котором они были использованы в описываемой реализации языка на многоядерных компьютерах. Кроме того, в язык были введены расширения для описания операций ввода-вывода, а также для выполнения операций над массивами.

Рассмотрена теоретическая модель параллельного вычисления значений функций и возникающие при этом формы параллелизма, имеющие важное значение при его реализации.

3. ТИПОВОЙ КОНТРОЛЬ FPTL ПРОГРАММ

FPTL является типизированным языком программирования и условия определения типов, полученных путем применения операций композиции функций, были предварительно рассмотрены в главе 2. При построении функциональных программ на FPTL, в частности при определении функций в общем виде как систем функциональных уравнений $F_i = \tau_i, i = \overline{1,n}$ в термах могут использоваться базисные функции, извлекаемые из вводимых определений абстрактных типов данных конструкторы и деструкторы, а также так называемые встроенные функции, реализованные в компьютерах: арифметические, логические и др. Эти функции могут иметь как конкретные типы (например, описанный в главе 2 конструктор натуральных чисел), так и быть полиморфными, как, например, арифметические функции и функция выбора аргументов. Поэтому помимо проверки синтаксиса программы, возникает проблема проверки ее типовой корректности. На практике в реализации языка типовой контроль может быть произведен до выполнения программы, т.е. статически, что возможно не для всех языков программирования, либо динамически, т.е. в процессе выполнения программы [28]. Естественно, что для параллельных программ, целью которых является уменьшение времени их выполнения, целесообразно применять статический типовой контроль (если он возможен). В начальной реализации языка FPTL на многоядерных компьютерах применялся динамический алгоритм типового контроля, который, внося существенные накладные расходы, значительно увеличивал время выполнения программы. В то же время в FPTL типовой контроль можно осуществлять до выполнения программы. С этой целью в рамках диссертационной работы был разработан и реализован алгоритм проверки типовой правильности FPTL-программы, который приводится в данной главе.

В разделе 3.1 настоящей главы будут даны основные определения, использованные для описания правил типовой корректности программ на языке FPTL. В разделе 3.2 приводится формальное описание правил типовой корректности FPTL-программ. В разделе 3.3 описан алгоритм проверки типовой корректности FPTL-программ.

3.1 Основные определения

Напомним, что функции в FPTL задаются в виде систем в общем случае рекурсивных функциональных уравнений: $F_i = \tau_i, i = \overline{1,n}$, где F_i – определяемые функции (функциональные переменные), τ_i – термы, построенные применением описанных в главе 2 операций композиции к множеству базисных функций F_{basis} и функциональных переменных $F = \{F_i | i = \overline{1,n}\}$.

Определение. Пусть T – множество константных типов, которое состоит из множества встроенных в FPTL типов (*real*, *int*, *float*, *bool* и др.) и абстрактных (определяемых пользователем) типов; X – множество типовых переменных, таких, что каждая переменная $x \in X$ принимает в качестве значений типы из T. Пусть далее $M^k = \underbrace{M \times M \times ... \times M}_{k \text{ раз}}$-множество кортежей длины k, построенных из элементов множества $M = T \cup X$. Очевидно, что M^0 состоит из единственного пустого кортежа, обозначаемого (как было указано выше) λ.

Кортеж типов длины k записывается следующим образом: $z_1 \times z_2 \times ... \times z_k$, $z_i \in M, i = \overline{1,k}$. Кортеж типов, не содержащий типовых переменных, называется константным. Кортеж типов, содержащий переменные, интерпретируется как множество константных кортежей, полученных путем всевозможных подстановок константных типов из T вместо вхождения в кортеж переменных типов.

Определение. Содержательно тип (m,n)-арной функции определяется как множество однозначных отображений $\left\{t_1^{(i)} \times t_2^{(i)} \times \ldots \times t_m^{(i)} \to t_1'^{(i)} \times t_2'^{(i)} \times \ldots \times t_n'^{(i)} \middle| i = 1,2,\ldots\right\}$, где справа и слева от → стоят константные кортежи. Множество однозначных отображений может быть как конечным, так и счетным. Синтаксически тип функций задается в форме $\left\{z_1^{(i)} \times z_2^{(i)} \times \ldots \times z_m^{(i)} \to z_1'^{(i)} \times z_2'^{(i)} \times \ldots \times z_n'^{(i)} \middle| i = \overline{1,k}\right\}$. В интерпретации эта запись представляет множество константных отображений, полученных всевозможными подстановками константных типов в типовые кортежи слева и справа от →. Предполагается, что одинаковым переменным, входящим в кортежи, присваиваются одни и те же типовые константы при подстановке.

Требование однозначности в определении типа функции существенно, поскольку в случае, если оно не выполняется, мы получаем неопределенность. Приведем иллюстрирующий это пример: пусть тип $f^{(m,n)}$ равен $\{int \times int \to real,\ int \times int \to int\}$. Представим ситуацию, когда на входе f имеется кортеж из двух целых чисел. Становится понятно, что мы не можем однозначно определить выходной типовой кортеж, поскольку неочевидно, ожидать нам на выходе целое или вещественное число. Однозначность не выполняется для функций, имеющих типы $\{x \to y\}$ и $\{x \to t\}, t \in T; x, y \in X$.

Определение. Функция называется полиморфной, если входной или выходной кортеж хотя бы одного из ее типовых элементов содержит типовую переменную, или если тип функции содержит более одного типового элемента.

Функция выбора i-го элемента из кортежа $I(i,m)$ полиморфна и имеет тип $\{x_1 \times x_2 \times \ldots \times x_i \times \ldots \times x_m \to x_i\}$. Базисная функция сложения add полиморфна, поскольку ее тип состоит из более чем одного

типового элемента: $\{int \times int \to int, int \times real \to real, real \times int \to real, real \times real \to real\}$.

Далее будут сформулированы условия правильной типизации функций в языке FPTL. Пусть функции представлены в форме систем функциональных уравнений вида $F_i = \tau_i, i = \overline{1,n}$. Будем исходить из предположения, что типы всех входящих в правые части τ_i базисных функций (встроенных функций, функций-конструкторов и деструкторов в определении абстрактных типов данных) заданы.

3.2 Условия типовой корректности

Введем отношение порядка на множестве типов, положив: $t \leq t, x_i \leq x_j, t \leq x \; \forall t \in T, \forall x, x_i, x_j \in X$. Отношение порядка на множестве кортежей вводится следующим образом: $y_1 \times y_2 \times ... \times y_k \geq z_1 \times z_2 \times ... \times z_k$, если $y_i \geq z_i, i = \overline{1,n}$. Будем считать, что тип функции $f_1^{(m,n)}$ меньше или равен типу функции $f_2^{(m,n)}$ $(type(f_1) \leq type(f_2))$, если выполнено условие: для каждого типового элемента $y_1^{(i)} \times y_2^{(i)} ... \times y_m^{(i)} \to z_1^{(i)} \times z_2^{(i)} \times ... \times z_n^{(i)}$, принадлежащего $type(f_1)$, существует типовой элемент $y_1'^{(i)} \times y_2'^{(i)} ... \times y_m'^{(i)} \to z_1'^{(i)} \times z_2'^{(i)} \times ... \times z_n'^{(i)}$, принадлежащий $type(f_2)$, такой, что: $y_1^{(i)} \times y_2^{(i)} ... \times y_m^{(i)} \leq y_1'^{(i)} \times y_2'^{(i)} ... \times y_m'^{(i)}$ и $z_1^{(i)} \times z_2^{(i)} \times ... \times z_n^{(i)} \leq z_1'^{(i)} \times z_2'^{(i)} \times ... \times z_n'^{(i)}$. Положим, что $type(f_1) = type(f_2) \Leftrightarrow type(f_1) \leq type(f_2)\text{^}type(f_1) \geq type(f_2)$.

Пусть $\alpha = y_1 \times y_2 \times ... \times y_m \to z_1 \times z_2 \times ... \times z_n$- типовой элемент. Тогда типовой кортеж $y_1 \times y_2 \times ... \times y_m$ называется входным и обозначается $IN(\alpha)$. Аналогично, кортеж $z_1 \times z_2 \times ... \times z_n$ называется выходным и обозначается $OUT(\alpha)$.

Для типа функции $\beta = \left\{ y_1^{(i)} \times y_2^{(i)} ... \times y_m^{(i)} \to z_1^{(i)} \times z_2^{(i)} \times ... \times z_n^{(i)} \;\middle|\; i = \overline{1,k} \right\}$ определим $IN(\beta)$ и $OUT(\beta)$ как объединение множеств входных и

выходных кортежей каждого типового элемента соответственно: $IN(\beta) = \{IN(\alpha)|\alpha\epsilon\beta\}, OUT(\beta) = \{OUT(\alpha)|\alpha\epsilon\beta\}$.

Определим во введенных выше обозначениях условия корректной типизации для операций композиции.

1. $\tau = \tau_1 \cdot \tau_2$ – терм правильно типизирован, если $IN(type(\tau_2)) \geq OUT(type(\tau_1))$;

$$type(\tau_1 \cdot \tau_2) = \left\{ (\alpha \to \beta) \middle| \begin{matrix} \alpha \in IN\big(type(\tau_1)\big), \beta \in OUT\big(type(\tau_2)\big) \\ \exists\beta_1\exists\beta_2(\beta_1 \leq \beta_2 \wedge (\alpha \to \beta_1) \in type(\tau_1) \\ \wedge (\beta_2 \to \beta) \in type(\tau_2) \end{matrix} \right\}$$

К условию правильной типизации для функций, полученных путем операции последовательной композиции, добавляется дополнительное условие, связанное с использованием функций выбора аргумента $I(m,i), m > 0, i \geq 0$. Как было указано, эти функции полиморфны и выполняют тождественное преобразование между данными на -м входе и выходе функции. Для функции f, имеющей только константные типы, тип функции $I(i,m) \cdot f$ не является однозначным.

Поэтому особым образом определяется тип функций $I(i,m) \cdot \tau$:

$$type(I(i,m) \cdot \tau) = \{x_1 \times x_2 \times \ldots \times x_m \to OUT(\tau)|x_i \in IN(\tau)\}$$

2. $\tau = \tau_1 * \tau_2$ – правильно типизирован, если $IN\big(type(\tau_1)\big) = IN\big(type(\tau_2)\big)$;

$$type(\tau_1 * \tau_2) = \left\{ (\alpha \to \beta) \middle| \begin{matrix} \alpha \in IN\big(type(\tau_1)\big) \wedge \\ \exists\beta_1\exists\beta_2(\alpha \to \beta_1 \in type(\tau_1) \wedge \\ (\alpha \to \beta_2 \in type(\tau_2)) \wedge \beta = \beta_1 \times \beta_2) \end{matrix} \right\}$$

3. $\tau = \tau_1 + \tau_2$ – правильно типизирован, если $type(\tau_1) = type(\tau_2)$.
4. $\tau_1 \to \tau_2$ – правильно типизирован, если $IN\big(type(\tau_1)\big) = IN(type(\tau_2))$;

$$type(\tau_1 \to \tau_2) = \big\{(\alpha \to \beta)\big|(\alpha \to \beta) \in \big(type(\tau_2)\big) \wedge \alpha \to bool \in type(\tau_1)\big\}$$

Определим условия строгой правильной типизации функций, заданных в виде системы функциональных уравнений. Эти условия вытекают из правил типизации функций, полученных применением

операций композиции функций и необходимости равенства типов функциональных переменных F_i и термов $\tau_i, i = \overline{1,n}$.

Добавим к условиям правильной типизации, определенных выше для операций композиции, еще два условия правильной типизации для функциональных уравнений:

1. Типы термов τ_i и определяемых ими функциональных переменных F_i должны совпадать.
2. Все вхождения функциональной переменной F_i в функциональные уравнения должны иметь одинаковый тип.

Заметим, что функция может быть неправильно типизирована (в определенном выше смысле), однако, при её применении к данным конкретного типа она, тем не менее, вычислит результат.

Перейдем к рассмотрению алгоритма типового контроля FPTL-программ.

3.3 Алгоритм типового контроля

По-прежнему исходим из общего задания функций в FPTL в форме системы функциональных уравнений $F_i = \tau_i, i = \overline{1,n}$. Считаем, что типы всех базисных функций заданы.

Перед проведением типового контроля производится ряд преобразований исходной системы функциональных уравнений:

сначала для каждой функциональной переменной F_i выполняются для всех входящих в τ_i функциональных переменных подстановки τ_j вместо каждого вхождения F_j в τ_i. Строятся $F_i^{(1)} = \left[\tau_j / F_j \,\middle|\, j = \overline{1,n}, j \neq i\right]\tau_i, i = \overline{1,n}$. Процесс повторяется, строя $F_i^{(2)}$ путем подстановок вместо всех новых функциональных переменных, входящих в $F_i^{(1)}$(переменных, которые появились в $F_i^{(1)}$ в результате осуществленных на первом шаге подстановок), соответствующих им термов.

Описанный процесс продолжается до того момента, пока для каждого F_i не будут получены $F_i^{(k_i)}$ такие, что в $F_i^{(k_i)}$ не существует вхождений функциональных переменных, для которых ранее не выполнялась подстановка. Полученную систему функциональных уравнений $F_i = F_i^{(k_i)}, i = \overline{1,n}$ назовем *приведенной*. Полученная система функциональных уравнений эквивалентна исходной системе.

Далее каждый терм $F_i^{(k_i)}$ представляется в эквивалентной форме: $F_i^{(k_i)} = \tau_1^{(i)} + \tau_2^{(i)} ... + \tau_{n_i}^{(i)}$, где термы $\tau_j^{(i)}$ не содержат вхождений операции $+$. Для этого используются следующие правила эквивалентности.

