М.П. Глуханков, П.А. Дортман, А.А. Павлов, А.П. Стасенко

ТРАНСЛИРУЮЩИЕ КОМПОНЕНТЫ СИСТЕМЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ SFP*

1. ВВЕДЕНИЕ

В настоящее время для проведения крупных научных расчетов все чаще используются различные суперкомпьютеры, имеющие параллельную архитектуру. Как правило, эти мощные машины расположены в вычислительных центрах, к которым программисты имеют доступ преимущественно по сети через FTP и Telnet-протоколы. Использование суперкомпьютера для отладки программ не является эффективным, к тому же вычислителю нет смысла привязываться к какой-то одной системе.

В Институте Систем Информатики СО РАН создается система функционального программирования SFP. Эта система позволит прикладным программистам создавать и отлаживать параллельные функциональные программы на обычном персональном компьютере, которые затем можно будет исполнять на супервычислителе. Более того, функциональный язык высокого уровня Sisal версии 3.0 [34], используемый в этой системе, позволяет в функциональные программы включать вставки на императивных языках. В настоящее время в SFP реализуется возможность вызова функций, написанных на языке Си. Язык Sisal предоставляет автоматическое управление параллелизмом, которое является результатом его функциональной семантики. В системе SFP Sisal-программы преобразуются в специально разработанные внутренние графовые представления, на которых проводятся различные оптимизирующие преобразования, производится интерпретация программ и которые транслируются в программы на языке Си. Эти представления хоть и называются внутренними, но есть возможность их просмотра при помощи системы Higres [32, 33]. Кроме того, предполагается создание специальной системы визуализации для предоставления программисту возможности просмотра процесса исполнения программы во время её интерпретации, а также управления процессом оптимизации.

---

* Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 01-01-794) и Министерства образования РФ.

В настоящей работе рассматриваются уже разработанные важнейшие части системы SFP, а именно: основанное на иерархических графах внутреннее представление программ, удобное для анализа и автоматического распараллеливания, блоки для синтаксического и лексического анализа, построения внутреннего представления, система тестирования SFP.

В разд. 2 представлена схема системы SFP, рассмотрено, какие этапы преобразований проходит исходная программа в этой системе. Разд. 3 кратко знакомит с языком программирования Sisal 3.0. Разд. 4 посвящен внутреннему представлению программ в системе. Разд. 5–8 представляют некоторые составляющие части системы. В разд. 9–12 рассматриваются проблемы тестирования системы SFP. И, наконец, в заключении представлены направления будущей работы.

2. СХЕМА РАБОТЫ СИСТЕМЫ SFP

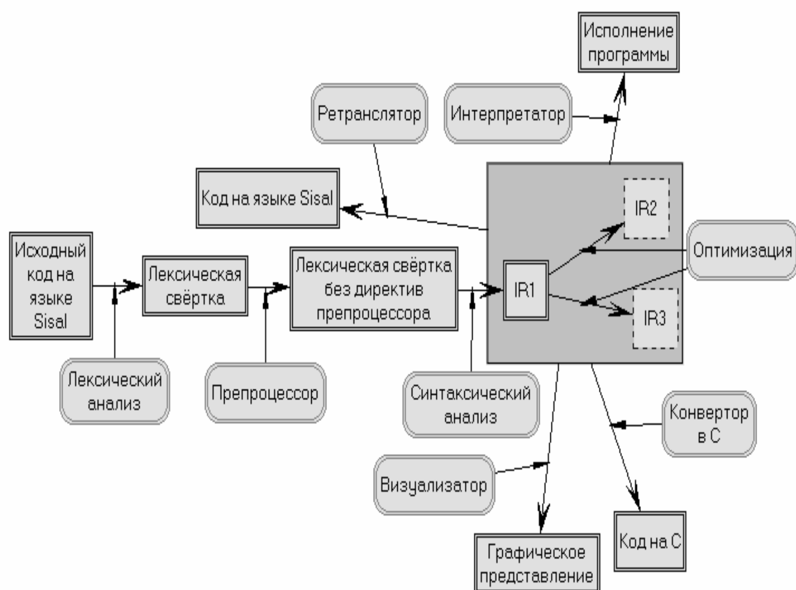

Рис. 1

На рис. 1 показаны основные части системы и их взаимодействие в процессе работы. Здесь прямоугольные вершины символизируют различные

уровни представления программ, т.е. то, с чем работают части системы. Закругленные вершины обозначают части системы, в результате работы которых представление программ переходит в новое состояние. Стрелочки между прямоугольными вершинами обозначают переход из одного состояния в другое, стрелочки, ведущие от закругленных вершин, обозначают, что соответствующий переход происходит под действием соответствующей функциональной части системы. Жирными рамками выделены те части и объекты системы, которые рассмотрены в настоящей работе.

Отметим, что вся работа программиста с системой происходит с помощью графического пользовательского интерфейса.

