Types and Programming Languages[1]

[1]This material is adapted with permission from [1, 4].

Contents

Chapter 1

Introduction

1.1 Semantic styles

- Formal mehods used in Software Engineering and Programming Languages design employ various mathematical structures.
  - They are essential in the basic engineering activities (modeling, specification, design and verification) esspecially when the focus is on quality attributes such as reliability or performance.
- *Semantics* is a core concept in Formal Methods.
- Traditionally, the research focus was on the *dynamic semantics* of languages and systems.
- Three semantic styles are consecrated:
  - Denotational semantics (relying on compositional definitions and a mathematical theory of domains of meanings, often called *denotations*)
  - Axiomatic semantics (laws or axioms are used to define meanings)
  - Operational semantics
- In this curse the method of *operational semantics* is used to describe the dynamic semantics of languages and systems.
  - In operational semantics the behavior is described by means of *transitions* between configurations of systems.
  - Proofs are constructed from representations of (program or system) executions, rather than by reasoning about compositional (denotational) meanings of language phrases (as in the denotational approach).
    - In general, an operational semantics is *not* a compositional description of a language or system behavior.
  - However, it seems that at present most researchers prefer the method of operational semantics.
- For the static aspects of the semantics we use the concept of a *type system*; the presentation of this concept is based on the monograph [1].

1.2 Types and programing languages

- According to [1] (B. Pierce, *Types and Programming Languages,* MIT Press, 2002):

  > *A type system is a tractable syntactic method for proving the absence of certain program beaviors by classifying phrases according to the kind of values they compute.*

- In the sequel we investigate "statically typed languages".
  - The static aspects of the semantics rely on compile-time analysis of programs.
- [1] provides a comprehensive introduction to the domain, with numerous examples, (solved) exercises and case studies. This course is an adaptation of [1, 4].
  - The imlementations presented in [1] are written in OCaml (available from `http://www.cis.upenn.edu/~bcpierce/tapl/`)
  - [4] is based on [1], but with examples in Scala.
  - In this course, some concepts, interpreters and type checkers from [1] will be explained based on prototype implementations written in Haskell [10, 41].
- We will study formal models for programming languages and type checkers
  - Type checking is one of the most successful applications of formal methods in computer science
    - Detect errors early
    - Enforce abstractions
    - Improve code readability
    - Guarantee safety
    - Improve efficiency

1.3 Administrative stuff

- Currently, the slides are used for a Master course comprising 14 lectures (28 hours) and 14 attached seminars (14 hours).
- Each student has to write an essay or a research paper.
- The final mark is computed based on the following components:
  - written examination: 75% (based on [1] and the slides)
  - paper: 25%
- The paper could be
  - A research work
  - An essay or a technical report (5-10 pages) based on individual study and experiments
    - For experiments it is recommended that students use Scala [8, 42] or Haskell [10, 41].

- The bibliography for the paper includes books and articles on the following topics:
  - Advanced topics in types and programming languages, e.g., from [1] (parts III-VI) or [2]
  - Dependent types [7]
  - Behavioral types, in the sense promoted by project BETTY [43]
    - Foundations of session types [28, 16, 32, 40, 19, 17, 27]
    - Experiments or small projects elaborated by using Session Java [29, 30, 15, 31, 46], or related tools [47]
  - Stochastic process algebras (and model checking): PRISM [11, 33, 26, 35, 34], PEPA [5, 44], Bio-PEPA [23, 45], Stochastic Pi [37, 18].
  - K framework [38] and runtime verification (RV) [12, 20, 48] (including RV tools such as: AspectJ, Java MOP, TraceMatches, Ruler-lite, TraceContract, RV-Match, RV-Predict, RV-Monitor)
- The above references are all available either from the library of the department (room M04, Baritiu Street, 28) or available from the Internet.

1.4 Acknowledements

We are very gratefull to professor Benjamin Pierce (from University of Pennsylvania, author of the main references of the course [1, 3]) for the permission to use [1, 3] and to adapt the slides available from `http://www.cis.upenn.edu/~bcpierce/tapl/index.html`. We are also very gratefull to professor Frank Piessens (from Catholic University of Leuven) for the permission to use and adapt the slides that he created [4]. Our course notes are an adaptation of materials from [1, 3, 4] with examples in Haskell.

Chapter 2

Untyped Arithmetic Expressions

We consider the toy language **NB** of numeric and boolean expressions introduced in section 3 of [1]. Note the use of meta-variables (t) in the definition given below.

