Кафедра Информационных Технологий
Технико-Экономическая Академия Кино и Телевидения

Согласовано

Директор АМО ЗИЛ

____________ *Иванов И. И.*

“_____” ______________ _________

Утверждаю

Зав. кафедрой

____________ *Гусев И. И.*

“_____” ______________ _________

Типизированный интерпретатор просто-типизированного λ-исчисления
Пояснительная записка к дипломному проекту

АБВГД XXXXXXXX

Главный инженер АМО ЗИЛ

____________ *Гусев*

“_____” ______________ _________

2010

Аңдатпа

Интерпретаторлар қазіргі ақпараттанудың негізі болып табылады, соңдықтаң олардың дәлдігінің маңызы жоғары. Ол интерпретатордың кейбір қасиеттіне тәуелді. Кейбір программалық инварианттарды деректердін кезелген түрлерінен бейгелугі болады. Ал орындалуын тексеріп тұруға болады.

Бұл жұмыста интерпретатор қыруға верификация түрлерімен қарапайым түрдегі лямбда-есептеулер қолданыңған. Дипломник ішні көрсетілім түрлеріне негізделген жұмысшы интерпретаторың құрды. Прогресс және сақтау сиякты есептеуш қасиетті мета-тіл түрлері жүйесіне негезделген және дәрелденген.

Ұсынылған шешемді қолданба-бағытталған тілдерді іске асыруға қолдануға болады. Сон мен бірге тілдік-бағытталған синтаксистік лексикалық талдауштар, атрибуттық грамматикалар сиякты қолдануға болады.

Аннотация

Интерпретаторы являются основой современной информатики, и поэтому их корректность имеет высокую важность. Корректность зависит от некоторых инвариантов. Известно, что некоторые программные инварианты удобно выразить с помощью типов, а их выполнение гарантировать с помощью проверки типов.

В данной работе основанная на типах верификация применяется к построению интерпретатора просто-типизированного лямбда-исчисления. Автором был сконструирован рабочий интерпретатор на основе типизированного внутреннего представления, состоящий из проверки типов и вычислителя. Свойства вычислителя, такие, как прогресс и сохранение, доказаны формально в системе типов мета-языка.

Предлагаемое решение возможно использовать для реализации встраиваемых предметно-ориентированных языков, а также языково-ориентированных инструментов, например, синтаксических и лексических анализаторов, атрибутных грамматик, и т.п.

ABSTRACT

Interpreters are at the heart of modern computing environment, and their correctness is of high importance. Correctness depends on a number of invariants, and it is known that certain program invariants can be conveniently captured with types and enforced by the type checker.

In this diploma we are to demonstrate the application of lightweight, type-based verification to the construction of a simply-typed lambda calculus interpreter (with extensions, such as general recursion, products and sums). We were able to integrate a typeful internal representation into a working interpreter, comprised of a type-checker and an evaluator. Progress and type-preservation properties of the evaluator are formally proven in the type system of the meta-language using dependent types.

The proposed solution can be used for the implementation of (embedded) domain-specific languages, and also language-based tools (parsers, lexers, attribute grammars, ORMs, etc.).

Содержание

Введение

В целом, программирование является процессом, подверженным серьезным ошибкам, и на практике получено много доказательств того, что применение системы типов в языке программирования позволяет обнаруживать некоторый класс программных ошибок во время компиляции, до запуска программы.

В данной работе основной интерес проявляется к обеспечению корректности интерпретаторов посредством типов. Такой интерес обусловлен тем, что от интерпретаторов зависит большое количество программ.

Традиционно, интерпретаторы пишут на Си или Си++, и в этом случае сложно получить гарантии (формальной) корректности. Подход, предпринятый в данной работе, состоит в том, чтобы закодировать правила вывода системы типов объектного языка в представлении абстрактного синтаксиса в мета-языке, получив т.н. *типизированное внутреннее представление*.

При разработке интерпретатора первым возникает вопрос о представлении объектного языка в терминах мета-языка. В случае, когда мета-язык является функциональным языком программирования, таким, как OCaml [1] или Haskell [2], обычно определяют (алгебраический) тип данных для представления программ на объектном языке. При этом возникают проблемы, связанные с тем, что вся информация о типах объектного языка теряется в типе его представления. Более того, поддержка переменных, связывания и подстановки требует особого внимания.

В данной работе внутреннее представление основано на типизированном абстрактном синтаксисе первого порядка, где программные переменные заменены индексами де Брауна. [3, 4] При таком подходе не только тип объектной программы, но и типы свободных переменных объектной программы отражаются в типе ее представления.

Ключевым результатом работы является реализация проверки типов и интерпретатора, использующие типизированное внутреннее представление, которое отражает дерево вывода типов просто-типизированного лямбда-исчисления [5] с расширениями, в ATS [6].

1 Теоретическая часть

1.1 История функционального программирования

В 60-х гг. XX в. λ-исчисление начало привлекать интерес группы исседователей вне сообщества логиков – ученых, занимавшихся теорией и практикой языков программирования.

С 1956 по 1960 Джон Маккарти из США разрабатывал язык программирования Лисп [7] для обработки списков с возможностью определения абстракций посредством функций. Целью Маккарти было применение Лиспа к проблемам нечислового вычисления, и особенно к новой области искусственного интеллекта, а также способствовать новому стилю организации программ, ныне называемым функциональным программированием.

В начале 60-х гг. в Англии Питер Ландин предложил использовать λ-термы для кодирования конструкций языка программирования Алгол-60. [8] Если в случае Лиспа точному соответствию с λ-исчислением препятствовала динамическая область видимости, то в случае с Алголом блочная структура точно соответствовала связыванию имен в λ-исчислении. Фактически работа Ландина позволила взглянуть на само λ-исчисление как на язык программирования, причем особенно подходящий для теоретических целей. В дальнейшем Ландин предложил язык программирования ISWIM, [9] который стал предшественником языков семейства ML.

В 1978 Бэкус определил FP (язык комбинаторов и аппарат, в рамках которого возможно размышлять о программах) в своей лекции по случаю вручения Премии Тьюринга. [10] Эта лекция привлекла внимание к области функционального программирования.