Рассмотрим жизненный цикл Sisal-программы в системе более подробно. Пользуясь текстовым редактором интерфейсной части, пользователь создаёт или загружает текстовый файл с исходным кодом. При попытке пользователя транслировать эту программу, под действием лексического анализатора происходит лексический анализ программы, и как результат в системе появляется лексическая свертка программы. Если в программе использовались директивы препроцессора, то они исполняются препроцессором уже на уровне лексической свертки, таким образом, после обработки препроцессором мы получаем лексическую свертку без директив препроцессора.

Внутреннее представление программ является важнейшим представлением программ в системе. Его напрямую использует большинство частей системы. Предполагается наличие трех видов внутреннего представления — IR1, IR2 и IR3. Отметим, что реально на настоящий момент разработано только IR1 представление, IR2 и IR3 будут разработаны позже. Представление IR1 служит для представления Sisal-программы в виде иерархического ациклического графа, представляющего поток данных в программе. В этом графе вершины соответствуют операциям, а дуги определяют передачу данных между вершинами, каждая дуга несет значение вполне определенного типа. Подробнее о внутреннем представлении см. разд. 4. Представление IR2 похоже на представление IR1, но в него добавлено распределение данных по памяти. Подобное представление IF2 описано в работе [28]. Над IR2 можно будет производить оптимизацию программ по памяти. Представление IR3 — это представление для императивных программ.

Система работает с внутренним представлением в зависимости от инструкций пользователя. Программист может просматривать внутреннее представление, используя систему визуализации графов Higres. Более того, используя графическое представление, пользователь может даже интерактивно изменять IR1-программу. Также пользователь может отлаживать IR1-

программу в режиме транслятора, и в целях отладки программа может быть ретранслирована из внутреннего представления в Sisal-программу. И, наконец, IR1-программа может быть переведена в программу на языке Си для компиляции и исполнения на суперкомпьютере.

3. ЯЗЫК SISAL 3.0

Sisal — язык функционального программирования, разработанный к 1983 г. в результате сотрудничества Манчестерского Университета, Ливерморской национальной лаборатории (США), Университета штата Колорадо и Digital Equipment Corporation [9]. В 1985 г. появилась версия 1.2 языка [10], для нее существует работающий оптимизирующий компилятор (OSC). В 1991 г. Ливерморская лаборатория и университет Колорадо опубликовали версию 2.0 языка [12, 14], которая после модернизации в 1995 г. получила название Sisal-90 [16, 17, 23], но так и не была никем реализована. В 2001 г. была представлена версия 3.0 языка Sisal [34]. В этой версии введены некоторые новые синтаксические конструкции, которые вносят в язык Sisal новые возможности, как, например, поддержка многоязыкового программирования (можно встраивать функции, написанные на языке Си), появились модульность и глобальные переменные.

Семантика языка Sisal имеет ряд важных свойств, которые благоприятствуют анализу и доказательству корректности программ. Во-первых, имена прозрачны для ссылок, имена переменных олицетворяют значения, а не ячейки памяти, а также нет прямой работы с памятью. Во-вторых, Sisal — язык однократного присваивания. В версиях языка до 2.0 включительно Sisal был математически правильным языком, т.е. функции не имели побочных эффектов. Но в двух последних версиях добавлены глобальные переменные. Версия Sisal 3.0 содержит такие конструкции, как функции высшего порядка, множества типов, массивы, потоки, союзы, записи, комплексные числа, включает в себя векторные операции над массивами, семантика которых аналогична семантике векторных операций языка Фортран-90. Кроме того, программы могут быть разбиты на модули.

4. ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ ПРОГРАММ (IR)

Внутреннее представление, являющееся системой Си++ классов, основано на представлении Sisal-программ в виде множества ациклических графов, с использованием идеи IF1 представления [26, 27]. Такие графы (точ-

нее было бы назвать их графовыми объектами) состоят из 4-х видов компонент: вершин, дуг, типов и границ графа. Вершины обозначают операции, такие как сложение и деление, дуги символизируют значения, передающиеся от вершины к вершине, типы могут быть закреплены за каждой дугой. Границы графа окружают группы вершин и дуг. Например, граф для выражения (X*X+Y*Y) представлен, как это показано на рис. 2.

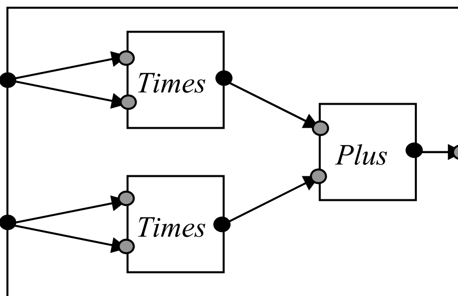

Рис. 2

Внешний прямоугольник обозначает границу графа. Этот граф имеет два порта для входных значений (*x* и *y*) и один выходной порт для результата. Прямоугольники внутри графа обозначают вершины. Все операции в примере имеют два входа и один выход, но, вообще говоря, число входящих и выходящих портов зависит от операции. Например, операция для построения массива может брать любое количество входных значений. Порты для каждой операции нумеруются и порядок нумерации важен.

