Санкт-Петербургский государственный университет

Пономарев Николай Алексеевич

Выпускная квалификационная работа

Разработка транслятора модельного функционального языка в Interaction Nets

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5162.2021 «Технологии программирования»*

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Григорьев С. В.

Рецензент:
инженер-исследователь, Лаборатория YADRO СПбГУ, Косарев Д. С.

Санкт-Петербург
2025

Saint Petersburg State University

Nikolai Ponomarev

Bachelor's Thesis

An implementation of a translator from a model functional language to Interaction Nets

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5162.2021 "Programming Technologies"*

Scientific supervisor:
C.Sc., docent S.V. Grigorev

Reviewer:
research engineer at "St. Petersburg State University" D.S. Kosarev

Saint Petersburg
2025

Оглавление

Введение

Многие алгоритмы искусственного интеллекта и анализа графов основаны на линейной алгебре или могут быть переформулированы в её терминах, позволяя использовать развитую экосистему для работы с линейной алгеброй. Поскольку вычисления в линейной алгебре часто независимы друг от друга, разумно использовать возможности параллельного программирования для ускорения работы алгоритмов. Кроме того, реальные данные часто являются разреженными [5], что позволяет использовать алгоритмы разреженной линейной алгебры.

К сожалению, распараллеливание разреженной линейной алгебры — сложная задача для традиционных архитектур вычислителей таких, как CPU или GPU, из-за нелокальных обращений к памяти и непредсказуемого количества независимых подзадач [6, 9, 20]. В настоящее время для решения этих проблем всё чаще применяют ускорители на специализированных архитектурах [1, 2, 11, 31, 34].

Interaction Nets — модель вычислений, которая была описана Yves Lafont в 1989 году [15]. В этой модели программа представляется в виде графа, а одним из важных свойств является естественная поддержка нерегулярного параллелизма.

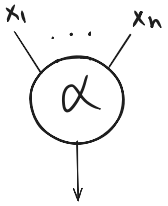
Для Interaction Nets был написан не один интерпретатор, например [18, 26], однако попыток реализовать ускоритель на его основе не предпринималось, кроме того существующие интерпретаторы используют собственные языки программирования далёкие от распространённых. Поэтому в рамках проекта Lamagraph[1] исследуются возможности по разработке параметризуемого многоядерного сопроцессора для разреженной линейной алгебры на основе Interaction Nets и ML-подобного функционального языка для программирования сопроцессора.

[1]Репозитории проекта доступны по ссылке: https://github.com/Lamagraph/ (дата обращения: 18 мая 2025 г.)

1. Interaction Nets

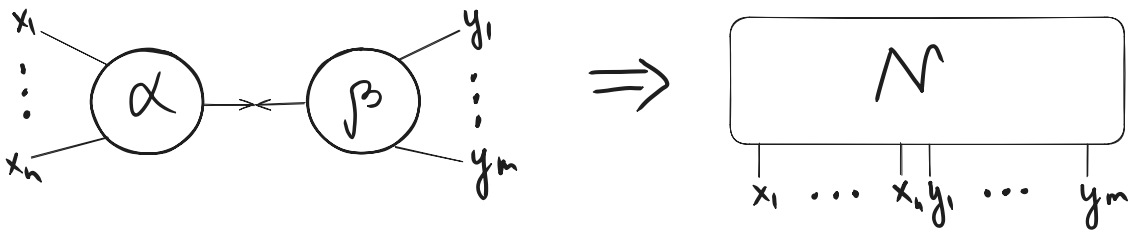
Данный раздел является кратким описанием системы Interaction Nets, более подробно про неё можно узнать в [14, 25, 27].

Описание системы. Interaction Nets представляет собой систему переписывания графов. Зафиксируем множество символов Σ, которым будем обозначать узлы графа. Помеченную символом из Σ вершину будем называть *агентом*, связи между агентами — *проводами*, а места соединения агентов проводами — *портами*. Для каждого агента зафиксируем его арность ar, которая будет означать количество *дополнительных* портов символа. Так, если символ $\alpha \in \Sigma$ имеет арность $\mathrm{ar}(\alpha) = n$, где $n \in \mathbb{N}$, то у символа имеется $n + 1$ портов: n дополнительных и один выделенный — *главный*.



Узлы могут изображаться с помощью кругов, треугольников или прямоугольников. *Сеть*, построенная на Σ, является неориентированным графом с символами из Σ в его вершинах. Ребра соединяют порты вершин так, что в каждый порт приходит не более одного ребра. Порт не соединенный ни с одним ребром называется *свободным*, а множество таких портов называется *интерфейсом*.

Пара агентов $(\alpha, \beta) \in \Sigma \times \Sigma$, соединенных своими главными портами называется *активной парой* (редексом). Правило $((\alpha, \beta) \Longrightarrow N)$ заменяет активную пару (α, β) на сеть N. Для каждой пары агентов существует не более одного правила редукции, при этом в процессе редукции интерфейс сохраняется.



Одним из важных свойств системы является *свойство ромба*: порядок переписываний не важен и все последовательности переписывания имеют одну и ту же длину. Из этого следуют практически значимые вещи: преобразовывать граф можно в любом порядке, кроме того, это можно делать параллельно.

Выбор базиса агентов. Поскольку описание Interaction Nets не фиксирует набор агентов, пользователю предоставлены большие возможности по созданию собственных наборов и правил их редукции.

Так, например, в статье [15] Yves Lafont в качестве примера использует набор агентов $\Sigma = \{\text{Cons}, \text{Nil}, \text{Append}\}$ и правила редукции, соответствующие спискам в функциональных языках программирования. А в статье [14] обсуждается базис $\Sigma = \{\gamma, \delta, \varepsilon\}$, который по своим свойствам схож с базисом SKI в λ-исчислении.

Отдельно можно отметить наличие множества базисов для трансляции λ-исчисления в Interaction Nets: [3, 8, 16, 17, 29, 33].

Стратегия вычислений. В отличие от λ-исчисления, модель Interaction Nets не предполагает наличие нескольких стратегий редукции сама по себе. Тем не менее, поскольку в Interaction Nets возможно закодировать другие формальные системы, то стратегия редукции может проявиться в Interaction Nets в различных формах записи агентов и правил их редукции. Так, например, в Interaction Nets можно закодировать как строгую [29], так и ленивую [30] стратегии для λ-исчисления.

2. Обзор существующих решений

Со времен публикации первых статей был разработан не один исполнитель Interaction Nets. Обзор на момент 2014 года можно найти в работе [27]. Мы же приведём обзор более новых работ в данной области.

Inpla и Train. Inpla[2] — интерпретатор одного из крупных учёных в области Shinya Sato, начатый в его PhD [27], и поддерживающий параллельное исполнение [18].

