

Санкт-Петербургский государственный университет

Кафедра Системного Программирования

Группа 21.Б15-мм

Дефорестация для векторного языка программирования

Друмов Максим Андреевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
доцент кафедры СП, к.ф.-м.н. Д. Ю. Булычев

Санкт-Петербург
2024

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Philip Wadler	6
2.2. Andy Gill	7
3. Реализация	10
Заключение	13
Список литературы	14

Введение

Цифровой сигнальный процессор (*DSP*) представляет собой специализированный микропроцессор, разработанный для обработки оцифрованных сигналов [2]. Он является ключевым элементом в современных системах связи, аудио- и видеообработки, радиосвязи и других областях, где требуется анализ и обработка сигналов в цифровой форме.

Программирование таких процессоров является сложной задачей, не поддающейся легкой автоматизации. Это обусловлено уникальной архитектурой данных процессоров, требующей тщательного планирования деятельности параллельно работающих блоков.

Несмотря на длительное использование *DSP*, трансляторы для традиционных языков программирования, таких как *C/C++*, оказываются недостаточно эффективными, значительно уступая в эффективности ручному программированию на языке ассемблера.

Для решения этой проблемы ведется работа над векторным языком — языком, в котором основным типом данных являются массивы и основные операции предназначены для обработки массивов. Особенностью данного конкретного языка является то, что в нем все данные имеют статический размер и статическое распределение.

В языке реализовано некоторое количество функций для работы с массивами, такие как: *map*, *fold*, *rev*, *zip*, *unzip* и т.д. При использовании их в комбинации выполнение каждой порождает отдельный проход по массиву. Для примера возьмем функцию, которая приведет к формированию промежуточной структуры данных, включающую два применения функции *map* к одному массиву:

$$\text{map } f \ (\text{map } g \ \text{arr})$$

При отсутствии оптимизаций сначала выполняется применение функции *g* к массиву *arr*, а затем применение функции *f* к результату первой функции. Этого можно избежать при использовании дефорестации. Дефорестация — это программное преобразование, при котором устраняются или оптимизируются промежуточные структуры данных. Эта оптимизация направлена на уменьшение использования ресурсов

и повышение производительности программы [3]. С помощью правила *map-map* код может быть оптимизирован применением композиции к функциям написанных *map*. После дефорестации наш пример примет вид:

$$\text{map } (f \circ g) \text{ arr}$$

Код, оптимизированный таким образом, не создает промежуточных структур данных и использует память только для сохранения полученных результатов. Наша задача заключается в разработке механизма дефорестации, который будет адаптирован для векторного языка программирования. Дефорестация в данном контексте представляет собой программное преобразование, направленное на устранение или оптимизацию промежуточных структур данных, что содействует эффективно-му выполнению алгоритмов и уменьшению нагрузки на память.

1. Постановка задачи

Целью работы является разработка и реализация дефорестации для векторного языка программирования.

Для ее выполнения поставлены следующие задачи:

1. Обзор реализаций дефорестации
2. Изучение существующей реализации
3. Разработка и реализация дефорестации
4. Тестирование

2. Обзор

Обзор существующих реализаций дефорестации необходим для получения представления, как могут выглядеть решения. Рассмотрены популярные подходы, которые представляют алгоритмы дефорестации.

2.1. Philip Wadler

В 1981 году Филип Уодлер приступил к разработке метода, который автоматически исключал бы промежуточные структуры данных из функциональных программ [5]. Этот метод он назвал Дефорестация и впервые введен в 1990 году.

Алгоритм Уодлера устанавливал основой набор операторов, которые можно использовать для выражения большого класса списковых вычислений: *map*, *red(use)* и *gen(erate)*. Red очень похож на *foldr*, а *gen* схож с *unfoldr*. Далее вместо red будем использовать обозначение *foldr*. А *gen* хоть и близок по задачам с *unfoldr*, но все равно достаточно отличается, поэтому будем писать *gen*.

Уодлер определил набор правил перезаписи функций, над установленными функциями высшего порядка. Эти правила определяют шаблоны операторов списков, которые создают, используют промежуточные списки, а затем объединяют операции, чтобы исключить необходимость в промежуточных списках.