Стрелочки в примере обозначают дуги. Дуги символизируют данные, передаваемые между вершинами (или между вершинами и границей графа). Дуги в рассматриваемом примере несут значения *x* и *y* внутрь графа, передают их квадраты из вершин *Times* в вершину *Plus* и отправляют результат за пределы графа. Дуги также несут информацию о типах данных, передаваемых по ним, но это не отражено на рисунке.

Каждую функцию программы мы пытаемся представить таким IR1-графом, и семантика языка Sisal способствует этому. Но простых вершин не всегда достаточно, для представления сложных языковых конструкций приходится использовать составные вершины.

Составных вершин не так много. Всего используются 5 типов составных вершин: для реализации альтернативы, выбора элемента из объединения и три типа для реализации различных циклов.

Таким образом, программа на языке Sisal представляется в виде конечного множества иерархических графов. Представление конкретной программы в виде IR1-графа мы называем IR1-программой.

Представление IR1 реализовано с помощью системы классов на языке Си++. Объектная модель очень удобна для представления таких графов. Реализованы интерфейсы для построения графов IR1, а также различные интерфейсы, необходимые для реализации анализа внутреннего представления и проведения над ним оптимизирующих преобразований. Объекты IR1 содержат много дополнительной информации для отладки, ретрансляции, интерпретации, оптимизации и других операций. За счёт добавления дополнительных интерфейсов объектная реализация может быть легко расширена до более сложной, позволяющей решать новые задачи без изменений уже реализованных частей системы.

5. ТРАНСЛЯТОР

В этом и следующих разделах описаны некоторые основные части системы SFP.

При рассмотрении трансляции языков функционального программирования существенным является этап синтаксического и семантического анализа, так как лексический анализ выполняется при помощи общих с анализом других языков методов.

Рассматривая интересующие нас этапы трансляции программ на языке Sisal 3.0, можно выделить несколько характерных особенностей его трансляции.

Во-первых, трансляция осуществляется в специальное графовое, ориентированное на манипуляцию с параллельными потоками данных, внутреннее представление IR1, что позволяет в значительной степени сохранить наглядное соответствие между конструкциями языка и объектами внутреннего представления.

Во-вторых, глобальная вложенность синтаксических конструкций языка гарантирует нам, что любая его конструкция имеет возвращаемые значения, что, в свою очередь, даёт нам возможность организовать синтаксический разбор концептуально в виде набора функций, разбирающих конкретную конструкцию языка и возвращающих её результаты вызвавшей её конструкции. Это приводит к более структурированной организации исходного кода транслятора, что повышает его общую читаемость, возможность внесения изменений и исправления ошибок.

В-третьих, конструкции на языке Sisal 3.0 строго типизированы, что позволяет уже на этапе внутреннего представления присвоить каждому потоку данных его тип, что, однако, несколько усложняет этап семантического анализа, но значительно упрощает дальнейшие преобразования внутреннего представления.

В-четвёртых, стоит отметить, что язык Sisal 3.0 не является “чистым” с точки зрения функционального программирования, так как он поддерживает операции присваивания. Но это не создаёт дополнительных проблем ввиду того, что все присваивания являются однократными, т. е. мы не можем, к примеру, присвоить значение какому-то отдельному полю записи, а вынуждены определять запись, указывая все её поля. Операция присваивания реализована в Sisal 3.0 достаточно интересным способом. Она может появляться только внутри специальной синтаксической конструкции let, которая в одной своей части содержит список присваиваний, свойственный императивным языкам, а в другой — набор выражений, использующий имена, определенные в первой части конструкции, количество которых определяет её результирующую арность.

6. КОНВЕРТОР ИЗ ВНУТРЕННЕГО ПРЕДСТАВЛЕНИЯ ЯЗЫКА SISAL 3.0 В ЯЗЫК СИ

При разработке системы функционального программирования на базе языка Sisal 3.0 одной из задач была возможность получать выходной код на языке Си. Эта задача ставилась и в более ранних версиях систем программирования Sisal. Построение конвертора позволило добиться необходимой переносимости кода, обеспечивая его компиляцию практически на всех известных в настоящий момент платформах. Более того, наличие такого конвертора позволило достичь понимания (пусть и преобразованного кода языка Sisal) практически любым программистом. Конвертор получает на вход внутреннее представление IR1-программы на языке Sisal, которая уже прошла проверку на грамматическую и синтаксическую правильность. При этом конвертор начинает построение готовой программы, строя функцию main, а затем производя последовательный разбор и построение выходного кода для всех вложенных в функцию main выражений. Разбор других функций, находящихся на одном уровне с main, производится аналогично разбору main. В дальнейшем, при обработке тела любой из функций (в IR1 они представлены в виде сложных вершин, отвечающих за объемлющие функции) происходит переход к обработке этих вершин. При этом типы пере-