Definition 2.1 *[Syntax of terms for* **NB***]*

$$t ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid 0 \mid \mathsf{succ}\ t \mid \mathsf{pred}\ t \mid \mathsf{iszero}\ t$$

Remark 2.2 *In this language:*

- 1 *is written as* $\mathsf{succ}\ 0$
- 2 *is written as* $\mathsf{succ}\ (\mathsf{succ}\ 0)$*, etc.*

Notation 2.3 *We use the symbol* $\blacktriangleright$ *to display the result of evaluating examples. For example:* $\mathsf{if\ false\ then}\ 0\ \mathsf{else}\ (\mathsf{succ}\ 0) \blacktriangleright \mathsf{succ}\ 0$*. (If, for brevity, we write* $(\mathsf{succ}\ 0)$ *as* 1*, the evaluation can be written as follows:* $\mathsf{if\ false\ then}\ 0\ \mathsf{else}\ 1 \blacktriangleright 1$*).*

2.1 Syntax

- What does the definition (of **NB** terms) given above mean exactly?
- The BNF notation is considered a shorthand for the following:

Definition 2.4 *[Terms, inductively] The set of* terms *is the smallest set* $\mathcal{T}$ *such that:*

1. $\{\mathsf{true}, \mathsf{false}, 0\} \subseteq \mathcal{T}$*;*
2. *if* $t_1 \in \mathcal{T}$ *then* $\{\mathsf{succ}\ t_1, \mathsf{pred}\ t_1, \mathsf{iszero}\ t_1\} \subseteq \mathcal{T}$*;*
3. *if* $t_1 \in \mathcal{T}$*,* $t_2 \in \mathcal{T}$ *and* $t_3 \in \mathcal{T}$ *then* $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \in \mathcal{T}$*.*

Definition 2.5 *[Terms, by inference rules] The set of terms is defined by the following rules:*

$$\mathsf{true} \in \mathcal{T} \qquad \mathsf{false} \in \mathcal{T} \qquad 0 \in \mathcal{T}$$

$$\frac{t_1 \in \mathcal{T}}{\mathsf{succ}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\mathsf{pred}\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\mathsf{iszero}\ t_1 \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \in \mathcal{T}}$$

- Note:
  - Strings versus *Abstract Syntax Trees* (AST's) ? Formally, we work with AST's
  - Terminology: *axiom*, *inference rule* (or *rule schema*, because it uses metavariables)
- A more concrete characterization of the syntax of **NB** is given in the following:

Definition 2.6 *[Terms, concretely] Define an infinite sequence of sets, $S_0, S_1, S_2, \ldots$, as follows:*

$$S_0 = \emptyset$$

$$S_{i+1} = \{\mathsf{true}, \mathsf{false}, 0\}$$
$$\cup\{\mathsf{succ}\ t_1,\ \mathsf{pred}\ t_1,\ \mathsf{iszero}\ t_1 \mid t_1 \in S_i\}$$
$$\cup\{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \mid t_1, t_2, t_3 \in S_i\}$$

Now let

$$S = \bigcup_{i \in \mathbb{N}} S_i.$$

Proposition 2.7 $\mathcal{T} = S$.

Remark 2.8 *In syntactic and semantic specifications we use inference* rules *of the form:*

$$\frac{premise_1 \quad \cdots \quad premise_n}{conclusion}$$

Some rules have no premises. An example is rule $\mathsf{true} \in \mathcal{T}$ *given in definition 2.5. Such rules (i.e., rules without premises) are named* axioms.

2.2 Induction on terms

- Inductive definitions on terms are pervasive in computer science. Two examples:
  - The *size* of a term:
    - $size(\mathsf{true}) = size(\mathsf{false}) = size(0) = 1$
    - $size(\mathsf{succ}\ t_1) = size(\mathsf{pred}\ t_1) = size(\mathsf{iszero}\ t_1) = 1 + size(t_1)$
    - $size(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = 1 + size(t_1) + size(t_2) + size(t_3)$
  - The *depth* of a term:
    - $depth(\mathsf{true}) = depth(\mathsf{false}) = depth(0) = 1$
    - $depth(\mathsf{succ}\ t_1) = depth(\mathsf{pred}\ t_1) = depth(\mathsf{iszero}\ t_1) = 1 + depth(t_1)$
    - $depth(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = 1 + max\{depth(t_1), depth(t_2), depth(t_3)\}$
- The constructive characterization of terms gives us an important tool for proving things about terms, the *principle of induction on terms*, or *principle of structural induction.*

Theorem 2.9 *[Principle of induction on terms] If, for each term s, given $P(r)$ for all immediate subterms r of s we can show $P(s)$, then $P(s)$ holds for all s.*

- Variants include: induction on depth and size.

