МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

РОСТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра алгебры и дискретной математики

ДИПЛОМНАЯ РАБОТА

студента 5-го курса 1-ой группы

Петренко Виктора Владимировича

НОВОЕ ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ
ОТКРЫТОЙ РАСПАРАЛЛЕЛИВАЮЩЕЙ СИСТЕМЫ

Научный руководитель:
доц. каф. алгебры и
дискретной математики, к.ф.-м.н., с.н.с. Штейнберг Б. Я.

Рецензент:
доц. каф. информатики и
вычислительного эксперимента, к.т.н. Крицкий С. П.

г. Ростов-на-Дону

2005

Оглавление

1. Введение ... 3
1.1. Старое внутреннее представление ОРС ... 5
2. Структура нового внутреннего представления ОРС ... 7
2.1. Дерево типов данных ... 9
2.2. Дерево идентификаторов ... 14
2.3. Дерево выражений ... 18
2.4. Дерево операторов ... 20
2.5. Итераторы и ссылки ... 23
2.6. Пометки узлов ... 24
2.7. Средства проверки и отладки ... 25
3. Представление свойств языков программирования во внутреннем представлении ... 27
4. Графы ОРС и библиотека преобразований программ, их взаимодействие с внутренним представлением ... 29
5. Визуализация внутреннего представления ... 31
6. Обзор внутренних представлений аналогичных систем ... 34
6.1. Polaris ... 34
6.2. SUIF ... 35
7. Заключение ... 37
8. Литература ... 38

1. Введение

В данной работе представлена основа программной реализации Открытой распараллеливающей системы – новое внутреннее представление.

Открытая распараллеливающая система (ОРС) – проект, разрабатываемый на кафедре Алгебры и дискретной математики группой студентов и аспирантов. Она предназначена для автоматического распараллеливания программ с процедурных языков программирования (Фортран, Паскаль, Си) на параллельные компьютеры, ориентированные на математические вычисления [1].

Внутреннее представление (ВП) – это структура данных для хранения информации о программе, а также алгоритмы, облегчающие выполнение преобразований программ [2]. Новое ВП ОРС удовлетворяет следующим требованиям.

- Является универсальной структурой данных для хранения программ процедурных языков (рассматривались Си, Фортран и Паскаль).
- Выполнено в виде объектно-ориентированной модели, которая служит базисом для написания преобразований программ. При этом похожие свойства процедурных языков скрыты за единым интерфейсом, а специфичные сохранены для этапа выполнения преобразований.
- Эта модель также предоставляет базис для построения и анализа информационных зависимостей в программе.
- Предусматривает расширяемость для подключения компиляторов переднего плана с новых языков программирования к ОРС.

- Предоставляет возможность для генерации машинного кода из внутреннего представления.

Первоначальное внутреннее представление ОРС было разработано несколько лет назад [3]. Был написан специальный генератор компиляторов, разбирающий программы с Фортрана, Паскаля и Си. К сожалению, в той работе основной акцент был сделан на разбор языков, а адаптация внутреннего представления для реализации преобразований и анализа информационных зависимостей в программе выполнялась на поздних стадиях реализации проекта. Поэтому, старое внутреннее представление предоставляет недостаточные средства для реализации преобразований.

При разработке нового внутреннего представления изначально ставилась задача создания эффективного базиса для реализации преобразований и программы построения графов информационных зависимостей.

При реализации проекта были выполнены следующие работы:

- Спроектировано новое универсальное внутреннее представление программ и реализовано в виде классов на языке C++.
- Проведены работы по интеграции внутреннего представления и нового компилятора переднего плана с языка Си, основанного на инструменте ANTLR [5].
- Разработан механизм передачи информации между различными частями ОРС в виде специальных пометок в узлах внутреннего представления.
- Реализованы отладочные средства и средства проверки целостности структуры внутреннего представления во время работы программы.

- Разработан инструмент для визуализации и дополнительных преобразований фрагментов программ во внутреннем представлении.

Программная часть проекта написана на языке C++. При проектировании использовались «паттерны проектирования» [10], приемы современного проектирования на C++ [11] и приемы классического объектно-ориентированного анализа [12]. Программная реализация ОРС на момент написания этой работы составляет ~91000 строк, из них кода сторонних библиотек (ANTLR, Advanced WTL Controls) ~29000. Внутреннее представление ~9000 строк.

1.1. Старое внутреннее представление ОРС

Старое внутреннее представление ОРС было разработано в 2002 году [3], на раннем этапе развития Открытой распараллеливающей системы. В то время был реализован собственный генератор компиляторов, который разбирал программы с сильно сокращенных подмножеств языков Си, Паскаль и Фортран.

Основным требованием к старому внутреннему представлению ОРС была быстрая реализация, позволяющая начать разработку преобразований. Среди недостатков старого внутреннего представления следует отметить:

1. Отсутствие поддержки пользовательских типов данных, а также структур в языках программирования (struct в Си и record в Паскале). Не существовало понятия «дерево типов».
2. Недостаточное сведение общих свойств языков в общий базис.
3. Отсутствие адекватного механизма пометок узлов внутреннего представления.
4. Недостаточное использование статической типизации языка C++ для поддержки целостности, синтаксической и семантической корректности программы.

Вначале необходимо отметить, что при создании генератора компиляторов и как следствия – старого внутреннего представления, была решена сложная задача, спроектировано представление, хранящее программы на языках Си, Паскаль и Фортран. Это позволило начать реализовывать преобразования. На старом внутреннем представлении было реализовано девять распараллеливающих преобразований [2], в том числе два – «Введение временных массивов» и «Растягивание скаляров» автором. Появившийся в результате выполнения этой работы опыт, позволил обнаружить приведенные выше недостатки старого ВП, и прийти к пониманию каким должно быть новое ВП.

Остановимся на выделенных недостатках подробнее. Старое внутреннее представление содержало существенное ограничение на входные языки – отсутствие возможности описать пользовательские типы данных. Это ограничение, например, не позволяло разработать конвертеры в программу с MPI-инструкциями.

В старом внутреннем представлении не была закончена работа по сведению сходных конструкций языков в общий базис. Так, например, в старом ВП имеется оператор FortranDO, который может быть сведен к оператору FOR таких языков как Си и Паскаль.

Также в старом ВП отсутствовал механизм пометки узлов. Хотя для операторов цикла было введено целое число, биты которого указывали наличие или отсутствие какого-то свойства, в общем, такой подход неудачен, негибок и трудно расширяем.

Старое внутреннее представление недостаточно использовало возможности языка C++ по статическому контролю типов данных. Так например, все циклы в старом ВП были представлены как объекты класса OPERATOR и целочисленное поле указывало какой именно оператор представляет данный объект.

2. Структура нового внутреннего представления ОРС

Новое внутреннее представление Открытой распараллеливающей системы является композицией из четырех деревьев: типов, идентификаторов, выражений и операторов.

За основу для конструирования внутреннего представления взят паттерн «Composite». Основная идея заключается в следующем. Записывается базовый класс[1] для всех типов узлов, встречающихся в дереве. Этот класс может содержать общие для всех типов узлов свойства и методы. Например, если узлы дерева представляют идентификаторы в программе, то в базовый класс удобно поместить строковое представление (имя) идентификатора, потому что каждый узел характеризуется таким именем. Такой базовый класс называется *компонентом*.

Следом за базовым классом определяются классы, представляющие узлы дерева. Эти классы связаны с базовым отношением наследования. Классы, представляющие узлы дерева могут ссылаться на другие узлы дерева (как, например, в двусвязном списке одно звено ссылается на левого и правого соседа). Узлы, которые не имеют ссылок на другие узлы, как обычно, будем называть *листьями*.

Каждое из четырех деревьев ВП хранит информацию о какой-то части программы. Дерево типов содержит информацию о типах в языке программирования. Дерево идентификаторов – об именах в программе, их распределение по областям видимости. Дерево выражений представляет выражения в программе, узлы – операции и операнды. Дерево операторов представляет операторы в программе.

Прежде чем переходить к рассмотрению деревьев, следует упомянуть о способе конструирования узлов. Для этой цели применяется паттерн «Factory» [10]. Поскольку язык C++ не содержит сборщик мусора,

---

[1] Речь идет о классах объектно-ориентированного языка программирования, например, C++.

