УДК 519.6

ПОСТРОЕНИЕ ФОРМАЛЬНОЙ МОДЕЛИ Т-СИСТЕМЫ И ИССЛЕДОВАНИЕ ЕЕ КОРРЕКТНОСТИ

А. Н. Водомеров[1]

В статье исследуются методы распараллеливания программ, применяемые в Т-системе — средстве автоматического динамического распараллеливания вычислительных приложений. Отличительная особенность Т-системы от аналогичных средств — ориентация на программы, написанные на широко распространенных языках C, C++. В работе построена формальная модель для базовых конструкций Т-системы с помощью операционного подхода. В рамках модели доказывается корректность используемых механизмов. Разработанная модель была положена в основу новой реализации Т-системы.

Ключевые слова: формальная модель, Т-система, динамическое распараллеливание, корректность, параллельные программы.

1. Введение. В настоящей работе исследуются методы распараллеливания прикладных программ, используемые в Т-системе, предназначенной для автоматического динамического распараллеливания вычислительных приложений [2, 8]. Т-система позволяет без непосредственного участия разработчика преобразовывать программы, написанные для последовательного выполнения на одном компьютере, в программы, способные параллельно работать на нескольких процессорах или различных вычислительных устройствах [8]. Такой подход вызывает в настоящее время значительный интерес в связи с широким использованием слабосвязанных вычислительных комплексов, состоящих из большого числа модулей с различным программно-аппаратным обеспечением [11]. В подобных случаях поиск оптимального (в смысле минимизации времени работы программы) распределения вычислительной нагрузки представляет собой отдельную, достаточно сложную задачу, которую не удается эффективно решить в рамках традиционных подходов, например с использованием MPI. Применение Т-системы позволяет также обеспечить отказоустойчивость и организовать работу в условиях изменяющегося в процессе счета множества доступных узлов. Последнее обстоятельство немаловажно при работе в информационно-вычислительных структурах, построенных на технологиях GRID.

Одним из важнейших требований к таким программным средствам, как Т-система, является обеспечение их *корректности* — свойства, гарантирующего, что результаты исходной и распараллеленной программ совпадают. В течение продолжительного времени разработчики OpenTS (последней реализации Т-системы) ставили перед собой лишь задачи достижения высокой производительности [2, 8]. Вопрос о корректности системы или ее математическом моделировании не ставился.

В середине 2004 года использование OpenTS для распараллеливания прикладных программ для решения ряда практически значимых задач выявило в OpenTS несколько “тонких” ошибок. Они имели принципиальный характер и требовали подробного анализа для своего исправления. Особенность подобных ошибок состоит в том, что они проявляются не всегда, а только при определенном порядке прихода сетевых сообщений. Как следствие, вероятность их возникновения очень мала, поэтому обнаружить их традиционными методами тестирования программ достаточно сложно. По этой причине актуальной стала задача *исследования корректности работы подобных средств формальными методами.*

Следует отметить, что ситуация, возникшая в процессе разработки OpenTS, является весьма типичной для программного обеспечения, в том числе для средств распараллеливания приложений. Большая часть программ разрабатывается без специально сконструированной формальной модели, описывающей их работу. Например, вопрос о корректности MultiLisp [12] — среды параллельного выполнения программ на Lisp — был впервые рассмотрен в работах [10, 13], через 11 лет после ее реализации.

2. Т-система. Т-система — подход к автоматическому распараллеливанию программ, разрабатываемый в Институте программных систем РАН и МГУ им. М. В. Ломоносова [1 – 3, 8]. Первые версии были созданы в 1990-х годах, с тех пор система была несколько раз переписана заново. Последняя представленная в публикациях версия получила название OpenTS. Далее, употребляя термин Т-система, будем рассматривать именно эту версию (если не оговорено противное).

[1] Научно-исследовательский институт механики МГУ им. М. В. Ломоносова, 119192, Москва; e-mail: alexander.vodomerov@gmail.com

В T-системе используется подход на основе так называемых *прозрачных аннотаций*, которые добавляются в исходный текст программы и не меняют его смысла, однако дают системе выполнения (runtime) указания о возможном параллелизме. Впервые данная идея была описана в работе Бэйкера (H. Baker) и Хьюита (C. Hewitt) 1977 года [9], где был предложен механизм вызова функций call-by-future.

На настоящий момент существует ряд программных средств, реализующих данный подход, например, GUM, NeXeme. Большинство из них распараллеливают программы на функциональных языках (Haskell и Scheme соответственно). Как следствие, они не могут быть применены к прикладным вычислительным приложениям, которые, как правило, разрабатываются на языках низкого и среднего уровней, таких как C, C++, Fortran.