2.2.1 Well-founded induction

- Mathematical induction is a convenient tool for recursive functions design (for functions defined on finite structures).
- The most common forms of induction are
  - The well-known principle (or axiom) of induction on natural numbers:
    - Suppose that P is a predicate on the set of natural numbers $\mathbb{N}$. Then, if $P(0)$ and, for all $i \in \mathbb{N}$, $P(i)$ implies $P(i+1)$, then $P(n)$ holds for all $n \in \mathbb{N}$
  - The structural induction principle (which can be proved by using the principle of induction on natural numbers)
- They can all be treated as instances of a general form of induction, called *well-founded induction.*

Definition 2.10 *Suppose we have a set $(a, b \in)A$ with a preorder $\leq$ (a preorder on a set A is a binary relation on A which is reflexive and transitive). We write $b < a$ when $b \leq a$ and $a \neq b$; in this case we say that a is strictly greater than b (or, equivalently, b is strictly lesser than a). We say that $\leq$ is* well-founded *if it contains no infinite (strictly) decreasing chains. For example, the usual order on the set of natural numbers $\mathbb{N}$, with $0 < 1 < 2 < \ldots$ is well-ordered, but the same order on the set $\mathbb{Z}$ of integers, $\ldots < -2 < -1 < 0 < 1 < 2 \ldots$, is not.*

Remarks 2.11

- *An equivalent definition is that a binary relation $\leq$ on a set A is well-founded iff every nonempty subset B of A ($\emptyset \neq B \subseteq A$) has a minimal element, where $a \in B$ is* minimal *if there is no $a' \in B$ with $a' < a$.*
- *A preorder on a set $(a, b \in)A$ which is also antisymmetric (i.e. $a \leq b$ and $b \leq a$ implies $a = b$) is called a* partial order. *A partial order $\leq$ is called a* total order *if it also has the property that, for each $a, b \in A$, eiher $a \leq b$ or $b \leq a$. In general a well-founded relation need not be a total order !*

Axiom 2.12 *[Generalized or well-founded induction principle] Let $\leq$ be a well-founded binary relation on set $(a, b \in)A$ and let P be a predicate on A. If $P(a)$ holds whenever we have $P(b)$ for all $b < a$, then $P(a)$ is true for all $a \in A$.*

Remark 2.13 *More familiar forms of induction can be obtained by using the following well-founded relations:*

- *$b < a$ if $b + 1 = a$, for natural number induction,*
- *$e' < e$ if e' is an immediate sub-expression of e, for structural induction.*

2.3 Evaluation relation

- In this course we only use operational semantics.
- In particular, we use the so-called "small-step structural operational semantics".

- Small-Step Structural Operational Semantics can be characterized as follows:
  - We define an abstract machine, consisting of:
    - A set of states
    - A transition relation that defines how the state changes over time
  - In the simple case we are considering now, the state is just the program
    - Computation is rewriting ("simplification") of the program
- We leave aside numbers for the moment, and consider a very simple language **B** of just boolean expression
- Apart from the syntax of terms, we also introduce a class of "end-states" or *values.*
- Values are possible final results of evaluation.

Definition 2.14 *[Syntax of terms and values for* **B***]*

(a) (Terms) $t ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$

(b) (Values) $v ::= \mathsf{true} \mid \mathsf{false}$

- The semantics is defined based on an evaluation relation on terms $\rightarrow$
- The elements of the relation $\rightarrow$ are pairs of terms
  - We write $t \rightarrow t'$ to express that $(t, t') \in \rightarrow$
  - $t \rightarrow t'$ expresses the fact that t evaluates to t' in one step.

Definition 2.15 *[Evaluation relation specification for* **B***]*

(E-IfTrue) $\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightarrow t_2$

(E-IfFalse) $\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightarrow t_3$

(E-If) $$\frac{t_1 \rightarrow t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}$$

- Some experts prefer to use the term *reduction* for this relation, instead of *evaluation.* By using this terminology:
  - we speak of "reduction steps" instead of "evaluation steps"
  - $t \rightarrow t'$ expresses the fact that t reduces to t' in one step
- Also, sometimes one uses the following terminology:
  - E-IfTrue and E-IfFalse are *computation rules*
  - E-If is a *congruence rule*

Definition 2.16 *The one-step evaluation relation for* $\mathbf{B} \rightarrow$ *is the* smallest *binary relation on terms satisfying the three rules given above (E-IfTrue, E-IfFalse, E-If).*[1] *When the pair* (t, t') *is in the evaluation relation, we say that* $t \rightarrow t'$ *is* derivable.

[1] For the other languages that we study in this course we will omit this definition. We will always assume implicitly that we only take the smallest relation that satisfy the rules of the given specification. In fact, it is comon practice to avoid repeating this definition and to take the inference rules as constituting the definition of the relation all by themselves.