1. $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
2. $(A + B) \cdot C = (A \cdot C) + (B \cdot C)$
3. $A * (B + C) = (A * B) + (A * C)$
4. $(A + B) * C = (A * C) + (B * C)$
5. $A \to (B + C) = (A \to B) + (A \to C)$
6. $(A + B) \to C = (A \to C) + (B \to C)$

Определение. Систему функциональных уравнений будем считать семантически корректной, если в представлении, полученном выше, в правой части для каждого терма $F_i^{(k_i)}$ существует по крайней мере один $+$-терм $\tau_j^{(i)}$, не содержащий вхождений функциональных переменных.

Требование семантической корректности на практике не является сильно ограничивающим, поскольку программы, которые не обладают этим свойством, могут выполняться бесконечно, не приводя к получению результата. Очевидно, что в случае, если программа семантически корректна, процесс подстановки, описанный выше, не может длиться бесконечно. В дальнейшем будем рассматривать только семантически корректные системы функциональных уравнений.

После построения приведенной системы функциональных уравнений для семантически корректной программы в каждом уравнении $F_i = \tau_i, i =$

$\overline{1,n}$ имеется хотя бы один терм τ_j, такой что он не содержит функциональных переменных. В случае, если в процессе построения приведенной системы уравнений получилось более одного слагаемого, не содержащего функциональных переменных, проверяется выполнение условий правильной типизации для операции +, т.е. равенство типов слагаемых.

Если это условие не выполняется, считается, что типизация не является правильной. Если же это условие выполняется в каждом уравнении приведенной системы уравнений, то полагается, что тип $F_i, i = \overline{1,n}$ равен типу одного из его слагаемых и для каждого вхождения функциональной переменной $F_j, j = \overline{1,n}$ в соответствующее слагаемое $F_i^{(k_i)}$ определяется его тип. Если после этого все слагаемые в $F_i^{(k_i)}, i = \overline{1,n}$ имеют тип, равный типу F_i, считается, что типизация рассматриваемой системы функциональных уравнений правильна. В противном случае считается, что правильность типизации нарушена.

В реализации языка FPTL на многоядерных компьютерах вместо двух операций композиции функций → и + используется более эффективная с точки зрения параллельного выполнения программ тернарная операция, эквивалентная условному оператору в последовательных языках программирования. Эта операция записывается в форме $\tau_1 \to \tau_2, \tau_3$, где τ_1 – терм-условие. Легко показать, что она эквивалентна представлению $(\tau_1 \to \tau_2) + (\overline{\tau_1} \to \tau_3)$. Поэтому перед применением алгоритма контроля типовой правильности функциональной программы все вхождения в нее термов, полученных путем применения тернарной операции, приводятся к данной эквивалентной форме.

Покажем работу алгоритма типового контроля на примере следующей программы:

$$\begin{cases} F_1 = f_1 \cdot F_1 + f_2 \cdot F_2 + f_3, \\ F_2 = f_4 \cdot F_2 \cdot f_5 + F_1 \cdot f_6 \end{cases}$$

Пусть $type(f_1) = \{int \to int\},\ type(f_2) = \{int \to real\},\ type(f_3) = \{int \to int\}, type(f_4) = \{int \to real\},\ type(f_5) = \{real \to int\}, type(f_6) = \{int \to real\}$.

Производим подстановку вместо F_2 терм τ_2 в правой части уравнения $F_1 = f_1 \cdot F_1 + f_2 \cdot F_2 + f_3$. Получаем $F_1^{(1)} = f_1 \cdot F_1 + f_2 \cdot (f_4 \cdot F_2 \cdot f_5 + F_1 \cdot f_6) + f_3 = f_1 \cdot F_1 + f_2 \cdot f_4 \cdot F_2 \cdot f_5 + f_2 \cdot F_1 \cdot f_6 + f_3$. Все подстановки для F_1 сделаны, заметим, что в $F_1^{(1)}$ имеется терм, не содержащий функциональных переменных. Получаем, что $type(F_1) = type(f_3) = \{int \to int\}$.

В правой части уравнения $F_2 = f_4 \cdot F_2 \cdot f_5 + F_1 \cdot f_6$ нет терма, не содержащего вхождений функциональных переменных. Выполняем подстановки для F_2: $F_2^{(1)} = f_4 \cdot F_2 \cdot f_5 + (f_1 \cdot F_1 + f_2 \cdot F_2 + f_3) \cdot f_6 = f_4 \cdot F_2 \cdot f_5 + f_1 \cdot F_1 \cdot f_6 + f_2 \cdot F_2 \cdot f_6 + f_3 \cdot f_6$. В результате подстановки появился терм $f_3 \cdot f_6$, не содержащий вхождений функциональных переменных. В соответствии с правилами определения типов для операции композиции получаем, что $type(f_3 \cdot f_6) = \{int \to real\}$. Из этого следует, что тип функции F_2 равен $\{int \to real\}$.

Проверяем выполнение условий типовой корректности для всех слагаемых в $F_1^{(1)}$ и $F_2^{(1)}$, содержащих вхождения функциональных переменных, считая их тип определенным ранее. $type(f_1 \cdot F_1) = \{int \to int\}$. При проверке типовой корректности выражения $f_2 \cdot f_4 \cdot F_2 \cdot f_5$ мы встречаем ошибку типизации – тип функции f_2 равен $\{int \to real\}$, как и тип функции f_4 равен $\{int \to real\}$. Делаем вывод о неверной типизации.

Пример можно изменить так, чтобы получить верную типизацию. Если все типы функций $f_i = \{int \to int\}, i \in \{1,2,3,4,5,6\}$, то $type(F_1) = \{int \to int\}, type(F_2) = \{int \to int\}$.

3.4 Заключение

Описанный выше алгоритм типового контроля реализован как дополнительный подключаемый модуль в системе выполнения FPTL-программ. Применение алгоритма статического типового контроля позволило в среднем на 30% сократить время выполнения функциональных программ на языке FPTL, в сравнении с предыдущей реализацией, основанной на проверке типов непосредственно во время выполнения программы.

Далее перейдем непосредственно к описанию реализации системы выполнения FPTL-программ.

4. РЕАЛИЗАЦИЯ ЯЗЫКА FPTL НА МНОГОЯДЕРНЫХ КОМПЬЮТЕРНЫХ СИСТЕМАХ

Язык FPTL претерпел несколько реализаций. Одна из ранних реализаций [3] оказалась недостаточно эффективной, так как она была выполнена на компьютерах с распределенной памятью (многоядерные компьютеры в то время еще не были широко распространены). Кроме того, описанная в разделе 2.1.3 модель параллельного вычисления значений функций, как показала практика, требует больших (по времени) накладных расходов. Это связано с тем, что при осуществлении подстановок деревьев вместо функциональных переменных каждый раз приходилось выявлять части дерева, которые готовы к вычислению (редексы), что требует значительного времени и памяти. Попытка избавиться от этого недостатка сначала привела к идее статического составления приоритетных списков, в которых были бы записаны эти редексы. Одним из вариантов решения данной проблемы была разработка такой модели, в которой отсутствовал бы (или был существенно упрощен) поиск частей деревьев. Исследование этого вопроса привело к описываемому в этой главе решению.

Для повышения эффективности процессов управления параллельным выполнением FPTL-программ были внесены изменения в синтаксис и семантику самого языка, основными из которых является введение тернарной операции условной композиции взамен двух операций $\rightarrow$ и +, которые, главным образом, предназначены для описания условных конструкций. Новая тернарная операция является аналогом условного оператора *if-then-else* в традиционных языках программирования и представляется теперь в FPTL как $(p \longrightarrow f_1, f_2)(x)$, где p – предикат, а f_1 и f_2– функции, значение одной из которых будет использовано в зависимости от того, истинно или ложно значение $p(x)$. Хотя это несколько сужает выразительные возможности языка, тем не менее программист возвращается к принятому в языках программирования

заданию условных конструкции и, что более важно, это позволяет более эффективно их выполнять.

В данной главе будет описана реализация программной системы, созданной для эффективного выполнения функциональных программ на многоядерных компьютерах. В разделе 4.1 будет приведено общее описание архитектуры системы выполнения FPTL-программ. В разделе 4.2 будет дано описание модели внутреннего представления программ, а в разделе 4.3 описан алгоритм работы интерпретатора. Раздел 4.4 полностью посвящен описанию реализации системы управления выполнением параллельных процессов. В разделах 4.5-4.6 дается описание реализации внутреннего представления данных в системе выполнения. Разделы 4.7-4.10 посвящены описанию вспомогательных аспектов реализации, таких как управление памятью, работа с внешними процедурами и описание примененных программных средств.

4.1 Архитектура системы выполнения FPTL-программ на многоядерных компьютерах

В составе системы выполнения функциональных программ можно выделить следующие подсистемы.

1. *Лексический анализатор*. Его назначение – преобразование исходного текста программы в список лексем [29].
2. *Синтаксический анализатор*. Задача этой подсистемы – синтаксический разбор списка лексем. В случае отсутствия в тексте программы синтаксических ошибок строится абстрактное синтаксическое дерево [29].
3. *Подсистема семантической проверки*. Данная подсистема производит проверку семантической корректности функциональной программы: производится поиск функциональных переменных и типов данных, не определенных в тексте программе.

4. *Подсистема поиска рекурсивных определений*. Производит поиск узлов синтаксического дерева, соответствующих рекурсивным функциям.
5. *Генератор сетевого представления* строит сетевое представление функциональной программы, определяет арности всех входных и выходных кортежей данных. Также производит создание базисных функций, соответствующих конструкторам и деструкторам. Кроме того, на этом этапе производится проверка типовой корректности FPTL-программы, алгоритм которой описан в главе 3, и определяются типы кортежей на всех узлах сетевого представления функциональной программы.
6. *Интерпретатор*. Интерпретатор производит вычисление значений функций, представленных в виде сетей.
7. *Подсистема управления параллельным выполнением* осуществляет динамическое планирование порождаемых при работе интерпретатора параллельных процессов (заданий) и назначает их на выполнение соответствующим ресурсам (нитям).
8. *Библиотека базисных функций*. Содержит программную реализацию базисных функций языка FPTL.
9. *Подсистема вызова внешних процедур*. Производит динамическое подключение исполняемых модулей, содержащих реализацию внешних процедур.
10. *Подсистема управления памятью*. Отвечает за выделение и освобождение памяти при выполнении функциональной программы, производит автоматическое управление памятью (сборку мусора).

На уровне программной реализации все данные подсистемы объединены в один исполняемый файл. Архитектура системы выполнения FPTL программ представлена на рис. 4.1.

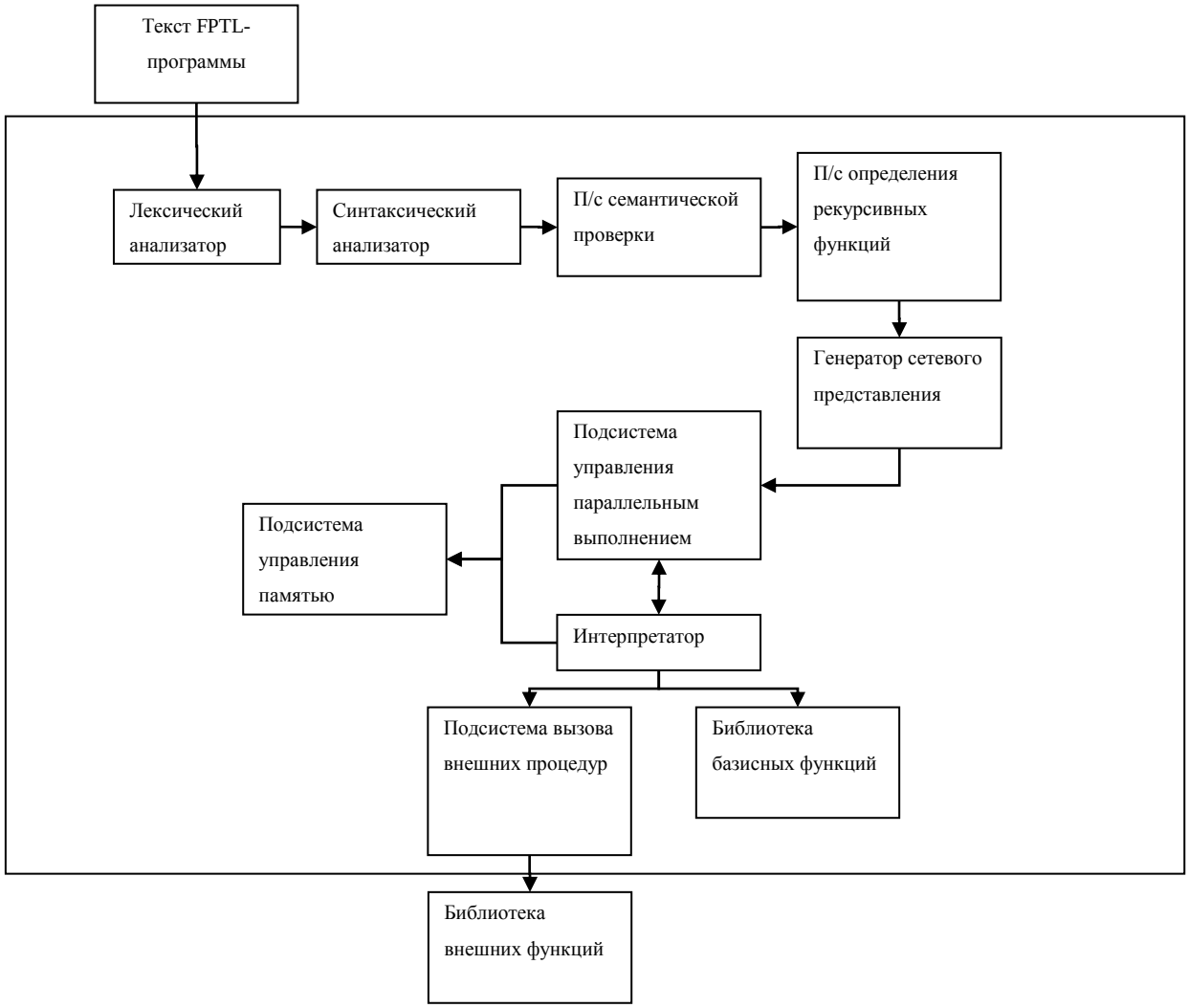

Рис. 4.1. Структура системы выполнения FPTL-программ.