Inpla предполагает описание сетей на собственном языке программирования, который тем не менее достаточно сложен для использования. Транслятор Train[3] решает данную проблему, конвертируя код из функционального языка программирования в код для Inpla. Весь комплекс реализован на C.

HVM 1, 2, 3 и Bend. Семейство проектов от стартапа Higher Order Company[4]. HVM (Higher-order Virtual Machine) является средой исполнения Interaction Nets, эксплуатирующей параллельность, и существует в нескольких версиях, отличающихся языком реализации, стратегией вычислений и средой исполнения.

HVM1 Написана на Rust, с ленивой стратегией вычисления на CPU. Считается устаревшей.

HVM2 Написана на Rust, со строгой стратегией, поддерживает как CPU через кодогенерацию в C, так и GPU через генерацию в CUDA. На данный момент является стабильной версией.

HVM3 Написана на Haskell; поддерживает как ленивую, так и строгую стратегии вычисления на CPU. Является наследником HVM1 и HVM2, и создается, чтобы их заменить; находится в активной разработке.

[2]Репозиторий проекта: https://github.com/inpla/inpla/ (дата обращения: 15 февраля 2025 г.)

[3]Репозиторий проекта: https://github.com/inpla/train/ (дата обращения: 15 февраля 2025 г.)

[4]GitHub организация проекта: https://github.com/HigherOrderCO/ (дата обращения: 17 февраля 2025 г.)

HVM1 использовал собственный высокоуровневый язык для описания программ. В HVM2 и HVM3 же используется низкоуровневое промежуточное представление для описания Interaction Nets. Авторы предполагают, что пользователи вместо него будут использовать высокоуровневый язык Bend, который затем будет транслироваться в низкоуровневое представление.

lambda. lambda[5] — интерпретатор λ-исчисления, реализованный на JavaScript, поддерживающий четыре стратегии трансляции в Interaction Nets [26]. Принимает программы на собственном языке программирования, похожем на λ-исчисление.

interact. interact[6] — интерпретатор, написанный на Scala. Имеет свой язык программирования, похожий на OCaml и Python, где каждая функция становится агентом сети.

Выводы. Таким образом, несмотря на существование множества различных интерпретаторов, большинство из них используют свои собственные языки программирования, часто синтаксически далёкие от массовых, и не декларируют используемые наборы агентов. Кроме того, на данный момент попыток разработать ускоритель на основе Interaction Nets не предпринималось.

[5]Репозиторий проекта: https://github.com/codedot/lambda/ (дата обращения: 17 февраля 2025 г.)

[6]Репозиторий проекта: https://github.com/szeiger/interact/ (дата обращения: 18 февраля 2025 г.)

3. Постановка задачи

Целью работы является разработка транслятора модельного функционального языка в Interaction Nets. Для её выполнения были поставлены следующие задачи:

1. Реализовать интерпретатор модельного ML-подобного языка.
   (a) Конкретный синтаксис языка.
   (b) AST и синтаксический анализатор.
   (c) Алгоритм вывода типов.
   (d) Рассахаривание в обогащенное λ-исчисление.
   (e) Интерпретатор обогащенного λ-исчисления.
2. Реализовать интерпретатор Interaction Nets, поддерживающий сбор статистики.
3. Реализовать транслятор из обогащенного λ-исчисления в Interaction Nets.

4. Общая архитектура проекта

Проект Lamagraph ставит перед собой цель изучить возможности по созданию специализированных ускорителей на основе Interaction Nets.

К проекту выдвинуты следующие требования.

- Возможность параметризовать компилятор и вычислительное ядро типами агентов сети и правилами их редукции.
- Возможность сбора статистики такой, как размер сети, количество редукций, время исполнения и другой.
- Возможность постановки сравнительных экспериментов.
- Использование единого стека технологий — гомогенность.
- Получение полнофункционального прототипа, содержащего все компоненты, важнее, чем детальная проработка какого-то отдельного компонента.
- Расширяемость и модифицируемость. Должна быть возможность вносить изменения в любые компоненты.

Крупномасштабная архитектура проекта изображена на рисунке 1 и состоит из трёх крупных блоков.

Compiler Транслятор ML-подобного языка программирования. Содержит в себе интерпретатор и генерирует промежуточное представление, пригодное к дальнейшей трансляции в Interaction Nets.

Hardware Генератор описания аппаратуры для ПЛИС, параметризуемый базисом агентов сети.

Middle Транслятор из промежуточного представления в байт-код, пригодный для исполнения на процессоре, сгенерированном в блоке Hardware.

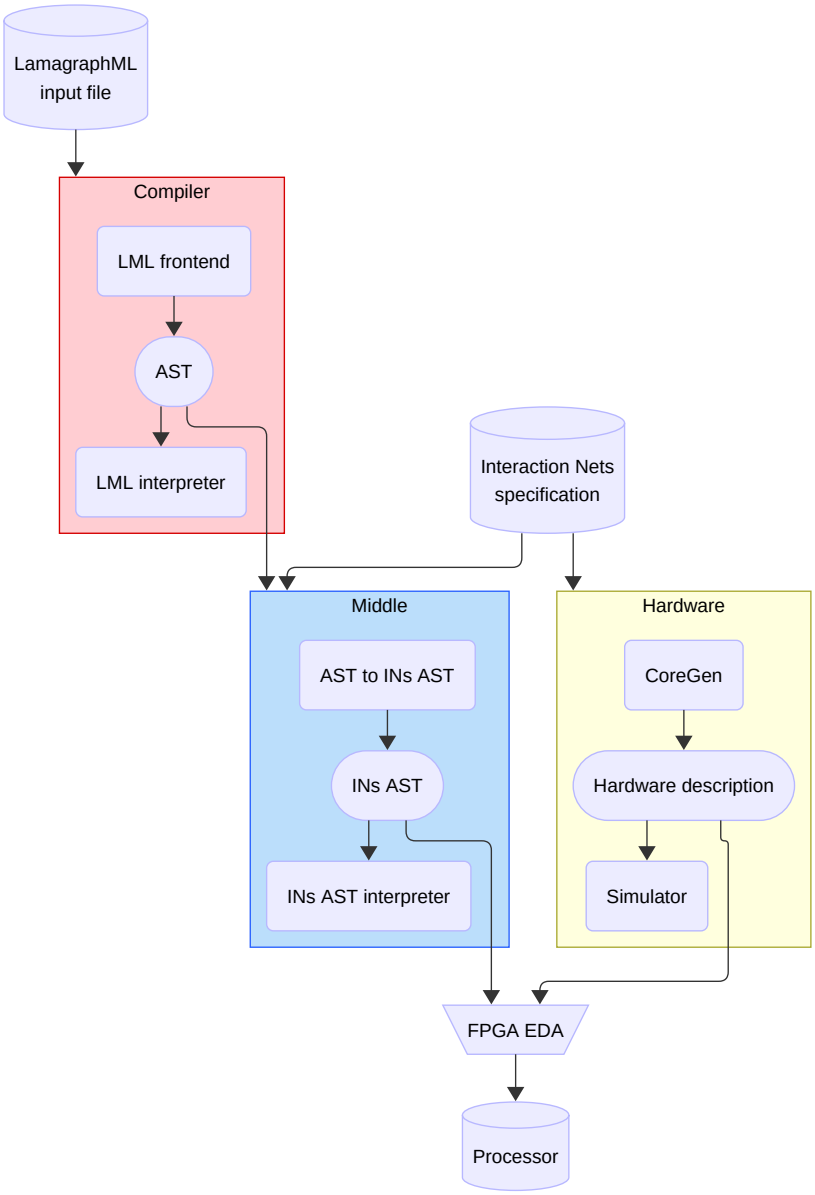


Рис. 1: Общая архитектура проекта Lamagraph.