- A pair (t, t') is in the evaluation relation ($(t, t') \in \rightarrow$) iff it is justified by the rules
- This justification can be made explicit as a *derivation tree*
  - The *leaves* of a derivation tree correspond to the axioms of the specification (the computation rules E-IfTrue and E-IfFalse, in this case)
  - The *internal nodes* of a derivation tree correspond to the inference rules of the specification (the congruence rule E-If, in this case)

Exercise 2.17 *Study the derivation tree given in [1] at page 36, and build the derivation tree that proves the following transition:*

$$\mathsf{if}\ t_1\ \mathsf{then\ false\ else\ true} \rightarrow \mathsf{if}\ t_1'\ \mathsf{then\ false\ else\ true}$$

where

$$t_1 = \mathsf{if}\ t_2\ \mathsf{then\ true\ else\ false}$$

$$t_1' = \mathsf{if\ true\ then\ true\ else\ false}$$

$$t_2 = \mathsf{if\ false\ then\ false\ else\ true}$$

- A powerful technique for proving properties of the evaluation relation is:
  - *Induction on derivations*
- Example: prove the determinacy of the evaluation relation

Theorem 2.18 *[Determinacy of one-step evaluation]: If $t \rightarrow t'$ and $t \rightarrow t''$ then $t' = t''$.*

Exercise 2.19 *The proof of the above theorem (determinacy of one-step evaluation) can proceed by induction on a derivation of $t \rightarrow t'$ (the proof is given in [1] at page 37). Spell out the principle of induction on derivations ([1], page 38, exercise 3.5.5).*

Definition 2.20 *A term t is in* normal form *(sometimes we say "t is a normal form") if no evaluation rule applies to it, i.e., if there is no t' such that $t \rightarrow t'$.*

Theorem 2.21 *Every value is in normal form.*

Remarks 2.22

- *It is natural to design models where values are treated as final results of evaluations*
  - *The property stated by the last theorem given above (every value is in normal form) should be valid in any language.*
  - *For the simple language* **B** *the converse property is also true:*
    - *All normal forms are values (this follows easily by structural induction)*
- *However, in general, normal forms need not be values (see the definition of "stuckness" given below)*

Definition 2.23 *The multi-step evaluation relation $\rightarrow^*$ is the reflexive, transitive closure of the one-step evaluation relation $\rightarrow$.*

- The following properties can be established for the language **B** [1]
  - normal forms are unique (if $t \rightarrow^* u$ and $t \rightarrow^* u'$, where u and u' are both normal forms, then $u = u'$)
  - evaluation always terminates (for every term t there is some normal form t' such that $t \rightarrow^* t'$)
- In the sequel, we consider again the language **NB**
- We extend the above definitions given for **B** with rules for numeric expressions
- The syntax of terms in **NB** was introduced previously, but we repeat it here and we introduce the classes of values and numeric values

Definition 2.24 *[Syntax of terms and values in* **NB***]*

(Terms) $t ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid 0 \mid \mathsf{succ}\ t \mid \mathsf{pred}\ t \mid \mathsf{iszero}\ t$

(Values) $v ::= \mathsf{true} \mid \mathsf{false} \mid nv$

(Numeric values) $nv ::= 0 \mid \mathsf{succ}\ nv$

Definition 2.25 *[Evaluation relation specification for* **NB***]*

(E-IfTrue) $\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \rightarrow t_2$

(E-IfFalse) $\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \rightarrow t_3$

(E-If) $$\frac{t_1 \rightarrow t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightarrow\ \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}$$

(E-Succ) $$\frac{t_1 \rightarrow t_1'}{\mathsf{succ}\ t_1 \rightarrow\ \mathsf{succ}\ t_1'}$$

(E-PredZero) $\mathsf{pred}\ 0 \rightarrow 0$

(E-PredSucc) $\mathsf{pred}\ (\mathsf{succ}\ nv_1) \rightarrow nv_1$

(E-Pred) $$\frac{t_1 \rightarrow t_1'}{\mathsf{pred}\ t_1 \rightarrow\ \mathsf{pred}\ t_1'}$$

(E-IsZeroZero) $\mathsf{iszero}\ 0 \rightarrow \mathsf{true}$

(E-IsZeroSucc) $\mathsf{iszero}\ (\mathsf{succ}\ nv_1) \rightarrow \mathsf{false}$

(E-IsZero) $$\frac{t_1 \rightarrow t_1'}{\mathsf{iszero}\ t_1 \rightarrow\ \mathsf{iszero}\ t_1'}$$

- What properties remain true?
  - Values are normal forms? Yes.
  - All normal forms are values? No.
    - $(\mathsf{succ\ false})$ is in normal form
    - $(\mathsf{succ\ false})$ is *not* a value

- Note the use of numeric values in the rules that specify the evaluation relation.
  - Is it possible to use the rule E-PredSucc to reduce the expression pred (succ false) to false?
  - How do you evaluate the expression pred (succ (pred 0))?
- Normal forms that are not values (like (succ false)) deserve special attention.