программист вынужден самостоятельно следить за освобождением памяти. При проектировании ВП было принято два важных решения:

1. Все объекты-узлы всех деревьев создаются в динамической памяти.
2. Узел при удалении ответственен за освобождение памяти всех своих поддеревьев.

Приведем некоторое обоснование. Хотя язык C++ позволяет создавать объекты в автоматической памяти (на стеке), что в некоторых случаях более эффективно, чем создание в динамической памяти, но создавать все узлы ВП в автоматической памяти, очевидно, невозможно из-за непродолжительного времени жизни объектов, созданных на стеке. Поэтому, при конструировании одного и того же дерева, память может выделяться как динамически, так и автоматически. Это очень сильно затрудняет корректное освобождение памяти, потому что приходится четко следить за временем жизни динамических объектов и каким-то образом помечать автоматические. Поэтому, несмотря на некоторую возможную неэффективность, было принято решение всюду использовать динамическое выделение памяти. В связи с этим, все классы, представляющие узлы имеют защищенный конструктор и открытые статические методы, возвращающие указатели на созданные узлы.

Теперь о стратегии освобождения памяти. На ранних этапах разработки нового ВП рассматривалась идея организации ссылок между узлами через «умные указатели» (Smart Pointers) [11]. Однако при практическом применении оказалось, что в данной ситуации использование таких указателей неэффективно. Во-первых, они создают перерасход памяти, во-вторых, при освобождении памяти могут возникать циклические ссылки между объектами, которые очень трудно отслеживать, а ошибки, связанные с их невнимательным применением крайне трудно находить. Таким образом,

было принято решение использовать простую и четкую стратегию освобождения узлов – каждый узел отвечает за удаление своих поддеревьев.

2.1. Дерево типов данных

Дерево типов используется как для представления простых типов данных (целый, символьный), так и для представления составных (массивы, структуры, перечисления). Используется одна и та же система классов для всех языков, хранящихся во внутреннем представлении.

На Рисунок 1. Схема наследования классов типов представлена диаграмма классов для дерева типов:

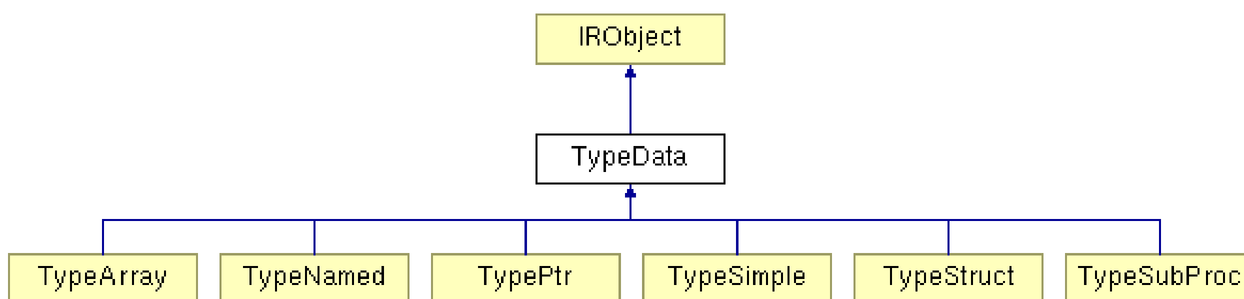

Рисунок 1. Схема наследования классов типов

Класс IRObject является корнем иерархии для большинства классов внутреннего представления. Основная его роль – хранить пометки узлов. Таким образом, каждый класс, который наследуется от IRObject, может содержать пометки.

Абстрактный класс TypeData является компонентом – корнем иерархии классов дерева типов. Он содержит чисто виртуальный метод[2] clone(). Этот метод создает и возвращает свою копию и копию всех своих поддеревьев («глубокую копию»). Операцию clone() можно определить рекурсивно: для узла-листа clone() создает копию этого узла. Для узла-композиции clone() проходит по всем ссылкам из этого узла, вызывая для них clone(), таким образом, получая копию всех поддеревьев, затем создает копию самого узла и «прицепляет» вновь созданные поддеревья.

[2] Чисто виртуальный метод (pure virtual method) – виртуальный метод, который не имеет реализации. Если класс содержит хотя бы один чисто виртуальный метод, его называют абстрактным классом. Невозможно создать объект абстрактного класса. Такой класс служит для представления понятия «интерфейс».

Теперь рассмотрим последовательно все узлы, наследованные от TypeData. Самый простой из них – TypeSimple. Этот узел представляет элементарные типы данных в языке программирования (целый, символьный, с плавающей запятой).

Внутри класса TypeSimple объявляется перечисление (enum), в котором перечислены все элементарные типы. Внутри класса имеется поле типа этого перечисления, которое и хранит информацию, какой конкретный элементарный тип представляет этот узел. Имеются методы для получения и установки типа узла: getType() и setType().

Точность элементарных типов данных реализовывается пометками в узлах TypeSimple. Компилятор переднего плана вставляет туда такую информацию как размер типа в байтах и необходимое выравнивание. В большинстве машинно-независимых распараллеливающих преобразований, реализованных и планируемых реализовать для ОРС, такая информация используется редко. Однако, она важна для машинно-зависимых преобразований и генерации кода.

Следующий рассматриваемый узел – TypePtr. Этот узел используется для представления типа «указатель на тип». Этот класс имеет поле, в котором хранится ссылка на поддерево, представляющее тип (называемый базовым), на который указывает этот узел. Имеются методы для получения и установки базового типа – getBaseType() и setBaseType(). При установке нового базового типа, дерево, соответствующее старому базовому типу автоматически удаляется.

Например, запись типа в программе на Си «int *» будет выглядеть следующим образом:

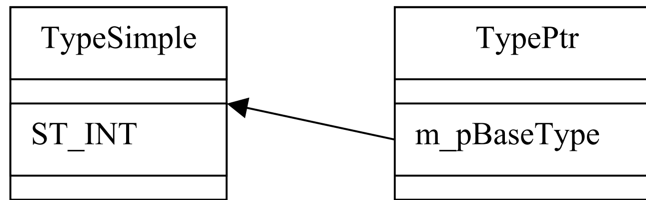

Рисунок 2. Представление «int *»

А для типа «int**» следующим образом:

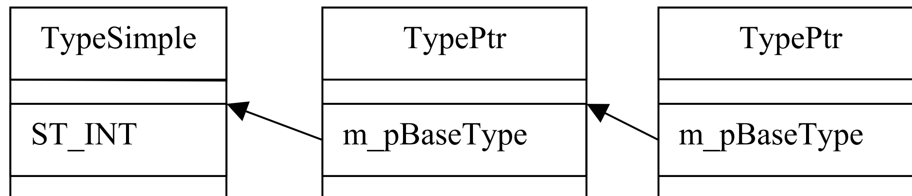

Рисунок 3. Представление «int **»

Следующий узел – TypeSubProc. Этот узел представляет тип «подпрограмма». Класс хранит информацию об аргументах подпрограммы, о типе возвращаемого значения, а также о модификаторах, которые использовались при объявлении подпрограммы (например, cdecl, inline). Для представления аргументов используется класс SubProcParams.

Класс SubProcParams представляет собой список аргументов подпрограммы. Каждый аргумент является экземпляром класса SubProcParam, поэтому для каждого аргумента хранится имя и информация о типе. Класс SubProcParams предоставляет методы для добавления, удаления и итерирования по списку аргументов.

Тип возвращаемого значения подпрограммы хранится точно так же, как базовый тип для узла-указателя на тип. Имеются методы для получения и установки типа возвращаемого значения getRetValueType() и setRetValueType().

Модификаторы подпрограммы реализованы при помощи шаблонного класса Modifiers. Этот класс является обобщенным контейнером для классов-

модификаторов, это означает, что он может быть использован для хранения информации практически о любых модификаторах. Имеется также класс ModifiersIter – это итератор по модификаторам. С помощью него можно осуществлять проход по списку модификаторов.