Проект является командным. Блоки Compiler и Hardware достаточно независимы друг от друга. Блок Middle соединяет два других, поэтому его разработка должна быть скоординирована. В данной работе речь пойдёт о блоке Compiler и части блока Middle.

5. Подробности реализации

В силу требования гомогенности, удобства работы с древовидными типами данных, естественно возникающими в компиляторах, и наличия языка описания аппаратуры Clash[7], используемого в блоке Hardware, в качестве языка реализации проекта был выбран Haskell. Архитектура изображена на рисунке 2.

Для сборки проекта и версионирования зависимостей используется Stack[8], предоставляющий снимки репозитория пакетов с гарантированно совместимыми пакетами. Для тестирования используется фреймворк Tasty[9], позволяющий через единый интерфейс писать и запускать различные виды тестов: модульные, golden, property-based. Поскольку тестирование трансляторов с помощью модульных тестов довольно сложно, по умолчанию все части компилятора покрываются интеграционными golden[10] тестами.

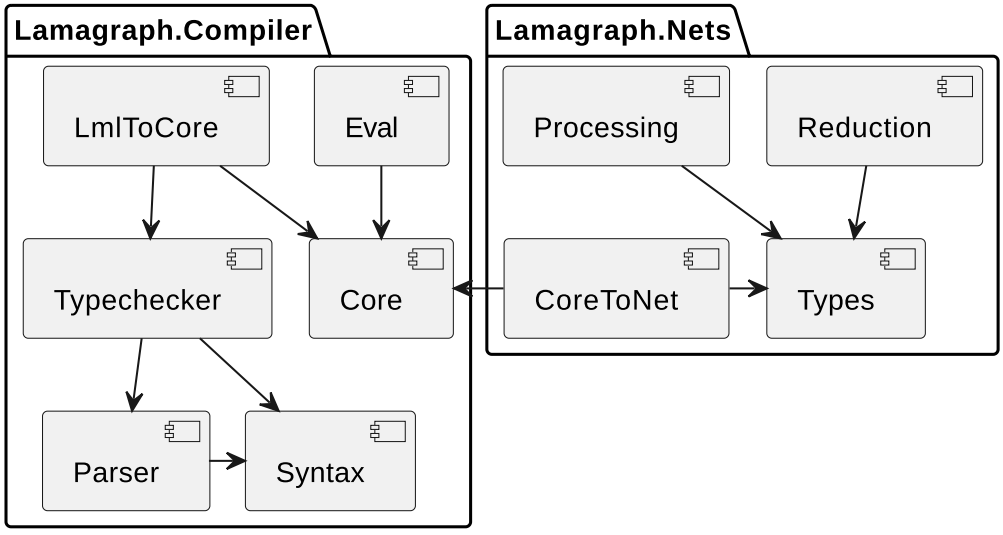


Рис. 2: Архитектура транслятора

[7]Сайт проекта: https://clash-lang.org/ (дата обращения 18 мая 2025 г.)

[8]Сайт проекта: https://docs.haskellstack.org/en/stable/ (дата обращения 25 февраля 2025 г.)

[9]Описание проекта на Hackage: https://hackage.haskell.org/package/tasty/ (дата обращения 25 февраля 2025 г.)

[10]Описание golden тестов: https://ro-che.info/articles/2017-12-04-golden-tests (дата обращения 25 февраля 2025 г.)

Синтаксис языка. Синтаксис языка основан на OCaml. Однако упрощен для простоты реализации[11]. Так, например, в языке остались стандартные для функциональных языков конструкции, такие как сопоставление с образцом, рекурсивные и взаимнорекурсивные функции, а также алгебраические типы данных. В отличие от OCaml отсутствует поддержка классов и функторов, а система модулей максимально упрощена и напоминает систему модулей в F#.

Для представления AST (дерева абстрактного синтаксиса) используется паттерн Trees That Grow (TTG) [28]. Он позволяет с помощью механизма type families [32] гибко параметризовать дерево необходимыми аннотациями, более того аннотации могут различаться в разных узлах дерева, тем самым поддерживая безопасность кода.

Парсер. Для синтаксического анализа используется стандартная для Haskell связка лексера Alex и парсер-генератора Happy, которые являются аналогами flex и bison и используются в крупных проектах, например GHC.

На данном этапе работы транслятора аннотации с помощью TTG не используются.

Для тестирования лексера используются модульные тесты. Для парсера используется property-based тестирование с использованием библиотеки Hedgehog[12]. Данный метод основан на том, что синтаксический анализ в AST и печать AST должны давать тождественное отображение при композиции.

Вывод типов. Поскольку язык ML-подобный, используется система типов Хиндли-Милнера [10, 19].

После вывода типов в аннотациях TTG сохраняется тип каждого узла дерева.

[11] С полной грамматикой можно ознакомиться в репозитории проекта: https://github.com/Lamagraph/interaction-nets-in-fpga/blob/main/lamagraph-compiler/src/Lamagraph/Compiler/Syntax.hs (дата обращения 25 февраля 2025 г.)

[12] Репозиторий проекта: https://github.com/hedgehogqa/haskell-hedgehog/ (дата обращения: 19 февраля 2025 г.)

```haskell
data Literal = LitInt Int | LitChar Char | LitString Text

type DataCon = Name

newtype Var = Id Name

data Expr b
  = Var b
  | Lit Literal
  | App (Expr b) (Expr b)
  | Lam b (Expr b)
  | Let (Bind b) (Expr b)
  | Match (Expr b) b (NonEmpty (MatchAlt b))
  | Tuple (Expr b) (NonEmpty (Expr b))

type MatchAlt b = (AltCon, [b], Expr b)

data AltCon = DataAlt DataCon | LitAlt Literal | TupleAlt | DEFAULT

data Bind b = NonRec b (Expr b) | Rec (NonEmpty (b, Expr b))

type CoreExpr = Expr Var
type CoreMatchAlt = MatchAlt Var
type CoreBind = Bind Var
```

Листинг 1: Представление Core в алгебраических типах данных.