4.2 Сетевое представление функциональных программ

В этом параграфе будет описано сетевое представление функциональной программы, с которым непосредственно работает интерпретатор.

Формально, сеть - это графическое представление функций в программе, которые задаются в виде системы уравнений: $F_i = \tau_i,\ i = \overline{1,n}$. Сетевое представление заданных таким образом функций представляет собой множество сетей, однозначно связанных с термами τ_i и стоящихся индуктивно следующим образом.

1. Если терм τ есть базисная функция или функциональная переменная, то ее представление имеет вид (рис. 4.2):

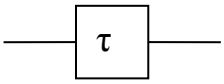

Рис. 4.2. Представление базисной функции или функциональной переменной.

2. Если $\tau = \tau_1 * \tau_2$, то графическое представление τ имеет вид (рис. 4.3):

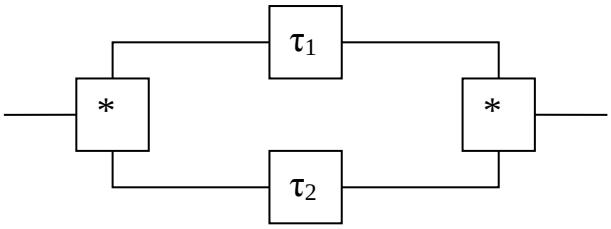

Рис. 4.3. Представление операции параллельной композиции.

3. Если $\tau = \tau_1 \rightarrow \tau_2, \tau_3$, то графическое представление τ имеет вид (рис. 4.4):

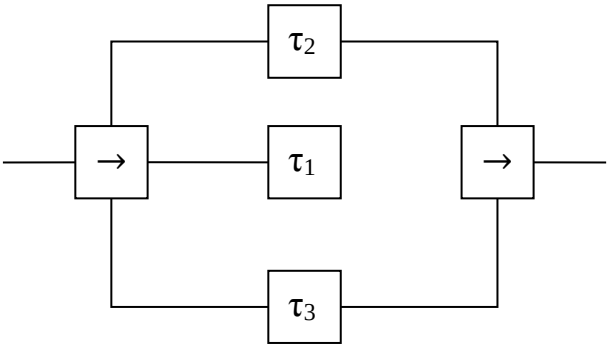

Рис. 4.4. Представление операции условной композиции.

В памяти компьютера сеть представляется многосвязной списковой структурой. Рассмотрим подробно внутреннее представление каждого из узлов сети. Для каждого них введем понятие типа узла – тэга, по которому узлы можно однозначно идентифицировать. Структуру узла будем описывать в следующем виде:

$$T(p_1, p_2, \dots, p_n)$$

где T – имя типа узла, p_i, $i = \overline{1,n}$ – названия полей узла.

1. Узел, содержащий базисную функцию:

$$BF(f, next)$$

Здесь поле f – это адрес базисной или библиотечной функции в справочнике базисных функций, поле $next$ – адрес следующего узла сети.

2. Узел, содержащий функциональную переменную:

$$FV(F, next)$$

Здесь F –имя соответствующей функциональной переменной, $next$ – адрес следующего узла сети.

3. Открывающий узел операции параллельной композиции $*$.

$$FORK(\tau_{top}, \tau_{bottom}, next)$$

Здесь поля τ_{top} и τ_{bottom} представляют адреса верхнего и нижнего элементов схемы (рис. 4.3), справа от открывающего узла $*$, поле $next$ – адрес узла, следующего за закрывающим *-узлом.

4. Закрывающий узел операции параллельной композиции $*$.

$$JOIN(next)$$

Здесь $next$ – адрес следующего узла сети.

5. Узел условной композиции →.

$$COND(\tau_p, \tau_t, \tau_e)$$

Здесь τ_p – указатель на узел сети, соответствующий условию, τ_t, – указатель на узел сети, соответствующий *then*-части условной конструкции, τ_e – указатель на узел, соответствующий *else*-части. Закрывающий узел операции условной композиции является фиктивным и в реализации вместо него участки сети τ_t, и τ_e непосредственно ссылаются на элемент, следующий за закрывающим узлом.

Кроме того, введен дополнительный узел, который всегда располагается в правом конце схемы.

6. Узел возврата из рекурсивной функции.

$$RET$$

Этот узел не содержит полей.

Приведем пример сетевого представления функции, вычисляющей значение факториала числа (рис. 4.5).

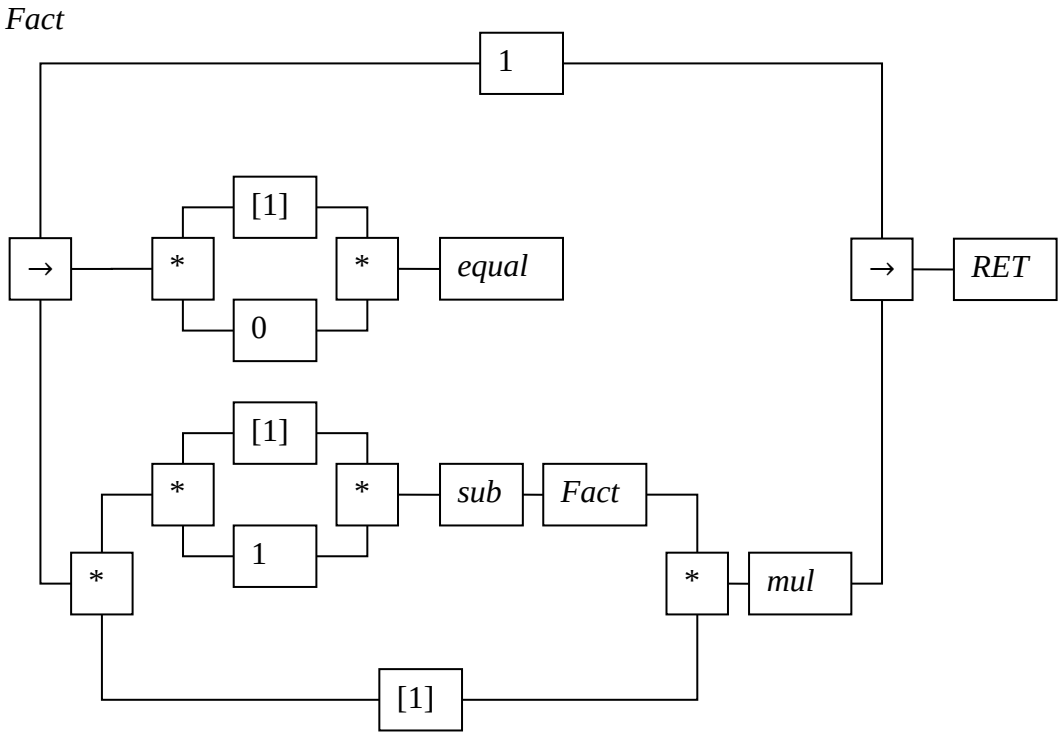

Рис. 4.5. Сетевое представление функциональной программы вычисления факториала числа.

Далее будет описан алгоритм вычисления значений функций, представленных сетями.

4.3 Реализация интерпретатора

Для вычисления значения функций, представленных в виде схем, в системе выполнения FPTL-программ используется интерпретатор.

Для описания алгоритма работы интерпретатора и последующего описания системы управления параллельным выполнением FPTL-программ, введем также понятие *задания* – самостоятельного процесса, который порождается при вычислении значения функции.

Заданием имеет следующую структуру:

$$Task(\tau, RS, DS, ready)$$

Здесь τ – адрес узла сети, RS – стек адресов возврата, DS – стек данных, $ready$ – флаг готовности. Задание представляет собой вычислительный контекст интерпретатора. Рассмотрим алгоритм работы интерпретатора. В описании алгоритма специально опущены некоторые элементы – они будут раскрыты далее.

```
EVALUATE (τ, T):
1  switch τ
2    case of BF(f, next):
3        f(T.DS)
4       EVALUATE(next, T)
5    case of FV(F, next):
6        PUSH(T.RS, next)
7        EVALUATE(τ_F, T)
8    case of FORK(τ_top, τ_bottom, next):
9        T_c := CREATE-TASK(T, τ_top)
10       EVALUATE(τ_bottom, T)
11       WAIT-TASK(T_c)
12       EVALUATE(next, T)
13   case of JOIN(next):
14       return
15   case of COND(τ_p, τ_t, τ_e):
16       EVALUATE(τ_p, T)
17       D := CHECK-COND(T.DS)
18       if D = true
19           EVALUATE(τ_t, T)
20       else EVALUATE (τ_e, T)
21   case of RET():
22       τ_next := POP(T.RS)
23       EVALUATE(τ_next, T)
```

Поясним работу алгоритма. Процедура EVALUATE принимает на вход узел сетевого представления функции τ и задание T. Последующие действия выполняются в зависимости от типа переданного узла (конструкция **switch - case of** реализует семантику сопоставления с

образцом). В строках 2-4 производится вычисление значения базисной или внешней функции f. Входные данные функции f берутся из стека данных DS задания T, выходные данные также помещаются в этот стек, после этого происходит переход к следующему узлу сети (строка 4) посредством рекурсивного вызова процедуры EVALUATE. В строках 5-7 производится подстановка правой части τ_F функционального уравнения, вместо функциональной переменной F. При этом узел, на который должен быть осуществлен дальнейший переход после выполнения подстановки, запоминается на стеке адресов возврата RS задания T. В строках 8-12 производится порождение нового задания. (Подробно этот процесс будет раскрыт далее в разделе 4.4.3.) В строках 15-20 производится интерпретация условной конструкции: сначала вычисляется значение функции предиката, заданного сетью τ_p, затем проверяется условие и делается выбор дальнейшего перехода к узлу сети τ_t или τ_e. Фиктивный узел RET обозначает окончание вычисления значения функции и производит действия для возобновления дальнейшего процесса вычислений: извлекает адрес следующего элемента сети из стека адресов возврата (строки 21-23) и производит переход к его интерпретации.

В приведенном выше псевдокоде были использованы следующие вспомогательные процедуры:

- PUSH (S, v) – добавление значения v в стек S.
- POP (S) – извлечение верхнего элемента из стека S.
- COPY(D, S) – копирование результата вычисления из стека данных S в стек данных D.
- CHECK-COND(S) – проверка результата на стеке. Возвращает ложь, если результат представляет значение $false$ или неопределенность ω, истину в противном случае.

Описание процедур CREATE-TASK и WAIT-TASK будет приведено в разделе 4.4.3.

4.4 Реализация управления параллельными вычислениями

В следующих параграфах будет описана реализация подсистемы управления процессом параллельного выполнения FPTL-программ.

4.4.1 Общие моменты

В основе реализации системы управления параллельными процессами лежат средства нитевого программирования, причем в любой момент выполнения FPTL-программы используется фиксированное число нитей. Каждая нить, далее называемая *рабочей нитью* (РН), выполняет задание (см. раздел 4.3) – процесс вычисления значения функции, заданной сетью. При этом во время интерпретации *-узлов схемы, могут порождаться другие задания. Им будет соответствовать один из параллельных процессов вычислений, порожденных в программе операцией *. По сути, соединенные операцией * два терма эквивалентны порождению двух процессов вычислений, первый из которых будет продолжен выполняться в контексте текущего задания, а для второго порождается новое задание. Порожденные задания добавляются в специальную *рабочую очередь* (РО), существующую отдельно для каждой рабочей нити. Кроме того, в системе выполнения FPTL-программ имеется *планировщик заданий*, который производит поиск новых заданий для рабочей нити, если она простаивает.

В качестве обоснования использования именно такого подхода (использования фиксированного количества нитей) при реализации параллельного выполнения функциональных программ приведем следующие аргументы.

1. Время создания нити может быть достаточно велико. В действительности, если не использовать концепцию задания, а перейти непосредственно к порождению нитей, то такой подход

приведет к масштабному увеличению накладных расходов, связанных с созданием нитей.

2. Большое количество одновременно работающих в системе нитей приводит к резкому увеличению доли времени работы планировщика ядра операционной системы относительно общего времени выполнения программы.

Рассмотрим детально реализацию каждого из перечисленных выше компонентов системы: организацию работы рабочих нитей, рабочих очередей, взаимодействие между нитями и очередями и алгоритм работы планировщика.

4.4.2 Рабочие нити

Каждая рабочая нить представляет собой независимый процесс выполнения процедуры интерпретатора, описанного выше в разделе 4.3, и алгоритма планирования. Для сохранения состояния интерпретатора (сохранения значений локальных переменных процедуры EVALUATE) используется собственный стек данных рабочей нити [12].

Количество рабочих нитей задает программист перед выполнением функциональной программы и в процессе выполнения оно не изменяется. Для того чтобы полностью задействовать имеющиеся в компьютере процессоры (ядра), число РН должно быть не меньше, чем количество физических ядер в компьютере.

Поскольку параллельная работа с данными в едином адресном пространстве в общем случае требует использования механизмов синхронизации, целесообразно разделить структуры данных, используемые всеми рабочими нитями на две группы.

1. *Неизменяемые данные всех нитей*. К ним относятся сетевые представления функциональных уравнений, библиотека базисных и внешних функций, реализация конструкторов и деструкторов. Эти структуры данных используются интерпретатором и создаются в

единственном экземпляре перед запуском функциональной программы, не изменяются в процессе ее выполнения. Доступ к этим данным предоставляется в режиме только для чтения. Важно отметить, что такой порядок работы с данными при параллельной работе нитей не может привести к состоянию гонки, поэтому использование механизмов синхронизации доступа к памяти в этом случае не требуется.