В середине 70-х гг. XX в. исследователи Университета г. Эдинбурга (Гордон, Милнер и др.) работали над системой автоматизированного доказательства теорем под названием LCF. [11] Система состояла из дедуктивного исчисления $PP\lambda$ (полиморфное предикатное λ-исчисление), а также интерактивного языка программирования ML (от meta-language, т.е. *мета-язык*), использовавшегося для описания стратегий поиска доказальств, инспирированный языком ISWIM (близком к λ-исчислению) и обладающий оригинальной системой типов. Вскоре обнаружилось, что этот язык можно применять в качестве языка программирования общего назначения.

1.2 λ-исчисление

Формальная система, ныне называемая λ-исчислением была изобретена логиком Алонзо Чёрчем в 20-х гг. XX в. Его целью являлось разработка более естественного основания для логики, чем теория типов Рассела или теория множеств Цермело. Он решил взять функцию в качестве основы, в примитивы входили *абстракция* и *аппликация*. Бестиповое λ-исчисление, рассматриваемое в качестве логики, оказалось противоречивым.

Работа Чёрча была мотивирована стремлением создать *исчисление* (неформально под этим понмается синтаксис для термов и множество правил переписывания для их преобразования), которое бы отражало интуитивное понимание поведения *функций*. Данный подход в корне отличается от рассмотрения функций как множеств (множеств пар «аргумент, значение»), потому что целью являлось отражение *вычислительного* аспекта функций.

Абстрактный синтаксис *бестипового* λ*-исчисления* (термин, используемый для того, чтобы отличать это исчисление от других версий λ-исчисления) включает в себя *лямбда-выражения*, определяемые следующим образом:

$$e ::= x \in V \mid \lambda x.e \mid e_1 e_2$$

Множество V задает имена переменных (например, x_1, x_2, и т.д.). Выражения вида $\lambda x.e$ называются *абстракциями*, а $(e_1 e_2)$ — *аппликациями*. Первые отражают понятие функции, вторые — применения функции. По соглашению операция аппликации принимается левоассоциативной, поэтому $(e_1 e_2 e_3)$ означает $((e_1 e_2) e_3)$.

Правила переписывания λ-исчисления зависят от понятия *подстановки* выражения e_1 вместо всех свободных вхождений переменной x в выражении e_2, записываемом как $[e_1/x]e_2$. В большинстве систем, использующих подстановку, включая и предикатное, и λ-исчисление, необходимо проявлять внимание к конфликтам имен. Из-за этого строгое формальное определение подстановки является несколько громоздким.

Чтобы понять, как выполняется подстановка, необходимо разобраться с понятием *свободных переменных* в выражении e, которое записывается как $fv(e)$ и определяется с помощью структурной индукции по абстрактному синтаксису следующими правилами:

$$fv(x) = \{x\}$$
$$fv(e_1 e_2) = fv(e_1) \cup fv(e_2)$$
$$fv(\lambda x.e) = fv(e) \setminus \{x\}$$

Переменную x называют *свободной* в выражении e если и только если $x \in fv(e)$. Тогда подстановку $[e_1/x]e_2$ можно определить индуктивно следующим образом:

$$[e/x_i]x_j = \begin{cases} e, & \text{если } i = j \\ x_j, & \text{если } i \neq j \end{cases}$$
$$[e_1/x](e_2 e_3) = ([e_1/x]e_2)([e_1/x]e_3)$$
$$[e_i/x_i](\lambda x_j.e_2) = \begin{cases} \lambda x_j.e_2, & \text{если } i = j \\ \lambda x_j.[e_1/x_i]e_2, & \text{если } i \neq j \text{ и } x_j \notin fv(e_1) \\ \lambda x_k.[e_1/x_i]([x_k/x_j]e_2), & \text{в ином случае, где } k \neq i,\ k \neq j, \\ & \text{а также } x_k \notin fv(e_1) \cup fv(e_2) \end{cases}$$

Последнее правило таково потому, что в рассматриваемом случае может произойти конфликт имен, который разрешается переименованием связанной переменной. Следующий пример показывает все три правила в действии:

$$[y/x]((\lambda y.x)(\lambda x.x)x) \equiv (\lambda z.y)(\lambda x.x)y$$

Определив подстановку, λ-исчисление можно завершить следующими тремя правилами переписывания:

а) α-конверсия (переименование): $\lambda x_i.e \Leftrightarrow \lambda x_j.[x_j/x_i]e$, где $x_j \notin fv(e)$

б) β-конверсия (подстановка): $(\lambda x.e_1)e_2 \Leftrightarrow [e_2/x]e_1$

в) η-конверсия: $\lambda x.(ex) \Leftrightarrow e$, если $x \notin fv(e)$

Эти правила, вместе со стандартными правилами отношения эквивалентности для рефлексивности, симметричности и транзитивности, создают теорию *конвертируемости* для λ-исчисления.

Понятие *редукции* является тем же самым, что и конвертируемость, но ограниченную таким образом, чтобы β-конверсия и η-конверсия применялись «в одну сторону»:

а) β-редукция: $(\lambda x.e_1)e_2 \Rightarrow [e_2/x]e_1$

б) η-редукция: $\lambda x.(ex) \Rightarrow e$, если $x \notin fv(e)$

Пишут $e_1 \overset{*}{\Rightarrow} e_2$, если e_1 можно вывести из e_2 посредством последовательного применения одного или более правила (α-конверсии, β- или η-редукции). Иными словами, $\overset{*}{\Rightarrow}$ является рефлексивным, транзитивным замыканием $\Rightarrow$, включая α-конверсию.

Если лямбда-выражение нельзя редуцировать посредством применения β- или η-редукции, то говорят, что оно *в нормальной форме.*

У некоторых выражений нет нормальной формы, например, результатом единственно возможной редукции

$$(\lambda x.(xx))(\lambda x.(xx))$$

является идентичное выражение, вследствие чего процесс редукции не завершается.

Несмотря на это, нормальная форма является тем, что интуитивно понимают под «значением» выражения. Возникают естественные вопросы. Если у выражения есть нормальная форма, всегда ли можно её найти? Является ли нормальная форма выражения уникальной? Ответы на эти вопросы дают теоремы Чёрча-Россера.