T-система ориентирована на распараллеливание практически важных прикладных задач, требующих высокопроизводительных вычислений, написанных на языках C и C++ (известны также попытки использования ее для программ на Fortran). OpenTS позволяет добиться высокой производительности как на SMP-машинах, так и на вычислительных кластерах. В настоящее время ведется разработка новой версии T-системы NewTS, способной эффективно работать на слабосвязанных вычислительных комплексах, построенных на архитектурно-технологических подходах GRID [11].

T-система реализует *динамическое распараллеливание* [8] путем балансировки нагрузки во время работы. Единицей вычислительной работы, способной перемещаться между узлами, является *T-функция* — функция, объявленная и реализованная специальным образом, позволяющим вычислять ее параллельно с основной программой на другом процессоре в SMP-машине или на другом узле. Преобразование обычных функций в T-функции осуществляется автоматически.

Распараллеливание программы с использованием T-системы производится следующим способом. Пользователь расставляет в своей программе *модификаторы* — специальные ключевые слова, которые не меняют смысл программы, однако дают компилятору и среде поддержки времени выполнения (runtime) некоторые указания (например, о том, что некоторые участки кода могут выполняться параллельно). Основными модификаторами являются `tfun`, `tval`. Модификатор `tfun` указывается перед функцией и означает, что объявляемая функция является T-функцией. Результатом вызова T-функции является специальное значение, которое называется “неготовым”. После того как функция закончит вычисление, значение становится “готовым” (содержащим конкретное число, строку и другую подобную величину). Для работы с неготовыми значениями используются специальные *T-переменные*, которые объявляются с помощью модификатора `tval`. При попытке выполнения с неготовой переменной действий, которые требуют ее явного значения, например, арифметических операций, функция “засыпает” до тех пор, пока требуемое значение не будет вычислено. Приведем простой пример:

```
tfun double long_func(double x)
{
   ... // сложные вычисления
}

tfun double func(double x)
{
   tval double a = long_func(x);
   ... // произвести вычисления
   double b = another_long_func(y);
   // long_func будет параллельно вычисляться
   // на другом узле или процессоре
   c = a + b;  // ожидание завершения long_func
   ...
}
```

3. Модель T-системы.

3.1. Общий план исследования. Для исследования методов распараллеливания прежде всего необходимо выбрать способ моделирования исполнения последовательных и параллельных программ. Методы формального описания программных конструкций представляют собой чрезвычайно обширную тему. Сравнительный анализ различных подходов и возможности их применения к T-системе выходит за рамки данной статьи.

Сформулируем лишь общий план, по которому в дальнейшем будут излагаться результаты исследования T-системы.

1. В качестве основного инструмента для моделирования работы программ используется *операционный подход*, а также его различные вариации, например, структурный операционный подход (SOS) [16].

2. Выбирается некоторое небольшое, но в то же время представительное подмножество языка, достаточное для написания широкого класса целевых программ. Основная задача в процессе выбора данного подмножества состоит в том, чтобы минимизировать его размер и тем самым упростить семантику языка.

3. Формально задается синтаксис языка и его параллельного расширения.

4. Семантика исходного последовательного языка задается с помощью выполнения на некоторой абстрактной машине.

5. Аналогичным образом описывается процесс выполнения параллельной программы.

6. Строится отображение между машинами и исследуются его свойства.

7. Доказывается корректность метода распараллеливания.

3.2. Выделение подмножества языка. T-система производит распараллеливание программ на языках C и C++. Как следствие, возникает задача формального описания программ на этих языках. Исследования семантики C и C++ широко представлены в литературе. Наиболее полные модели были построены в работах Норриша (Michael Norrish) [14] и Папаспироу (Nicholas S. Papaspyrou) [15] с помощью операционного и денотационного подходов соответственно. Все исследователи отмечают, что семантика языка C чрезвычайно сложна. В работе [14] утверждается, что разработанная семантика настолько громоздка, что для ее применения к доказательству свойств программ или языка необходимы средства автоматического вывода теорем (automatic theorem prover). Вопросы формального описания языка C рассматривались также в работах отечественных исследователей. Среди них отметим работы В. А. Непомнящего, И. С. Ануреева, И. Н. Михайлова, А. В. Промского [4 – 7].

В связи с изложенным выше возникает необходимость существенного упрощения языка, сведения его к некоторому "минимальному" подмножеству, который в то же время остается достаточным для выражения основных конструкций, используемых в вычислительных программах. Такое подмножество языка называется его *ядром* (core).