В силу ограниченности времени, на данном этапе отсутствует поддержка пользовательских алгебраических типов данных. Тем не менее поддержка типов `list` и `option` присутствует.

Промежуточное представление. AST, получаемое после синтаксического анализа и вывода типов, содержит большое количество различных типов узлов с аннотациями. Для дальнейшей обработки используется упрощенное представление на основе GHC Core[13], также часто называемое обогащённым λ-исчислением[14]. Его описание представлено на листинге 1.

Представление, используемое в данной работе, отличается от GHC Core наличием выделенного конструктора для пар, а также отсутствием типизации.

Наличие выделенного конструктора для пар обусловлено различием

[13]Подробнее можно прочитать по ссылке: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-syn-type (дата обращения: 19 февраля 2025 г.)

[14]Подробнее узнать про способы обогащения λ-исчисления можно в [22, раздел 3.2]

между ML и Haskell — в первом пары тоже выделены и могут быть какой угодно арности, во втором же пара является синтаксическим сахаром для алгебраического типа-суммы и ограничена арностью 63.

От типизации пришлось отказаться в угоду простоты реализации, а также в виду отсутствия оптимизаций со стороны компилятора, где наличие типов упрощает и делает более безопасным их применение.

Трансляция высокоуровневого AST в промежуточное представление. На данный момент алгоритм трансляции достаточно прямолинеен. Связано это с тем, что в левой части let-связываний поддерживаются только простые шаблоны (переменные), а в match-выражениях не поддерживаются вложенные шаблоны, защитные выражения и ИЛИ-шаблоны.

Интерпретатор обогащенного λ-исчисления. Интерпретатор реализован на основе идей из [24]. Для реализации функций высшего порядка используются замыкания, для рекурсивных let-связываний используется «ленивость» Haskell для создания рекурсивных замыканий, остальная часть работы интерпретатора довольно прямолинейна.

В текущей реализации интерпретатор наследует порядок исполнения от языка, на котором написан, в случае Haskell — это call-by-need. Это можно исправить с помощью продолжений (continuations), однако поскольку схема трансляции и порядок исполнения Interaction Nets по умолчанию не определен, на данном этапе было решено оставить call-by-need.

Кроме того, на данный момент отсутствует поддержка взаимной рекурсии.

Интерпретатор Interaction Nets. Стандартным представлением для Interaction Nets является графовое. Однако для представления графов в функциональных языках не удаётся использовать их возможности по работе с алгебраическими типами данных и сопоставлением

с образцом[15], поэтому потребовалось найти альтернативное представление. Таковым стало текстовое представление Fernández и Mackie [7]. Для него также существует абстрактная машина [23], которая и была реализована.

Транслятор обогащенного λ-исчисления в Interaction Nets. Как было сказано в разделе 1, существует не один способ трансляции λ-исчисления в Interaction Nets. Поскольку мы транслируем ML-подобный язык, то первой схемой трансляции была выбрана схема call-by-value из [29]. Для λ-абстракции и аппликации используются выделенные агенты, переменные же выражаются с помощью проводов. В случае, если какая-то переменная используется более одного раза, используются агенты-дупликаторы.

В процессе реализации была обнаружена проблема: существующие схемы трансляции работают только с чистым λ-исчислением. Попытки расширить схему трансляции предпринимались в [12], однако такой подход требует использование динамического базиса агентов. Но поскольку вычислитель прошивается на ПЛИС, то базис агентов должен быть фиксирован, и подход предложенный в статье нам не подходит.

Проблему можно попробовать решить на уровне λ-исчисления, используя кодировки Чёрча [4] или Скотта [13]. Однако скорее всего сети в таком случае будут получаться сложнее, чем могли бы, и будут требовать больше шагов редукции, кроме того такой подход отчасти противоречит параметризуемости Interaction Nets. Поэтому трансляция в Interaction Nets конструкций обогащённого λ-исчисления — предмет дальнейшей работы.

Однако одну конструкцию обогащенного λ-исчисления, let-связывание, удалось транслировать. Для нерекурсивых связываний использовалась схема, похожая на схему работы с переменными; для рекурсивных, функция дополнительно применялась к комбинатору неподвижной точки.

[15] Алгебраические графы, описанные в [21], позволяют использовать сопоставление с образцом для обработки графов. Однако неизвестно, можно ли выразить все особенности Interaction Nets с помощью данной кодировки (например, упорядоченность дополнительных портов), кроме того неизвестно насколько естественно выражаются необходимые для редукции трансформации.

6. Тестирование транслятора

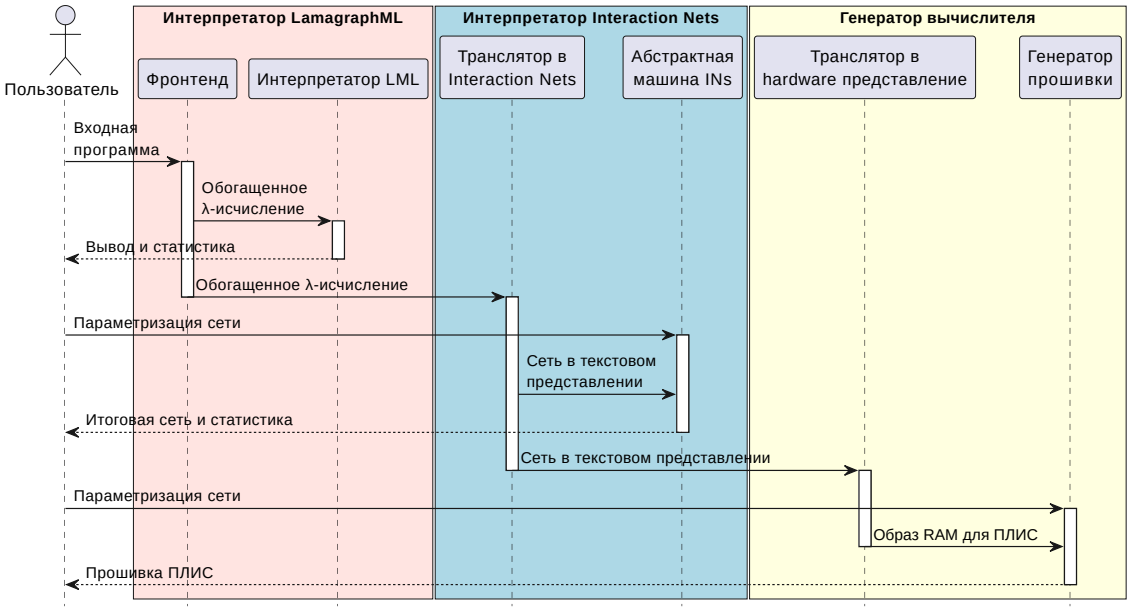


Рис. 3: Диаграмма взаимодействия пользователя с программно-аппаратным стеком Lamagraph.