Из экономии места различные теоремы (такие, как упомянутые выше теоремы, теорема о висячей точке, тезис Чёрча) не рассматриваются. За более подробным обсуждением этих и других вопросов по λ-исчислению следует обращаться к [12].

1.3 Естественный вывод

Логика высказываний формализует аргументацию, вовлекающую *связки*, такие, как «и», «или», «подразумевать» и т.д. Используя связки, сложные высказывания конструируют из атомарных высказываний и переменных.

Формально абстрактный синтаксис формул определяется следующим образом:

$$A ::= x \in V \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \implies A_2 \mid A_1 \iff A_2 \mid \bot \mid \neg A$$

Множество V задает имена переменных (например, X_1, X_2, и т.д.). В связки вкладывается следующий смысл:

– $\wedge$ означает *и* (конъюнкция)

- $\vee$ означает *или* (дизъюнкция)
- $\implies$ означает *если* (импликация)
- $\iff$ означает *если и только если* (эквивалентность)
- $\perp$ означает *ложь*
- $\neg$ означает *отрицание*

По соглашению большими латинскими буквами $(A, B, C, ...)$ обозначают произвольные формулы. Приоритет $\wedge$ выше, чем приоритет $\vee$, $\implies$, $\iff$; самый высокий приоритет у $\neg$. Операции конъюнкции и дизъюнкции левоассоциативны, импликация правоассоциативна, эквивалентность неассоциативна.

Общей формой доказательства является вывод *умозаключения* на основе нескольких *предпосылок*:

$$B_1, ..., B_n \vdash A$$

что значит A *истинно, если все* $B_1, ..., B_n$ *истинны.* Список предпосылок $B_1, ..., B_2$ может быть пустым, или содержать одну или более предпосылок.

Греческими буквами Γ и Δ обозначают произвольные списки высказываний, то есть

$$\Gamma ::= \emptyset \mid \Gamma, A \mid \Gamma_1, \Gamma_2$$

Объединение двух списков Γ и Δ записывают Γ, Δ (дубликаты из обоих списков удаляются).

В системе естественного вывода каждая логическая связка характеризуется одним или более правилами *введения*, которые определяют способ вывода того, что конъюнкция, импликация, и т.п. истинна. Правило *устранения* связки указывает, какие истины можно получить на основании истинности конъюнкции, импликации и т.п. Правила введения и устранения иметь некоторые свойства, чтобы гарантировать *обоснованность* системы.

Первое правило

$$\frac{}{A \vdash A}$$

означает простую тавтологию: если A истинно, то оно истинно.

Введение импликации записывают:

$$\frac{\Gamma, B \vdash A}{\Gamma \vdash B \implies A}$$

С интуитивной точки зрения, это правило отражает метод условного доказательства: если из Γ и B можно вывести A, то из Γ можно вывести $B \implies A$.

Правило *modus ponens* записывают

$$\frac{\Gamma \vdash B \implies A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A}$$

Первая предпосылка указывает, что из истинности B следует истинность A если предпосылки Γ истинны. Вторая предпосылка указывает, что B истинно при условии, что предпосылки Δ истинны, а заключение указывает, что A истинно в том случае, когда предпосылки Γ, Δ истинны.

Правила для введения и устранения конъюнкции определяются следующим образом:

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,2$$

Правило $\wedge$-введения утверждает, что если из предпосылок Γ следует A, а из предпосылок Δ следует B, то из сложенных списков Γ, Δ можно вывести $A \wedge B$. Правило $\wedge$-устранения №1 указывает, что если из Γ следует $A \wedge B$, то значит, из Γ можно вывести и A. Правило $\wedge$-устранения №2 аналогично правилу №1.

Остальные связки определяются подобным образом. За более полным изложением следует обращаться к [13].

2 Постановка задачи

3 Практическая часть

Текст 2.

3.1 Просто-типизированное λ-исчисление

Тип является коллекцией вычислительных сущностей, которые имеют некоторые общие свойства. Например, тип int назначается всем выражениям, результатом вычисления которых будет целое число, а тип $int \implies int$ назначается всем функциям от целых к целым.

Типы можно рассматривать как ёмкие, приблизительные описания вычислений: типы являются *статической* аппроксимацией поведения программы при выполнении. Системы типов являются легковесным формальным методом для размышления о поведении программ.

1 Синтаксис

Синтаксис просто-типизированного λ-исчисления подобен синтаксису бестипового λ-исчисления за исключением абстракций. В абстракции $\lambda x : \tau.e$ тип τ является ожидаемым типом аргумента x.

Термы, находящиеся в нормальной форме, выделены в отдельную категорию *значений.*

$$e ::= x \in V \mid \lambda x : \tau.e \mid e_1 e_2 \mid n$$
$$v ::= \lambda x : \tau.e \mid n$$
$$\tau ::= \mathrm{int} \mid \tau_1 \implies \tau_2$$

2 Отношение типизации

Присутствие типов не изменяет способ вычисления лямбда-выражений, отношение редукции определяется по аналогии с бестиповым λ-исчислением.

Типы используются для ограничения множества вычисляемых выражений. Система типов просто-типизированного λ-исчисления позволяет удостовериться, что вычисление любой программы, прошедшей проверку типов, не зайдет в тупик. Вычисление выражения e называют *зашедшим в тупик*, если e не является значением, но его невозможно редуцировать до e_1.

Отношение между выражениями и типами $\Gamma \vdash e : \tau$ читается как «выражение e имеет тип τ в контексте Γ. Типовый контекст является списком переменных и их типов. В умозаключении $\Gamma \vdash e : \tau$ все свободные переменные e связаны с типами в контексте Γ.

Формально, типовый контекст определяется как

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

Пусть даны контекст Γ и выражение e. Если существует некоторый тип τ, такой, что $\Gamma \vdash e : \tau$, говорят, что e *присваивается тип* τ *в контексте* Γ.