Основной целью настоящего исследования является изучение механизмов, поддерживаемых T-системой, а не различных тонкостей языка. Многие аспекты языка, которым, как правило, уделяется большое внимание при построении семантики, слабо влияют на распараллеливание. В частности, можно пренебречь статической типизацией и типами размещения переменных.

Выполним над программой следующие действия:

— заменим все неявные приведения типов явными (в виде вызовов встроенных функций);

— все типы, объявленные с помощью `typedef`, заменим их содержимым;

— переименуем все типы и переменные так, чтобы исключить совпадения;

— перенесем все объявления локальных переменных в начала функций (считаем, что типы данных не имеют конструкторов, выполняющих какие-либо нетривиальные действия);

— преобразуем конструкции вида `x = x++` или `x += a` в `x = x + 1`;

— преобразуем циклы `while`, `do`, `for` и операторы `break`, `continue` в условные переходы по меткам;

— заменим условные операторы, выполняющие сложные действия, на операторы условного перехода;

— раскроем составные операторы (с учетом вышеописанных преобразований, для этого достаточно убрать фигурные скобки);

— упростим вычисление сложных выражений, введя дополнительные переменные: например, `a = f(x)+g(y)` преобразуем в `tmp1 = f(x); tmp2 = g(y); a = tmp1 + tmp2`;

— заменим логические операторы `&&`, `||` на условные операторы `if`: например, `if (x && g(x)) ...` на `if (x) { y = g(x); if (y) ... }`.

Перечисленные действия соответствуют преобразованиям, проводимым при компиляции, поэтому не меняют смысла программы. С другой стороны, результирующая программа значительно проще для анализа.

Еще одна особенность T-системы связана с указателями. В имеющихся реализациях T-системы передача указателей в аргументах функции или в возвращаемом значении не поддерживается. Вместо них пользователь может воспользоваться T-указателями, объявляемыми с ключевым словом `tptr`. К сожалению, в OpenTS объекты класса `TPtr` имеют семантику значений (value semantics), а не указателей (pointer semantics). Это обстоятельство не позволяет использовать их как полноценную замену обычным указателям в языках C и C++. В действительности T-указатели не предоставляют никаких новых возможностей по сравнению с T-переменными и могут быть реализованы в виде небольшой надстройки над ними. В этой связи указатели и T-указатели не включаются в модель на данном этапе исследований.

3.3. Ограничения, связанные с распараллеливанием. Выбранный в T-системе метод распа-

раллеливания, состоящий в разбиении программы на функции и их параллельном запуске, приводит к некоторым ограничениям, среди которых можно выделить следующие:

— отсутствие глобальных переменных;

— программа не получает информации извне (чтение файлов, передача данных по сети, взаимодействие с пользователем);

— отсутствие недетерминированных функций, например, `random`.

Необходимо отметить, что данные ограничения носят принципиальный характер и свойственны всем системам распараллеливания, в том числе MultiLisp и его аналогам. Для обеспечения корректности все функции, обращающиеся к глобальной переменной, должны следовать именно в том порядке, в котором бы они выполнялись при запуске исходной последовательной программы. Данное обстоятельство делает распараллеливание невозможным. В существующих программных реализациях, например OpenTS или MultiLisp, пользователю разрешается задействовать глобальные переменные и производить операции ввода-вывода. Однако корректность работы программы при этом не гарантируется, а задача обеспечения правильности результата возлагается на программиста (например, необходима синхронизация файлов посредством общей файловой системы). Поскольку основная цель состоит в исследовании корректности распараллеливания, подобные ситуации рассматриваться не будут.

Ограниченный размер статьи позволяет исследовать только базовые конструкции. Многие важные аспекты языка, в том числе массивы, структуры, указатели, переменные функционального типа, выделение памяти, не рассматриваются. Описание подобных возможностей требует включения большого объема технических деталей.

3.4. Абстрактная машина C. Опишем операционную семантику подмножества C, выбранного с учетом изложенных выше соображений. Определим множество значений базовых типов языка:

$$B = \mathrm{Int} \cup \mathrm{Char} \cup \mathrm{Bool} \cup \mathrm{Float},$$

где Int — множество значений типа `int`, Char — множество значений типа `char`, и так далее. В качестве Int могут выступать целые натуральные числа от -2^{31} до $2^{31}-1$, числа от -2^{63} до $2^{63}-1$, или все множество $\mathbb{Z}$. Конкретный вид Int, Char и самого B нас не интересует, так как дальнейшие рассуждения от него не зависят. Без уменьшения общности можно считать, что B не пересекается с $\mathbb{N}$ (множеством номеров операторов).