Алгоритм заключается в многократном применении этих правил, чтобы открыть больше возможностей для дефорестации:

1. *map-map*

$$\begin{aligned} \text{map } f \ (\text{map } g \ \text{arr}) &\rightarrow \\ \text{map } (f \circ g) \ \text{arr} \end{aligned}$$

2. *foldr-map*

$$\begin{aligned} \text{foldr } f \ \text{acc} \ (\text{map } g \ \text{arr}) &\rightarrow \\ \text{foldr } (f \ (g \ x) \ \text{acc}) \ \text{acc} \ \text{arr} \end{aligned}$$

3. *map-gen*

```
map (gen stop yield next arr) →  
gen stop (f (yield x)) next arr
```

4. *foldr-gen*

```
foldr f a (gen stop yield next arr) →  
let rec h a b =  
    if stop b  
    then a  
    else h (f (yield b) a) (next arr)
```

Преимущества:

1. Не требуется ввод данных программистов и поэтому подходит для включения в компилятор
2. Все преобразования переводят программы на исходном языке, в другие более быстрые программы также на исходном языке
3. Простота – каждое правило легко к пониманию

Недостатки:

1. Если потребуется добавить новые операторы для списков, необходимо будет добавить новое правило для каждой комбинации нового оператора с каждым из ранее заданных
2. Ограниченность – многие вычисления списков не могут быть выполнены с использованием этой структуры, что повлечет появление промежуточных списков.

2.2. Andy Gill

Энди Джилл представил иной подход к дефорестации в 1993 году под названием *build-foldr fusion* [1]. Этот подход выполняет ряд оптимизаций, аналогичных предыдущему методу Уодлера, но при этом устраняет его недостатки.

Данный алгоритм так же выполняет оптимизации из исходного языка в этот же язык. Данный подход подчиняется и основан на одном правиле:

foldr-build

$$\text{foldr } f \ a \ (\text{build } g) \rightarrow \\ g \ f \ a$$

Что такое *build*?

$$\text{build } g = g \ (\text{::}) \ []$$

В *build* главная часть находится в *g*. *g* это любой производитель списка, который требуется нам, абстрагированный от *cons* и *nil*. Это значит, что вместо вызова *cons* и *nil* он вызывает *cons* и *nil*, которые получает через аргументы.

Чтобы применять это правило не только к оператору *foldr*, необходимо нужные нам функции представить в виде *foldr*, что позволит нам масштабировать этот алгоритм дефорестации. Например, представим функцию *map*:

$$\text{map } f \ \text{arr} = \\ \text{build } (\text{fun } \text{cons } \text{nil} \rightarrow \\ \text{foldr } (\text{fun } x \ a \rightarrow \text{cons } (f \ x) \ a) \ \text{nil } \text{arr})$$

Таким образом, мы можем добиться через преобразование функций в *foldr*-подобные, к которым можем применять правила из рассматриваемого алгоритма.

Преимущества:

1. Как и оригинальный алгоритм, вышерассмотренный алгоритм обладает теми же преимуществами: Простота, Автоматический, Оптимизация из исходного языка в него же
2. Нет ограничений на ввод, это значит что он может работать со всеми операторами. Правда, это не гарантирует что алгоритм оптимизирует этот код, но, по крайней мере, не сломается и не зациклится

3. При добавлении нового оператора списков, не требуется реализовывать поддержку всех комбинаций с уже существующими

Недостатки:

1. Ограничен с функциями которые используют аккумулятор, например *foldl*. Мы можем представить *foldl* в виде *foldr*, но это переместит промежуточный список в аккумулятор, что ничего нам не даст в плане оптимизации. Так же с *zip*, правило сможет про-дефорестировать первый список, но ничего не сможет сделать со вторым.

3. Реализация

Для реализации были рассмотрены способы дефорестации, описанные Филипом Уодлером и Энди Джиллом. Подход, предложенный Энди Джиллом предпочтительнее в связи с тем, что он позволяет продефорестировать большее количество функций и обладает лучшей масштабируемостью [1]. Но его применение невозможно в нашем случае, поскольку в разрабатываемом векторном языке программирования отсутствует поддержка использования массивов в качестве аккумулятора функции *fold*. А метод Джилла основан на представлении функций, предназначенных для обработки массивов, с использованием *fold*.

Вариант дефорестации, представленный Филипом Уодлером, поддерживает четыре правила, по которым происходит избавление от промежуточных массивов [5]. Он охватывает такие функции, как *map*, *fold* и *unfold*. Несмотря на то, что в разрабатываемом языке программирования отсутствует функция *unfold*, данный метод является более подходящим. Рассматриваемый подход позволяет эффективно удалять промежуточные структуры данных, что значительно повышает производительность. В представленном решении дефорестируются следующие правила: *map-map* и *fold-map*, а *map-unfold* и *fold-unfold* не поддерживаются.