Отношение $\Gamma \vdash e : \tau$ определяется индуктивно посредством следующих правил.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,(var) \qquad \frac{}{\Gamma \vdash n : int}\,(int) \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \implies \tau_2}\,(abst)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \implies \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\,(app)$$

Целому числу всегда назначается тип int. Переменная x имеет, с которым она связана в контексте (разумеется, контекст должен содержать x). Абстракция $\lambda x : \tau_1 . e$ имеет тип $\tau_1 \implies \tau_2$ если выражение e имеет тип τ_2 при условии, что x имеет тип τ_1. И наконец, аппликация $e_1 e_2$ имеет тип τ_2, если e_1 имеет тип $\tau_1 \implies \tau_2$, а e_2 имеет тип τ_1.

Проверка типов позволяет удостовериться, что если программе может быть присвоен тип, то она не «застрянет» при выполнении. Это свойство можно описать более формально:

Обоснованность Если $\vdash e : \tau$ и $e \overset{*}{\Rightarrow} e'$, то либо e' является значением, либо существует такое e'', что $e' \overset{*}{\Rightarrow} e''$.

Обычно эту теорему доказывают при помощи двух лемм: о *сохранении* и о *прогрессе*. Лемма о сохранении гласит, что если выражение e назначен тип, и его можно редуцировать или α-конвертировать в e', то e' тоже будет назначен тип. Иными словами, редукция сохраняет типизацию. Лемма о прогрессе гласит, что если выражению e назначен тип, то либо это значение, либо существует такое e', что $e \Rightarrow e'$.

Обе леммы можно записать более формально:

Сохранение Если $\vdash e : \tau$ и $e \Rightarrow e'$, то $\vdash e' : \tau$.

Прогресс Если $\vdash e : \tau$, то либо e является значением, либо $e \Rightarrow e'$

Из соображений экономии места доказательства теорем, а также многие другие вопросы не раскрываются. Более подробное изложение можно найти в [14].

3 Операционная семантика

Операционная семантика позволяет указать смысл программы путем описания того, какие шаги интерпретации необходимо предпринять для вычисления результата. Эти шаги называются *семантикой* программы.

Структурная операционная семантика была предложения Г. Плоткиным в [15]. Основной идеей является определение поведения программы в терминах поведения ее частей, то есть на основе синтаксиса, отсюда название. Спецификация семантики принимает форму правил вывода.

Правила показывают отношение между выражением и результатом его вычисления. «e вычисляется в v» пишут $e \Downarrow v$.

$$\frac{}{i : int \Downarrow i : int}\,(evalint) \qquad \frac{}{\lambda x : \tau.e \Downarrow \lambda x : \tau.e}\,(evallam)$$

$$\frac{e_1 \Downarrow \lambda x : \tau_1.e : \tau_1 \Rightarrow \tau_2 \quad e_2 \Downarrow v : \tau_1}{e_1 e_2 \Downarrow [v/x]e : \tau_2}\,(evalapp)$$

Правила ($evalint$) и ($evallam$) показывают, что результатом вычисления функций и целых чисел являются они сами. Согласно правилу ($evalapp$), вычисление применения функции к аргументу вовлекает в себя вычисление выражения для получения функции, затем вычисление аргумента, и наконец, подстановку.

3.2 Типизированное внутреннее представление

Список литературы

1. *Rémy Didier*. Using, Understanding, and Unraveling the OCaml Language // Applied Semantics. Advanced Lectures. LNCS 2395. / Ed. by Gilles Barthe. — Springer Verlag, 2002. — Pp. 413–537. http://gallium.inria.fr/ remy/cours/appsem/.
2. A history of Haskell: being lazy with class / P. Hudak, J. Hughes, S.P. Jones, P. Wadler // Proceedings of the third ACM SIGPLAN conference on History of programming languages / ACM. — 2007. — Pp. 12–55.
3. *De Bruijn NG*. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem // Indagationes Mathematicae (Proceedings) / Elsevier. — Vol. 75. — 1972. — Pp. 381–392.
4. *Chen C., Shi R., Xi H.* Implementing typeful program transformations // *Fundamenta informaticae*. — 2006. — Vol. 69, no. 1. — Pp. 103–121.
5. *Church A.* A formulation of the simple theory of types // *Journal of symbolic logic*. — 1940. — Pp. 56–68.
6. *Xi Hongwei*. Applied Type System (extended abstract) // Post-workshop Proceedings of TYPES 2003. — 2004. — Pp. 394–408.
7. *McCarthy J., Levin M.I.* LISP 1.5 programmer's manual. — The MIT Press, 1965.
8. *Landin PJ*. Correspondence between ALGOL 60 and Church's Lambda-notation: part I // *Communications of the ACM*. — 1965. — Vol. 8, no. 2. — Pp. 89–101.
9. *Landin PJ*. The next 700 programming languages // *Communications of the ACM*. — 1966. — Vol. 9, no. 3. — Pp. 157–166.
10. *Backus J.* Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs // ACM Turing award lectures / ACM. — 2007. — P. 1977.
11. *Gordon M., Milner R., Wadsworth C.P.* Edinburgh LCF: a mechanized logic of computation, volume 78 of Lecture Notes in Computer Science. — 1979.
12. *Barendregt H.P.* The lambda calculus: its syntax and semantics. — North Holland, 1984.
13. *Mendelson E.* Introduction to mathematical logic. — Chapman & Hall/CRC, 1997.

14. *Barendregt H.P.* Lambda calculi with types // *Handbook of logic in computer science.* — 1992. — Vol. 2. — Pp. 117–309.

15. *Plotkin G.D.* A structural approach to operational semantics. — 1981.