2. *Локальные данные нити*. К этим данным относятся рабочие очереди нитей и порожденные интерпретатором задания. Доступ к ним предоставляется как для чтения, так и для записи. При этом если при обращении к данным имеется возможность гарантировать, что в любой момент времени работа с ними ведется из одной единственной нити, использование элементов синхронизации также не требуется. В противном случае для синхронизации доступа требуется применение семафоров (мьютексов), атомарных операций или барьеров чтения-записи [30; 31].

Перед началом выполнения функциональной программы в РО одной из нитей добавляется *первоначальное* задание. В процессе выполнения функциональной программы, состояние рабочих нитей может изменяться следующим образом.

1. РН активна (выполняется), если есть задание хотя бы в одной очереди всех нитей.
2. Прерывание нити и поиск нового задания происходит в двух следующих случаях:
   a) РН полностью выполнила назначенное ей задание. Этому состоянию соответствует возврат из процедуры EVALUATE;
   b) при достижении закрывающей операции * и только в том случае, если другой линейный участок этой операции, исходящий от соответствующей открывающей операции *, еще выполняется

другой нитью. Этому состоянию соответствует строка 11 описанной выше процедуры EVALUATE.

3. Как только будет выполнено первоначальное задание, а это свидетельствует о том, что были выполнены и все рекурсивно порожденные задания, всем РН подается команда остановки.

Опишем далее принципы организации рабочих очередей заданий.

4.4.3 Очереди заданий

Как было отмечено в разделе 4.4.1, каждая рабочая нить имеет свою очередь порожденных заданий, называемую рабочей очередью (РО). РН берет новое задание для выполнения из конца своей очереди по принципу LIFO. Однако если эта очередь пуста, то нить вынуждена искать новое задание в очереди другой нити, которое извлекается из очереди по принципу FIFO. Это, во-первых, позволяет осуществлять миграцию более сложных с вычислительной точки зрения заданий на другие нити и, во-вторых, локализовать данные при выполнении заданий, взятых из собственной очереди.

С учетом вышесказанного, приведем алгоритм работы процедур CREATE-TASK и WAIT-TASK, которые были использованы раньше в процедуре EVALUATE.

CREATE-TASK (T, τ):
1 T_{child} := $Task(\tau, [], T.DS, [], false)$
2 WQ-PUSH (PO_t, T_{child})
3 **return** T_{child}

Процедура CREATE-TASK создает новое здание T_{child}, которое содержит указатель на узел схемы τ и добавляет его в очередь PO_t рабочей нити t.

WAIT-TASK (T):
1 **while** $T.ready \neq$ **true**
2 SCHEDULE()

Процедура WAIT-TASK производит ожидание готовности порожденного в процедуре CREATE-TASK задания. Если задание не готово, то происходит вызов алгоритма планирования (строка 2) для поиска другой работы. Таким образом, удается избежать простаивания нити. Процедура SCHEDULE, реализующая алгоритм планирования будет описана далее в разделе 4.4.4.

Заметим, что можно было бы использовать общую очередь для заданий, сохраняя описанную логику управления их выполнением. Однако обращение к общей очереди множества нитей требует синхронизации доступа к ней, которая является достаточно затратной по времени операцией. Также следует отметить, что принципы использования раздельных очередей были описаны в [32; 33; 40], а также используется в системах параллельного программирования [20; 34; 35; 45].

Для организации доступа к очереди используются следующие процедуры:

WQ-PUSH(Q, T) – добавляет задание T в конец РО Q.

WQ-TAKE() – извлекает задание из конца РО Q (извлечение по принципу LIFO). Если очередь пуста, возвращается значение nil.

WQ-STEAL(Q) – извлекает задание из начала РО Q. Если очередь пуста, возвращается значение nil.

Схематично, работа описанных процедур представлена на рис. 4.6.

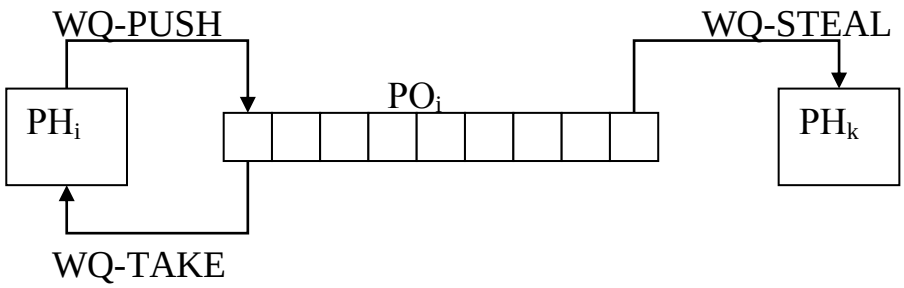

Рис 4.6. Организация взаимодействия нитей с очередью заданий.

Отметим особенности реализации вышеописанных процедур. Рабочая нить PH_i добавляет и извлекает задания из конца очереди, в то время как другие рабочие нити PH_k, $k \neq i$ могут только извлекать задания из начала очереди. Такая организация работы с РО позволяет свести к минимуму использование элементов синхронизации параллельного доступа к памяти. При данном подходе состояние гонки возникает лишь в двух конкретных случаях:

1. Очередь содержит одно единственное задание. Две нити пытаются одновременно извлечь его с использованием методов WQ-TAKE и WQ-STEAL.
2. Две нити одновременно пытаются извлечь задание с помощью метода WQ-STEAL.

Два вышеприведенных случая приводят к ситуации, когда одно и то же задание будет выполнено различными нитями. Такое поведение делает алгоритм управления параллельным выполнением недетерминированным, что недопустимо для его эффективной работы. Для устранения состояния гонки в этих двух случаях используется синхронизация с использованием мьютекса.

Следует отметить используемый принцип, по которому регулируется размер РО. Изначально РО имеет фиксированный размер в 32 элемента. Если при добавлении очередного задания, в очереди не оказывается свободного места, происходит полная блокировка доступа к очереди для всех РН (производится с помощью мьютекса), создается новая РО вдвое большего размера, в которую копируются все задачи из изначальной очереди, после этого доступ нитей возобновляется уже к новой РО.

Опишем реализацию процедур работы с РО. Структура очереди представляется следующим образом:

$$Queue(head, tail, size, D, M)$$

Здесь $head$ – позиция чтения и записи с конца очереди, $tail$ – позиция чтения из начала очереди, $size$ – текущий размер очереди, D – массив для хранения элементов очереди, M – мьютекс для синхронизации доступа из разных нитей.

Процедура добавления в конец очереди выглядит следующим образом:

```
WQ-PUSH(Q, T):
1   tail := Q.tail
2   if tail < Q.size
3       Q.D[tail] := T
4       Q.tail := tail + 1
5   else LOCK(Q.M)
6       Увеличить размер массива D в 2 раза.
7       Q.D[tail] := T
8       Q.tail := Q.tail + 1
9       UNLOCK(Q.M)
```

Сначала производится проверка наличия свободного места в очереди (строки 1-2). В случае если очередь не полна, производится добавление задания T в конец очереди (строки 3-4). В противном случае, производится увеличение размера массива D, образующего очередь, в 2 раза и добавление задания T в конец увеличенной очереди.

Рассмотрим псевдокод процедуры взятия задания из конца очереди.

```
WQ-TAKE(Q):
1   tail := Q.tail
2   if tail ≤ Q.head
3       return nil
4   else tail := tail − 1
5       Q.tail := tail
6       if tail ≥ Q.head
7           return Q.D[tail]
8       else T := nil
9           LOCK(Q.M)
10          if tail ≥ Q.head
11              T := Q.D[tail]
12          else tail := tail + 1
13              Q.tail := tail
```

14 UNLOCK($Q.M$)
15 **return** T

Стоки 1-3 соответствуют ситуации, когда очередь пуста. Иначе, если очередь содержит более одного задания, производится его извлечение (строки 5-7). Если в очереди содержится ровно одно задание, производится получение к ней эксклюзивного доступа посредством блокировки мьютекса M, после этого попытка извлечения задания из очереди повторяется (стоки 10-13). Процедуры LOCK(M) и UNLOCK(M) используются для захвата и освобождения мьютекса M соответственно [12].

Псевдокод процедуры получения задания из начала очереди выглядит следующим образом.

WQ-STEAL (Q):
1 $T := nil$
2 LOCK($Q.M$)
3 $head := Q.head$
4 $Q.head := Q.head + 1$
5 **if** $head < Q.tail$
6 $T := Q.D[head]$
7 **else** $Q.head := head$
8 UNLOCK($Q.M$)
9 **return** T

В начале приведенного псевдокода производится блокировка мьютекса M для обеспечения эксклюзивного доступа к началу очереди. После этого, если очередь не пуста, производится извлечение задания из ее начала.

4.4.4 Алгоритм работы планировщика

Работу планировщика по выбору новых заданий можно описать следующей процедурой, упомянутой ранее в процедуре WAIT-TASK.

SCHEDULE():
1 $T :=$ WS-TAKE(Q_t)
2 **if** $T \neq nil$

```
3       EVALUATE(T.τ, T)
4       T.ready := true
5   else for each s in Threads \ {t} do
6       T' := WQ-STEAL(Q_s)
7       if T' ≠ nil
8           EVALUATE(T'.τ, T')
9           T' := true
10          return
```

Здесь Q_t – рабочая очередь нити t, выполняющей процедуру SCHEDULE, $Threads$ – множество всех рабочих нитей.

Алгоритм работает следующим образом. Сначала нить t, выполняющая процедуру SCHEDULE, пытается получить задание из своей РО Q_t (рис. 4.7а). Если РО Q_t пуста, то нить последовательно просматривает рабочие очереди Q_s других нитей s, отличных от нити t. (рис. 4.7б). Если в какой-либо из очередей других нитей, предположим Q_k, имеется задание, то нить t берет его на выполнение и затем выходит из процедуры SCHEDULE. В случае, если очереди всех рабочих нитей пусты, нить t отдает свой квант времени системе. Это позволяет избежать полной загрузки ядер вычислительной системы в случае выполнения чисто последовательных программ, при которых новые задачи не порождаются.

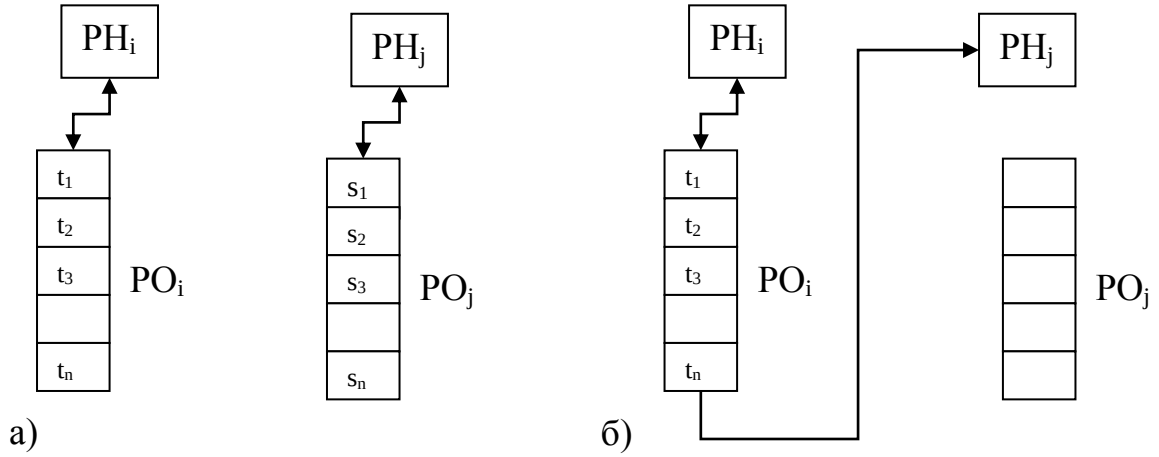

Рис. 4.7. Действия рабочих нитей PH_i и PH_j, $(i \neq j)$ при: а) наличии заданий в своих очередях, б) пустой рабочей очереди у PH_j.

Полный алгоритм действий РН можно представить следующим образом.

THREAD-PROC ():
1 T ≔ «Первоначальное задание»
2 **while** $T.ready = false$
3 SCHEDULE()

Здесь нить в цикле выполняет процедуру SCHEDULE (строки 1-2), пока не будет достигнуто условие останова – готовность первоначального задания.

4.4.5 Выбор сложности задания

Описанный выше алгоритм параллельного вычисления значений функций имеет один существенный недостаток: каждая операция параллельной композиции приводит к порождению нового задания. При этом не учитывается вычислительная сложность части сети, представляющей это задание. В реальных условиях это неизбежно приведет к ситуации, когда накладные расходы на планирование параллельного выполнения задания окажутся выше, чем время выполнения этого задания. Такой случай возникает, например, если левая и правая часть оператора * содержит только базисные функции (рис. 4.8а), и встречается довольно часто. Одним из способов решения данной проблемы является подход, при котором для параллельного выполнения назначаются только участки сетей, содержащие две и более рекурсивные функции по каждому из путей.