Определим множество допустимых выражений E: $e ::= b \mid id \mid e_1 \oplus e_2 \mid e_1 \odot e_2 \mid \delta(e_1)$, где $b \in B$ — значение базового типа, $id \in Id$ — имя переменной, e_i — подвыражения. Под $\oplus$ понимается бинарный арифметический оператор (`+`, `-`, `*`, `/`, `%`, `>>`, `<<`). Удобно также включить в $\oplus$ операторы сравнения: `<`, `>`, `==`, `!=`. Знаком δ обозначаются встроенные унарные функции (`!`, `-`, `sqrt`, `exp`, `log`, `sin`, `cos` и т.п.). Как уже отмечалось ранее, на данный момент мы не рассматриваем структуры, указатели и, как следствие, операторы доступа к полю структуры и к элементам массива, операторы разыменования и взятия адреса. С учетом этих ограничений, множество возможных значений Val_C (во избежание путаницы индекс обозначает название абстрактной машины) совпадает с множеством значений базовых типов B.

Выделим следующие основные операторы языка:

$$c ::= ; \mid id \ = \ e \mid id \ = \ funId(e_1, e_2, \ldots, e_n) \mid \texttt{return}\ e \mid \texttt{goto}\ l \mid \texttt{if}\ e\ \texttt{goto}\ l,$$

где $funId \in FunId$ — имена функций, $e_i \in E$ — выражения, $c \in C$ — команды, $l \in \mathbb{N}$ — номер оператора.

Заметим, что вычисление функции исключено из множества выражений. Такое странное на первый взгляд решение не ограничивает общности. В действительности при вычислении сложных выражений, вызывающих функции, промежуточные результаты сохраняются в некоторых временных переменных (которые обычно размещаются в регистрах процессора). Подобный подход позволяет значительно упростить вычисление выражений. В частности, рассматриваемые выражения не имеют побочных эффектов (side-effect). Если включать вызовы функций в выражения, то при обобщении на параллельный случай потребовалось бы включать в семантику выражений обращения к памяти, возможность возникновения ошибок и зацикливания в процессе их вычисления. При выбранном способе отмеченные трудности переносятся на описание команд.

Функция описывается своим именем, аргументами и телом (последовательностью команд):

$$f = (n, args, [c_1, \ldots, c_k]) \in \Phi,$$

где $n \in funId$ — имя функции, а $args$ — список ее аргументов (имен переменных). Во избежание неопределенных значений потребуем, чтобы каждая функция заканчивалась оператором `return`. При таком подходе программа задается конечным множеством функций: $P = \{f_1, f_2, \ldots, f_k\} \in \Pi, \quad f_i \in \Phi$.

Обозначим для удобства $P(n)$ тело функции, имеющей имя n в программе P. Для описания семантики данного мини-языка рассмотрим следующую абстрактную машину, которую в дальнейшем будем называть *машиной C*. Ее состояния st представляют собой стек вызовов функций: $st = \{s_1, s_2, \ldots, s_l\}$, где s_i — состояние i-ой функции в стеке вызовов, содержащее номер текущей выполняемой строки k, текущие значения переменных (окружение, environment) Σ и имя функции n: $s = (k, \Sigma, n), \quad k \in \mathbb{N}, \quad \Sigma \in Env = Id \to Val_C, \quad n \in FunId$.

Ниже приведены правила переходов. Через $\mathcal{E} : E \times Env \to Val_C$ обозначена функция вычисления выражений, описание которой не приводим ввиду ограниченного объема статьи. В начальном состоянии в стеке содержится одна функция `main`: $s_0 = \{(1, \varnothing, \text{“main''})\}$. В каждый момент времени выполняется функция, находящаяся на вершине стека. Конечным состоянием является то, из которого невозможен ни один переход.

$\{(k, \Sigma, n), \ldots\} \quad c = ; \longrightarrow$
$\{(k+1, \Sigma, n), \ldots\}$ $(skip_C)$

$\{(k, \Sigma, n), \ldots\} \quad c = id = e \longrightarrow$
$\{(k+1, \Sigma[id = v], n), \ldots\}, v = \mathcal{E}(e)$ $(assign_C)$

$\{(k, \Sigma, n), \ldots\} \quad c = \texttt{goto}\ l \longrightarrow$
$\{(l, \Sigma, n), \ldots\}$, если $l \leqslant |P(n)|$ $(goto_C)$

$\{(k, \Sigma, n), \ldots\} \quad c = \texttt{if}\ e\ \texttt{goto}\ l \longrightarrow$
$\{(l, \Sigma, n), \ldots\}$, если $l \leqslant |P(n)|$ и $\mathcal{E}(E) = \mathbf{true}$ $(if_{C,t})$