Приложение А

(обязательное)

Исходный код

```
// compile with:
// atscc -D_GC_GCATS interp.dats
staload _ = "prelude/DATS/option.dats"

datasort tp = tpbool | tpnat // base types
  | tpfun of (tp, tp)
  | tpcon of (tp, tp) // conjunction
  | tpdis of (tp, tp) // disjunction
// | forall of (tp -> tp)
datasort tps = tpsnil | tpsmore of (tps, tp)

dataprop TPI (int, tp, tps) =
  | {T:tp} {G:tps} TPIone (0, T, tpsmore (G, T))
  | {T1,T2:tp} {G:tps} {n:nat}
    TPIshi (n+1, T1, tpsmore (G, T2)) of TPI (n, T1, G)

(* ****** ****** *)
// internal representation

datatype EXP (G:tps, tp, int) =
  // core
  | {i:nat} {T:tp} EXPvar (G, T, 0) of (TPI (i, T, G) | int i)
  | {T1,T2:tp} {n1,n2:nat}
      EXPapp (G, T2, n1+n2+1) of
        (EXP (G, tpfun (T1, T2), n1), EXP (G, T1, n2))
  | {T1,T2:tp} {n:nat}
      EXPlam (G, tpfun (T1, T2), n+1) of
        EXP (tpsmore (G, T1), T2, n)
  // extensions
  // general recursion
  | {T:tp} {n:nat}
      EXPfix (G, T, n+1) of EXP (tpsmore (G, T), T, n)
  // booleans
  | EXPtrue (G, tpbool, 0)
  | EXPfalse (G, tpbool, 0)
```

```
  | {n1,n2,n3:nat} {T:tp}
      EXPif (G, T, n1+n2+n3+1) of
        (EXP (G, tpbool, n1), EXP (G, T, n2), EXP (G, T, n3))
  // naturals
  | EXPzero (G, tpnat, 0)
  | {n:nat} EXPsucc (G, tpnat, n+1) of EXP (G, tpnat, n)
  | {T:tp} {n1,n2,n3:nat}
      EXPcase (G, T, n1+n2+n3+1) of
        (EXP (G, tpnat, n1), EXP (G, T, n2),
         EXP (tpsmore (G, tpnat), T, n3))
  // conjunction
  | {T1,T2:tp} {n1,n2:nat}
      EXPcon (G, tpcon (T1, T2), n1+n2+1) of
        (EXP (G, T1, n1), EXP (G, T2, n2))
  | {T1,T2:tp} {n:nat}
      EXPfst (G, T1, n+1) of EXP (G, tpcon (T1, T2), n)
  | {T1,T2:tp} {n:nat}
      EXPsnd (G, T2, n+1) of EXP (G, tpcon (T1, T2), n)
  // disjunction
  | {T1,T2:tp} {n:nat}
      EXPinl (G, tpdis (T1, T2), n+1) of EXP (G, T1, n)
  | {T1,T2:tp} {n:nat}
      EXPinr (G, tpdis (T1, T2), n+1) of EXP (G, T2, n)
  | {T1,T2,T3:tp} {n1,n2,n3:nat}
      EXPdis (G, T3, n1+n2+n3+1) of
        (EXP (G, tpdis (T1, T2), n1), EXP (tpsmore (G, T1), T3, n2),
         EXP (tpsmore (G, T2), T3, n3))
  // quantification
(*
  | {G:tps} {f:tp->tp} {n:nat} EXPtlam (G, forall f, n+1) of
      ({t:tp} EXP (G, f t, n))
  | {G:tps} {f:tp->tp} {t:tp} {n:nat} EXPtapp (G, f t, n+1) of
      EXP (G, forall f, n)
*)

typedef EXP0 (G:tps, T:tp) = [n:nat] EXP (G, T, n)

(* ****** ****** *)
// type checking

// singleton type for tp
```