менных переводятся в стандартные типы языка Си, что облегчает переносимость. Перевод осуществляется не в максимально эффективный код на Си, а в максимально похожий на свой исходный текст на Sisal, удобный для понимания программисту (с этой целью максимально сокращено введение новых имен, которые затрудняют понимание, а вновь создаваемые имена имеют понятную привязку к именам исходных файлов и функций). Кроме того, предусмотрена возможность использования в качестве внутреннего представления не IR1, а уже подвергшихся оптимизации IR2. Возможно также использовать вставки модулей языка Си. Внутреннее представление такого модуля является вершиной, имеющей лишь входные и выходные дуги, внутренности такого модуля — черный ящик, раскрываемый лишь на этапе компиляции Си-программы.

Хотелось бы отметить некоторые проблемы, возникшие при разработке конвертора, обусловленные тем, что Sisal является функциональным языком: подстановки функций, функции сокращения, глобальные переменные. Также хотелось бы отметить проблему с вызовами функций, решение которой предусматривает введение неявных дуг у функций (контекста). Кроме того, в этом конверторе найдены решения тех проблем, которых в силу синтаксиса не было в предыдущих версиях языка Sisal, таких как передача глобальных переменных, а также использование при вызове функций контекста. Конвертор реализован на языке Си++ в виде иерархии классов, которые производят соответствующее конструирование программ на Си.

7. ОТЛАДЧИК

Средства отладки системы функционального программирования позволяют интерактивно отлаживать параллельные Sisal-программы. К средствам отладки относятся: графический визуализатор графового внутреннего представления программ, ретранслятор внутреннего представления и сам отладчик. Все они интегрированы со средой разработки программ, легко запускаются и управляются из неё посредством нажатия кнопок на панели инструментов или запуска команд меню.

Отладчик представляет собой средство для пошагового исполнения Sisal-программ. Под шагом здесь понимается вычисление результирующих значений на всех выходных портах одной из вершин графового представления программы.

Отладчик позволяет просматривать промежуточные результаты вычислений на любом шаге, а также вести отчёт пошагового исполнения, в запи-

сях которого указываются вершины, их тип, соответствующие операторы, входные и выходные значения.

Перед началом каждого шага у пользователя (программиста) имеется возможность выбрать для него вершину среди тех вершин, у которых на всех входных портах вычислены значения. При этом выбор может быть осуществлён либо по средствам выбора вершины в окне графического визуализатора, либо с помощью выбора вершины из списка вершин-кандидатов, либо через указание соответствующего оператора в тексте программы. Для широкого использования последнего метода применяется ретранслятор внутреннего представления, так как после применения оптимизирующих преобразований соответствие между вершинами графового представления и исходной программой может быть утеряно.

Отладчик может выбирать вершину для вычисления на новом шаге и автоматически. При таком исполнении существует возможность визуализации процесса вычислений на графовом или текстовом представлении. Можно также просто просматривать типы вычисляемых операций, имена функций и результаты. Можно изменять скорость исполнения, указывая время простоя между шагами вычисления, устанавливать точки останова или указывать место в программе, до которого следует произвести вычисления и остановиться.

Есть возможность просмотра стека вызовов функций для каждой ветви исполнения.

Отладчик позволяет отлаживать программы до того, как написаны реализации некоторых функций. Если в программе встречается вызов такой функции, то отладчик может спросить результат её выполнения у программиста либо использовать значения по умолчанию. Этот же метод может быть использован для получения результатов вызовов функций, реализованных в других модулях или на других языках программирования.

Для запуска отладчика достаточно нажать одну из кнопок: “пошаговое исполнение программы с самого начала”, “исполнять до указанного места” или “исполнять программу в режиме отладки”. В последнем случае исполнение будет приостанавливаться в точках останова.

Во время интерпретирования отчёт об исполнении выводится в отдельное окно. Содержание отчёта программист может настраивать самостоятельно в зависимости от потребностей, нажав “кнопку настройка” отчёта. В окне настройки указывается: нужно ли выводить номер строки в программе для каждого выполняемого оператора, тип оператора, входные и выходные значения, имена функций в начале и в конце исполнения, список имён, для которых необходимо выводить значения, в момент их присваивания.

8. РЕТРАНСЛЯТОР

После построения транслятором IR1-представления, проведения над ним оптимизирующих преобразований (получения IR2 и IR3) может возникнуть потребность в восстановлении всего текста Sisal-программы или её участка по её графовому представлению. Для этой цели в SFP предусмотрена операция ретрансляции, которую выполняет ретранслятор. Данный инструмент может использоваться, например, при отладке Sisal-программы.