На данном этапе развития проекта проведение сравнительных экспериментов невозможно, тем не менее пользователь уже может взаимодействовать с инструментом.

Так, передав на вход программу на LamagraphML пользователь может получить вывод интерпретатора, а также вывод абстрактной машины и соответствующую статистику, что проиллюстрировано на рисунке 3.

Анализ текущих возможностей

Анализ проводился с учётом реализованной схемы трансляции чистого λ-исчисления в Interaction Nets со стратегией call-by-value. Для вычисления использовались два набора правил: для последовательного вычисление аргументов применения и для параллельного. Оба набора описаны в [29] и отличаются лишь правилом для взаимодействия агентов $\Downarrow$ и a.

Допустимые термы. Для проведения анализа было написано 10 программ. Первоначально планировалось писать программы только на высокоуровневом LamagraphML, однако в процессе выяснилось, что не все термы, используемые в бестиповом λ-исчислении могут быть протипизированы в системе типов Хиндли-Милнера без использования алгебраических типов данных (например, терм `pred`[16]). Поэтому часть программ была вручную написана на бестиповом Core. Кроме того, для корректного завершения программ в стратегии call-by-value условные конструкции необходимо было сделать «отложенными», введя дополнительную λ-абстракцию[17].

Данная проблема — одна из самых важных для проекта на данный момент, поскольку Core и Interaction Nets планировалось использовать, как промежуточные представления, предоставляя пользователю только LamagraphML и соответствующие возможности высокоуровневого языка.

Один из вариантов решения данной проблемы — введение числовых литералов, примитивных арифметических операцией и сопоставления с образцом. Такое решение потребует расширения набора агентов и правил редукции, с учётом того, что арифметические операции строгие (eager) в своих аргументах. Кроме того, необходимо изучить, каким образом возможна реализации такого расширения в компоненте Hardware.

Корректность результата. При анализе получаемых сетей была выявлена ещё одна проблема: call-by-value стратегия по определению не редуцирует под λ-абстракцией из-за чего получаемые сети довольно сложно анализировать на корректность ответа.

Так, например, в нотации из [7], результат вычисления итеративного факториала от 0 (см. листинг 2) выглядит, как

$$\langle \Uparrow (\lambda(f, \lambda(x, a(f, x)))) \mid \rangle,$$

[16]Подробности можно прочитать в заметке Олега Киселева: https://okmij.org/ftp/Computation/lambda-calc.html#predecessor (дата обращения: 22 мая 2025 г.)

[17]Более подробная мотивация такого подхода описана по ссылке: https://cs.stackexchange.com/a/121350 (дата обращения: 22 мая 2025 г.)

```
let zero = fun f -> fun x -> x
let one = fun f -> fun x -> f x

let succ = fun n -> fun f -> fun x -> f (n f x)
let mult = fun m -> fun n -> fun f -> fun x -> m (n f) x

let pair = fun x -> fun y -> fun k -> k x y
let fst = fun p -> p (fun x -> fun y -> x)
let snd = fun p -> p (fun x -> fun y -> y)

let fact_aux = fun n -> n (fun p -> pair (succ (fst p)) (mult (succ (fst p)) (snd p)))
↪  (pair zero one)

let fact = fun n -> snd (fact_aux n)

let res = fact zero
```

Листинг 2: `IterFact(0)`, реализованный на LamagraphML

а для факториала от 1 уже так:

$$\Big\langle \Uparrow (\lambda(f, \lambda(x, a(a(\lambda(c(u, v), \lambda(w, a(u, a(a(\lambda(\varepsilon, \lambda(z, z)), v), w))))), \\ a(\lambda(t, \lambda(r, a(t, r))), f)), x)))) \mid \Big\rangle.$$

В качестве решения проблемы можно, как уже упоминалось выше, добавить поддержку числовых литералов. Также можно попробовать транслировать сеть обратно в λ-терм и осуществить оставшиеся редукции руками или с помощью интерпретатора, реализующего порядок вычислений, редуцирующий внутри λ-абстракции.

Отдельного внимания заслуживает мысль о смене схемы трансляции и использовании другой стратегии редукции. С учётом существования нескольких способов кодирования «оптимальной» редукции [3, 8, 16, 33], смена стратегии вычислений может позволить редуцировать сети до нормальный формы, соответствующего λ-терма.

Сбор статистики. Поскольку Interaction Nets — параллельная модель вычислений, то при сборе статистики абстрактной машины нам было интересно не только количество малых шагов, но и метрики, связанные с параллельностью: количество больших шагов и максимальная ширина большого шага.

Здесь под «малым шагом» подразумевается редукция одной активной

Терм	Параллельная стратегия	Количество малых шагов	Максимальная ширина большого шага	Количество больших шагов
`RecFact(0)`	Нет	472	18	103
`RecFact(0)`	Да	498	18	111
`RecFact(1)`	Нет	274	13	85
`RecFact(1)`	Да	302	12	94
`RecFact(2)`	Нет	291	13	91
`RecFact(2)`	Да	321	12	101
`IterFact(0)`	Нет	179	12	61
`IterFact(0)`	Да	199	12	62
`IterFact(1)`	Нет	265	9	176
`IterFact(1)`	Да	323	11	124
`IterFact(2)`	Нет	607	12	295
`IterFact(2)`	Да	703	14	190
`IterFact(3)`	Нет	1097	22	414
`IterFact(3)`	Да	1231	22	256
`IterFact(4)`	Нет	1766	33	533
`IterFact(4)`	Да	1938	33	322
`IterFact(5)`	Нет	2650	42	652
`IterFact(5)`	Да	2860	42	388
`IterFact(6)`	Нет	3785	53	771
`IterFact(6)`	Да	4033	53	454

Таблица 1: Статистика работы абстрактной машины Interaction Nets.

пары, под «большим шагом» — редукция, при которой сокращаются все возможные активные пары. Под «шириной большого шага» имеется в виду количество малых шагов в одном большом. Больших шагов совершается много, и каждый имеет свою ширину, поэтому в статистику попадает только максимальная ширина большого шага.

Отдельно стоит отметить, что последовательная и параллельная схема вычисления аргументов — это объекты уровня Interaction Nets, а большой и малый шаги — уровня абстрактной машины. Таким образом последовательная схема вычисления тоже имеет большие шаги шириной больше одного, но они вызваны не самой стратегией вычисления λ-термов, а вспомогательными агентами, например, дупликаторами.