Изм.	Лист	№ докум.	Подп.	Дата	АБВГД XXXXXXXX	Лист 19

```
datatype TP (tp) =
  | TPbool (tpbool)
  | TPnat (tpnat)
  | {T1,T2:tp} TPcon (tpcon (T1, T2)) of (TP T1, TP T2)
  | {T1,T2:tp} TPdis (tpdis (T1, T2)) of (TP T1, TP T2)
  | {T1,T2:tp} TPfun (tpfun (T1, T2)) of (TP T1, TP T2)

// equality on types
dataprop TPEQ (tp, tp, bool) =
  | {T1,T2:tp} TPEQnone (T1, T2, false)
  | {T:tp} TPEQsome (T, T, true)

fun eq_tp_tp {T1,T2:tp} (a: TP T1, b: TP T2)
  : [b:bool] (TPEQ (T1, T2, b) | bool b) = case+ (a, b) of
  | (TPbool (), TPbool ()) => (TPEQsome () | true)
  | (TPnat (), TPnat ()) => (TPEQsome () | true)
  | (TPfun (a1, a2), TPfun (b1, b2)) => begin
      case+ (eq_tp_tp (a1, b1), eq_tp_tp (a2, b2)) of
      | ((TPEQsome () | true), (TPEQsome () | true)) => (TPEQsome () | true)
      | (_, _) => (TPEQnone () | false)
      end
  | (TPcon (a1, a2), TPcon (b1, b2)) => begin
      case+ (eq_tp_tp (a1, b1), eq_tp_tp (a2, b2)) of
      | ((TPEQsome () | true), (TPEQsome () | true)) => (TPEQsome () | true)
      | (_, _) => (TPEQnone () | false)
      end
  | (TPdis (a1, a2), TPdis (b1, b2)) => begin
      case+ (eq_tp_tp (a1, b1), eq_tp_tp (a2, b2)) of
      | ((TPEQsome () | true), (TPEQsome () | true)) => (TPEQsome () | true)
      | (_, _) => (TPEQnone () | false)
      end
  | (_, _) => (TPEQnone () | false)

datatype CTX (tps) =
  | CTXnil (tpsnil)
  | {T:tp} {G:tps} CTXcons (tpsmore (G, T)) of (string, TP T, CTX G)

fun ctx_lookup {G:tps} (id: string, c: CTX G)
  : Option ([T:tp] [i:nat] @(TPI (i, T, G) | int i, TP T))
  = case+ c of
  | CTXnil () => None ()
```

```
  | CTXcons (x, t, c) => if id = x then Some @(TPIone () | 0, t)
    else case+ ctx_lookup (id, c) of
      | Some @(pf | v, t) => Some @(TPIshi pf | v+1, t)
      | None () => None ()

// this is what we get from our parser
datatype EXP0 =
  | EXP0var of string
  | {T:tp} EXP0lam of (string, TP T, EXP0)
  | EXP0app of (EXP0, EXP0)
  | {T:tp} EXP0fix of (string, TP T, EXP0)
  | EXP0false | EXP0true
  | EXP0if of (EXP0, EXP0, EXP0)
  | EXP0zero | EXP0succ of EXP0
  | EXP0case of (EXP0, EXP0, string, EXP0)
  | EXP0con of (EXP0, EXP0)
  | EXP0fst of EXP0 | EXP0snd of EXP0
  | EXP0inl of EXP0 | EXP0inr of EXP0
  | {T:tp} EXP0inl of (TP T, EXP0)
  | {T:tp} EXP0inr of (TP T, EXP0)
  | EXP0dis of (EXP0, string, EXP0, string, EXP0)

fun typecheck {G:tps} (e: EXP0, c: CTX G)
  : Option ([T:tp] @(TP T, EXP0 (G, T))) = case+ e of
  | EXP0var x => (case+ ctx_lookup (x, c) of
    | Some @(pf | v, t) => Some @(t, EXPvar (pf | v))
    | None () => None ())
  | EXP0lam (x, t, b) => (case+ typecheck (b, CTXcons (x, t, c)) of
    | Some @(t', b) => Some @(TPfun (t, t'), EXPlam b)
    | None () => None ())
  | EXP0app (e1, e2) =>
    begin case+ typecheck (e1, c) of
    | Some @(TPfun (t1, t2), e1) =>
      begin case+ typecheck (e2, c) of
        | Some @(t1', e2) => let
            val (pf | r) = eq_tp_tp (t1, t1')
          in
            if r then let
              prval TPEQsome () = pf
            in
              Some @(t2, EXPapp (e1, e2))
```

```
          end else let
            prval TPEQnone () = pf
          in None () end
        end
      | None () => None ()
    end
  | _ => None ()
  end
| EXP0fix (x, t, b) => (case+ typecheck (b, CTXcons (x, t, c)) of
  | Some @(t', b) => let
      val (pf | r) = eq_tp_tp (t, t')
    in
      if r then let
        prval TPEQsome () = pf
      in Some @(t, EXPfix b) end else let
        prval TPEQnone () = pf
      in None () end
    end
  | _ => None ())
| EXP0false () => Some @(TPbool (), EXPfalse ())
| EXP0true () => Some @(TPbool (), EXPtrue ())
| EXP0if (a,e1,e2) => (case+ typecheck (a, c) of
  | Some @(t, a) => let
      val (pf | r) = eq_tp_tp (t, TPbool ())
    in
      if r then let
        prval TPEQsome () = pf
      in
        case+ (typecheck (e1, c), typecheck (e2, c)) of
        | (Some @(t1, e1), Some @(t2, e2)) => let
            val (pf | r) = eq_tp_tp (t1, t2)
          in
            if r then let
              prval TPEQsome () = pf
            in
              Some @(t1, EXPif (a, e1, e2))
            end else let
              prval TPEQnone () = pf
            in None () end
          end
        | (_, _) => None ()
```

```
      end else let
        prval TPEQnone () = pf
      in None () end
    end
  | None () => None ())
| EXP0zero () => Some @(TPnat (), EXPzero ())
| EXP0succ e => (case+ typecheck (e, c) of
  | Some @(t, e) => let
      val (pf | r) = eq_tp_tp (t, TPnat ())
    in
      if r then let
        prval TPEQsome () = pf
      in Some @(t, EXPsucc e) end else let
        prval TPEQnone () = pf
      in None () end
    end
  | None () => None ())
// case x of z => foo | s(x) => baz
| EXP0case (a, e1, x2, e2) => (case+ typecheck (a, c) of
  | Some @(t, a) => let
      val (pf | r) = eq_tp_tp (t, TPnat ())
    in
      if r then let
        prval TPEQsome () = pf
      in
        case+ typecheck (e1, c) of
        | Some @(t1, e1) => begin
          case+ typecheck (e2, CTXcons (x2, TPnat (), c)) of
          | Some @(t2, e2) => let
              val (pf | r) = eq_tp_tp (t1, t2)
            in
              if r then let
                prval TPEQsome () = pf
              in
                Some @(t1, EXPcase (a, e1, e2))
              end else let
                prval TPEQnone () = pf
              in
                None ()
              end
            end
```

```
          | None () => None ()
          end
        | None () => None ()
      end else let
        prval TPEQnone () = pf
      in None () end
    end
  | None () => None ())
| EXP0con (e1, e2) => (case+ (typecheck (e1, c), typecheck (e2, c)) of
  | (Some @(t1, e1), Some @(t2, e2)) =>
      Some @(TPcon (t1, t2), EXPcon (e1, e2))
  | (_, _) => None ())
| EXP0fst e => (case+ typecheck (e, c) of
  | Some @(TPcon (t1, t2), e) => Some @(t1, EXPfst e)
  | _ => None ())
| EXP0snd e => (case+ typecheck (e, c) of
  | Some @(TPcon (t1, t2), e) => Some @(t2, EXPsnd e)
  | _ => None ())
| EXP0inl (TPdis (t1, t2), e) => (case+ typecheck (e, c) of
  | Some @(t1', e) => let
      val (pf | r) = eq_tp_tp (t1, t1')
    in
      if r then let
        prval TPEQsome () = pf
      in Some @(TPdis (t1, t2), EXPinl e) end else let
        prval TPEQnone () = pf
      in None () end
    end
  | None () => None ())
| EXP0inr (TPdis (t1, t2), e) => (case+ typecheck (e, c) of
  | Some @(t2', e) => let
      val (pf | r) = eq_tp_tp (t2, t2')
    in
      if r then let
        prval TPEQsome () = pf
      in Some @(TPdis (t1, t2), EXPinr e) end else let
        prval TPEQnone () = pf
      in None () end
    end
  | None () => None ())
| EXP0dis (e, x1, f1, x2, f2) => (case+ typecheck (e, c) of
```

```
    | Some @(TPdis (t1, t2), e) => begin
        case+ (typecheck (f1, CTXcons (x1, t1, c)), typecheck (f2, CTXcons
          (x2, t2, c))) of
        | (Some @(t1', f1), Some @(t2', f2)) => let
            val (pf | r) = eq_tp_tp (t1', t2')
          in
            if r then let
              prval TPEQsome () = pf
            in Some @(t1', EXPdis (e, f1, f2)) end else let
              prval TPEQnone () = pf
            in None () end
          end
        | (_, _) => None ()
      end
    | _ => None ())
  | _ => None ()
// | EXP0dis (e, t, f1, f2) => (case+ typecheck (e, c) of