Входными данными для ретранслятора могут быть графы внутренних представлений IR1, IR2 или IR3. Результатом работы ретранслятора должен являться текст программы на языке Sisal, при повторной трансляции которой будет получено внутреннее представление эквивалентное исходному.

Структуры внутреннего представления делятся на две группы. Первая группа — это структуры, для которых есть полностью соответствующие операции языка. Вторая группа — структуры, для которых в языке нет аналогичных. Первые при ретрансляции переводятся непосредственно в свои аналоги, а вторые — в более сложные конструкции.

При ретрансляции возникает проблема именования выражений. Во-первых, во внутреннем представлении значения передаются между вершинами по дугам, и неизвестно, нужно ли именовать значение и, если нужно, то какое имя выбрать. Во-вторых, хотелось бы, чтобы отретранслированная программа была максимально похожа на исходную, а значит, имена в них должны по возможности совпадать. В-третьих, после проведения оптимизирующих преобразований соответствие с исходной программой может быть утеряно, и будет необходимо отказаться от одних имен, а другие — добавить.

В SFP для решения этой задачи во внутреннем представлении все объекты могут иметь имена. Транслятор задаёт эти имена в соответствии с исходной программой, а ретранслятор может использовать их для именования выражений в отретранслированной программе. Ретранслятор предоставляет настройку именования. Он может определить, каким выражениям необходимо дать имена, но при необходимости можно заставить его давать имена ещё и тем выражениям, у которых есть имена во внутреннем представлении. Можно также потребовать генерацию имён для абсолютно всех выражений. Ретранслятор либо генерирует имена, либо использует имеющиеся во внутреннем представлении. Для генерации имён применяется шаблон, который задаётся в окне настроек ретранслятора. Например, в имя функции можно добавить имя модуля, а в имя локального выражения можно включить имя функции.

Средство ретрансляции интегрировано со средой разработки программ, и ретрансляцию можно произвести в любой момент, если внутреннее представление уже построено, нажатием кнопки на панели инструментов. Во время отладки ретрансляция при необходимости выполняется автоматически (см. разд. 7).

9. ПОДХОД К ТЕСТИРОВАНИЮ SFP

Для тестирования транслятора языка Sisal 3.0 создана система тестов по методике первого покрытия [1–4]. В методике первого покрытия тесты проектируются на основе частично формализованной (ЧФ) модели текста документации. Основные положения метода можно сформулировать следующим образом.

Определим (неформально) комплект первого покрытия как комплект тестов, достаточно равномерно затрагивающий все аспекты и свойства языка программирования (ЯП). Простейший способ частичной формализации критерия (полноты) первого покрытия — потребовать, чтобы для каждого утверждения определенного вида, содержащегося в тексте документации, в комплекте нашелся хотя бы один нацеленный на проверку этого утверждения тест.

Это, конечно, весьма слабый критерий полноты (хотя бы потому, что проверяемые утверждения должны явно присутствовать в тексте документации, а не выводиться из него; к тому же каждое содержательное утверждение может допускать различные интерпретации в зависимости от контекста и т.п.).

Тем не менее опыт тестирования трансляторов показывает, что даже такой слабый критерий при применении к сложным ЯП требует, во-первых, серьезных затрат на создание комплекта, во-вторых, специальной организации работы, и за счет недостатков современной технологии создания трансляторов на практике оказывается достаточным для формирования комплектов тестов, способных находить серьезные дефекты реализаций ЯП.

Суть метода в систематическом сканировании текста стандарта с целью создания и сопровождения следующих файлов.

1. ЧФ-модель текста документации. Из текста документации выделяются и фиксируются в этой модели подлежащие проверке утверждения. Структура модели соответствует структуре документации. Такая модель играет роль перечня подозрений; по сравнению с

полным текстом документации она существенно короче и содержит лишь необходимую для данного вида тестирования информацию.

2. Комплект тестов (текущая версия). Каждое подозрение, зафиксированное в первом файле, детализируется в виде группы тестов. Уровень детализации определяется тестовиком, однако, учитывая, что перечень подозрений удовлетворяет относительно слабому критерию полноты и не может быть существенно улучшен в рамках данной методики, рекомендуется придерживаться принципа "один тест на подозрение".

Созданная система тестов содержит как корректные тесты по отношению к проверяемой документации, так и тесты, некорректные по отношению к проверяемой документации (цель — обнаружение дефекта в реакции на неправильные входные данные). Для каждого теста определёны условия верного результата прогона.