Definition 2.26 *A closed term is* stuck *if it is in normal form but not a value.*[2]

- Value terms represent the (normal) *end-results* of computation
- Stuck terms model *run-time errors*
  - A key goal of type systems will be to remove such run-time errors

2.4 A Haskell Implementation of Arithmetic Expressions

- The Haskell code given below implements the syntax and semantics of the language **NB** introduced formally in this chapter 2.
- We implement the class of **NB** terms by using the following `data` declaration:

```
data Term = TmTrue | TmFalse | TmZero
   | TmIf Term Term Term | TmSucc Term
   | TmPred Term | TmIsZero Term

instance Show Term where
   show TmTrue           = "true"
   show TmFalse          = "false"
   show TmZero           = "0"
   show (TmSucc t1)      = "(succ " ++ (show t1) ++ ")"
   show (TmPred t1)      = "(pred " ++ (show t1) ++ ")"
   show (TmIsZero t1)    = "(iszero " ++ (show t1) ++ ")"
   show (TmIf t1 t2 t3)  = " if " ++ (show t1) ++ " then " ++
                           (show t2) ++ " else " ++ (show t3) ++ " "
```

- Values and numeric values are determined with the aid of the predicates `isval` and `isnumericval`, respectively.

```
isval :: Term -> Bool
isval TmTrue  = True
isval TmFalse = True
isval t       = isnumericval t

isnumericval :: Term -> Bool
isnumericval TmZero     = True
isnumericval (TmSucc t) = isnumericval t
isnumericval _          = False
```

[2] A term is said to be *closed* if it contains no (free) variables; see also chapter 3.

- An evaluation relation need not be a function. In general, a term may reduce to zero, one or more terms. To express this idea, we implement the rules that specify the evaluation relation by defining a Haskell function `eval1` of the type:

  ```
  eval1 :: Term -> [Term]
  ```

- The abstract machine that specifies the semantics of **NB** is deterministic
  - The function `eval1 t` always produces:
    - Either a list with exactly one term `[t']`
    - Or an empty list `[]` when the term `t` is in normal form
- In the definition of function `eval1` we use *guards* (see [6] page 16) and *pattern guards*.
  - Haskell 2010 changes the syntax for guards by replacing the use of a single condition with a list of qualifiers
  - These qualifiers, which include both conditions and pattern guards of the form `pat <- exp`, serve to bind/match patterns against expressions
  - You can find more information about patterns guards at [41]

```
eval1 (TmIf TmTrue  t2 t3)    = [t2]
eval1 (TmIf TmFalse t2 t3)    = [t3]
eval1 (TmIf t1      t2 t3)    = [TmIf t1' t2 t3 | t1' <- eval1 t1]
eval1 (TmSucc t1)             = [TmSucc t1' | t1' <- eval1 t1]
eval1 (TmPred TmZero)         = [TmZero]
eval1 (TmPred t1)
   | TmSucc nv1 <- t1,
     isnumericval nv1         = [nv1]
   | otherwise                = [TmPred t1' | t1' <- eval1 t1]
eval1 (TmIsZero TmZero)       = [TmTrue]
eval1 (TmIsZero t1)
   | TmSucc nv1 <- t1,
     isnumericval nv1         = [TmFalse]
   | otherwise                = [TmIsZero t1' | t1' <- eval1 t1]
eval1 _                       = []
```

- The function `eval` takes a term `t` and finds its normal form by repeatedly calling `eval1 t`, until `eval1` returns an empty list
  - When `eval1 t` returns an empty list
    - Either the term `t` is a value
    - Or the term `t` is stuck

```
eval :: Term -> Term
eval t =
   case eval1 t of
        [] -> if isval t then t
              else error ("Term "++ (show t) ++ " is stuck")
        [t'] -> eval t'
        _    -> error "Nondeterministic evaluation (impossible!)"
```

- Let

```
t1 :: Term
t1 = TmSucc (TmPred (TmSucc (TmPred (TmSucc TmZero))))

t2 :: Term
t2 = TmIf (TmIsZero t1) (TmSucc t1) (TmPred t1)

t3 :: Term
t3 = TmIf TmTrue  (TmSucc t1) (TmPred t1)

t4 :: Term
t4 = TmIf TmFalse (TmSucc t1) (TmPred t1)

t5 :: Term
t5 = TmSucc (TmPred TmFalse)
```