//    | Some @(TPdis (t1, t2), e) => (case+ (typecheck (f1, c)

(* ****** ****** *)
// big-step interpreter

datatype either (a:t@ype, b:t@ype) =
  | inleft (a, b) of a
  | inright (a, b) of b

datatype VAL (tp) =
  | VALtrue (tpbool)
  | VALfalse (tpbool)
  | VALzero (tpnat)
  | VALsucc (tpnat) of VAL (tpnat)
  | {t1,t2:tp} VALcon (tpcon (t1, t2)) of (VAL t1, VAL t2)
  | {t1,t2:tp} VALdis (tpdis (t1, t2)) of either (VAL t1, VAL t2)
  | {D:tps} {T1,T2:tp} {m:nat} VALclo (tpfun (T1, T2)) of
      (EXP (tpsmore (D, T1), T2, m), ENV D)
// | {D:tps} {f:tp->tp} {m:nat} VALtclo (forall f) of
//      ({t:tp} EXP (D, f t, m), ENV D)

and ENV (tps) =
  | ENVnil (tpsnil)
  | {G:tps} {T:tp}
```

```
      ENVcons (tpsmore (G, T)) of (VAL T, ENV G)
  | {G:tps} {T:tp}
      ENVfix (tpsmore (G, T)) of (EXP0 (tpsmore (G, T), T), ENV G)

// typedef VAL0 (T:tp) = [n:nat] VAL (T, n)
// typedef ENV0 (G:tps) = [n:nat] ENV (G, n)

fun lookup {G:tps} {T:tp} {n:nat}
  (pf: TPI (n, T, G) | e: ENV G, n: int n)
  : VAL T =
  if n = 0 then let
    prval TPIone () = pf
  in
    case+ e of
    | ENVcons (x, _) => x
    | ENVfix (x, e) => eval (ENVfix (x, e), x)
  end else let
    prval TPIshi pf = pf
    val e = case+ e of
      | ENVcons (_, e) => e
      | ENVfix (_, e) => e
  in
    lookup (pf | e, n-1)
  end

and eval {G:tps} {T:tp}
  (e: ENV G, a: EXP0 (G, T))
  : VAL T = case+ a of
  // core
  | EXPvar (pf | n) => lookup (pf | e, n)
  | EXPlam b => VALclo (b, e)
  | EXPapp (a1, a2) => apply (e, a1, a2)
  // extensions
  | EXPfix b => eval (ENVfix (b, e), b)
  (* booleans *)
  | EXPtrue () => VALtrue ()
  | EXPfalse () => VALfalse ()
  | EXPif (a, b, c) => (case+ eval (e, a) of
      | VALtrue () => eval (e, b)
      | VALfalse () => eval (e, c))
  (* naturals *)
```

```
  | EXPzero () => VALzero ()
  | EXPsucc a => VALsucc (eval (e, a))
  | EXPcase (a, b, c) => (case+ eval (e, a) of
      | VALzero () => eval (e, b)
      | VALsucc v => eval (ENVcons (v, e), c))
  (* conjunction *)
  | EXPcon (a1, a2) => VALcon (eval (e, a1), eval (e, a2))
  | EXPfst a => let
      val VALcon (r, _) = eval (e, a)
    in r end
  | EXPsnd a => let
      val VALcon (_, r) = eval (e, a)
    in r end
  (* disjunction *)
  | EXPinl a => VALdis (inleft (eval (e, a)))
  | EXPinr a => VALdis (inright (eval (e, a)))
  | EXPdis (a, b, c) => let
      val VALdis r = eval (e, a)
    in case+ r of
      | inleft r => eval (ENVcons (r, e), b)
      | inright r => eval (ENVcons (r, e), c)
    end
(*
  | EXPtlam b => VALtclo (b, e)
  | EXPtapp a => let
      val VALtclo (a', e') = eval (e, a)
    in
      eval (e', a' {..})
    end
*)

and apply {G:tps} {T1,T2:tp}
  (e: ENV G, a1: EXP0 (G, tpfun (T1, T2)), a2: EXP0 (G, T1))
  : VAL T2 = let
  val VALclo (a1, e') = eval (e, a1)
  val a2 = eval (e, a2)
in
  eval (ENVcons (a2, e'), a1)
end

(* ****** ****** *)
```

```
// some simple programs

fun print_val {T:tp} (v: VAL T): void = case+ v of
  | VALtrue () => print "T"
  | VALfalse () => print "F"
  | VALzero () => print "Z"
  | VALsucc v => (print "S("; print_val v; print ")")
  | VALcon (a, b) =>
      (print "and("; print_val a; print ",␣"; print_val b; print ")")
  | VALdis (inleft x) => (print "inl("; print_val x; print ")")
  | VALdis (inright x) => (print "inr("; print_val x; print ")")
  | VALclo (a, e) => print "<closure>"

fun print_exp {G:tps} {T:tp} (e: EXP0 (G, T)): void =
  case+ e of
  | EXPvar (pf | i) => (print "var("; print i; print ")")
  | EXPlam e => (print "lam("; print_exp e; print ")")
  | EXPapp (e1, e2) =>
    (print "app("; print_exp e1; print ",␣"; print_exp e2; print ")")
  | EXPfix e => (print "fix("; print_exp e; print ")")
  | EXPfalse () => print "F"
  | EXPtrue () => print "T"
  | EXPif (a, b, c) =>
    (print "if("; print_exp a; print ","; print_exp b; print ",";
     print_exp c; print ")")
  | EXPzero () => print "Z"
  | EXPsucc e => (print "S("; print_exp e; print ")")
  | EXPcase (a, b, c) =>
    (print "case("; print_exp a; print ",␣"; print_exp b; print ",␣";
     print_exp c; print ")")
  | EXPcon (a, b) =>
    (print "and("; print_exp a; print ",␣"; print_exp b; print ")")
  | EXPfst x => (print "fst("; print_exp x; print ")")
  | EXPsnd x => (print "snd("; print_exp x; print ")")
  | EXPinl x => (print "inl("; print_exp x; print ")")
  | EXPinr x => (print "inr("; print_exp x; print ")")
  | EXPdis (a, b, c) =>
    (print "or("; print_exp a; print ","; print_exp b; print ","; print_exp c;
     print ")")

// an abbreviation
```

```
fun exp2app (e: EXP0, x: EXP0, y: EXP0): EXP0 = EXP0app (EXP0app (e, x), y)

// truth-logical functions

// bool_and(x,y) = if x then false else y
val bool_and = EXP0lam ("x", TPbool (), EXP0lam ("y", TPbool (),
  EXP0if (EXP0var "x", EXP0false (), EXP0var "y")))
// bool_or(x,y) = if x then y else false
val bool_or = EXP0lam ("x", TPbool (), EXP0lam ("y", TPbool (),
  EXP0if (EXP0var "x", EXP0var "y", EXP0false ())))
// bool_not(x) = if x then false else true
val bool_not = EXP0lam ("x", TPbool (),
  EXP0if (EXP0var "x", EXP0false (), EXP0true ()))
// bool_imp(x,y) = or(not(x), y)
val bool_imp = EXP0lam ("x", TPbool (), EXP0lam ("y", TPbool (),
  exp2app (bool_or, EXP0app (bool_not, EXP0var "x"), EXP0var "y")))

// arithmetic
val fun_inc = EXP0lam ("x", TPnat (), EXP0succ (EXP0var "x"))
val inc = EXP0app (fun_inc, EXP0succ (EXP0succ (EXP0zero ())))
// pred(x) = case x of z => z | s(x) => x
val fun_pred = EXP0lam ("x", TPnat (),
  EXP0case (EXP0var "x", EXP0zero (), "x", EXP0var "x"))
// iszero(x) = case x of z => true | s(x) => false
val fun_iszero = EXP0lam ("x", TPnat (),
  EXP0case (EXP0var "x", EXP0true (), "x", EXP0false ()))

// 0+y=y
// x+y=((x-1)+y)+1
// add = \y:nat. fix(f:nat->nat. \x:nat. if(iszero(x),y,succ(f(pred(x)))))
val add = EXP0lam ("y", TPnat (),
  EXP0fix ("f", TPfun (TPnat (), TPnat ()), EXP0lam ("x", TPnat (),
    EXP0if (EXP0app (fun_iszero, EXP0var "x"), EXP0var "y",
      EXP0succ (EXP0app (EXP0var "f", EXP0app (fun_pred, EXP0var "x")))))))

// 0*y=0
// x*y=y+(x-1)*y
// mul = \y:nat. fix(f:nat->nat, \x:nat.
//          case x of Z => Z | S(x) => add(y,f(x,y)))
val mul = EXP0lam ("y", TPnat (),
  EXP0fix ("f", TPfun (TPnat (), TPnat ()), EXP0lam ("x", TPnat (),
```

```
    EXP0case (EXP0var "x", EXP0zero (),
      "x'", exp2app (add, EXP0var "y", EXP0app (EXP0var "f", EXP0var
        "x'"))))))

implement main (argc, argv) = let
  val one = EXP0succ (EXP0zero ())
  val two = EXP0succ one
  fun test (x: EXP0): void = case+ typecheck (x, CTXnil) of
    | Some @(t, x) => begin
        print "input term: "; print_exp x; print_newline ();
        print "evaluated term: "; print_val (eval (ENVnil, x));
        print_newline ()
      end
    | None () => print "type checking failed\n"
in
  test inc;
  test (EXP0app (bool_not, EXP0false ()));
  test (EXP0app (EXP0app (bool_imp, EXP0true ()), EXP0false ()));
  test (EXP0app (fun_iszero, EXP0zero ()));
  test (EXP0app (fun_iszero, EXP0succ (EXP0zero ())));
  test (EXP0app (fun_pred, EXP0zero ()));
  test (EXP0app (fun_pred, EXP0succ (EXP0succ (EXP0zero ()))));
  test (exp2app (add, EXP0zero (), EXP0zero ()));
  test (exp2app (add, one, two));
  test (exp2app (mul, EXP0zero (), EXP0zero ()));
  test (exp2app (mul, two, two))
end
```