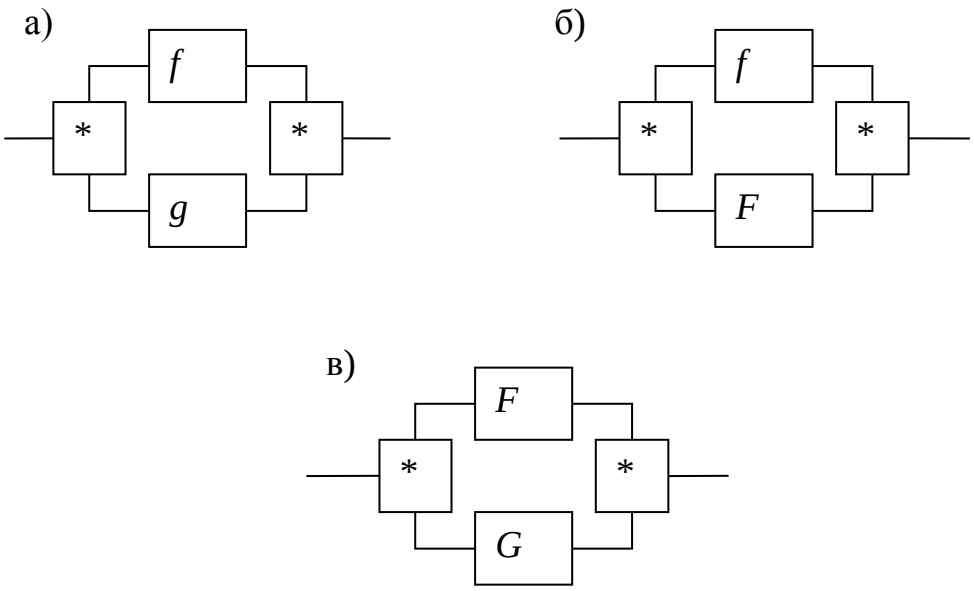

Рис. 4.8. Возможные варианты сетей с оператором *. Здесь f и g – базисные функции; F и G – функциональные переменные, правые части которых содержат рекурсию.

Для реализации данного подхода, в каждом открывающем узле операции * будем хранить еще дополнительный флаг, сигнализирующий о том, требуется ли параллельное вычисление. Значение этого флага определяется после дополнительного анализа сетевого представления функциональной программы перед ее выполнением. Внутреннее представление модифицированного *-узла сети имеет следующий вид:

$$FORK(\tau_{top}, \tau_{bottom}, next, parrallel)$$

Здесь поле $parrallel$ принимает значение «истина», если требуется параллельное вычисление двух участков сети. Часть процедуры EVALUATE, отвечающая за работу с модифицированным узлом $FORK$, реализуется следующим образом:

EVALUATE(τ, T):

…

1 **case of** $FORK(\tau_{top}, \tau_{bottom}, next, parrallel)$:

2 **if** $parallel = true$

3 T_{child} := CREATE-TASK(T, τ_{top})

4 EVALUATE(τ_{bottom})

5 WAIT-TASK(T_{child})

6 **else** EVALUATE(τ_{top})

7 EVALUATE(τ_{bottom},)

8 EVALUATE($next$)

…

В этом псевдокоде для каждого открывающего *-узла проверяется флаг $parrallel$ (строка 2) и в зависимости от его состояния либо происходит порождение нового задания (строки 3-5), либо обе части сети обрабатываются последовательно в контексте одного задания (строки 6-8).

Для примера, на рисунке 4.8 ситуациям а) и б) будет соответствовать последовательный случай, в то время, как в ситуации в) для вычисление значений функций, задаваемых уравнениями F и G, будет порождаться новое задание.

Отметим, что способность любой системы управления параллельными процессами динамически регулировать сложность (зернистость) порождаемых процессов является чрезвычайно важной проблемой для достижения минимального времени выполнения параллельных программ [36].

4.5 Внутреннее представление данных

Данные в языке FPTL, как было сказано выше, можно разделить на две категории: встроенные типы данных и задаваемые пользователем. К встроенным типам данных относятся типы int, real, bool, string, Array. К типам, задаваемым пользователям, относятся абстрактные типы данных (АТД). Для дальнейшего описания реализации внутреннего представления и работы с данными в языке FPTL уместно ввести разделение типов данных на две группы.

- *Элементарные* типы данных. Элементарным типом данных будем называть такой тип данных, который может быть представлен одной неделимой областью памяти. К этим типам данных относятся int, real и bool.
- *Составные* типы данных. К составным типам данных в FPTL относятся string, Array и АТД. Эти типы данных не могут быть представлены, как неделимые области памяти и их реализация требует учитывать некоторые аспекты работы интерпретатора (например, систему сборки мусора). Составные типы данных реализуются через промежуточную структуру данных – дескриптор составного типа данных. Всего имеются 3 типа дескрипторов: дескриптор строки, дескриптор массива и дескриптор АТД. Приведем описание каждого из них.

  1. Дескриптор строкового типа:

$$String(S, P_{start}, P_{end})$$

Здесь поле S – указатель на массив символов строки, P_{start} – индекс первого символа строки в массиве S, P_{end} – индекс последнего символа строки в массиве S. Такое представление позволяет использовать один общий буфер для нескольких разных строк, например, в случае если одна строка является подстрокой другой.

2. Дескриптор массива:

$$Array(size, E)$$

Здесь $size$ – количество элементов массива, E – указатель на буфер, в котором содержатся элементы данных массива.

3. Дескриптор АТД:

$$ADT(ctor, D)$$

Здесь $ctor$ – имя конструктора, с помощью которого АТД был создан, D - указатель на буфер элементов данных, к которым конструктор был применен.

Для удобства реализации интерпретатора, внутреннее представление всех типов данных в языке FPTL упаковывается в одну структуру, размер которой равен максимальному из размеров всех элементарных типов данных и дескрипторов составных типов. Такой способ упаковки данных позволяет избежать работы с областями памяти разного размера, упрощая реализацию интерпретатора. Помимо этого, в целях упрощения отладки программ структура элемента данных также содержит информацию о его типе.

4.6 Представление кортежей данных

В языке FPTL все базисные функции производят операции над кортежами данных, реализация внутреннего представления которых существенно влияет на уменьшение времени выполнения операций над ними. Двумя основными операциями, выполняемыми над кортежами данных, являются операция конкатенации двух кортежей и операция выбора одного элемента из кортежа. Были исследованы следующие варианты реализации внутреннего представления кортежей.

1. *Реализация кортежей на основе массивов фиксированной длины*. При данном подходе элементы кортежа данных упаковываются в массив, длина которого соответствует размеру кортежа. Операция выбора элемента из кортежа при этом реализуется тривиально - это выбор элемента из массива. При конкатенации данные из двух исходных кортежей копируются в новый массив с длиной равной сумме длин исходных массивов. Так как создание результирующего массива требует выделения новой области памяти, каждая операция конкатенации кортежей требует обращения к системе управления памятью. Это является главным недостатком данного способа реализации внутреннего представления кортежей.
2. *Реализация кортежей на основе односвязных списков.* В данной реализации кортеж представляет собой односвязный список

элементов данных: каждый элемент данных хранит в себе ссылку на следующий элемент. Конечный элемент ссылается на самого себя. Для реализации конкатенации двух кортежей требуется пройтись по списку от головного до конечного элемента, и присвоить ссылке на следующий за последним элементом левого кортежа адрес первого элемента правого кортежа. Чтобы избежать поиска последнего элемента левого кортежа, можно в дополнение сохранять адрес последнего элемента кортежа. Преимуществом данной реализации является простая реализация конкатенации кортежей. Недостатками являются: необходимость последовательно просмотра списка для реализации операции извлечения элемента кортежа и потребность в сохранении дополнительного указателя на следующий элемент кортежа. Кроме того, создание каждого нового элемента кортежа требует обращения к системе управления памятью.

3. *Реализация кортежей на основе динамически расширяемых массивов.* Данная реализация концептуально напоминает работу с локальными переменными в стеке, применяемую в императивных языках программирования. В языке FPTL в качестве локальных переменных выступают входные кортежи данных. Именно такое представление кортежей и было использовано в текущей реализации.

Схема работы с данными при стековом представлении кортежей описывается следующими положениями.

- Изначально выделяется массив небольшого размера для хранения элементов кортежей. Работа с массивом ведется по принципу стека.
- Для каждого элемента или участка сетевого представления функциональной программы τ известны арности $A_{in}(\tau)$ входного кортежа $IN(\tau)$ и $A_{out}(\tau)$ выходного кортежа $OUT(\tau)$. Подсчет арностей производится на этапе создания сетевого представления функции.

- Пусть базисная функция f, принимает на вход кортеж данных e арности n, а результатом ее вычислений является кортеж данных x арности k. Тогда при вычислении значений функции f входным кортежем данных являются n верхних элементов стека. При вычислении значения базисной функции кортеж входных данных не извлекается из стека, а делается его копия. После вычисления значения базисной функции k элементов результирующего кортежа помещаются в стек. (Рис. 4.9)

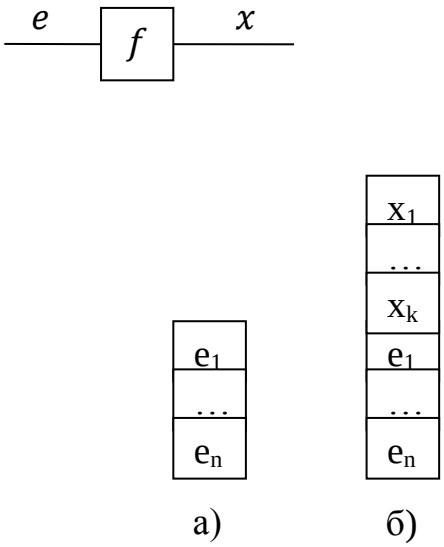

Рис. 4.9. Состояния стека данных а) – до и б) – после вычисления значений базисной функции f.

- При продвижении по сетевому представлению функции слева направо после получения результирующего кортежа w арности t ($w = g(x)$, где g - базисная функция), производится свертывание стека, т.к. выходной кортеж левой части схемы становится больше не нужен. При этом производится перемещение элементов кортежа w на место элементов кортежа x и размер стека уменьшается на k элементов. Эта операция позволяет исключить разрастание стека: при такой организации работы стек не будет содержать данных,

использование которых не требуется для дальнейших вычислений. (Рис. 4.10)

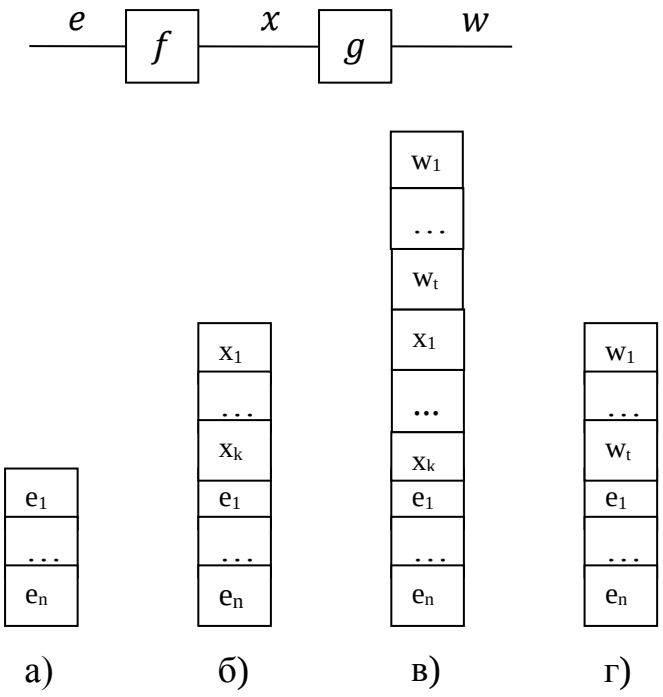

Рис. 4.10. Состояния стека данных: а) – изначальное, б) – после вычисления значения баз. функции f, в) – после вычисления значения баз. функции g, г) – после свертки стека.

- Если при обработке *-узла сети порождается новое задание, то оно содержит свой собственный стек данных, в который изначально копируется входной кортеж, а после выполнения задания извлекается результирующий кортеж и добавляется в стек исходного задания.
- Если при обработке *-узла новое задание не порождается, то сначала вычисляется кортеж, соответствующий результату интерпретации нижней части *-узла. Этот кортеж извлекается из стека и запоминается в локальной переменной интерпретатора. Затем вычисляется и помещается в стек результирующий кортеж,

полученный после выполнения верхней части *-узла. После этого в стек добавляется сохраненный ранее верхний кортеж.

- Если при добавлении очередного элемента выясняется, что в стеке больше нет места, происходит его увеличение: выделяется новая область памяти в 2 раза большего размера, в нее копируется элементы старого стека, затем старая область памяти возвращается системе управления памятью.

Организация работы с кортежами данных с помощью стека позволяет добиться следующих преимуществ.

1. Снижается количество запросов к системе управления памятью, поскольку обращения к ней производятся только при заполнении стека.
2. Сохраняется локальность данных: память выделяется большой непрерывной областью и элементы данных кортежей располагаются «поблизости». Такое размещение данных хорошо сочетается с принципами организации работы кэш-памяти процессора.

4.7 Вычисление значений конструкторов и деструкторов

Перед этапом создания сетевого представления функциональной программы для каждого конструктора и деструктора создаются специальные процедуры-переходники. Алгоритм их работы может быть представлен следующим образом:

EVAL-CONSTRUCTOR (c, DS):

1 D := ALLOCATE($A_{in}(c)$)

2 COPY(d, DS, $A_{in}(c)$)

3 v := $ADT(c, D)$

4 PUSH(DS, v)

EVAL-DESTRUCTOR(c, DS):

1 v := GET(DS)

2 **if** $v.name = c$

3 **for each** e_i **in** $v.D$ **do**

4 PUSH(DS, e_i)

5 **else** PUSH(DS, ω)

Здесь DS – стек данных задания, c - имя конструктора, процедура ALLOCATE($A_{in}(c)$) выделяет буфер размера, равного арности входного кортежа конструктора c, процедура COPY производит копирование $A_{in}(c)$ элементов из стека данных в буфер, GET(DS) получает копию верхнего элемента стека. Адреса созданных процедур-переходников заносятся в справочник базисных функций и используются при вычислениях (в процедуре EVALUATE).