- One can perform the following experiments:

```
Main> eval t1
(succ 0)
Main> eval t2
0
Main> eval t3
(succ (succ 0))
Main> eval t4
0
Main> eval t5
*** Exception: Term (succ (pred false)) is stuck
```

Chapter 3

The Untyped Lambda-Calculus

- We now switch to a more interesting programming language than the expression language we considered so far
- The (untyped) lambda-calculus (λ-calculus) is:
  - A Turing-complete language
  - And its key abstractions - function definition and application - are closely related to abstractions found in programming languages

Remarks 3.1

- *The lambda-calculus is of foundational importance in Computer Science*
  - *It is mainly used in the semantic investigation of sequential languages*
- *The untyped lambda-calculus was developed by Alonzo Church in the 1920s and 1930s [22] (a comprehensive presentation of this calculus is provided in [14])*
- *In denotational semantics, the mathematical domain $\mathbb{D}$ that can express denotations of untyped lambda-calculus terms can be defined as solution of the following domain equation*[1] *(first solved in the 1970s by Dana Scott [39]):*

  $$\mathbb{D} \cong \mathbb{D} \to \mathbb{D}$$

  - *At present, most researchers seem to prefer operational semantics*
  - *In this course we only use operational semantics*
- *An important calculus that can be used to express distributed and mobile computation is the pi-calculus (π-calculus), introduced by Robin Milner [9]*

3.1 Basics

- We begin the study of lambda calculus by investigating a very simple language
  - We will use the symbol λ for the language of pure untyped lambda-calculus given in the following definition

[1] Notice that the solution is obtained up to *isomorphism* '$\cong$'.

- A term in this language can be a variable (x), where x is taken from a countable set $\mathcal{V}$ of variable names, a lambda-abstraction ($\lambda x\,.\,t$), or an application ($t\ t$).

Definition 3.2 *[Syntax of λ (the pure untyped lambda-calculus)]* $t ::= x \mid \lambda x\,.\,t \mid t\ t$

Exercise 3.3 *The syntax of λ can be implemented in Haskell as follows:*

```
type X    = String
data Term = TmVar X  | TmAbs X Term | TmApp Term Term
```

By using an appropriate design pattern develop an UML class diagram representing the syntactic constructions of the language λ.

- It is customary to use the following syntactic conventions [1]
  - Application associates to the left ($t\ u\ v$ means $(t\ u)\ v$, not $t\ (u\ v)$)
  - Bodies of λ-abstractions extend as far to the right as possible ($\lambda x\,.\,\lambda y\,.\,x\ y$ means $\lambda x\,.\,(\lambda y\,.\,x\ y)$, not $\lambda x\,.\,(\lambda y\,.\,x)\ y$)
- Scope and free variables
  - In the term $\lambda x\,.\,t$, the variable x is *bound* in t
    - t is the *scope* of the binding
  - A variable is *free* if it is not bound by any enclosing abstraction
  - A term without free variables is said to be *closed*
    - Closed terms are also called *combinators*. Two examples:
      - $id = \lambda x\,.\,x$ (identity, the simplest combinator)
      - $k = \lambda x\,.\,\lambda y\,.\,x$
- Evaluating λ terms always boils down to performing function application:
  - An actual parameter (term) is substituted for the formal parameter in the body of a lambda abstraction, using the following rule called *beta-reduction* (β-reduction):

$$(\lambda x\,.\,t_{12})\ t_2 \rightarrow [x \mapsto t_2]t_{12}$$

  - The term $(\lambda x\,.\,t_{12})\ t_2$ is sometimes called a *redex* ("reducible expression")
- The *substitution* operation $[x \mapsto t_2]t_{12}$ replaces in t_{12} all free occurrences of the variable x with t_2 (the substitution operation is introduced formally in Definition 3.20)

Remark 3.4 *There are several possible evaluation strategies for λ-calculus*

- *Full beta-reduction (any redex may be reduced at any time)*
- *Normal order: leftmost, outermost redex is always reduced first (recall that $id = \lambda x\,.\,x$, $k = \lambda x\,.\,\lambda y\,.\,x$). Example (redexes are underlined at each step):*

$$\underline{k\ (k\ (\lambda z\,.\,k\ z\ id)\ id)}\ id \rightarrow$$
$$\underline{(\lambda y\,.\,k\ (\lambda z\,.\,k\ z\ id)\ id)\ id} \rightarrow$$

$\underline{k\ (\lambda z\,.\,k\ z\ id)}\ id \rightarrow$

$\underline{(\lambda y\,.\,(\lambda z\,.\,k\ z\ id))\ id} \rightarrow$

$\lambda z\,.\,\underline{k\ z}\ id \rightarrow$

$\lambda z\,.\,\underline{(\lambda y\,.\,z)\ id} \rightarrow$

$\lambda z\,.\,z \not\rightarrow$

- *We use the notation $t \not\rightarrow$ to express that the term t cannot be reduced any more (t is regarded as a normal form)*