Приложение Б

(обязательное)

Вывод программы

```
input term: app(lam(S(var(0))), S(S(Z)))
evaluated term: S(S(S(Z)))
input term: app(lam(if(var(0),F,T)), F)
evaluated term: T
input term:
  app(app(lam(lam(app(app(lam(lam(if(var(1),var(0),F))),
  app(lam(if(var(0),F,T)), var(1))), var(0)))), T), F)
evaluated term: F
input term: app(lam(case(var(0), T, F)), Z)
evaluated term: T
input term: app(lam(case(var(0), T, F)), S(Z))
evaluated term: F
input term: app(lam(case(var(0), Z, var(0))), Z)
evaluated term: Z
input term: app(lam(case(var(0), Z, var(0))), S(S(Z)))
evaluated term: S(Z)
input term: app(app(lam(fix(lam(if(app(lam(case(var(0), T,
  F)), var(0)),var(2),S(app(var(1), app(lam(case(var(0), Z,
  var(0))), var(0)))))))), Z), Z)
evaluated term: Z
input term: app(app(lam(fix(lam(if(app(lam(case(var(0), T,
  F)), var(0)),var(2),S(app(var(1), app(lam(case(var(0), Z,
  var(0))), var(0)))))))), S(Z)), S(S(Z)))
evaluated term: S(S(S(Z)))
input term: app(app(lam(fix(lam(case(var(0), Z,
  app(app(lam(fix(lam(if(app(lam(case(var(0), T, F)),
  var(0)),var(2),S(app(var(1), app(lam(case(var(0), Z,
  var(0))), var(0)))))))), var(3)), app(var(2), var(0))))))),
  Z), Z)
```

```
evaluated term: Z
input term: app(app(lam(fix(lam(case(var(0), Z,
  app(app(lam(fix(lam(if(app(lam(case(var(0), T, F)),
  var(0)),var(2),S(app(var(1), app(lam(case(var(0), Z,
  var(0))), var(0)))))))), var(3)), app(var(2), var(0))))))),
  S(S(Z))), S(S(Z)))
evaluated term: S(S(S(S(Z))))
```