4.8 Реализация вызова внешних функций

Вызов внешних функций (*foreign functions*) также реализован через специальные процедуры-переходники. Для каждой внешней функции создается процедура-переходник, которая извлекает входные данные из стека данных задания, преобразовывает их в формат входных параметров процедуры, производит вызов процедуры, преобразовывает ее выходные параметры и помещает их обратно в стек данных задания. В описании узлов сетевого представления под адресом внешней функции понимается адрес ее процедуры-переходника. Все используемые в программе библиотеки внешних функций загружаются в память непосредственно перед выполнением FPTL-программы.

4.9 Управление памятью и сборка мусора

Очевидно, что память, используемая для выполнения функциональной программы, не может быть полностью распределена статически перед ее выполнением. Более того, сложный рекурсивный

характер функциональных программ требует, как правило, больших и заранее неопределенных объемов памяти для их выполнения. В процессе выполнения функциональной программы возникает потребность динамически выделять и освобождать области памяти. Эта память используется для хранения порожденных заданий, стека для хранения элементов кортежей данных, хранения строк, массивов, элементов АТД и т.д. В реализациях большинства современных программных систем динамическая память выделяется из области, называемой *кучей*. Язык программирования или стандартная библиотека представляет примитивный интерфейс для работы с кучей. Обычно он реализуется в двух процедурах: процедура allocate(n) выделяет объем памяти заданного размера n из кучи, процедура free(p) освобождает выделенную по адресу p память и возвращает ее обратно в кучу. Память, используемая для содержания кучи, выделяется с помощью средств операционной системы. Работа с памятью в многоядерных системах имеет определенные особенности. Обращение к куче из нескольких нитей требует использования примитивов синхронизации. Большая интенсивность запросов к куче приводит к увеличению накладных расходов на обеспечение синхронизации. Это становится «бутылочным горлышком» всей системы выполнения и влечет за собой катастрофические потери пропускной способности (накладные расходы составляют порядка 35% общего времени выполнения программы [31; 32]). Другими словами, стандартные средства для работы с динамической памятью не обладают требуемой масштабируемостью. Для решения этой проблемы в среде выполнения FPTL-программ используется механизм локального по отношению к нити выделения памяти (*thread-local allocation*). Этот механизм заключается в том, что для каждой нити создается своя собственная куча, обращения к которой для выделения памяти не требуют

использования примитивов синхронизации. Это позволяет полностью исключить упомянутые выше накладные расходы.

Другой задачей системы управления памятью является контроль над временем жизни выделенной памяти. Поскольку язык FPTL, как и большинство высокоуровневых функциональных языков программирования, не предоставляет средства для ручного управления памятью, то эту обязанность вынуждена взять на себя система выполнения. Существует несколько подходов к автоматическому управлению памятью. Рассмотрим некоторые из них.

Подсчет ссылок. В реализации данного подхода [39], вместе с выделенной памятью хранится счетчик указателей на нее. При создании нового указателя, ссылающегося на заданную область памяти, счетчик увеличивается. Если указатель уничтожается, счетчик уменьшается. Если значение счетчика становится равным нулю, память освобождается. Алгоритм в самой простой его форме не работает для циклических ссылок. Но главным его недостатком является необходимость использования примитивов синхронизации при обращении к счетчику ссылок из разных нитей. Возникающие при этом накладные расходы серьезно уменьшают общую производительность системы.

Отслеживающая сборка мусора. Мусором называются выделенные области памяти, на которые в данный момент выполнения программы не ведет ни одна ссылка. Алгоритм заключается в следующем [42].

1. Если при очередном выделении памяти обнаруживается, что свободная память исчерпалась, нить входит в состояние сборки мусора.
2. Нить, вошедшая в состояние сборки мусора, подает сигнал на остановку всех остальных выполняющихся нитей, дожидается их остановки и начинает непосредственно процесс сборки мусора.

3. В ходе этого процесса, происходит сканирование областей памяти, которые используются нитью. После этого вся выделенная память делится на две категории: *достижимые* области памяти и *мусор*.
4. Вся выделенная память, помеченная как мусор, возвращается в кучу. После этого выполнение всех нитей возобновляется.

Главным недостатком отслеживающей сборки мусора является необходимость остановки выполнения программы, так называемая *пауза сборки мусора* (*garbage collector pause*). Эта пауза сильно увеличивает время выполнения параллельных программ, активно использующих динамическую память (например, производящих операции над списковыми структурами данных). Для грубой оценки максимального ускорения, которого можно добиться при параллельном выполнении программ, использующей сборку мусора, можно воспользоваться законом Амдала [41].

Для уменьшения времени паузы сборки мусора прибегают к следующим техникам:

1. Распараллеливание процесса сборки мусора [42]. В этом случае процесс поиска достижимых областей памяти производится параллельно. Также может быть распараллелено возвращение мусора в кучу. Это позволяет некоторым образом сократить время паузы сборки мусора. Для сборки мусора в среде выполнения FPTL-программ используется именно такой вариант оптимизации.
2. Использование фоновой сборки мусора, работающей без остановки выполнения программы. Существующие алгоритмы довольно сложны в реализации и требуют поддержки со стороны аппаратного обеспечения или изменение ядра операционной системы. Один из вариантов реализации фоновой сборки мусора используется в коммерческой виртуальной машине JVM Azul [43].

Ко всему вышесказанному, следует отметить, что сборка мусора не требуется для данных элементарных типов: память для их хранения выделяется и освобождается вместе со стеком данных задания.

4.10 Языки и методы реализации

В таб. 4.1 дано описание средств разработки, использованных для реализации каждой подсистемы, входящей в состав системы параллельного выполнения FPTL-программ.

Подсистема	Реализация	Язык реализации	Используемые библиотеки
Лексический анализатор	GNU Flex	C	
Синтаксический анализатор	GNU Bison	C	
Семантический анализатор	Собственная.	C++	
Подсистема поиска рекурсивных уравнений	Собственная.	C++	
Интерпретатор	Собственная.	C++	
Рабочие очереди	Собственная.	C++	boost/atomic
Подсистема управления параллельным выполнением	Собственная.	C++	boost/thread
Сборка мусора	Boehm GC	C	
Библиотека базисных функций	Собственная.	C++	
Подсистема вызова внешних процедур	Собственная.	C	libffi

Таб. 4.1. Программные средства, использованные для реализации компонентов системы выполнения FPTL-программ.

Система выполнения FPTL-программ реализована на языке C++ за исключением подсистем, выполненных на основе сторонних библиотек. Был выбран именно этот язык программирования по ряду следующих причин:

- программы написанные на языке C++ транслируются непосредственно в машинный код для конкретной процессорной архитектуры. Это заметно увеличивает быстродействие интерпретатора, по сравнению с возможным вариантом его реализации на языках Java или C#, программы на которых транслируются в промежуточное представление и выполняются в дальнейшем под управлением виртуальных машин;
- в языке C++ имеются все необходимые средства и библиотеки, которые требуются для реализации низкоуровневой работы с нитями и обеспечения синхронизации доступа к памяти (библиотеки boost/thread и boost/atomic);
- средства компиляции и компоновки, а также перечисленные библиотеки языка C++ являются кроссплатформенными и присутствуют практически на всех широко используемых операционных системах и процессорных архитектурах.

Недостатками языка C++, использованного для реализации системы выполнения FPTL-программ являются:

- отсутствие встроенных средств для автоматического управления памятью. Для устранения этого недостатка в качестве реализации сборщика мусора была использована модифицированная библиотека Boehm GC [44]. Модификация заключалась в переходе от сканирования стека нити при поиске достижимых объектов к сканированию непосредственно стеков данных заданий. Данная модификация позволила избавиться от «консервативного» [44] характера работы сборщика мусора и сделать процесс сборки мусора более эффективным.
- Отсутствие встроенного интерфейса для работы с внешними программными модулями, который необходим для реализации механизма вызова внешних функций в языке FPTL. Для устранения

этого недостатка была использована сторонняя библиотека libffi, позволяющая реализовывать внешние программные модули, подключаемые во время выполнения программы.

Отметим, что упомянутые языки программирования, такие как Java и C#, лишены перечисленных недостатков. Однако отсутствие в них поддержки компиляции программы в машинный код и прямой работы с памятью может привести к существенным накладным расходам, что делает их использование для разработки подобного рода систем весьма рискованным.

Реализация системы выполнения FPTL-программ является кроссплатформенной. Имеются варианты сборок под операционные системы Windows, Linux и OS X.

4.11 Заключение

В настоящей главе было приведено описание системы выполнения FPTL-программ. Перечислим наиболее важные результаты, которые при этом получены.

- Разработана многокомпонентная архитектура системы и описана организация взаимодействия ее подсистем.
- Разработано внутреннее представление функциональных программ в виде сетей, обеспечивающее эффективное выполнение FPTL-программ на многоядерных компьютерах, и представление всех структур данных, используемых в языке.
- Разработаны алгоритмы:
    - интерпретатора, реализующего вычисления значений функций, представленных в виде сетей и производящего порождение новых параллельных процессов (заданий);
    - планировщика параллельных процессов, выполняющего распределение заданий по рабочим нитям;

  - очередей, используемых для эффективного хранения порожденных интерпретатором заданий;
  - управления сложностью (зернистостью) порожденных параллельных процессов (заданий).
- Рассмотрены вспомогательные аспекты организации работы системы выполнения FPTL-программ, а именно: автоматическое управление памятью, представление данных и вызовы внешних функций.
- Произведен обзор методов и программных средств, использованных для реализации приведенных алгоритмов и подсистем.

Далее перейдем к заключительной главе диссертационной работы – экспериментальной проверке эффективности реализованной системы.

5. ЭКСПЕРИМЕНТАЛЬНАЯ ПРОВЕРКА ЭФФЕКТИВНОСТИ РАСПАРАЛЛЕЛИВАНИЯ

В данной главе представлены результаты вычислительных экспериментов, проведенных с помощью разработанной системы выполнения FPTL-программ на многоядерных компьютерах, а также сравнение результатов экспериментов с аналогичными для системы выполнения языка Haskell.

В разделе 5.1 дается описание наборов программ, которые будут использованы в экспериментах, а также приводятся характеристики программно-аппаратной платформы экспериментов. В разделе 5.2 приведены результаты вычислительных экспериментов с использованием системы выполнения языка FPTL и описанных в разделе 5.1 тестовых программ. В разделе 5.3 приведено описание набора тестовых программ на языке Haskell, результаты вычислительных экспериментов с их использованием и сравнение с аналогичными результатами для языка FPTL.

5.1 Описание экспериментов

Целью проведенных вычислительных экспериментов являлась оценка эффективности предложенных методов и алгоритмов, реализованных в системе параллельного выполнения FPTL-программ.

Все эксперименты проводились на компьютерной системе с 8-ю физическими ядрами, представленными 2-мя четырехядерными процессорами Intel Xeon E5-2680 v2 с тактовой частотой 2.80ГГц. Общий объем оперативной памяти составлял 30 ГБ. Компьютерная система работала под управлением операционной системы Ubuntu Server. Для выполнения программ на языке Haskell использовалась система выполнения GHC 7.6.2. Результаты получены на основе трех прогонок с применением медианного фильтра к полученным данным.

Набор из семи функциональных программ на языке FPTL[2], взятых для экспериментов, описан в таблице 5.1. Эти программы были выбраны как примеры типовых задач, наиболее часто используемых для экспериментальной проверки эффективности реализации параллельных вычислений на многоядерных компьютерах. Алгоритмически, все перечисленные программы основаны на принципе «разделяй и властвуй», который гарантирует большие возможности для распараллеливания [46].

Программа	Описание
Fib	Вычисление 42-го числа Фибоначчи по рекурсивной схеме.
Integ	Вычисление интеграла функции $f(x) = 1/(x * e^x)$ на отрезке [10^{-6}, 10] с точностью 10^{-5} адаптивным методом трапеций.
Sort1	Быстрая сортировка односвязных списков размером в 1000000 случайных вещественных чисел, равномерно распределенных на отрезке [-1, 1]
Sort2	Быстрая сортировка массивов размером в 1000000 случайных вещественных числе, равномерно распределенных на отрезке [-1, 1]
Mul1	Умножение матриц размерности 600 (реализация на базе массивов).
Mul2	Умножение матриц размерности 600 (реализация на базе списков).
FFT	Быстрое преобразование Фурье суммы трех синусоидальных сигналов, заданных списком значений функции на 2^{17} точках.

Таб. 5.1. Описание набора тестовых функциональных программ на языке FPTL.

[2] Исходные тексты программ на языке FPTL доступны по адресу: https://github.com/sti0cli/fptl (дата обращения 30.09.2014).

В качестве критериев для оценки эффективности применялась зависимость времени выполнения и относительного ускорения выполнения параллельной программы от количества ядер компьютера. По характеру поведения этих критериев на различных наборах тестовых программ можно дать однозначную оценку эффективности и (масштабируемости) реализованной системы.

5.2 Результаты экспериментов для программ на языке FPTL

Отличительной особенностью тестовых программ численного интегрирования, вычисления -го числа Фибоначчи и умножения матриц является тот факт, что эти тестовые программы в основном работают над элементарными типами данных, за исключением задачи умножения матриц, представленных в виде списков. Тем не менее, размерность матрицы не столь велика, чтобы ощутить воздействие накладных расходов сборки мусора.