Следующий узел дерева типов – TypeArray. Этот класс предназначен для хранения информации о типах – массивах. Многомерные массивы также представляются этим узлом. В отличие от Си, где многомерные массивы представляются композицией одномерных, во внутреннем представлении ОРС, массивы представляются единым узлом, содержащим информацию обо всех измерениях, потому что этот способ хранения удобнее для многих распараллеливающих преобразований. Для каждого измерения известна нижняя и верхняя граница индекса (позволяет хранить полную информацию о массивах в Паскале). Для этого внутри класса массива определена структура Limits, которая имеет два поля: low и high.

Пример определения массива. Пусть имеется следующий фрагмент программы на Паскале:

```
type
      T = array[0..1][1..8] of int;
```

Во внутреннем представлении эта запись будет выглядеть следующим образом:

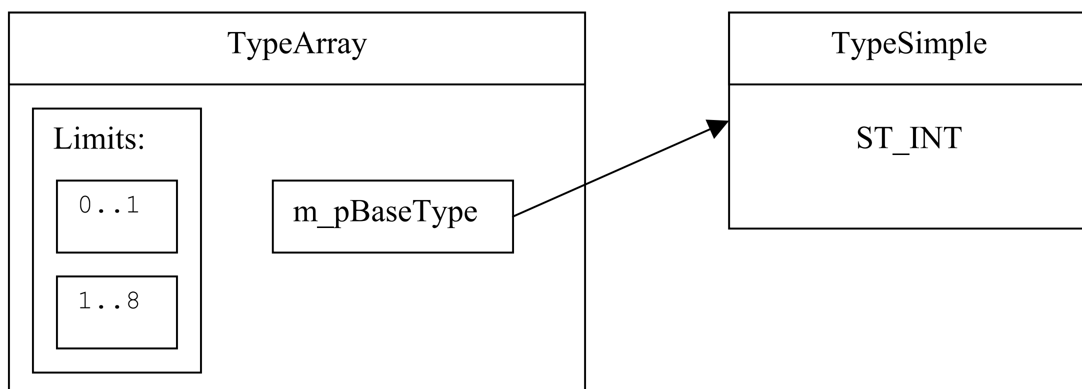

Рисунок 4. Представление массива

Класс TypeArray содержит все необходимые методы для работы с массивом. Для установки и получения базового типа массива: getBaseType() и setBaseType(). Для получения количества измерений массива – getDimsCount(). Для получения информации о каждом измерении, перегружен оператор [], который по номеру измерения возвращает ссылку на структуру Limits, соответствующую данному измерению.

Класс TypeStruct хранит информацию о структуре (struct) или объединении (union). Этот класс является контейнером для элементов структуры. Каждый элемент структуры описывается классом NameType. Этот класс будет рассмотрен позже в главе «Дерево идентификаторов», здесь же следует сказать, что этот класс содержит имя и информацию о типе элемента.

Класс TypeStruct предоставляет методы для добавления, вставки, удаления и итерирования по элементам структуры. Булевое поле m_bIsUnion внутри класса указывает, является ли это описание описанием структуры или объединения.

Последний узел дерева типов – TypeNamed. Этот класс используется для представления именованных типов. Т.е. типов, которые определены в некотором месте программы и ссылка на них необходима в другом месте. Например, пусть имеется следующий фрагмент программы на Паскале:

```
type
      T = integer;
      TPtr = ^T;
```

Класс TypeNamed имеет внутри два поля. Первое указывает на область видимости, в которой расположен именованный тип, а второе – имя этого типа.

Во внутреннем представлении эта запись будет выглядеть следующим образом:

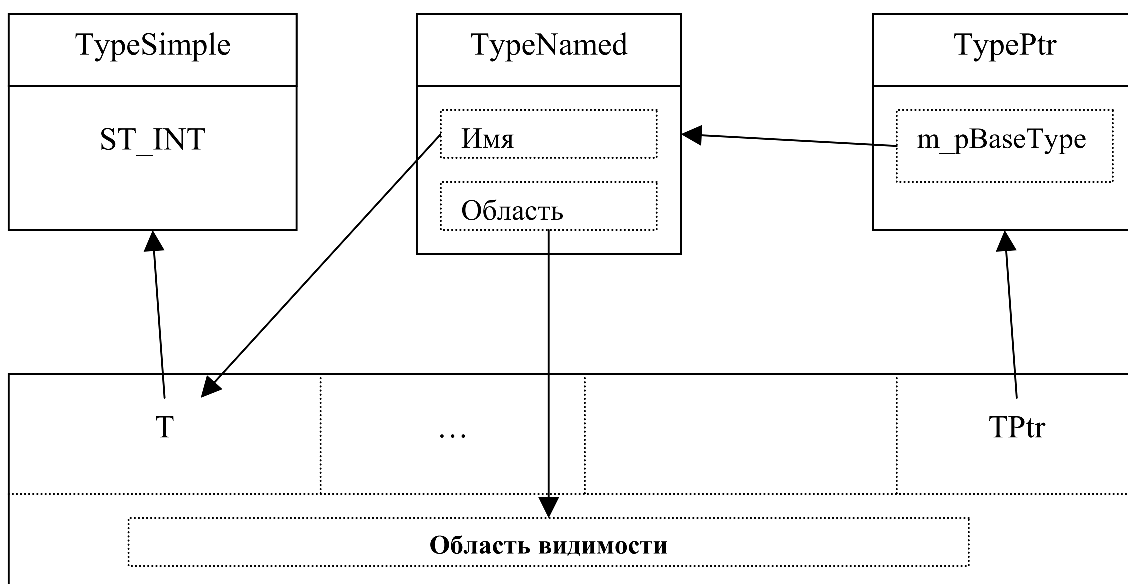

Рисунок 5. Представление ссылки на именованный тип

В области видимости содержатся два именованных типа: T и TPtr. Из схемы видно, что имя типа T связано с объектном класса TypeSimple, а ST_INT указывается что тип – целое число. TPtr является указателем (TypePtr), а тип на который он указывает – T. Чтобы сослаться на уже определенный тип и используется класс TypeNamed. Создается объект этого класса, «Имя» в нем содержит имя типа - «T», а «Область» в нем указывает на область видимости, в котором определен тип «T».

Таким образом, дерево типов позволяет хранить любые типы данных языков Си и Паскаль, в том числе пользовательские типы данных. Это возможно благодаря составлению композиции из блоков шести описанных выше разновидностей.

2.2. Дерево идентификаторов

Теперь рассмотрим дерево идентификаторов. Узлы этого дерева хранят информацию о поименованных сущностях в программе, таких как: переменные, типы данных, метки и подпрограммы.

На Рисунок 6. Схема наследования классов идентификаторовпредставлена диаграмма наследования классов идентификаторов:

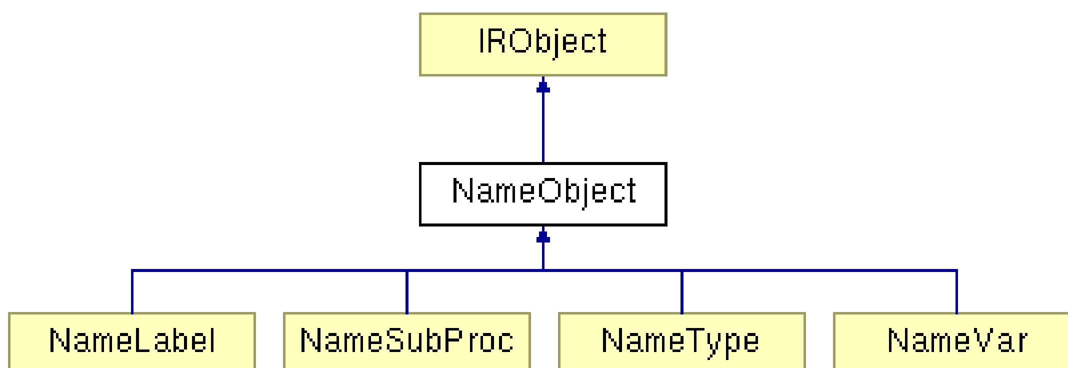

Рисунок 6. Схема наследования классов идентификаторов

Базовым классом для всех классов-узлов является класс NameObject. Он содержит общее для всех, наследованных от него классов свойство – имя идентификатора. Имя представляется обычной текстовой строкой. Класс также содержит метод getName(), который позволяет получить имя идентификатора.