Статистика представлена в таблице 1. Для количества малых и больших шагов: меньше — лучше, для максимальной ширины большого шага: больше — лучше. Некоторые программы приведены на листингах 2

```
let true = fun x -> fun y -> x
let false = fun x -> fun y -> y
let if = fun b -> fun t -> fun f -> (b t) f

let zero = fun f -> fun x -> x
let one = fun f -> fun x -> f x
let pred = fun n -> fun f -> fun x -> ((n (fun g -> fun h -> h (g f))) (fun u -> x)) (fun
↪  u -> u)
let mult = fun m -> fun n -> fun f -> fun x -> (m (n f)) x
let isZero = fun n -> (n (fun x -> false)) true

let rec fact = fun n -> ((if (isZero n)) (fun z -> one)) (fun z -> (mult n) (fact (pred
↪  n)))

let res = fact zero
```

Листинг 3: `RecFact(0)`, реализованный на Core

и 3. Остальные программы получаются очевидной заменой аргументов.

Можно заметить, что количество малых шагов при использовании параллельной стратегии больше, чем при обычной. Скорее всего это связано с тем, что для управления процессом вычисления текущий набор правил использует специальные метки, и именно работа с ними увеличивает количество редукций.

По количеству больших шагов сделать однозначные выводы сложно: для рекурсивного факториала (см. листинг 3) параллельная стратегия лишь увеличивает их количество, однако для итеративного уменьшает. Возможно, это связано с тем, что итеративый факториал содержит гораздо меньше λ-абстракций в аргументах и потому может быть более эффективно распараллелен.

Ширина большого шага практически не меняется от выбора стратегии, скорее всего это связано с тем, что при трансляции Core в Interaction Nets для общих подтермов генерируются агенты-дупликаторы, которые обязательно редуцируются. Для получения альтернативной статистики возможно использовать другую стратегию трансляции, которая бы не полагалась на явное копирование подтермов.

Дальнейшний анализ статистики. После проведения замеров был проведен дополнительный анализ полученной статистики на примере итеративного факториала.

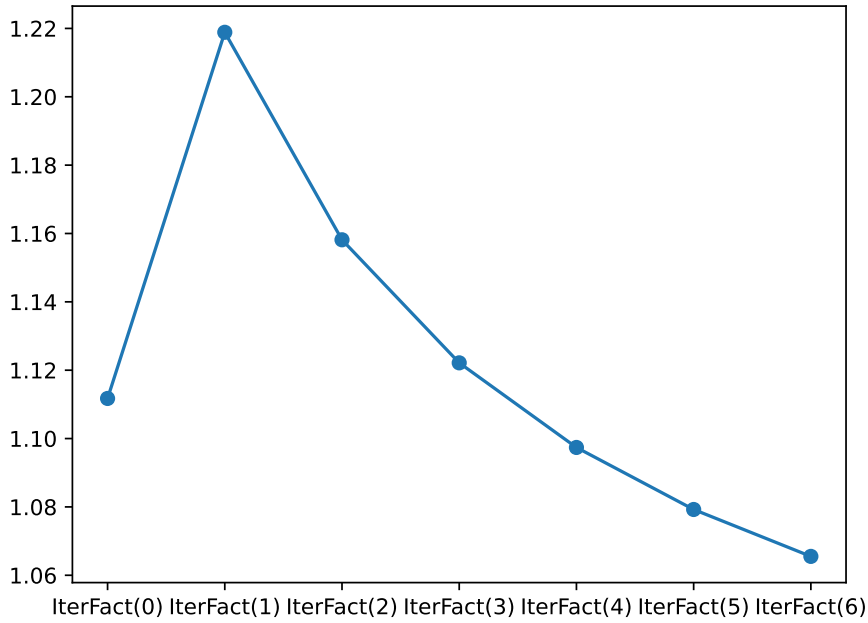


Рис. 4: Отношение количества малых шагов при использовании параллельного набора правил вычисления аргументов и последовательного

Так, на рисунке 4 изображен график отношения количества малых шагов при использовании параллельного набора правил вычисления аргументов и последовательного. По нему можно сделать вывод о том, что использование параллельного набора правил при небольшом объеме вычислений привносит дополнительные редукции, но с усложнением программы количество дополнительных редукций становится мало в процентном соотношении.

На рисунке 5 же представлен график отношения количества больших шагов при использовании последовательного набора правил вычисления аргументов и параллельного. На его основе можно сделать вывод, что параллельный набор правил даёт значительное уменьшение количества больших шагов (порядка 1.67 раз).

Рисунок 6 построен следующим образом: для каждого большого шага была измерена его ширина, затем посчитано сколько раз каждая ширина встречалась среди всех шагов, и затем построена гистограмма. Несмотря на то, что во всех случаях чаще всего ширина большого шага — 1, в

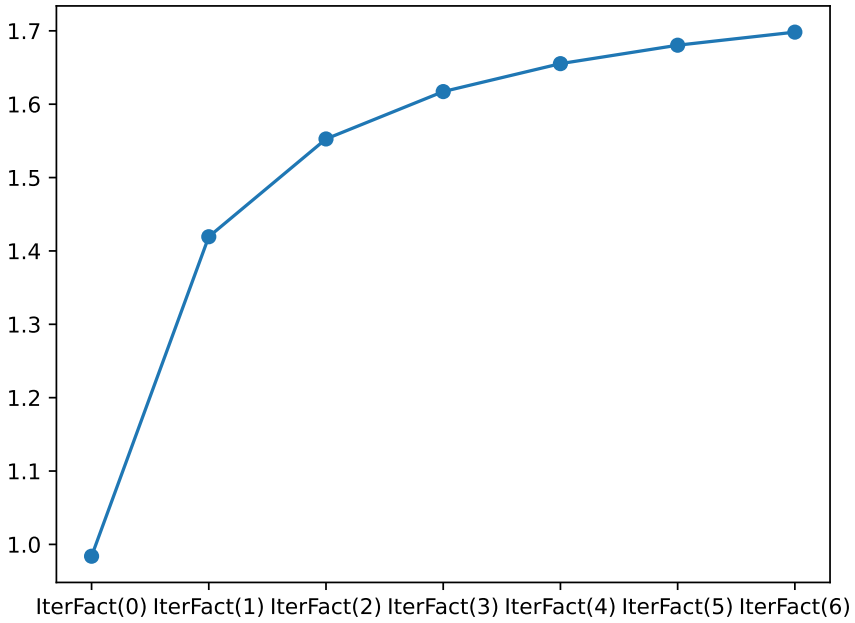


Рис. 5: Отношение количества больших шагов при использовании последовательного набора правил вычисления аргументов и параллельного