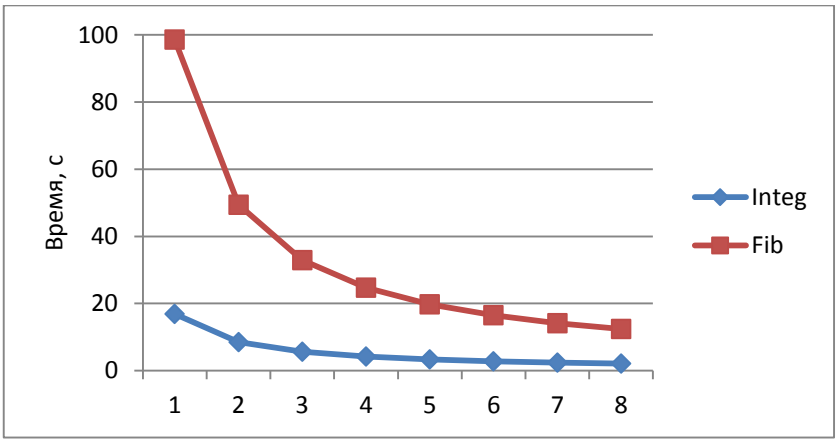

Рис. 5.1. Время выполнения программ *Integ* и *Fib* в зависимости от количества ядер.

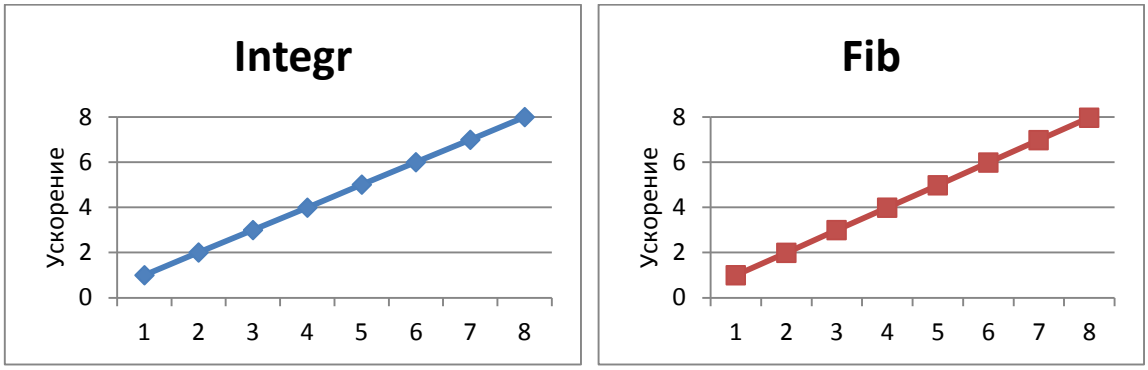

Рис. 5.2. Относительное ускорение выполнения программ *Integ* и *Fib* в зависимости от количества ядер.

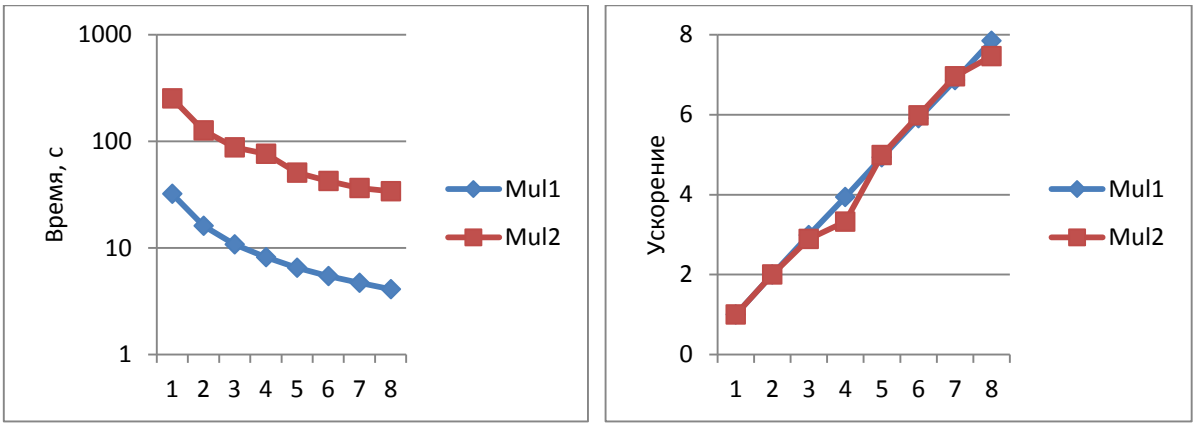

Рис. 5.3. Время и относительное ускорение выполнения программ умножения матриц в зависимости от количества ядер.

Как видно из графиков на рис. 5.1-5.3, задачи численного интегрирования, вычисления -го числа Фибоначчи и умножения матриц на языке FPTL обладают линейной масштабируемостью (ускорение времени выполнения представляет собой линейную функцию).

Основной особенностью задач сортировки списков и быстрого преобразования Фурье является их ориентированность на использование сложных типов данных: массивов и АТД, работа с которыми требует высокого уровня взаимодействия с системой управления памятью и сборкой мусора.

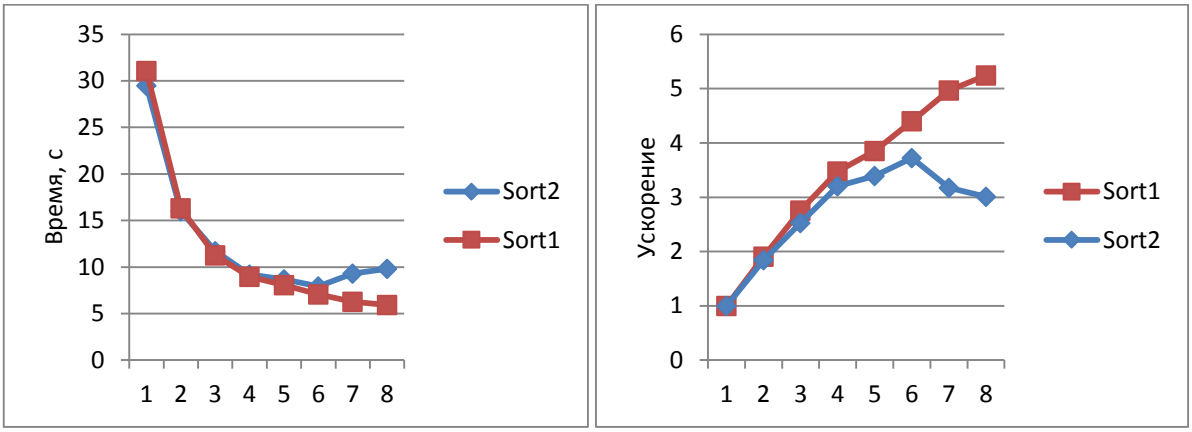

Рис. 5.4. Время и относительное ускорение выполнения программ сортировки в зависимости от количества ядер.

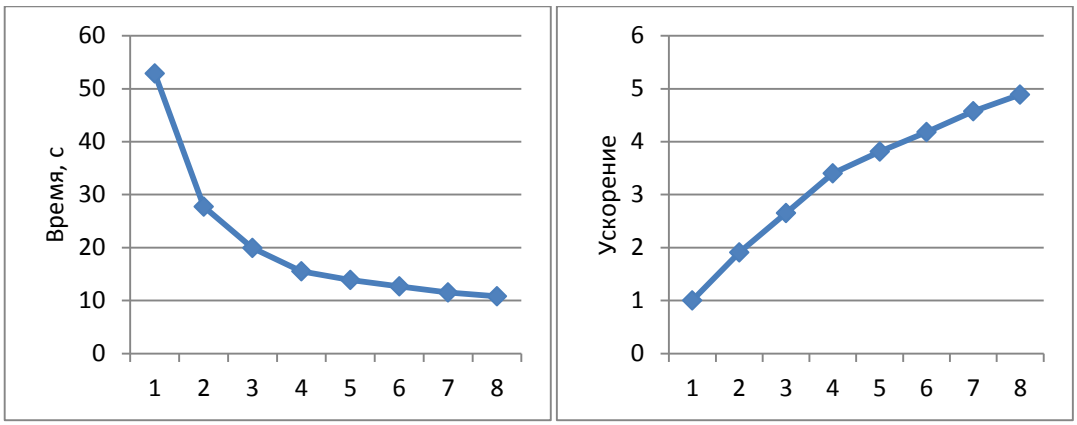

Рис. 5.5 Время и относительное ускорение выполнения программы FFT в зависимости от количества ядер.

На графиках рис. 5.4-5.5 видно, что реализация на языке FPTL быстрой сортировки и быстрого преобразования Фурье не обладает линейной масштабируемостью. Как было отмечено выше, данный факт обусловлен увеличением количества взаимодействий с подсистемой управления памятью: для представления элементов списков используются абстрактные типы данных, которые всегда создаются в динамической памяти. Так как алгоритм сборки мусора не обладает необходимой масштабируемостью, то поведение ускорения выполнения заметно отстает от линейного вида.

5.3 Результаты экспериментов для программ на языке Haskell

Для сравнения эффективности реализованной системы выполнения FPTL-программ с имеющимися аналогами, похожие эксперименты были проведены с использованием языка Haskell и его системы выполнения GHC [52]. Для языка Haskell были использованы следующие тестовые программы (таб. 5.2).

Программа	Описание
H_Fib	Вычисление 42-го числа Фибоначчи по рекурсивной схеме.
H_Integr	Вычисление интеграла функции $f(x) = 1/(x * e^x)$ на отрезке [10^{-6}, 10] с точностью 10^{-5} адаптивным методом трапеций.
H_Sort	Быстрая сортировка односвязных списков размером 10^6 случайных вещественных чисел, равномерно распределенных на отрезке [-1, 1].
H_Mul	Умножение матриц порядка 600. Реализация матриц на базе вложенных списков.

Таб. 5.2. Описание набора тестовых программ на языке Haskell.

Для этих программ были также получены зависимости времени и относительного ускорения выполнения программ от количества задействованных ядер. Графики полученных результатов отражены на рис. 5.6-5.7.

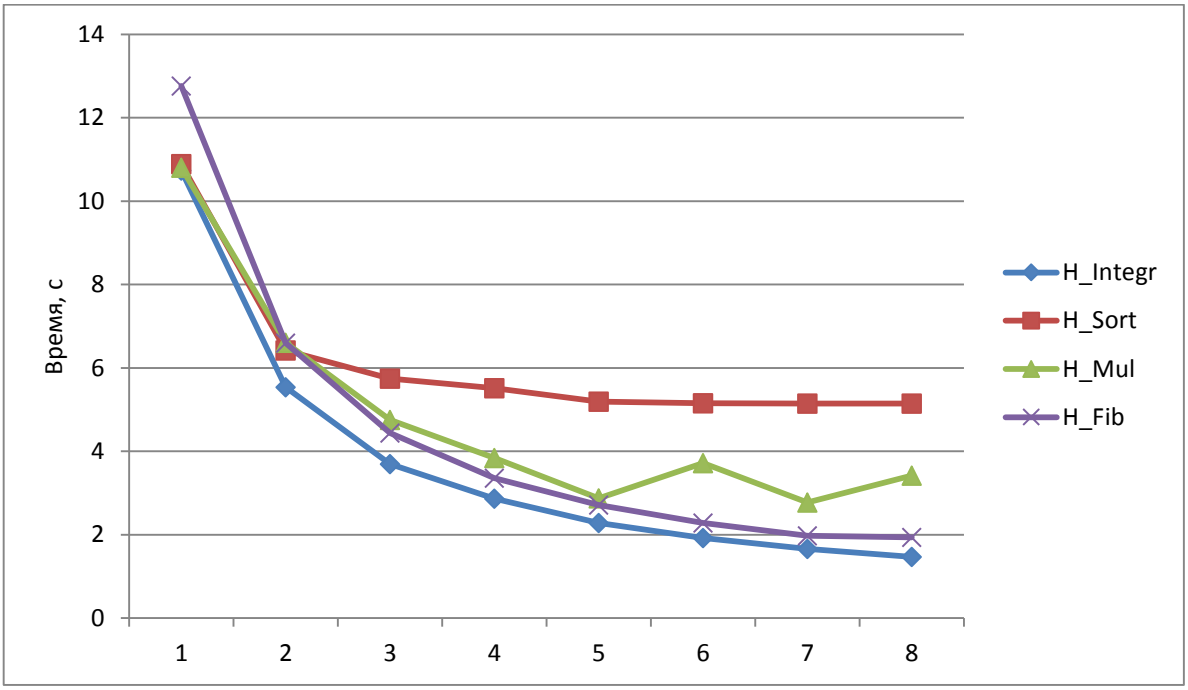

Рис. 5.6. Время выполнения тестовых программ на языке Haskell в зависимости от количества ядер.

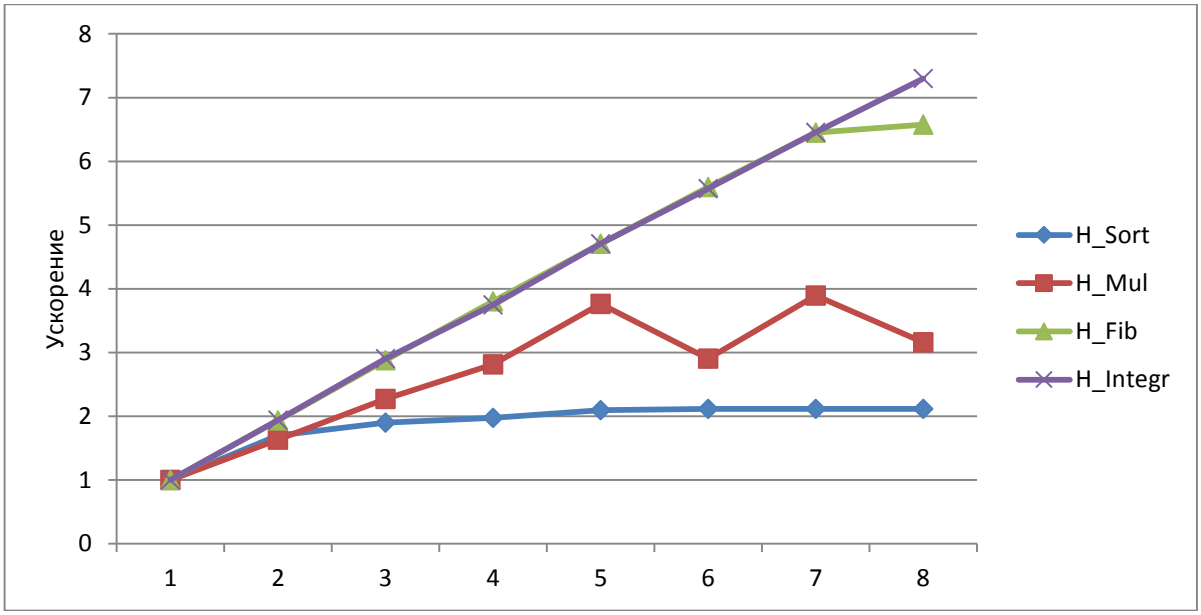

Рис. 5.7. Относительное ускорение выполнения тестовых программ на языке Haskell в зависимости от количества ядер.