Рассмотрим класс NameVar, он хранит информацию о объявленной в программе переменной. В этом классе содержится тип переменной – объект обобщенного класса TypeData. Это позволяет представлять переменные любого типа. Также в классе содержится булев признак, является ли переменная константной (модификатор const в языке Си), а также является ли переменная аргументом функции. Следует пояснить, как представляются аргументы функции во внутреннем представлении.

Если речь идет не о переменной из глобальной области видимости (глобальной переменной), то переменная является либо локальной для некоторой подпрограммы, либо параметром этой подпрограммы. Параметры подпрограммы во многом схожи с локальными переменными, поэтому также представлены классом NameVar. Однако, в случае необходимости, метод isArgument() позволяет определить является ли объект класса NameVar локальной переменной или аргументом подпрограммы. Такая проверка используется, например, при выводе внутреннего представления в Си. Для

локальных переменных необходимо генерировать их определение, в то время как информация о типах параметров записывается как типы аргументов подпрограммы.

Также при создании объекта класса NameVar имеется возможность указать его значение. Это необходимо для задания значения константных переменных в Си и типизированных переменных в Паскале.

Рассмотрим следующий класс дерева идентификаторов – NameType. Этот класс хранит информацию о поименованном типе данных: typedef в Си и раздел type в Паскале. В классе содержится объект обобщенного класса TypeData, что позволяет создавать именованные типы любой сложности в рамках дерева типов.

Следующий класс дерева идентификаторов – NameLabel. Этот класс содержит информацию о метках в программе: не используется явное определение в Си; label в Паскале. Он содержит указатель на оператор, который помечен этой меткой. Имеются методы getLabelStatement() и setLabelStatement() соответственно для получения и установки помеченного оператора.

Последний класс дерева идентификаторов – NameSubProc. Он представляет подпрограмму: функцию языка Си; procedure, function в Паскале. Класс содержит тип подпрограммы (указатель на TypeSubProc), а также опционально ссылку на тело подпрограммы (указатель на Block).

Необходимо заметить, что определение именованной подпрограммы может быть полным и неполным. Неполное определение – это «прототип функции» в языке Си и предварительное описание процедуры или функции в Паскале. Неполное определение, в отличие от полного, не содержит тело подпрограммы. Предполагается, что после неполного определения, в программе где-то далее встретится полное. Однако, поскольку фактический поиск тела подпрограммы может быть отложен до фазы работы линкера

(редактора связей), поддержка неполных определений необходима во внутреннем представлении.

В классе NameSubProc имеется также важный метод syncNamespace(), который необходимо рассмотреть подробнее. Как было сказано ранее, локальные переменные и параметры подпрограммы представлены классом NameVar. Однако, NameSubProc содержит тип подпрограммы, который определяет типы и название аргументов подпрограммы. Фактически, информация о параметрах подпрограммы хранится в двух местах – в прототипе и в виде объектов NameVar. Эта двойственность является вынужденной и оправдана схожестью для преобразований программ локальных переменных и параметров подпрограмм. Метод syncNamespace() используется при создании объекта класса NameSubProc для полного (вместе с телом) описания подпрограммы. syncNamespace() необходимо вызвать после задания прототипа подпрограммы, чтобы в область видимости тела подпрограммы были добавлены объекты класса NameVar для представления параметров подпрограммы. Также этот метод необходимо вызывать после каждого изменения в прототипе, чтобы соответствующие изменения были отражены в объектах NameVar.

Наконец, следует рассмотреть класс Namespace. Он представляет «ярус» области видимости идентификаторов, и хранит обобщенный список NameObject-ов, т.е. идентификаторов, объявленных в текущей области видимости.

Один объект класса Namespace связывается с объектом класса Program и является глобальной областью видимости программы. Также с каждым блоком (Block) ассоциируется объект класса Namespace и представляет локальную область видимости блока. В классе Namespace имеются методы для поиска, рекурсивного поиска (вверх по дереву идентификаторов), итерирования, удаления и добавления объектов области видимости.

2.3. Дерево выражений

Базовым классом для дерева выражений является класс ExprNode. В нем определены абстрактные методы isLValue (проверка, является ли выражение l-value) и isEqual (проверка выражений на структурную эквивалентность). Диаграмма наследования классов дерева выражений приведена на Рисунок 7. Схема наследования классов выражений:

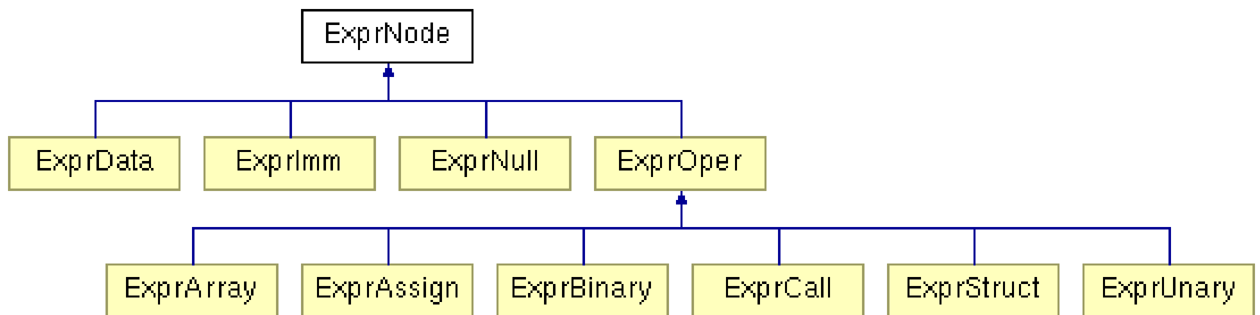

Рисунок 7. Схема наследования классов выражений

Из диаграммы видно, что классы ExprData, ExprImm, ExprNull и ExprOper являются непосредственными наследниками ExprNode. Рассмотрим назначение и структуру этих классов.

Класс ExprData представляет понятие вхождения переменной в выражение. Внутри он хранит NameReference, ссылку на идентификатор из области видимости. Класс реализует метод getData() для получения этого идентификатора.

Класс ExprImm представляет литерал, т.е. константу, записанную явно в коде программы. Для представления значения используется класс ImmValue, который способен хранить данное одного из следующих типов: int, double, char, string.

Класс ExprNull представляет понятие пустого выражения. Он используется при записи операторов в полном виде, например, если при конструировании оператора IF не задано условное выражение, объект класса ExprNull становится условным выражением этого оператора IF.

Класс ExprOper представляет операцию в выражении. Внутри класса определено перечисление (enum) со всеми возможными операциями:

```
// Unary
OT_UNPLUS,        // Unary +
OT_UNMINUS,       // Unary -

OT_UNPOSTPP,      // Unary postfix ++
OT_UNPREFPP,      // Unary prefix  ++
OT_UNPOSTMM,      // Unary postfix --
OT_UNPREFMM,      // Unary prefix  --

OT_SIZEOF,        // sizeof() operator

OT_ADDR,          // &
OT_UNREF,         // *

// Binary
OT_PLUS,          // +
OT_MINUS,         // -
OT_MUL,           // *
OT_DIV,           // /
OT_MOD,           // %

// Assign
OT_ASSIGN,        // =

OT_PLUS_ASSIGN,     // +=
OT_MINUS_ASSIGN,    // -=
OT_MUL_ASSIGN,      // *=
OT_DIV_ASSIGN,      // /=
OT_MOD_ASSIGN,      // %=
OT_LSHIFT_ASSIGN,   // <<=
OT_RSHIFT_ASSIGN,   // >>=

OT_BAND_ASSIGN,      // &=
OT_BOR_ASSIGN,       // |=
OT_BXOR_ASSIGN,      // ^=

// Equality
OT_LESS,          // <
OT_GREATER,       // >
OT_LESSEQ,        // <=
OT_GREATEREQ,     // >=
OT_EQ,            // ==
OT_NOTEQ,         // !=

OT_LSHIFT,        // <<
OT_RSHIFT,        // >>

// Logical
OT_LNOT,        // !
OT_LAND,        // &&
OT_LOR,         // ||

// Bitwise
OT_BNOT,        // ~
OT_BAND,        // &
OT_BOR,         // |
OT_BXOR,        // ^

// Special
```

```
OT_CALL,          // ()
OT_ARRAY,         // []
OT_STRUCT,        // .
OT_PTRSTRUCT,     // ->

// Special case
OT_POW            // pow(a, b) (at C language)
```

Класс реализует обобщенные методы для создания операций, установки аргументов операций. Однако, зачастую удобнее работать с операцией в соответствии с ее принадлежностью к одному из шести классов операций: обращение к массиву, присваивание, унарную и бинарную арифметические операции, вызов подпрограммы, обращение к элементу структуры. Для каждой операции из приведенного выше перечисления известно, к какому из шести классов операций она относится. Поэтому удобнее воспользоваться одним из соответствующих, унаследованных от ExprOper классов: ExprArray, ExprAssign, ExprUnary, ExprBinary, ExprCall или ExprStruct. Эти классы также имеют методы для получения и установки аргументов, но выполняют соответствующие проверки. Например, соотносят операцию и количество ее аргументов.