$\{(k, \Sigma, n), \ldots\} \quad c = \texttt{if}\ e\ \texttt{goto}\ l \longrightarrow$
$\{(k+1, \Sigma, n), \ldots\}$, если $l \leqslant |P(n)|$ и $\mathcal{E}(E) = \mathbf{false}$ $(if_{C,f})$

$\{(k, \Sigma, n), \ldots\} \quad c = id = funId(e_1, \ldots, e_r) \longrightarrow$
$\{s_1, (k, \Sigma, n), \ldots\}$, где
$s_1 = (1, \varnothing[id_1 = v_1, \ldots, id_r = v_r], funId)$, где $v_i = \mathcal{E}(e_i, \Sigma)$ $(call_C)$

$\{(k, \Sigma, n), s_2, \ldots\} \quad c = \texttt{return}\ e \longrightarrow$
$\{s_2', \ldots\}$, где $s_2' = (k_2 + 1, \Sigma_2[id = \mathcal{E}(e, \Sigma)]), n_2)$,
если s_2 имеет вид (k_2, Σ_2, n_2)
и в s_2 выполнялась команда $c_2 = id \ = \ funId(e_1, \ldots, e_l)$ $(return_C)$

$\Sigma[id = v] \stackrel{\text{def}}{=} (\Sigma \backslash (\{id\} \times B)) \cup \{(id, v)\}$

3.5. Абстрактная машина T. Теперь опишем, как происходит распараллеливание в наиболее простом случае. Для этого рассмотрим следующую абстрактную машину, которую будем называть *машиной T*. Она моделирует работу T-системы на многопроцессорном компьютере с общей памятью (SMP).

Базовые типы, выражения и основные операторы в машине T остаются такими же, как и в машине C. Каждая переменная теперь может быть как обычной переменной, так и T-переменной. С целью упрощения вопросы, связанные со статической типизацией, здесь не рассматриваются. В реализации T-системы при выполнении операций с T-переменными вызываются различные функции, например, оператор приведения типов или оператор разыменования (dereferencing), которые вызывают принудительное ожидание. Будем моделировать такое поведение вставкой специальной функции `wait`, которая ожидает готовности выражений. С учетом данного обстоятельства можно рассматривать обычные переменные как частный случай T-переменных, в которые записываются только готовые значения. Подобное решение позволяет добиться единообразия при моделировании переменных и заметного упрощения. Необходимо отметить, что данная возможность применима только в рамках NewTS. В OpenTS поведение T-переменных существенно отличается от поведения обычных переменных и по этой причине приходится рассматривать их раздельно.

В машине T переменные могут содержать либо обычные значения из множества B, либо указывать на специальный объект — *заменитель* (placeholder), который будет содержать значение через некоторое время. В первом случае говорят, что T-переменная имеет *готовое* значение, во втором — *неготовое*. Неготовое значение означает, что данное значение либо находится еще в процессе вычисления на другом процессоре или узле, либо уже подсчитано, но результат еще не поступил на данный узел. Заменители можно моделировать различными способами, наиболее простой из них — занумеровать все заменители и представлять такую ссылку в виде натурального числа. Множество всех заменителей представляет собой некоторое подобие общей памяти, доступ к которой имеют все работающие T-функции. С учетом перечисленных соображений получаем следующее множество значений: $Val_T = B \cup \mathbb{N}$, $Env_T = Id \to Val_T$. Каждый заменитель может либо содержать вычисленное значение, доступное для чтения всем функциям, либо не содержать никакого значения (для значений в процессе вычисления). Последнее состояние будем обозначать символом $\mathcal{U}$, множество состояний заменителей — $Store$: $Store = \mathbb{N} \to B \cup \{\mathcal{U}\}$. Значение

заменителя меняется только один раз, а именно в тот момент, когда завершается функция (для возвращаемых значений), вычисляющая его, или когда пользователь делает операцию `tdrop` (для переменных `tout`). Здесь наблюдается важное различие с OpenTS, где содержимое T-объектов (так называются заменители в OpenTS) менялось многократно, что вызывало большое количество вопросов по поводу, когда значение считается окончательным и доступным для функций-потребителей. Для этой цели использовались различные флажки и переменная `owner`, обозначающая текущего владельца T-объекта. В NewTS необходимость в подобных действиях отсутствует. Если в заменителе имеется значение, то оно доступно всем функциям. Данное обстоятельство имеет важное значение при работе на нескольких узлах. В случае SMP аппаратные средства реализуют общую память и изменения, сделанные в заменителе, сразу становятся видны всем функциям. В то же время, при взаимодействии узлов посредством обмена сообщениями такая возможность не обеспечивается, поэтому требуются специальные методы обеспечения согласованности (consistency) состояния заменителей на разных узлах. Многократная запись в T-объекты из различных мест программы в OpenTS существенно осложняет подобные алгоритмы. Необходимо отметить, что однократное присваивание заменителей не противоречит многократному присваиванию T-переменных. Заменители представляют собой внутренние объекты системы, доступа к которым пользователь не имеет.