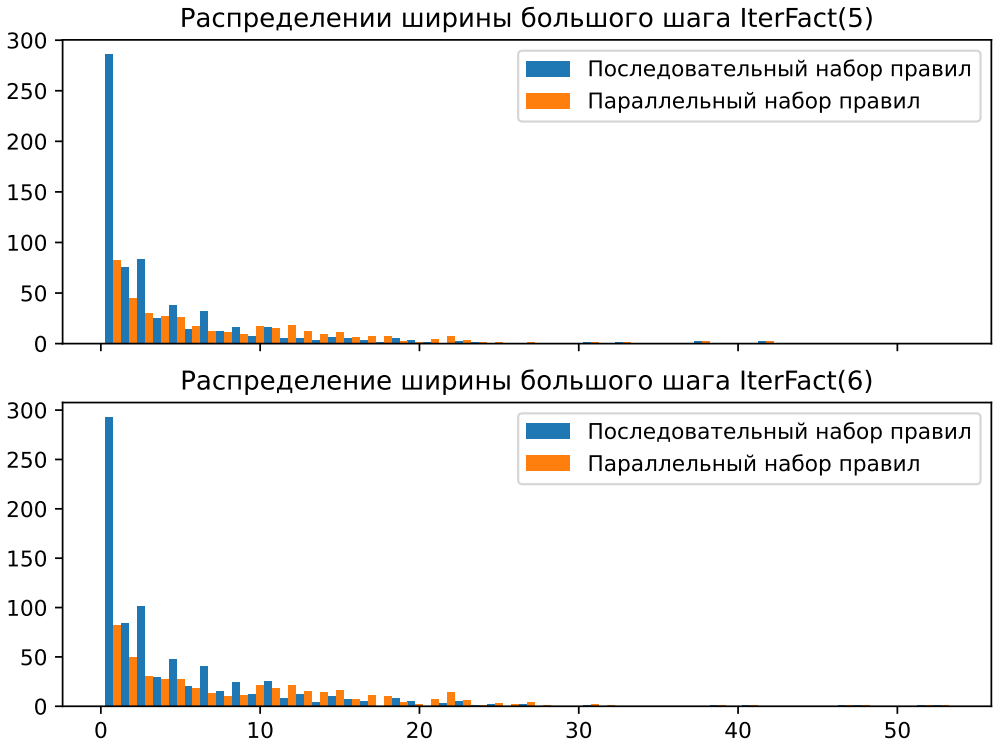


Рис. 6: Распределение ширин большого шага

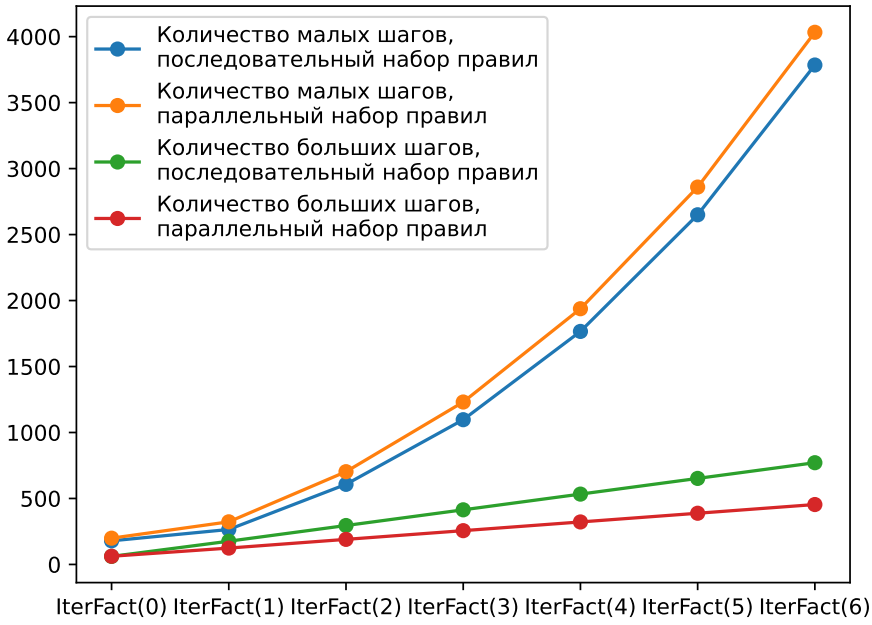


Рис. 7: График количества больших и малых шагов

параллельном наборе правил ширина от 2 до 20 наблюдается гораздо чаще, чем при использовании последовательного, что позволяет говорить о том, что параллельный набор правил увеличивает параллельность.

На рисунке 7 изображен сводный график количества больших и малых шагов.

Выводы. Тестирование позволило оценить текущее состояние проекта: программный стек целиком работоспособен, хоть и требует доработки. Бóльшая часть проблем оказалась связана с ограничениями существующих схем трансляции λ-термов в Interaction Nets, в частности, с отсутствием поддержки числовых литералов и алгебраических типов данных. К сожалению оценить реальную производительность в данный момент невозможно, тем не менее, полученная статистика исполнения позволяет говорить о том, что трансляция λ-термов в Interaction Nets позволяет повысить параллельность редукции λ-термов, при использовании соответствующих схем трансляции и наборов правил.

Заключение

В рамках выпускной квалификационной работы были достигнуты следующие результаты.

1. Реализован интерпретатор модельного ML-подобного языка.
   (a) Конкретный синтаксис языка.
   (b) AST и синтаксический анализатор.
   (c) Алгоритм вывода типов.
   (d) Рассахаривание в обогащенное λ-исчисление.
   (e) Интерпретатор обогащенного λ-исчисления.
2. Реализован интерпретатор Interaction Nets, поддерживающий сбор статистики в виде количества редукций с учётом возможных параллельных редукций.
3. Реализован транслятор чистого λ-исчисления в Interaction Nets со стратегией вычислений call-by-value.

Исходный код находится в репозитории: https://github.com/Lamagraph/interaction-nets-in-fpga/. Имя коммитера: `WoWaster`.

Планы на будущее

На данный момент реализован лишь минимально рабочий стек трансляции: от высокоуровневого языка в обогащенное λ-исчисление в Interaction Nets. В ближайшие планы по развитию проекта входит:

- закончить реализацию некоторых компонент, например добавить поддержку алгебраических типов данных в вывод типов и взаимной рекурсии в интерпретатор;
- развить схему трансляции в Interaction Nets до полной поддержки обогащенного λ-исчисления;
- реализовать библиотеку для разреженной линейной алгебры.

Список литературы

[1] Accelerating Reduction and Scan Using Tensor Core Units / Abdul Dakkak, Cheng Li, Isaac Gelado et al. // Proceedings of the ACM International Conference on Supercomputing. — P. 46–57. — arXiv : cs/1811.09736.

[2] Akkad Ghattas, Mansour Ali, Inaty Elie. Embedded Deep Learning Accelerators: A Survey on Recent Advances. — Vol. 5, no. 5. — P. 1954–1972. — URL: https://ieeexplore.ieee.org/document/10239336/?arnumber=10239336.

[3] Asperti Andrea, Giovannetti Cecilia, Naletto Andrea. The Bologna Optimal Higher-Order Machine. — Vol. 6, no. 6. — P. 763–810. — URL: https://www.cambridge.org/core/product/identifier/S0956796800001994/type/journal_article.