Чтобы реализовать новую функциональность, необходимо провести дополнительный проход по типизированному дереву. Этот обход предназначен для поиска конструкций, подходящих под заданные правила, и их трансформации. На выходе также получается типизированное дерево, в котором уже применены найденные правила. Процесс включает в себя композицию внешней и внутренней функций, что позволяет избежать создания лишних промежуточных массивов.

Рассмотрим пример применения правила *map-map*:

$\text{map } f \ (\text{map } g \ a) \rightarrow \text{map } (f \circ g) \ a$, где f — внешняя функция, g — внутренняя функция, а a — массив, над которым проводятся преобразования. В результате, вместо двух проходов по массиву, выполняется один проход с композицией функций, что существенно улучшает про-

изводительность.

Для написания модуля дефорестации использовалась библиотека *GT (Generic Transformers)* [4]. На первом этапе дефорестирования запускается специальный трансформер, задача которого — пройтись по конструкторам и создать хэш-таблицу. Эта таблица используется для мемоизации информации о каждом узле, который встречается в дереве конструкций. Цель этого шага — определить, какие узлы являются физически идентичными, то есть *shared nodes*. Это позволяет избежать дублирования вычислений. Когда трансформер встречается узел, он проверяет, есть ли уже информация об этом узле в хэш-таблице. Если такой узел уже встречался, трансформер берет из таблицы сохраненные данные о нем и возвращает их. Если узел ранее не был посещен, информация о нем добавляется в таблицу. Когда встречается конструктор **Map** или **Fold**, проверяется массив, над которым проводятся операции. Если массив является **Map**, то запускается проход по дереву внешней функции конструкции **Map/Fold**.

Для осуществления прохода по дереву функции (*deforest*), мы используем плагин *stateful* из библиотеки *GT*. Это обусловлено необходимостью учитывать, что не все конструкции требуют проверки наличия переменных, а также типизированное дерево представлено в виде кортежа, включающего в себя нетипизированное дерево **DFL.t** и тип **Type.t**. В такой ситуации, библиотека *GT* не может автоматически генерировать методы для данного типа дерева.

Чтобы решить эту проблему, воспользуемся особенностью плагина *stateful*: передача изменяемого окружения (*env*) на каждом шагу. В качестве окружения мы передаем кортеж, который состоит из типизированного узла и *env*, предназначенного для подсчета индекса де Брюйна, чтобы вставить функцию из внутреннего **Map** в нужные места. При обработке каждой конструкции передается и ее типизированный узел, чтобы предотвратить разъединение узлов и обеспечивает их физическое расположение в одном и том же месте в памяти. Это гарантирует корректность и консистентность данных во время выполнения оптимизаций, таких как композиция функций в **Map** и **Fold**.

Во время прохода `deforest` используется метод *super*, который выполняет проверку на физическую идентичность узла и подставляемой функции. Если они идентичны, то возвращается подставляемая внутренняя функция.

При обработке конструкции `Var` происходит проверка индекса де Брюйна. Для этого используется окружение (*env*), которое подсчитывает индекс при обнаружении лямбда-абстракции. Если индекс совпадает с индексом в переменной, то конструкция `Var` заменяется на внутреннюю функцию *g*.

После выполнения композиции функций, в первом обходе мы также мемоизируем итоговую функцию. Это необходимо для предотвращения дублирования кода при дальнейшей компиляции. Если во внешней функции заменено несколько переменных, может возникнуть несколько вхождений внутренней функции *g*, и все они будут физически идентичны. Мемоизация позволяет сохранить информацию об этих вхождениях, обеспечивая их единое представление в памяти и избегая избыточного кода в окончательной скомпилированной программе.

Заключение

В ходе работы были достигнуты следующие результаты:

1. Проведен обзор реализаций дефорестации
2. Изучена существующая реализация
3. Разработан и реализован дефорестатор
4. Протестирован дефорестатор

Список литературы

- [1] Andy Gill. — URL: <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/deforestation-short-cut.pdf?from=https://research.microsoft.com/en-us/um/people/simonpj/papers/deforestation-short-cut.pdf&type=exact> (дата обращения: 2024-06-02).
- [2] DSP. — URL: https://en.wikipedia.org/wiki/Digital_signal_processor (дата обращения: 2024-06-02).
- [3] Deforestation. — URL: [https://en.wikipedia.org/wiki/Deforestation_\(computer_science\)](https://en.wikipedia.org/wiki/Deforestation_(computer_science)) (дата обращения: 2024-06-02).
- [4] GT. — URL: <https://github.com/PLTools/GT> (дата обращения: 2024-06-02).
- [5] Philip Wadler. — URL: <https://homepages.inf.ed.ac.uk/wadler/topics/deforestation.html> (дата обращения: 2024-06-02).