Отметим, что в описываемой модели заменители содержат готовые значения из множества B. Возможен и другой вариант: $Store = \mathbb{N} \to Val_T \cup \{\mathcal{U}\}$. При таком определении $Store$ функции могут возвращать неготовые значения, не дожидаясь их вычисления. Для получения конкретного численного значения, возможно, придется обходить цепочку значений, что немного усложняет поведение T-переменных. Различия между указанными вариантами в действительности незначительны. В рамках работы над NewTS были реализованы оба варианта. Каких-либо существенных различий в функционировании имеющихся приложений обнаружено не было. По этой причине мы рассматриваем более простой случай.

К состоянию каждой работающей функции добавляется еще один параметр r, обозначающий номер заменителя, в который функция вернет свое значение при завершении. В остальном состояния машины T имеют аналогичный вид: $s = (k, \Sigma, n, r)$, $k \in \mathbb{N}$, $\Sigma \in Env$, $n \in FunId$, $r \in \mathbb{N}$, $st = \big(\{s_1, s_2, \ldots, s_n\}, \sigma\big)$, $\sigma \in Store$. Состояния, отличающиеся лишь порядком T-функций, считаются одинаковыми.

Функция, вычисляющая выражения, также может возвращать неготовые значения:

$$\mathcal{E} : E \times Env \times Store \to Val_T.$$

В отличие от машины C, не все выражения могут быть вычислены сразу. Например, результат сложения двух неготовых T-переменных не определен. Будем считать, что значение таких выражений не определено. Ниже приведены правила переходов в машине T.

$(\{(k, \Sigma, n), \ldots\}, \sigma) \quad c = ; \longrightarrow$
$\quad (\{(k+1, \Sigma, n), \ldots\}, \sigma)$ $(skip_T)$

$(\{(k, \Sigma, n), \ldots\}, \sigma) \quad c = id = E \longrightarrow$
$\quad (\{(k+1, \Sigma[id = v], n), \ldots\}, \sigma)$, где $v = \mathcal{E}(E)$ $(assign_T)$

$(\{(k, \Sigma, n), \ldots\}, \sigma) \quad c = \texttt{goto}\ l \longrightarrow$
$\quad (\{(l, \Sigma, n), \ldots\}, \sigma)$, если $l \leqslant |P(n)|$ $(goto_T)$

$(\{(k, \Sigma, n), \ldots\} \quad c = \texttt{if}\ e\ \texttt{goto}\ l \longrightarrow$
$\quad (\{(l, \Sigma, n), \ldots\}, \sigma)$, если $l \leqslant |P(n)|$ и $\mathcal{R}(\mathcal{E}(e)) = \mathbf{true}$ $(if_{T,t})$

$(\{(k, \Sigma, n), \ldots\} \quad c = \texttt{if}\ e\ \texttt{goto}\ l \longrightarrow$
$\quad (\{(k+1, \Sigma, n), \ldots\}, \sigma)$, если $l \leqslant |P(n)|$ и $\mathcal{R}(\mathcal{E}(e)) = \mathbf{false}$ $(if_{T,f})$

$(\{(k, \Sigma, n), \ldots\}, \sigma) \quad c = id = funId(e_1, \ldots, e_s) \longrightarrow$
$\quad (\{(1, \Sigma'_s, funId, r), (k+1, \Sigma[id = r], n), \ldots\}, \sigma[r = \mathcal{U}])$,
$\quad$ где $r = alloc(\sigma)$, Σ_i строятся по аргументам функции:
$\quad \Sigma'_0 = \varnothing$, далее рекурсивно по i:
$\quad \Sigma'_i = \Sigma'_{i-1}[id_i = \mathcal{E}(e_i)]$, где id_i — i-й аргумент funId $(call_T)$

$\{(k, \Sigma, n, r), s_2, \ldots\} \quad c = \texttt{return}\ e \longrightarrow$
$\quad (\{s_2, \ldots\}, \sigma[r = v])$, где $v = \mathcal{R}(\mathcal{E}(e, \Sigma, \sigma), \sigma)$ $(return_T)$

$$\mathcal{R} : Val_T \times Store \to B, \quad \mathcal{R}(v, \sigma) \stackrel{def}{=} \begin{cases} v, & \text{если } v \in B \\ \sigma(v), & \text{если } v \in \mathbb{N}, \quad \sigma(v) \in B \\ & \text{иначе не определено} \end{cases}$$

$$alloc(\sigma) \stackrel{def}{=} 1 + \max\{k \mid (k, v) \in \sigma\}$$

Начальное состояние $s_0 = (\{(1, \varnothing, \text{“}main''\text{”}, 0)\}, \{(0, \mathcal{U})\})$.