2.4. Дерево операторов

Базовым классом дерева операторов является класс Statement. Он содержит методы, общие для всех операторов, такие как getThisInBlock (получить итератор на этот оператор в блоке), getComprehBlock (получить итератор на блок, в котором располагается данный оператор), getParentStmt (получить родительский оператор), getParentBlock (получить родительский блок). Все операторы в дереве имеют ссылки наверх (uplinks), логика работы с этими ссылками также реализована в классе Statement.

Диаграмма наследования классов дерева операторов представлена на Рисунок 8. Схема наследования классов операторов.

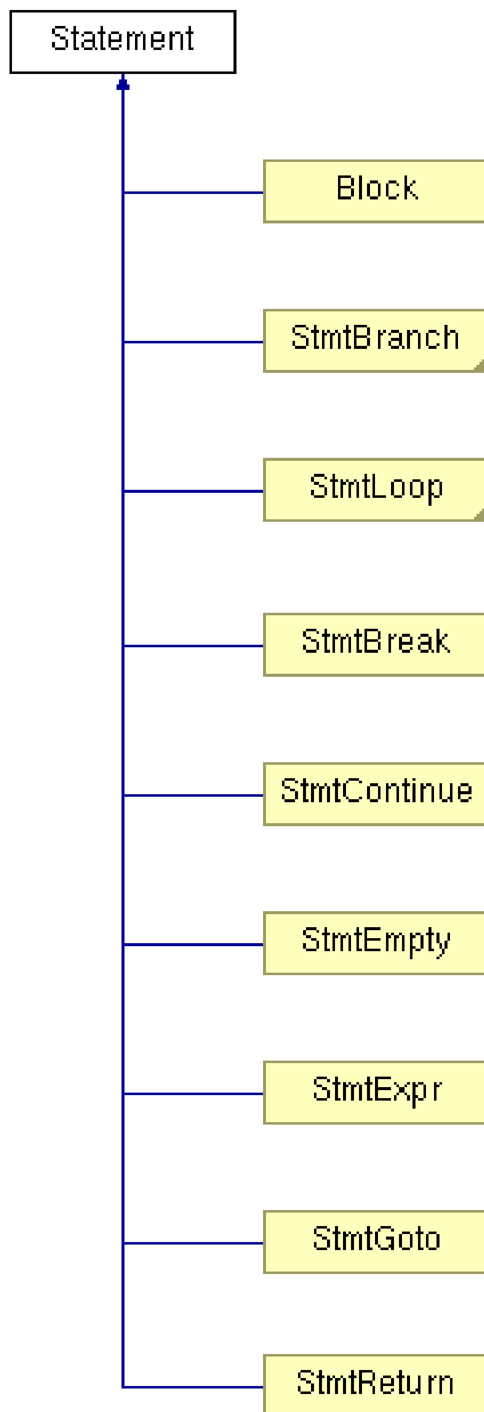

Рисунок 8. Схема наследования классов операторов

Вначале рассмотрим класс Block. Этот класс представляет понятие блока операторов в ЯВУ. Он хранит список операторов (в том числе возможно и других блоков), а также ссылку на область видимости, ассоциированную с этим блоком. В классе реализованы методы для вставки и удаления операторов, а также итерирования по списку операторов.

Теперь последовательно рассмотрим другие классы, непосредственно наследованные от Statement.

Классы StmtBreak и StmtContinue представляют соответственно операторы break и continue языка Си.

StmtEmpty – пустой оператор.

StmtExpr – оператор-выражение. В этом операторе хранится указатель на узел дерева выражения. Обычно этот класс служит адаптером для операции присваивания (ExprAssign).

StmtGoto – оператор безусловного перехода, goto. Содержит ссылку на объект в дереве идентификаторов, на метку, на которую осуществляется переход.

StmtReturn – оператор return в языке Си, также как и StmtExpr содержит ссылку на выражение.

Классы StmtBranch и StmtLoop имеют специальное назначение. Они являются базовыми соответственно для операторов ветвления и операторов цикла. Диаграммы наследования этих операторов приведены на Рисунок 9. Схема наследования классов операторов ветвления и Рисунок 10. Схема наследования классов циклов:

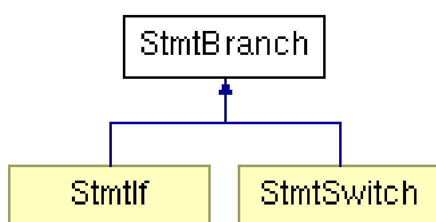

Рисунок 9. Схема наследования классов операторов ветвления

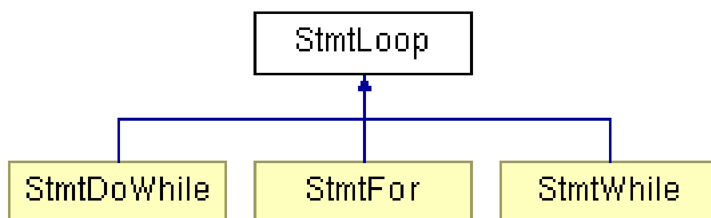

Рисунок 10. Схема наследования классов циклов

Класс StmtLoop реализует методы для получения и установки тела цикла. Таким образом, к телу любого оператора цикла можно получить доступ вне зависимости от точного типа цикла.

Классы StmtIf и StmtSwitch представляют соответственно операторы IF и SWITCH языка Си.

Классы StmtFor, StmtWhile, StmtDoWhile представляют соответственно операторы FOR, WHILE и DO..WHILE языка Си.

2.5. Итераторы и ссылки

Во внутреннем представлении для обеспечения навигации по дереву операторов и дереву идентификаторов разработаны итераторы. Для дерева выражений и дерева типов данных итераторы не реализованы, потому что в этих деревьях практически для всех узлов определено количество потомков и к ним можно обратиться по имени.

В дереве операторов итераторы применяются для обхода блока (класс Block). Для этого используется шаблонный класс ListIter[3] (константная версия ConstListIter), который инстанцируется классом Statement. Итератор реализует стандартную логику двунаправленного последовательного прохода по контейнеру, реализует методы для сравнения итераторов на равенство, присваивания и проверки на конец последовательности (метод isValid()).

Для доступа к дереву идентификаторов применяется две сущности: ссылки на идентификаторы в области видимости (NameReference) и итераторы по области видимости (NameIter). Класс NameReference внутри содержит указатель на некоторую конкретную область видимости и указатель на конкретный идентификатор в этой области видимости. Таким образом, он подходит для представления ссылки на идентификатор (например, из дерева выражений). Для удобства, объект этого класса дополнительно может находиться в состоянии Not Valid, т.е. не указывать ни

---

[3] Класс ListIter используется также для обхода членов структур и параметров подпрограмм.

на какой идентификатор. В класс определены вспомогательные методы для определения типа идентификатора:

```
NameKind getNameKind();
bool isVarName();
bool isTypeName();
bool isLabelName();
bool isSubProcName();
```

методы для получения объекта-идентификатора:

```
NameObject& getAsObject();
NameVar& getAsVar();
NameType& getAsType();
NameLabel& getAsLabel();
NameSubProc& getAsSubProc();
```

а также методы для получения строкового представления идентификатора, области видимости, в которой он определен и сравнения двух ссылок на идентификаторы.