10. АВТОМАТИЗАЦИИ В ТЕСТИРОВАНИИ

Для автоматизации создания тестов и их прогона созданы три программы. Одна из них позволяет разбивать файл с большим количеством текстов тестов на отдельные файлы. В первой строке файла должно находиться окончание названий файлов тестов, например *.sse* — точка и расширение имён файлов. Далее идут тексты тестов, но перед каждым тестом должна стоять строка-разделитель, начинающаяся с ////. За строкой-разделителем должна идти строка с именем файла для следующего теста. Полное имя файла для теста получается конкатенацией этой и первой строк. После имени теста идёт сам текст теста, который заканчивается либо строкой-разделителем, либо концом файла.

Например, пусть имеется файл all.txt, содержащий следующее:

.sse
//////
empty
////
2comment
%comment1
%comment2

Если дать команду *cutter.exe all.txt* , то будут созданы два файла *empty.sse* и *2comment.sse* , первый — пустой, а второй будет содержать две строки:

%comment1

%comment2

Здесь cutter.exe — это и есть первая программа — консольное приложение Win32.

Вторая программа позволяет быстро просматривать тексты файлов и редактировать их. Это текстовый редактор под Windows, очень похожий на блокнот, дополненный двумя кнопками на панели инструментов: предыдущий и следующий. Если нажать на одну из этих кнопок, то редактор откроет, соответственно, предыдущий или следующий файл, записанный в той же папке и с тем же расширением, что и текущий редактируемый файл. При этом, если в текст были внесены изменения, то редактор спросит, нужно ли сохранить его. Фактически программа позволяет “листать” файлы и редактировать их.

Третья программа позволяет автоматически прогонять большое количество тестов и может генерировать тесты сама по шаблонам, она описывается в следующей главе.

11. ПРОГОН ТЕСТОВ

Программа для автоматической генерации и прогона тестов имеет встроенный текстовый редактор для редактирования файла с текстом шаблона и информацией для прогона набора тестов (этот файл будем называть документом, программа позволяет указывать документ). То, куда будут помещаться сгенирированные тесты и откуда будут браться тесты для прогона, определяется именем файла документа. И в текущей рабочей папке используется папка с тем же именем (только без расширения), что и имя документа. Документ может содержать две части, которые не являются обязательными, но должны идти в тексте документа строго в порядке: информация для прогона, шаблон. Если информация для прогона отсутствует, то прогон не будет осуществляться. Если отсутствует шаблон, то не будет осуществляться генерация тестов.

Информация для прогона может состоять из одной или нескольких строк вида

<r>что угодно (комментарий)

либо

<e>PathName ext,time;

либо

<t> PathName ext,time,code;

Первый вид используется для комментариев. Второй — для запуска программы с полным именем PathName, и параметром — именем файла с тестом (имена автоматически читаются с диска). При этом расширение имени файла с тестом заменяется расширением ext. После запуска программы наша тестирующая программа ждёт time секунд и уничтожает процесс запущенной программы, если он не закончен.

Третий вид используется почти так же, как и второй, но дополнен code — кодом возврата тестируемой программы, который она должна возвратить, чтобы тест считался пройденным.

Прогон осуществляется следующим образом. Для каждого файла из каталога с именем документа и расширением имени, указанным в первой строке, начинающейся с <e> или <t>, запускается указанная в этой строке программа в соответствии с видом строки. После того как эта программа завершит своё выполнение или по истечении тайм-аута ее процесс уничтожится, запускается программа из следующей строки, начинающейся с <e> или <t>, и т. д. Прогон теста заканчивается, если встречается строка, начинающаяся не с <r>, <e> или <t>. Затем из папки выбирается следующий файл с расширением имени, указанным в первой строке, начинающейся с <e> или <t>, и т. д.

Во время прогона тестов автоматически создаётся файл отчёта о прогоне, где указываются: имя папки с тестами, список произошедших ошибок и, если тестирование прервано, то пометка в конце: “ Прервано пользователем”. При описании ошибки указывается имя отказавшей программы, имя теста и тип ошибки (либо код возврата, либо превышение времени исполнения).

Данным средством можно обрабатывать текст теста последовательно несколькими программами, подавая на вход следующей результат предыдущей. Например, можно одной программой производить препроцессирование, второй — лексическую свёртку, а третьей — синтаксический анализ.

12. ШАБЛОНЫ ТЕСТОВ

Часто при создании тестов нужно написать множество тестов, очень похожих друг на друга. Например, при тестировании лексического анализатора может понадобиться подстановка в текст тестовой программы различ-

ных символов или ключевых слов. Для автоматической генерации такого рода тестов разработаны шаблоны текстов. Они являются частью документа приложения, описанного в предыдущей главе.