Наиболее важное отличие машины T от машины C состоит в том, что работать может *любая, а не только первая* функция. Степень такого параллельного выполнения естественным образом ограничивается зависимостями данных друг от друга.

3.6. Корректность машины T. В данном разделе исследуется корректность машины T. Ввиду ограниченного объема статьи приводятся лишь формулировки утверждений без доказательств, которые содержат достаточно много технических деталей.

Теорема 1. *Пусть P — некоторая программа для машины C. Пусть P нормально завершается на машине C за конечное время. Тогда P — корректная программа для машины T. Существует порядок выполнения на машине T, при котором P завершается с таким же результатом и за такое же количество шагов.*

Для доказательства дальнейших утверждений воспользуемся достаточно известной техникой, являющейся обобщением свойства Черча–Росса (CR property, diamond property) в λ-исчислении. Впервые это свойство было использовано при доказательстве единственности нормальной формы λ-выражений. Впоследствии этот подход получил широкое обобщение, в частности, Келлер (R. Keller) ввел понятие конфлюэнтности для асинхронных параллельных процессов. Аналогичные методы используются во многих работах по корректности методов распараллеливания, например в [10, 13]. В представленном подходе используется слегка модифицированный вариант, так как требуется доказать тот факт, что выполнение программы не может продолжаться бесконечно.

Лемма 1. *Если в состоянии $\big(\{s_1, \ldots, s_i, \ldots\}, \sigma\big)$ в T-функции s_1 некоторое выражение e имело значение $\mathcal{E}(e) = v$, то после перехода в любой другой T-функции $s_i \to s_i'$, $i \neq 1$ выражение e сохранит свое значение.*

В непосредственной формулировке свойство Черча–Росса для машины T не выполняется. Причина этого состоит в том, что выделение памяти дает разные адреса заменителей и результирующие состояния оказываются различными. Подобные состояния необходимо отождествить. Для этого введем понятие изоморфизма состояний, которое, в некотором роде, является аналогом α-редукции в λ-исчислении. Любое отображение нумераций памяти $h : \mathbb{N} \to \mathbb{N}$ естественным образом продолжается до отображения значений:

$$\overline{h} : Val_T \to Val_T, \quad \overline{h}(v) = \begin{cases} b, & \text{если } v = b \in B; \\ h(p), & \text{если } v = p \in \mathbb{N}. \end{cases}$$

Применяя полученное $\overline{h}$ ко всем переменным всех функций в состоянии и ко всем значениям σ, получаем отображение состояний: $\overline{h} : State \to State$.

Определение. Назовем состояния A и B *изоморфными* ($A \cong B$), если существует отображение $h : \mathbb{N} \to \mathbb{N}$, такое, что оно биективно и его продолжение отождествляет A и B: $\overline{h}(A) = B$.

Лемма 2. *Пусть из некоторого состояния st_1 возможны два перехода в различные состояния: st_2 и st_3 соответственно. Тогда существуют изоморфные состояния st_4 и st_4', такие, что возможен переход из st_2 в st_4, а также из st_3 в st_4'. Другими словами, переходы в различных T-функциях коммутируют с точностью до изоморфизма.*

Теорема 2. *Рассмотрим некоторое состояние st_1. Пусть из st_1 существует порядок выполнения, который приводит в конечное состояние st_{fin}. Тогда любой путь выполнения, начинающийся в st_1, конечен, заканчивается в состоянии $st'_{fin} \cong st_{fin}$ и содержит столько же шагов.*

Теорема 3. *В условиях теоремы* 1 *любой порядок выполнения приводит к одинаковому результату* (*возвращаемому значению функции* `main`).

Теорема 3 дает утвердительный ответ на вопрос о корректности распараллеливания, проводимого T-системой (в рамках описанной модели). Можно доказать также более сильное утверждение о том, что каждому вызову функции в машине C в процессе работы программы соответствует некоторый вызов в машине T с эквивалентными аргументами.