Класс NameIter – это итератор по идентификаторам области видимости, который позволяет ставить фильтр на тип идентификаторов, по которым может осуществляться проход. При создании экземпляра класса разработчик выбирает, хочет ли он этим итератором проходить все идентификаторы в заданной области видимости или только идентификаторы определенного типа (например, только переменные или только подпрограммы). Итератор реализует стандартную логику последовательного прохода по контейнеру. Реализует операции сравнения двух итераторов на равенство. Объект класса NameIter может быть инициализировать объектом класса NameReference, а также по объекту класса NameIter можно получить объект класса NameReference. Таким образом, объекты этих двух классов являются взаимозаменяемыми и разработчик должен выбирать тот класс, который больше подходит для его задач. Если ему необходимо сослаться на идентификатор, предпочтительней использовать NameReference, если осуществить проход по области видимости – NameIter.

2.6. Пометки узлов

Пометки узлов во внутреннем представлении позволяют ассоциировать информацию с узлов одного из четырех деревьев и хранить ее вместе с

внутренним представлением. Причем, при реализации сериализации[4] внутреннего представления эта информация сохраняется.

Базовый механизм пометок узлов реализован в классе IRObject. Методы setNote(), getNote() и delNote() позволяют соответственно устанавливать, получать и удалять пометки узлов. Пометки идентифицируются по имени, т.е. все эти три метода одним из параметров принимают строку символов – имя пометки. Сама пометка представляется классом ImmValue (непосредственное значение). Этот класс позволяет хранить один из следующих типов данных: bool, int, double, char, string, указатель на IRObject*. Таким образом, пометки являются типизированными и допускают вложение друг в друга.

На текущий момент используются пометки для обозначения векторизуемых циклов и зависимости итераций гнезда циклов.

Текущая реализация компилятора переднего плана в ОРС поддерживает использование директив компилятора (#pragma), с помощью которых можно в тексте программы задать пометки узлов.

2.7. Средства проверки и отладки

В новом внутреннем представлении, как было сказано ранее, используется паттерн Factory для порождения узлов всех деревьев. Кроме рассмотренных ранее достоинств единообразия выделения и освобождения памяти, такой подход позволил также повысить надежность внутреннего представления. При реализации любого преобразования, всегда вызываются методы классов внутреннего представления. Эти методы содержат строгие проверки входных данных и некоторые эвристические проверки, которые предотвращают порождение неправильных конструкций. Следует заметить, что новое ВП, в отличие от старого, гораздо лучше использует статическую проверку типов С++, присущую этому языку строгую типизацию, поэтому

[4] Сериялизация – преобразование некоторой структуры данных в поток регулярной структуры. Например, в байтовый поток (бинарная сериализация) или XML-поток (текстовая сериализация).

многие ошибки, которые допускались при реализации работы со старым ВП, теперь выявляются на этапе компиляции. Так, например, узел дерева операторов – оператор FOR в новом ВП имеет строго четыре потомка, первые три из которых выражения, а четвертый – блок операторов. В то же время, в старом ВП оператор FOR представлялся обобщенным классом OPERATOR, и никаких проверок относительно потомков такого узла на этапе компиляции было произвести невозможно.

Теперь несколько слов о встроенном механизме отладки ВП. Каждый узел внутреннего представления имеет метод dumpState(), который возвращает строку с информацией об этом узле. Вывод представления этими методами легко читаем, потому что схож с синтаксисом языка Си.

Кроме того, все узлы ВП поддерживают модель обхода по паттерну Visitor [10]. Реализация этого паттерна сделана по описанию Acyclic Visitor из [11] и, вообще говоря, может применяться не только в отладочных целях. Однако, на текущий момент не решены некоторые специфичные технические проблемы с обходом константных объектов внутреннего представления и применение этого механизма, например, в графах информационных зависимостей небезопасно. Для тестирования преобразований, автором была разработана программа IntegrityTest, в которой для сбора статистики по операторам и вхождениям переменных в программу использовался именно механизм паттерна Visitor. Простота кода программы IntegrityTest подтверждает удобство применения этого способа обхода узлов внутреннего представления.

3. Представление свойств языков программирования во внутреннем представлении

Структура внутреннего представления во многом определяется классом языков, которые могут быть в нем записаны. Например, если речь идет о представлении программ с языка ассемблера, во внутреннем представлении должны быть элементы, представляющие такие низкоуровневые понятия как условная и безусловная передача управления, обращение к регистрам ЦП, прямое обращение к памяти и т.п. При этом во внутреннем представлении не будет понятия циклов, определения массивов или структур.

Внутреннее представление, предназначенное для хранения информации о программах на языках высокого уровня должно содержать высокоуровневые конструкции, такие как циклы, определение абстрактных типов данных, модулей. Новое внутреннее представление OPC хранит информацию о программах с языков высокого уровня, таких как Си, Паскаль, Фортран. При проектировании ВП были проанализированы эти языки программирования, и на основе их конструкций был сформирован базис, позволяющий представлять программы на любом из этих языков.

Большинство конструкций во внутреннем представлении перешли из языка Си. Например, цикл FOR Си и Паскаля (цикл DO Фортрана) во внутреннем представлении называется StmtFor и характеризуется узлом с четырьмя потомками: выражение инициализации (InitExpr), выражение условия (CondExpr), выражение инкремента (IncrExpr) и тела цикла (Body). Таким образом, например, цикл For Паскаля

```
for i := 1 to 10 do
  ...
```

может быть записан во внутреннем представлении:

```
InitExpr: i = 1
CondExpr: i <= 10
IncrExpr: ++i
Body: ...
```

В то же время определение массивов в языке Паскаль является более общим, чем в Си:

```
M : array[1..10, -5..5] of integer;
```

Поэтому узел дерева типов, хранящий массив во внутреннем представлении содержит информацию о верхней и нижней границе массива. Более того, в языке Си понятие многомерного массива вводится индуктивно, так, например, трехмерный массив представляется как массив двухмерных массивов, а двухмерный массив как массив массивов. Во внутреннем представлении ОРС многомерный массив – это один узел в дереве типов, которых хранит информацию о количестве измерений массива и границы индексов каждого.

Также важной характеристикой внутреннего представления является его исполнимость [15, с. 44]. Исполнимость ВП означает возможность исполнить программу, записанную во внутреннем представлении. При этом результаты такого исполнения должны быть идентичны результатам исполнения первоначальной программы при одинаковых наборах данных на входе обеих программ. Новое внутреннее представление ОРС, как и прежнее, является исполнимым.

Внутреннее представление проектировалось так, чтобы при преобразованиях программ сохранялась синтаксическая и семантическая целостность. Так операторы циклов и ветвления представлены в полном виде. Это означает, что при добавлении, например, оператора IF сразу создается выражение (пустое) для условия, а также пустые блоки для THEN и ELSE веток оператора. В частности, это решает проблему «кочующего ELSE». Таким образом, невозможна ситуация, когда оператор IF (в терминах языка Паскаль) имеет только ELSE, без THEN. Или когда в блоке имеется два ELSE, но только один IF.

4. Графы ОРС и библиотека преобразований программ, их взаимодействие с внутренним представлением

Графовые модели программ используются в оптимизирующих и распараллеливающих компиляторах давно. Так одной из первых теоретико-графовых моделей программ был управляющий граф, введенный в рассмотрение Р. Карпом в 1960 г. В Открытой распараллеливающей системе на момент написания этой работы реализованы пять графов: граф информационных связей Лампорта, решетчатый граф, граф вычислений, управляющий граф программы и граф вызовов подпрограмм.

Граф информационных связей Лампорта позволяет определять наличие зависимости между вхождениями переменных. Вершинами этого графа соответственно являются вхождения переменных, а дугами – зависимости между ними. Выделяется три типа зависимостей: истинная информационная зависимость, антизависимость, выходная зависимость [3].

Решетчатый граф также позволяет определять наличие информационных зависимостей и является более тонким инструментом, он содержит информацию о конкретной информационной зависимости на разных итерациях циклов. Нередко встречается ситуация, когда при некоторых значениях счетчиков циклов зависимость существует, но при всех остальных она отсутствует и можно осуществить распараллеливание.

Граф вычислений содержит информацию о порядке выполнения арифметических операций, а также операций обращения к памяти. Этот граф позволяет проследить порядок использования данных при вычислении выражений и применяется для формирования задания конвейеру, например, на суперкомпьютерах со структурно-процедурной организацией вычислений [17].