Шаблон идёт сразу после строк, начинающихся с <r>, <e> или <t>. Шаблон здесь — это текст, в котором могут встречаться теги. Тег — это текст, заключённый в угловые скобки: < , >. Чтобы в шаблоне поставить знак < , не считающийся началом тега, его нужно дублировать. Текст тега описывает набор значений. Для каждого значения тега будет сгенирирован тест. Тег можно записать в следующем виде <name=’file_name’> . Здесь *name* это имя тега, которое вместе со знаком ‘=’ можно опустить, file name — имя файла, строки которого будут значениями тега. Если в шаблоне присутствует лишь один тег, то будет создано столько тестов, сколько строк в файле file_name этого тега, при этом каждый тест будет получен из текста шаблона заменой тега на одно его значение. Если же тегов несколько, то для всевозможных комбинаций значений тегов будут созданы тексты тестов.

Имя нужно тегу для того, чтобы можно было в нескольких местах текста использовать одно и то же значение. Если имя тега определено, то далее в шаблоне можно указать тег вида <name> . Для того чтобы описать имя тега, а описание не было включено в тексты тестов, нужно использовать тег вида <*name=’file_name’> . Имя тега можно переопределять, тогда далее по тексту будет использоваться самое последнее определение.

Тесты могут создаваться не только для всевозможных комбинаций значений тегов, но и для значений, которые записаны в файлах file_name в строках с одинаковыми номерами. Для такого одновременного изменения значений нескольких тегов нужно после определения первого тега определить второй в виде <name=|’file_name’> или <*name=|’file_name’> . Вообще можно определять и более одного «паралельного» тега.

Пример.

Пусть есть шаблон

```
Function <fname=’names.txt’>(returns boolean)
<’args1.txt’>+<arg2=|’args2.txt’><*res=|’results.txt’>+0*<’args3.txt’>=<res
>
End function %<fname>
```

и файлы:

Names.txt содержащий:

Main

Args1.txt содержащий:
11
12

Args2.txt содержащий:
2
1

Results.txt содержащий:
13
13

args3.txt содержащий:
1
2

Тогда будут сгенерированы файлы:

1.sse содержащий:
```
Function main(returns Boolean)
   11+2+0*1=13
end function %main
```

2.sse содержащий:
```
Function main(returns Boolean)
   12+1+0*1=13
end function %main
```

3.sse содержащий:
```
Function main(returns Boolean)
   11+2+0*2=13
end function %main
```

4.sse содержащий:
```
Function main(returns Boolean)
   12+1+0*2=13
end function %main
```

13. ЗАКЛЮЧЕНИЕ

На данный момент система SFP реализуется для работы на машинах под управлением системы Windows. Но коллектив разработчиков при реализации основных частей системы, таких как внутреннее представление, трансляторы, анализаторы, интерпретатор, кодогенератор использует только стандартные библиотеки Си++, поэтому в дальнейшем, при желании, основные компоненты системы можно будет перенести на другие платформы. Нужно будет реализовать в основном только интерфейс взаимодействия системы с пользователем.

В дальнейшем будет реализована возможность построения программ для супервычислителя (предполагается использовать RMI), а также расширение возможности встраивания текстов на других языках в программы на языке Sisal.

СПИСОК ЛИТЕРАТУРЫ

1. **Kaufman V., Pavlov M., Rybin S.** The Testing of Ada Compiler Diagnostics // ACM Ada Letters — 1993. — V.13, N4. — P. 71–76.
2. **Кауфман В.Ш., Рыбин С.И.** Методика контроля диагностики трансляторов // Программирование. — 1990. — № 4. — С.28–37.
3. **Рыбин С.И.** Методика тестирования диагностики нарушений контекстных условий в стандартизованных трансляторах: Автореф. дисс. канд. физ.-мат. наук. — М: МГУ, 1987.
4. **Гусляев А.М., Кауфман В.Ш., Рыбин С.И.** О проектировании диагностических тестов для Ада-трансляторов // Новые методы разработки проблемно-ориентированных программных систем. — Владивосток: ИАПУ ДВНЦ АН СССР, 1986. — С. 61–71.
5. **Allan S.J., Oldehoeft R.R.** Parallelism in SISAL: Exploiting the HEP architecture // Proc. of the 19th Hawaii Intern. Conf. on System Science. —1986, January. — P. 538–548.
6. **Bohm A., Sargeant J.** Efficient dataflow code generation for sisal // Tech. Rep. / University of Manchester — 1985.
7. **D.E. Culler et al.** Fine grain parallelism with minimal hardware support: A compiler-controlled Threaded Abstract Machine // Proc. of the 4th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems — April 1991.
8. **D. Cann.** Retire FORTRAN? a debate rekindled. — CACM — Aug. 1992. — Vol. 35, N 8 — P. 81–89.
9. **McGraw, J. R. et. al.** Sisal: Streams and iterations in a single assignment language, Language Reference Manual, Version 1.1 // Lawrence Livermore Nat. Lab. Manual M-146. — Livermore, CA — 1983.