Теорема 2 дает также ответ на вопрос о возможном возникновении ошибок. Согласно лемме 2 и теореме 2 при любом порядке выполнения функций программа достигает конечного состояния со всеми выполненными функциями. Следовательно, в процессе выполнения программы всегда возможно применить правила, вычисляющие арифметические выражения и встроенные функции, а значит ошибок типа деления на ноль или извлечения корня из отрицательного числа не происходит.

Изложенная модель допускает естественное обобщение на случай работы на нескольких узлах. Ключевая особенность, делающая подобное расширение возможным, состоит в том, что значения присваиваются заменителям не более одного раза. В то же время, возникает задача моделирования сообщений и каналов связи. На настоящий момент такая модель разработана, но ее исследование проведено не полностью, поэтому мы не будем ее излагать. Доказательство корректности подобной модели может служить

материалом для отдельной статьи.

4. Заключение. В ходе исследования были получены следующие основные результаты:

— выделено подмножество языка, позволяющее описать класс задач, которые допускают распараллеливание с помощью T-системы;

— построена формальная модель распараллеливания, проводимого NewTS, в базовом, наиболее простом случае;

— доказана корректность разработанной модели в случае исполнения на машине с общей памятью;

— описанная модель реализована в программном коде NewTS, что позволило существенно сократить и упростить реализацию, а также исправить ряд важных ошибок.

СПИСОК ЛИТЕРАТУРЫ

1. *Абрамов С.М., Адамович А.И., Коваленко М.Р.* T-система — среда программирования с поддержкой автоматического динамического распараллеливания программ. Пример реализации алгоритма построения изображений методом трассировки // Программирование. 1999. **2**. 100–107.
2. *Абрамов С.М., Васенин В.А., Мамчиц Е.Е., Роганов В.А., Слепухин А.Ф.* Динамическое распараллеливание программ на базе параллельной редукции графов. Архитектура программного обеспечения новой версии T-системы // Труды Всероссийской научной конференции (30 октября – 2 ноября 2000 г., г. Черноголовка). М.: Изд-во МГУ, 2000. 261–265.
3. *Васенин В.А., Роганов В.А.* GRACE: распределенные приложения в Интернет // Открытые системы. 2001. № 5, 6.
4. *Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.* На пути к верификации C-программ. Часть 1. Язык C-light. Препринт ИСИ СО РАН, № 84. Новосибирск, 2001.
5. *Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.* На пути к верификации C-программ. Часть 2. Язык C-light-kernel и его аксиоматическая семантика. Препринт ИСИ СО РАН, № 87. Новосибирск, 2001.
6. *Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.* На пути к верификации C программ. Язык C-light и его формальная семантика // Программирование. 2002. № 6. 1–13.
7. *Промский А.В.* Формальная семантика C-light программ и их верификация методом Хоара. Диссертация на соискание ученой степени к.ф.-м.н. Новосибирск, 2004.
8. *Abramov S., Adamovich A., Inyukhin A., Moskovsky A., Roganov V., Shevchuk E., Shevchuk Yu., Vodomerov A.* OpenTS: An outline of dynamic parallelization approach // Lecture Notes in Computer Science. Vol. 3606. New York–Heidelberg–Berlin: Springer-Verlag, 2005. 303–312.
9. *Baker H. G., Hewitt C.* The incremental garbage collection of processes // ACM Conference on AI and Programming Languages. New York, 1977. 55–59.
10. *Flanagan C., Felleisen M.* The semantics of future and its use in program optimization // 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Houston, 1995.
11. *Foster I., Kesselman C., Tuecke S.* The anatomy of the GRID. Enabling scalable virtual organizations. Lecture Notes in Computer Science. Vol. 2150. New York–Heidelberg–Berlin: Springer-Verlag, 2001.
12. *Halstead R.H., Jr.* Implementation of Multilisp: Lisp on a multiprocessor // Proc. ACM Symposium on Lisp and Functional Programming. Austin, 1984. 9–17.
13. *Moreau L.* The semantics of future in the presence of first-class continuations and side-effects. Southampton (United Kingdom), 1995.
14. *Norrish M.* C formalised in HOL. Technical Report UCAM-CL-TR-453. University of Cambridge, Computer Laboratory. Cambridge, 1998.
15. *Papaspyrou N.S.* A formal semantics for the C programming language. Doctoral Disseration. National Technical University of Athens. Athens (Greece), 1998.
16. *Plotkin G.D.* A structural approach to operational semantics. Technical Report DAIMI FN-19. University of Aarhus, Computer Science Department. Aarhus (Denmark), 1981.

Поступила в редакцию
10.07.2006