Управляющий граф программы позволяет контролировать передачу управления между операторами программы. Он может быть полезен для

осуществления таких преобразований как развертка, раскрутка, разрезание (для них важен определенный порядок передачи управления в программе), а также для удаления неиспользуемых фрагментов программного кода.

Внутреннее представление ОРС не содержит в явном виде эти графы, вместо этого оно предоставляет сервис для их построения и осуществляет взаимодействие между графами и библиотекой преобразований. Так для графа Лампорта, решетчатого графа и графа вычислений для обозначения вхождений используется класс внутреннего представления ExprData. Для управляющего графа для обозначения операторов используется класс Statement, а для графа вызовов подпрограмм класс NameSubProc.

Преобразования программ в ОРС опираются на внутреннее представление. На момент написания этой работы на новом внутреннем представлении реализовано 14 преобразований: удаление мертвого кода, перестановка заголовков циклов, расщепление пространства итераций, гнездование циклов, слияние циклов, разрезание циклов, подстановка в многомерных циклах, распараллеливание рекуррентных циклов, переименование, разрыв итераций, перестановка фрагментов, введение временных массивов, растягивание скаляров. Следует отметить повышение надежности реализации преобразований и упрощение сопровождения кода.

5. Визуализация внутреннего представления

Визуализация внутреннего представления может быть необходима в различных ситуациях. Например, это упрощает отладку и понимание структуры ВП (при обучении разработчиков). Однако, имеется важное прикладное применение визуализации ВП в Открытой распараллеливающей системе – работа с фрагментами программы. Инструмент для визуализации внутреннего представления в виде текста программы с возможностью выделения фрагментов, был реализован еще для старого внутреннего представления и нашел применение в обучающей программе на основе ОРС [4]. Поясним принцип работы этого инструмента на примере.

В обучающей программе, чтобы построить решетчатый граф для фрагмента программы, необходимо выделить два вхождения переменной. Пользователь это делает при помощи мыши:

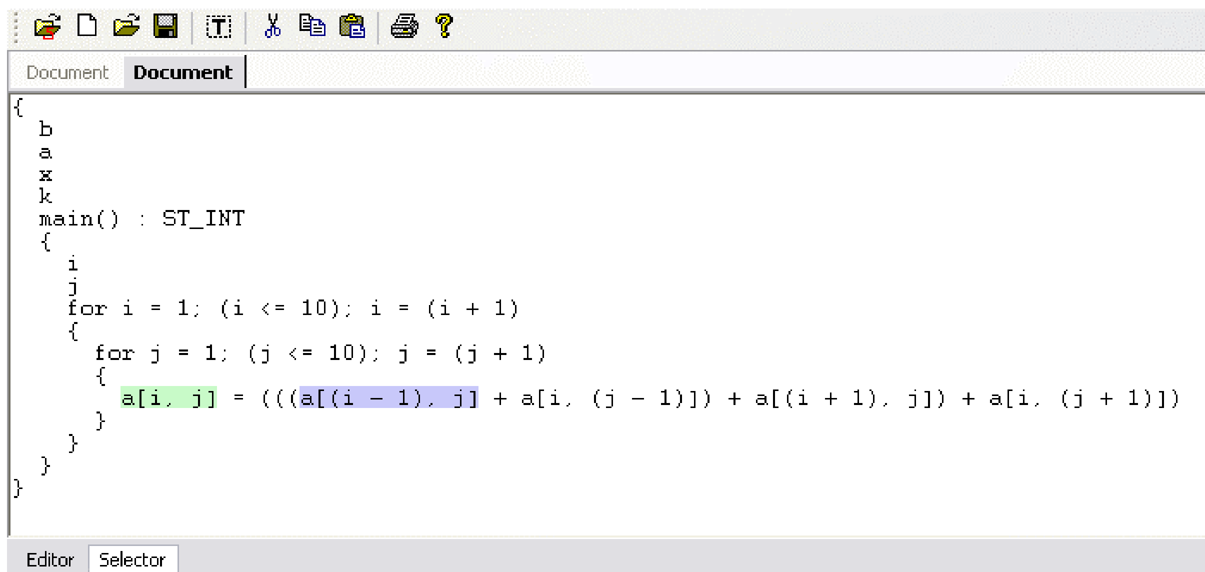

Рисунок 11. Инструмент для выделения фрагментов

И получает изображение решетчатого графа, построенного для двух выделенных вхождений переменных:

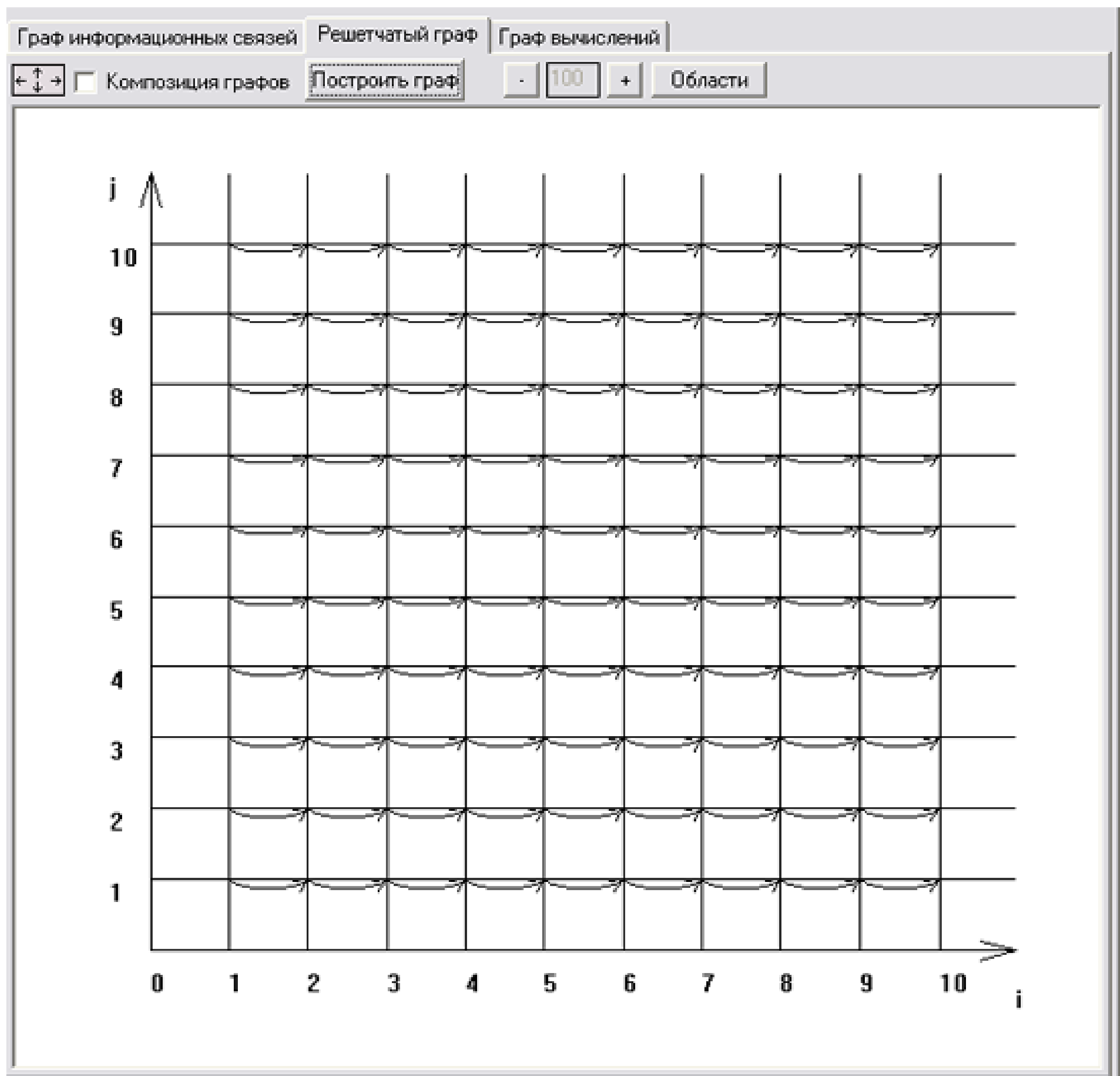

Рисунок 12. Решетчатый граф

Сложность реализации инструмента выделения фрагмента заключается в том, что необходимо проверить целостность фрагмента (например, что пользователь выделил именно вхождение, т.е. “a[i,j]”, а не “[i,j]”) и соотнести выделение с узлом во внутреннем представлении. Оказалось, что проще всего решить эти две проблемы, позволив пользователю выделять фрагменты на визуализированном внутреннем представлении программы.