- *Call-by-name is like normal order, but yet more restrictive (than normal order), in the sense that no reduction is allowed inside abstractions*

$\underline{k\ (k\ (\lambda z\,.\,k\ z\ id)\ id)\ id} \rightarrow$

$\underline{(\lambda y\,.\,k\ (\lambda z\,.\,k\ z\ id)\ id)\ id} \rightarrow$

$\underline{k\ (\lambda z\,.\,k\ z\ id)}\ id \rightarrow$

$\underline{(\lambda y\,.\,(\lambda z\,.\,k\ z\ id))\ id} \rightarrow$

$\lambda z\,.\,k\ z\ id \not\rightarrow$

- *Call-by-value: only the outermost redexes are reduced, and a redex is reduced only after its right-hand side has been reduced to a value*

$k\ (\underline{k\ (\lambda z\,.\,k\ z\ id)}\ id)\ id \rightarrow$

$k\ (\underline{(\lambda y\,.\,(\lambda z\,.\,k\ z\ id))\ id})\ id \rightarrow$

$\underline{k\ (\lambda z\,.\,k\ z\ id)}\ id \rightarrow$

$\underline{(\lambda y\,.\,(\lambda z\,.\,k\ z\ id))\ id} \rightarrow$

$\lambda z\,.\,k\ z\ id \not\rightarrow$

Remark 3.5 *We recall that a* value *is a term that cannot be reduced any further and represents a useful final result of computation. As it will be seen (Definition 3.15), for the language λ the only values are lambda-abstractions.*

Exercise 3.6 *Apply the full beta-reduction strategy to reduce the term $k\ (k\ (\lambda z\,.\,k\ z\ id)\ id)\ id$ to a normal form.*

Exercise 3.7 *Apply each of the evaluation strategies presented above to reduce the term $id\ (id\ (\lambda z\,.\,id\ z))$.*

- The focus in this course is on type systems

- From a type-theoretic perspective the choice of a particular evaluation strategy is of secondary importance

- Following [1], in the sequel we use *call-by-value*

3.2 Programming in the Lambda-Calculus

- Multiple arguments
  - Functions with more than one argument can be simulated using higher order functions
    - $\lambda(x, y) \,.\, s$ is simulated by $\lambda x \,.\, \lambda y \,.\, s$
    - $f\ (a, b)$ is simulated by $f\ a\ b$
  - Such expressions are sometimes called *curried functions*, in honor of the logician Haskell Curry, best known for his work in combinatory logic [24, 25]
- For example, by using Haskell language notation the curried version of the function

  ```
  \(x,y) -> (x+y)
  ```

  is

  ```
  \x -> \y -> (x+y)
  ```

  which can also be written as

  ```
  \x y -> (x+y)
  ```

- The (pure untyped) lambda-calculus is simple but expressive
- In this simple formalism it is possible to encode simple and complex data, including Boolean and numeric values, pairs, lists, trees, etc.
- Church Booleans

  $$tru = \lambda t \,.\, \lambda f \,.\, t$$
  $$fls = \lambda t \,.\, \lambda f \,.\, f$$

Exercise 3.8 *By using the Haskell description of λ terms given in Exercise 3.3 implement the Church booleans tru and fls as Haskell terms*

- The λ term *test*, $test = \lambda l \,.\, \lambda m \,.\, \lambda n \,.\, l\ m\ n$, is like a conditional expression that can be used (to test Church Boolean values) as in the following examples:

  $$test\ tru\ v\ w \rightarrow^* v$$
  $$test\ fls\ v\ w \rightarrow^* w$$

- Indeed

  $$tru\ v\ w = \underline{(\lambda t \,.\, \lambda f \,.\, t)\ v}\ w \rightarrow \underline{(\lambda f \,.\, v)\ w} \rightarrow v$$
  $$fls\ v\ w = \underline{(\lambda t \,.\, \lambda f \,.\, f)\ v}\ w \rightarrow \underline{(\lambda f \,.\, f)\ w} \rightarrow w$$

- One can also define functions on Booleans

  $$not = \lambda b \,.\, b\ fls\ tru$$

$$and = \lambda b\,.\,\lambda c\,.\,b\ c\ fls$$

Exercise 3.9 *Compute the logical expression:* $and\ (not\ fls)\ fls$

- Encoding pairs

$$pair = \lambda f\,.\,\lambda s\,.\,\lambda b\,.\,b\ f\ s$$
$$fst = \lambda p\,.\,p\ tru$$
$$snd = \lambda p\,.\,p\ fls$$

Exercise 3.10 *Using an appropriate reduction strategy verify that* $fst\ (pair\ v\ w) \rightarrow^* v$, *and* $snd\ (pair\ v\ w) \rightarrow^* w$.