[4] Barendregt Henk. The Impact of the Lambda Calculus in Logic and Computer Science. — Vol. 3, no. 2. — P. 181–215. — URL: https://www.cambridge.org/core/product/identifier/S1079898600007599/type/journal_article.

[5] Davis Timothy A., Hu Yifan. The University of Florida Sparse Matrix Collection. — Vol. 38, no. 1. — P. 1:1–1:25. — URL: https://doi.org/10.1145/2049662.2049663.

[6] Dedicated Hardware Accelerators for Processing of Sparse Matrices and Vectors: A Survey / Valentin Isaac–Chassande, Adrian Evans, Yves Durand, Frédéric Rousseau. — Vol. 21, no. 2. — P. 1–26. — URL: https://dl.acm.org/doi/10.1145/3640542.

[7] Fernández Maribel, Mackie Ian. A Calculus for Interaction Nets // Principles and Practice of Declarative Programming / Ed. by Gopalan Nadathur. — Springer Berlin Heidelberg. — Vol. 1702. — P. 170–187. — URL: http://link.springer.com/10.1007/10704567_10.

[8] Gonthier Georges, Abadi Martín, Lévy Jean-Jacques. The Geometry of Optimal Lambda Reduction // Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '92. — Association for Computing Machinery. — P. 15–26. — URL: https://dl.acm.org/doi/10.1145/143165.143172.

[9] High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV / Yixiao Du, Yuwei Hu, Zhongchun Zhou, Zhiru Zhang // Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. — ACM. — P. 54–64. — URL: https://dl.acm.org/doi/10.1145/3490422.3502368.

[10] Hindley R. The Principal Type-Scheme of an Object in Combinatory Logic. — Vol. 146. — P. 29–60. — jstor : 1995158.

[11] In-Datacenter Performance Analysis of a Tensor Processing Unit / Norman P. Jouppi, Cliff Young, Nishant Patil et al. // Proceedings of the 44th Annual International Symposium on Computer Architecture. — ISCA '17. — Association for Computing Machinery. — P. 1–12. — URL: https://dl.acm.org/doi/10.1145/3079856.3080246.

[12] Jiresch Eugen Robert Winfried. Extending Interaction Nets towards the Real World : Thesis / Eugen Robert Winfried Jiresch. — URL: https://repositum.tuwien.at/handle/20.500.12708/12949.

[13] Koopman Pieter, Plasmeijer Rinus, Jansen Jan Martin. Church Encoding of Data Types Considered Harmful for Implementations: Functional Pearl // Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages. — ACM. — P. 1–12. — URL: https://dl.acm.org/doi/10.1145/2746325.2746330.

[14] Lafont Yves. Interaction Combinators. — Vol. 137, no. 1. — P. 69–101. — URL: https://www.sciencedirect.com/science/article/pii/S0890540197926432.

[15] Lafont Yves. Interaction Nets // Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '90. — Association for Computing Machinery. — P. 95–108. — URL: https://dl.acm.org/doi/10.1145/96709.96718.

[16] Lamping John. An Algorithm for Optimal Lambda Calculus Reduction // Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '90. — ACM Press. — P. 16–30. — URL: http://portal.acm.org/citation.cfm?doid=96709.96711.

[17] Mackie Ian. An Interaction Net Implementation of Closed Reduction // Implementation and Application of Functional Languages / Ed. by Sven-Bodo Scholz, Olaf Chitil. — Springer Berlin Heidelberg. — Vol. 5836. — P. 43–59. — URL: http://link.springer.com/10.1007/978-3-642-24452-0_3.

[18] Mackie Ian, Sato Shinya. Parallel Evaluation of Interaction Nets: Case Studies and Experiments. — Vol. 73. — URL: https://eceasst.org/index.php/eceasst/article/view/2205.

[19] Milner Robin. A Theory of Type Polymorphism in Programming. — Vol. 17, no. 3. — P. 348–375. — URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[20] Mohammed Thaha, Mehmood Rashid. Performance Enhancement Strategies for Sparse Matrix-Vector Multiplication (SpMV) and Iterative Linear Solvers. — arXiv : cs/2212.07490.

[21] Mokhov Andrey. Algebraic Graphs with Class (Functional Pearl) // Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. — ACM. — P. 2–13. — URL: https://dl.acm.org/doi/10.1145/3122955.3122956.

[22] Peyton Jones Simon L. The Implementation of Functional Programming Languages. — Prentice Hall Internaltional (UK) Ltd. — URL: https:

//www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages-2/.

[23] Pinto Jorge Sousa. Sequential and Concurrent Abstract Machines for Interaction Nets // Foundations of Software Science and Computation Structures / Ed. by Jerzy Tiuryn. — Springer. — P. 267–282.

[24] Reynolds John C. Definitional Interpreters for Higher-Order Programming Languages // Proceedings of the ACM Annual Conference on - ACM '72. — Vol. 2. — ACM Press. — P. 717–740. — URL: http://portal.acm.org/citation.cfm?doid=800194.805852.

[25] Salikhmetov Anton. Interaction Nets in Russian. — arXiv : cs/1304.1309.

[26] Salikhmetov Anton. Token-Passing Optimal Reduction with Embedded Read-back. — Vol. 225. — P. 45–54. — arXiv : cs/1609.03644.

[27] Sato Shinya. Design and Implementation of a Low-Level Language for Interaction Nets : thesis / Shinya Sato. — URL: https://sussex.figshare.com/articles/thesis/Design_and_implementation_of_a_low-level_language_for_interaction_nets/23417312/1.

[28] Shayan Najd, Simon Peyton Jones. Trees That Grow. — URL: https://lib.jucs.org/article/22912.

[29] Sinot François-Régis. Call-by-Name and Call-by-Value as Token-Passing Interaction Nets // Typed Lambda Calculi and Applications / Ed. by Paweł Urzyczyn. — Springer Berlin Heidelberg. — Vol. 3461. — P. 386–400. — URL: http://link.springer.com/10.1007/11417170_28.

[30] Sinot François-Régis. Token-Passing Nets: Call-by-Need for Free. — Vol. 135, no. 3. — P. 129–139. — URL: https://www.sciencedirect.com/science/article/pii/S1571066106000934.

[31] Silvano Cristina, Ielmini Daniele, Ferrandi Fabrizio et al. A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms. — arXiv : cs/2306.15552.

[32] Type Checking with Open Type Functions / Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, Martin Sulzmann. — URL: https://www.researchgate.net/publication/221241290_Type_checking_with_open_type_functions.

[33] Vincent van Oostrom, Kees-Jan van de Looij, Marijn Zwitserlood. Lambdascope Another Optimal Implementation of the Lambda-Calculus. — URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=61042374787bf6514706b49a5a4f0b74996979a0.

[34] Zhu Yuhao, Mattina Matthew, Whatmough Paul. Mobile Machine Learning Hardware at ARM: A Systems-on-Chip (SoC) Perspective. — arXiv : cs/1801.06274.