Для того чтобы инструмент работал для нового внутреннего представления, потребовалось его доработать. Это оказалось возможным благодаря удачной архитектуре, заложенной с самого начала в инструмент визуализации. В нем используются унифицированные узлы (SelNodes),

которые хранят информацию о том, как отображать тот или иной фрагмент внутреннего представления и ссылку на узел во внутреннем представлении, соответствующий этому фрагменту. Для визуализации нового внутреннего представления была переписана процедура построения унифицированных узлов, остальной код остался практически без изменений.

6. Обзор внутренних представлений аналогичных систем

6.1. Polaris

Система Polaris, разработанная в центре суперкомпьютерных исследований университета Иллинойс, является распараллеливающим и оптимизирующим конвертером фортран-фортран. Внутреннее представление (ВП) этой системы реализовано на языке C++ и подробно документировано [6], благодаря чему имеется возможность рассмотреть сильные и слабые стороны принятых в нем архитектурных решений.

Прежде всего, следует отметить, что внутреннее представление Polaris ориентировано только на один язык программирования – Фортран-77, это во многом определило его архитектуру. Представление содержит ряд специфичных для фортрана конструкций, таких как Data Blocks и Common Blocks. Это позволяет эффективно реализовывать преобразования программ. Однако, у этого представления имеются недостатки. Отсутствует такая важная для современных процедурных языков (Си, Паскаль, Java) конструкция, как блок операторов (список операторов с ассоциированной с ним областью видимости имен). Пометки в узлах ВП не предусмотрены. Вместо них разработчики предоставили стек временных объектов, ассоциированный со всей программой (единицей трансляции), поэтому разработчик программы не может ссылаться на определенный элемент программы (оператор, операцию, вхождение) неким стандартным образом.

Граф информационных зависимостей в системе Polaris неразрывно связан с внутренним представлением [7]. Это означает, что при выполнении преобразований приходится его постоянно перестраивать, чтобы поддерживать его информацию о зависимостях в актуальном состоянии. В ОРС граф информационных зависимостей строится до выполнения преобразований заново. Это особенно актуально, потому что ОРС на момент написания этой работы уже содержит пять графовых моделей: граф

информационных связей, решетчатый граф, граф вычислений, граф вызова подпрограмм и управляющий граф. Синхронное перестроение всех этих графов заняло бы слишком много времени и усложнило бы реализацию преобразований внутреннего представления.

Таким образом, внутреннее представление системы Polaris ориентировано на один язык и требует избыточное время выполнения преобразований, связанное с модификацией графа информационных связей.

6.2. SUIF

Stanford University Intermediate Format (SUIF) – программная система для исследования распараллеливающих и оптимизирующих компиляторов [8]. Эта система также как и ОРС реализована на C++, в Интернете имеется документация на внутреннее представление [9] и исходный код.

Внутреннее представление SUIF рассчитано на языки Си и Фортран-77. Однако поддержка языка Фортран изначально в систему не закладывалась, программа с него конвертером f2c переводится в Си, а только потом во внутренний формат. Очевидно, что при таком подходе специфичная для программ на Фортране информация никак не отражается во внутреннем представлении.

В качестве основного способа расширения функциональности в системе SUIF используется написание нового прохода (compiler pass). При этом, в системе не предусмотрено никаких способов использовать имеющиеся, а также добавлять новые, алгоритмы поиска и анализа информационных зависимостей в программе. Кроме того, обмен информацией между проходами ведется через дисковые файлы, что дополнительно способствует снижению производительности оптимизирующего блока компилятора.

Внутреннее представление ОРС имеет ряд схожих черт с внутренним форматом SUIF, например, основанный на паттерне «Composite» [10] способ

представления программ и использование пометок узлов для передачи информации между отдельными частями компилятора.

7. Заключение

Дипломная работа посвящена новому внутреннему представлению Открытой распараллеливающей системы, его связи с графовыми моделями и преобразованиями программ. В работе показано, что разработанное внутреннее представление является универсальным, расширяемым, исполнимым и способно хранить программы процедурных языков программирования. Приведены доводы в пользу объектно-ориентированной модели внутреннего представления, приведено сравнение со старым внутренним представлением ОРС. Также приведено сравнение с внутренними представлениями наиболее известных распараллеливающих систем – Polaris и SUIF.

В работе описана программная реализация нового внутреннего представления, рассмотрены средства проверки целостности и отладки. Описан инструмент для визуализации внутреннего представления и пометки его фрагментов, показано использование этого инструмента в обучающей распараллеливанию программе.

Внутреннее представление, описанное в работе, и его программная реализация являются основой программной реализации библиотеки преобразований Открытой распараллеливающей системы.

8. Литература

1. Штейнберг Б.Я. Открытая распараллеливающая система.// РАСО’2001/ Труды международной конференции «Параллельные вычисления и задачи управления». М., 2-4 октября 2001 г., ИПУ РАН, с. 214-220.

2. Штейнберг Б.Я., Черданцев Д.Н., Науменко С.А., Бутов А.Э., Петренко В.В. Преобразование программ для Открытой распараллеливающей системе. // Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, “Наука и Освита”, 2003, № 4 , с. 97-105.

3. Штейнберг Б.Я., Макошенко Д.В., Черданцев Д.Н., Шульженко А.М. Внутреннее представление в Открытой распараллеливающей системе. // Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, “Наука и Освита”, 2003, № 4 , с. 89-97.

4. Штейнберг Б.Я., Черданцев Д.Н., Штейнберг Р.Б., Шульженко А.М., Бутов А.Э., Науменко С.А., Петренко В.В., Шилов М. В., Гуфан К.Ю., Тузаев А.В., Арутюнян О. Э., Морылёв Р.И. Обучающая распараллеливанию программа на основе ОРС. // Научно-методическая конференция «Современные информационные технологии в образовании: Южный Федеральный округ»/ 13-14 мая 2004 г., Ростов-на-Дону, Тезисы докладов, с. 248-250.

5. ANother Tool for Language Recognition. http://www.antlr.org/.

6. K. Faigin, J. Hoeflinger, D. Padua, P. Petersen, S. Weatherford. The Polaris Internal Representation.

7. Y. Paek, P. Petersen. A Data Dependence Graph in Polaris.

8. R. Wilson, R. French, Ch. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers

9. G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moorey, B. Murphy, C. Sapuntzakis. The SUIF Program Representation

10. Гамма Э., Хэлм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.: ил.

11. Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied – Addison Wesley, 2001. – 352 p.

12. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя/ Пер. с англ. – М.: ДМК Пресс, 2001 – 432 с.: ил.

13. Страуструп Б. Язык программирования С++, 3-е изд. / Пер. с англ. – СПб.; М.: «Невский диалект» - «Издательство БИНОМ», 1999 г. – 991 с.: ил.

14. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++, 2-е изд./ Пер. с англ. – М.: «Издательство БИНОМ», СПб.: «Невский диалект», 2000 – 560 с.: ил.

15. Лазарева С. А. Использование многоуровневого внутреннего представления в автоматическом распараллеливании программ для многопроцессорных ЭВМ, диссертация на соискание ученой степени кандидата технических наук, Ростовский госуниверситет, Ростов-на-Дону, 2000 г., 150 с.

16. Штейнберг Б.Я. Математические методы распараллеливания рекуррентных циклов для суперкомпьютеров с параллельной памятью . – Ростов н/Д: Изд-во Ростовского университета, 2004. – 192 с.: ил.

17. Штейнберг Р.Б. Вычисление задержек в стартах конвейеров для суперкомпьютеров со структурно-процедурной организацией вычислений. // Международная научно-техническая конференция «Интеллектуальные и

многопроцессорные системы»/ 22-27 сентября, 2003, Дивноморское, Россия, Тезисы докладов, с. 43-47.