Как видно из графиков на рис. 5.6, программы на языке Haskell выполняются примерно на порядок быстрее их аналогов на языке FPTL. Такое сильное различие во времени выполнения программ объясняется тем, что GHC-реализация Haskell применяет компиляцию исходной

программы [47], в то время как система выполнения FPTL-программ базируется на интерпретаторе.

Графики на рис. 5.7 показывают, что реализация вычислительных задач на языке Haskell также обладает хорошей масштабируемостью. Однако программы, оперирующие списковыми и абстрактными типами данных (H_Mul и H_Sort), масштабируются плохо. Как и в случае реализации на FPTL это обусловлено накладными расходами, вносимыми системой сборки мусора.

5.4 Заключение

В настоящей главе приведены результаты экспериментов по исследованию эффективности реализованной системы выполнения FPTL-программ и сравнение ее эффективности с системой выполнения программ на языке Haskell. Эти эксперименты приводят к следующим заключениям.

1. FPTL-программы в общем случае неплохо масштабируются на небольшом числе ядер (≤ 4), с увеличением числя ядер ускорение замедляется из-за увеличения накладных расходов на управление.
2. По критерию ускорения реализация FPTL не уступает реализации Haskell, а в некоторых случаях дает лучшие результаты. Это говорит о том, что разработанные механизмы управления параллельным выполнением программ реализованы в FPTL не менее эффективно, чем в Haskell. Однако, если сравнивать реализацию FPTL с Haskell по времени выполнения программ, то заметно очевидное преимущество реализации языка Haskell в силу того, что перед выполнением программы на языке Haskell компилируются. Это преимущество особенно заметно на задачах вычислительного характера и менее заметно для программ, оперирующих абстрактными типами данных. Естественным вариантом устранения этого недостатка является замена в FPTL затратного по времени режима интерпретации на режим компиляции.

3. При работе со сложными списковыми структурами данных сборка мусора вносит серьезные накладные расходы, что существенно влияет на ускорение времени параллельного выполнения программ. Это наблюдается как в реализации языка FPTL, так и в реализации языка Haskell.

ЗАКЛЮЧЕНИЕ. ОСНОВНЫЕ РЕЗУЛЬТАТЫ РАБОТЫ

Целью диссертационной работы являлись разработка и исследование эффективности системы выполнения функциональных параллельных программ на языке FPTL на многоядерных компьютерах. Перечислим основные результаты диссертационной работы:

1. Проведен сравнительный анализ методов распараллеливания вычислений в современных функциональных языках программирования и способов их реализации.
2. Разработаны и исследованы методы и алгоритмы эффективного параллельного выполнения функциональных программ на многоядерных компьютерах.
3. Разработан и реализован алгоритм контроля типовой корректности программ.
4. Создана интегрированная система параллельного выполнения FPTL-программ на многоядерных компьютерах, которая включает следующие подсистемы:
   - интерпретатор FPTL-программ,
   - управление параллельными процессами выполнения ФП,
   - статический анализ типовой корректности программы.
5. Проведено экспериментальное исследование эффективности созданной системы, которое доказало правильность принятых проектных решений и конкурентоспособность всей системы в целом.

ЛИТЕРАТУРА

1. *Borkar S.Y. and others*. Platform 2015: Intel processor and platform evolution for the next decade. // URL: http://goo.gl/43dbG3 (дата обращения 30.09.2014).
2. *Филд А., Харрисон П.* Функциональное программирование. // М.Ж Мир, 1993
3. *Бажанов С.Е., Кутепов В.П., Шестаков Д.А.* Язык функционального параллельного программирования и его реализация на кластерных системах. // Программирование. 2005, № 5.
4. *McCarthy J.* Recursive functions of symbolic expressions and their computation by machine. // Cambridge, Mass.: MIT, 1960
5. *Milner R.G.* The standard ML core language. Polymorphism // The ML/LCF/Hope Newsletter 1985. V. 2 № 2.
6. *Peyton Jones S. L.* The implementation of functional programming languages. // London: Prentice Hall, 1987
7. *Church A.* The calculi of lambda-conversion. // Ann. of Math. Studies, Princeton, N.J.: Princeton University Press, 1941. V. 6.
8. *Milner R.* A calculus of communicative systems. // LNCS, vol. 92. Springer, Heidelberg, 1980
9. *Hoare C.A.R.* Communicating sequential processes. // Communications of the ACM. Vol. 21 No. 9, 1978
10. F# Historical Acknowledgements. // URL: http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/ack.aspx (дата обращения 30.09.2014)
11. Linear ML // URL: https://github.com/pikatchu/LinearML/wiki (дата обращения 30.09.2013).
12. *Göetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D.* Java Concurrency in Practice. // Addison-Wesley, 2006
13. *Peyton Jones S., Singh S.* A Tutorial on Parallel and Concurrent Programming in Haskell // Microsoft Research, Cambridge, 2008

14. *Petricek T., Skeet J.* Real world functional programming. // Manning Publications Co., 2009
15. *Rui Shi, Hongwei Xi.* A Linear Type System for Multicore Programming, 2009
16. *Lee E. A.* The Problem with Threads. // Electrical Engineering and Computer Sciences University of California at Berkeley, 2006
17. *Shavit N., Touitou D.* Software transactional memory. // Distributed Computing, Vol. 10, No. 2, 1997
18. *Marlow S., Newton R., Peyton Jones S.* A monad for deterministic parallelism. // Haskell'11, 2011, Tokyo, Japan.
19. *Harris T., Marlow S., Peyton Jones S.* Haskell on shared-memory multiprocessor. // Proceeding in Haskell workshop, 2005
20. *Laijen D., Hall J.* Optimize managed code for multi-core machines. // MSDN Magazine, October 2007
21. *Harris T., Singh S.* Feedback Directed Implicit Parallelism. // ICFP'07, Freiburg, Germany, 2007.
22. *Бажанов С.Е., Кутепов В.П., Шестаков Д.А.* Структурный анализ и планирование процессов параллельного выполнения функциональных программ. // Изв. РАН. ТиСУ, 2005. № 6;
23. *Кутепов В.П., Фальк В.Н.* Функциональные системы: теоретический и практический аспекты. // Кибернетика, 1979, №1.
24. *Кутепов В.П.* Об интеллектуальных компьютерах и больших компьютерных системах нового поколения // Изв. РАН. Теория и системы управления, 1996. №5.
25. *Кутепов В.П., Фальк В.Н.* Направленные отношения: теория и приложения. // Изв. РАН. Техн. кибернетика. 1994. № 4, 5.
26. *Бажанов С.Е., Кутепов В.П., Шестаков Д.А.* Разработка и реализация системы функционального параллельного программирования на вычислительных системах. // Доклады международной научной

конференции «Суперкомпьютерные системы и их применение» SSA’2004. Мн.: ОИПИ НАН Беларуси, 2004.

27. *Кутепов В.П., Фальк В.Н.* Модели асинхронных вычислений значений функций в языке функциональных схем // Программирование. 1978. №3.
28. *Pierce B. C.* Types in programming languages. // The MIT Press, 2002
29. *Ахо А., Лам М., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструментарий. 2 изд. // Москва, Вильямс, 2008.
30. *Lamport L.* How to make a multiprocessor computer that correctly executes multiprocess programs. // IEEE Transactions on Computers, Vol. C-28 No. 9, 1979
31. *Dijkstra E.W.* Solution of a problem in concurrent programming control. // Communications of the ACM, Vol. 8 No. 9. 1965
32. *Frigo M., Leiserson C.E., Randall K.H.* The implementation of the Cilk-5 multithreaded language. // In proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, 1998.
33. Cilk Arts, Inc., Burlington, Massachistetts. Cilk++ Programmer’s Guide, 2008. // URL: http://goo.gl/MzbhKI (дата обращения 30.09.2014)
34. *Reiders J.* Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. // O’Relly Media, Inc., 2007
35. *Blumofe R.D., Joerg C.F., Kuszmaul B.C., Leiserson C.E., Randall K.H., Zhou Y.* Cilk: An efficient multithreaded runtime system. // Journal of Parallel and Distributed Computing. Vol. 37, 1996
36. *Blelloch G., Gibbons P., Matias Y.* Provably efficient scheduling for languages with fine-grained parallelism. // In processing of 7th Annual ACM Symposium on Parallel Algorithms and Architectures, 1996
37. *Manghwani R., He T.* Scalable memory allocation. New York University, http://cs.nyu.edu/~lerner/spring12/Preso05-MemAlloc.pdf

38. *Marlow S., Peyton Jones S.* Multicore Garbage Collection with Local Heaps. // ISMM'11, San Jose, California, USA, 2011

39. *Collins G. E.* A method for overlapping and erasure of lists. // Commun. ACM 3, 12 (Dec.), 1960, 655–657.

40. *Bluemofe R. D., Leiserson C. E.*. Scheduling multithreaded computations by work stealing.// NY: Journal of the ACM, 1999. V. 46 № 5.

41. *Amdahl M.* Validity of the single processor approach to archieving large scale computing capabilities. // A FIPS spring joint computer conference, 1967.

42. *Jones R., Hosking A.; Moss E.* The Garbage Collection Handbook: The Art of Automatic Memory Management. // CRC Applied Algorithms and Data Structures Series. Chapman and Hall/CRC, 2011

43. *Tene G., Iyengar B., Wolf M.* C4: The Continuously Concurrent Compacting Collector. // ISMM'11, June 4-5 2011

44. *Boehm H. J.* The Boehm-Demers-Weiser Conservative Garbage Collector. // HP Labs 2004
http://www.hpl.hp.com/personal/Hans_Boehm/gc/04tutorial.pdf

45. *Lea D.* A Java Fork/Join Framework. // Proceedings of the ACM 2000 conference on Java Grande, 2000

46. *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ. 3-е издание. // Вильямс, 2014

47. *Peyton Jones S. L., Hall C., Hammond K., Partain W., Wadler P.* The Glasgow Haskell Compiler: a technical overview. // Proceedings of Joint Framework for Information Technology Technical Conference, Keele, 1993.

48. *Воеводин В. В., Воеводин Вл.* В. Параллельные вычисления. // СПб.: БХВ-Петербург, 2002.

49. *Backus J.* Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. // Communications of the ACM. Vol. 21, No. 8, 1978

50. *Leijen D., Schulte W., Burckhardt S.* The Design of a Task Parallel Library. // OOPSLA 2009, Orlando, Florida, USA, 2009

51. *Кутепов В.П.* Исчисление функциональных схем и параллельные алгоритмы. // Программирование, 1976, №6.

52. The Glasgow Haskell Compiler // URL: http://www.haskell.org/ghc (дата обращения 30.09.2014).

ПРИЛОЖЕНИЕ 1: СПИСОК ВСТРОЕННЫХ ФУНКЦИЙ ЯЗЫКА FPTL

Имя функции	**Сигнатура**	**Описание**
id	any *… → any * …	Тождественная функция: f(x) = x
$[n]$	any *… → any	Выбор *n*-го элемента из кортежа
add	int * int → int	Сложение
sub	double * double → double	Вычитание
mul		Умножение
div		Деление
mod		Остаток от деления
equal		=
nequal		≠
greater		>
less		<
gequal		≥
lequal		≤
sqrt	double → double	Квадратный корень
sin		Синус
cos		Косинус
tan		Тангенс
asin		Арксинус
atan		Арктангенс
round		Округление к ближайшему целому
exp		Степень числа *e*
ln		Натуральный логарифм
abs	int → int	

	double → double	
Pi	λ → double	Получение числа π
E	λ → double	Получение числа e
cat	string * string → string	Конкатенация строк
search	string *string → string * … string * string → <undefined>	Поиск подстроки по регулярному выражению (второй аргумент) в исходной строке (первый аргумент)
match		Проверка соответствия по регулярному выражению
replace	string * string * string → string	Замена по регулярному выражению. Первый аргумент – исходная строка. Второй аргумент – регулярное выражение для поиска образца. Третий аргумент – заменяющая строка.
length	string → int	Длина строки.
getToken	string * string → string * string	Выделение лексемы из исходной строки (первый аргумент) по заданному регулярному выражению (второй аргумент). Возвращает выделенную лексему и оставшуюся часть строки. В качестве разделителей лексем считаются символы пробелов, табуляции, возврата и переноса строки.
rand	λ → double	Возвращает псевдослучайное число в интервале [0..1]

print	any *… → λ	Вывод кортежа на экран
printType	any * → λ	Вывод типа кортежа на экран
toString	int → string double → string	Преобразование в строку
toInt	string → int double → int	Преобразование в целое
toReal	string → double int → double	Преобразование в вещественное число
readFile	string → string	Чтение файла в строку. Первый аргумент – путь к файлу