10. **McGraw, J. R. et. al.** Sisal: Streams and iterations in a single assignment language, Language Reference Manual, Version 1.2 / Lawrence Livermore Nat. Lab. Manual M-146 (Rev. 1). — Livermore, CA 1985.
11. **Skedzielewski S. K., Welcome M. L.** Data flow graph optimization in IF1 // Lect. Notes Comput. Sci. — 1985. — Vol. 201. — P. 17–34.
12. **Cann D.C.** The optimizing SISAL compiler: Version 2.0. — Livermore — 1992. (Prepr. / Lawrence Livermore National Laboratory; UCRL-MA-110080).
13. **Cann D. C.** Compilation techniques for high performance high performance applicative compilation. — Colorado State, 1989. — (Tech. Rep. / Colorado State; NCS–89–108).
14. **Bohm A. P. W., Oldenhoeft R. R., Cann D. C., Feo J. T.** The SISAL 2.0 Reference Manual. — Livermore, CA, 1991. — (Prepr. / Lawrence Livermore Nat. Lab.; UCRL-MA-109098, LLNL).
15. **Евстигнеев В. А., Городняя Л. В., Густокашина Ю. В.** Язык функционального программирования SISAL // Интеллектуализация и качество программного обеспечения. — Новосибирск, 1994. — С. 21–42.
16. **Feo D. T., Miller P. J., Skedzielewski S. K., Denton S. M.** Sisal 90 User's Guide / Lawrence Livermore Nat. Lab. Draft 0.96. — Livermore, CA, 1995.
17. **Бирюкова Ю. В.** SISAL 90 руководство пользователя. — Новосибирск, 2000. — 84с. — (Препр./ Сиб. Отд-е. РАН ИСИ; № 72).
18. **Feo J. T.** Sisal. — Livermore, CA, 1992. — (Prepr. / Lawrence Livermore Nat. Lab.; UCRL-JC-110915, LLNL).
19. **Feo J. T., Cann D.C, Oldehoeft R.R.** A report on the Sisal language project // J. on Parallel and Distributed Computing. — 1990. — Vol. 10. — P. 349–366.
20. **McGraw J. R.** Parallel functional programming in Sisal: fictions, facts, and future. — Livermore, CA, 1993.— (Prepr. / Lawrence Livermore Nat. Lab.; LLNL).
21. **Kasyanov V. N., Evstigneev V.A. et al.** The system PROGRESS as a tool for parallelizing compiler prototyping // Proc. of Eighth SIAM Conf. on Parallel Processing for Scientific Computing (PPSC-97). — Minneapolis, 1997. — P. 301–306.
22. **Kasyanov V.N., Evstigneev V.A. et al.** Support tools for supercomputing and networking // Lect. Notes Comput. Sci. — 1999. — Vol.1593. — P. 431–434.
23. **Feo D. T., Miller P. J., Skedzielewski S. K., Denton S. M., Solomon C. J.** Sisal 90 // Proc. High Performance Functional Computing — Livermore, 1995. — P. 35–47.
24. **Трой Д.** Программирование на языке Си для персонального компьютера IBM PC / Пер. Б.А. Кузьмина под ред. И. В. Емелина. — М.: Радио и связь, 1991.
25. **Касьянов В. Н.** Оптимизирующие преобразования программ. — М.: Наука, 1988.
26. **Skedzielewski S. K., Glauert J.** IF1 — An intermediate form for applicative languages. Manual M-170 / Lawrence Livermore National Laboratory — Livermore, CA, 1985.
27. **Густокашина Ю.В., Евстигнеев В.А.** IF1 — промежуточное представление Sisal-программ // Проблемы конструирования эффективных и надежных программ. — Новосибирск, 1995. — С. 70–78.

28. **Welcome M., Skedzielewski S., Yates R.K., Ranelletti J.** IF2 — An applicative language intermediate form with explicit memory management / Lawrence Livermore Nat. Lab. Manual M-195. — Livermore, CA, 1986.
29. **Miller P. J.** TWINE: a portable, extensible SISAL execution kernel // Proc. Second SISAL User's Conf. — Livermore, 1992. — P. 243–256.
30. **Ranelletti J. E.** Graph transformation algorithms for array memory optimization in applicative languages. — Livermore, CA, 1987. — (Prepr. / Lawrence Livermore National Laboratory; UCRL-53832).
31. **Li Z., Kirkham C.** Efficient implementation of aggregates in united functions and obiects. — University of Manchester, 1995.
32. **Лисицын И.А.** Применение системы HIGRES для визуальной обработки иерархических графовых моделей // Проблемы систем информатики и программирования. — Новосибирск, 1999. — С. 64–77.
33. **Система** Higres. — Доступна по адресу: http://lis.iis.nsk.su/higres.
34. **Касьянов В.Н, Бирюкова Ю.В., Евстигнеев В.А.** Функциональный язык Sisal 3.0 // Поддержка супервычислений и интернет-ориентированные технологии. — Новосибирск, 2001. — С. 54–67.