- Church numerals

$$c_0 = \lambda s\,.\,\lambda z\,.\,z$$
$$c_1 = \lambda s\,.\,\lambda z\,.\,s\ z$$
$$c_2 = \lambda s\,.\,\lambda z\,.\,s\ (s\ z)$$
$$c_3 = \lambda s\,.\,\lambda z\,.\,s\ (s\ (s\ z))$$

  - Intuitively, if we write $s^n\ z$ for s ("successor") applied n times to z ("zero") then c_n is $\lambda s\,.\,\lambda z\,.\,s^n\ z$.

- Church numerals can be implemented in Haskell (by using the representation of λ terms given in exercise 3.3) as follows:

```
c0 :: Term
c0 = TmAbs "s" (TmAbs "z" (TmVar "z"))
c1 :: Term
c1 = TmAbs "s" (TmAbs "z" (TmApp (TmVar "s") (TmVar "z")))
c2 :: Term
c2 = TmAbs "s" (TmAbs "z"
                  (TmApp (TmVar "s") (TmApp (TmVar "s") (TmVar "z"))))
...
```

- Various functions can be defined on Church numerals
  - Successor: $scc = \lambda n\,.\,\lambda s\,.\,\lambda z\,.\,s\ (n\ s\ z)$
  - Zero test: $iszro = \lambda m\,.\,m\ (\lambda x\,.\,fls)\ tru$
  - Addition: $plus = \lambda m\,.\,\lambda n\,.\,\lambda s\,.\,\lambda z\,.\,m\ s\ (n\ s\ z)$

Remark 3.11

- *We encounter difficulties if we use a particular evaluation order. Under the call-by-value evaluation strategy:*

$$scc\ c_1 \rightarrow c_2',\quad \text{where } c_2' = \lambda s\,.\,\lambda z\,.\,s\ ((\lambda s'\,.\,\lambda z'\,.\,s'\ z')\ s\ z)$$

  - *Under full beta-reduction, $c_2' \to^* c_2$ (in two steps)*
  - *The call-by-value regime does not allow us to reduce the term c_2' yet (c_2' contains a redex, but the redex is under a lambda-abstraction)*

- *Although the language λ (of pure untyped lambda-calculus) is expressive enough for any programming task, it is convenient to extend the calculus with various features, including numbers, booleans and data structures*

3.3 Recursion

- Some λ terms do not have a normal form (such terms are said to *diverge*)
- For example, the term *omega* has no normal form (*omega* is a *divergent* combinator)

$$omega = (\lambda x\,.\,x\ x)\ (\lambda x\,.\,x\ x)$$
$$omega \to omega \to omega \cdots$$

- More surprisingly, although the language λ provides no explicit construction to express recursion, arbitrary recursive functions can be defined in the lambda calculus!
- Let $f : A \to A$ be a function. When $x \in A$ is such that $f(x) = x$, we call x a *fixed point* of f. In a framework where fixed points are unique, we can write $x = fix(f)$.[2]
  - Intuitively, we can define a higher-order mapping $fix : (A \to A) \to A$ such that $fix(f) = x$, which implies
    $$fix(f) = f(fix(f))$$
- Haskell [10, 41] is a purely functional programming language, based on an evaluation strategy known as *call-by-need*.[3]
- In Haskell it is very easy to define the fixed point combinator:

```
fix :: (a -> a) -> a
fix = \f -> f (fix f)
```

(i.e. `fix f = f (fix f)`)

Remark 3.12 *By expanding this definition of the fixed-point combinator we get:*

```
fix f = f (fix f) = f (f (fix f)) = f (f (f (fix f))) = ...
```

- By using `fix`, you can define a recursive function such as `factorial`, as fixed point of a non-recursive higher-order mapping given by

```
hofact :: (Int -> Int) -> Int -> Int
hofact = \fct -> \n -> if (n == 0) then 1 else n * fct (n-1)
```

[2] For example, according to the well-known Banach fixed point theorem [13], any contracting function $f : M \to M$ defined on a nonempty complete metric space M has a *unique* fixed point.

[3] Call-by-need is an optimized version of call-by-name, where if the function argument is evaluated, that value is stored for subsequent uses, in order to avoid the need for subsequent re-evaluation.

- or equivalently

```
hofact fct n = if (n == 0) then 1 else n * fct (n-1)
```

- We put

```
factorial :: Int -> Int
factorial = fix hofact

Main> factorial 5
120
```

- Intuitively, if `n>0` and we compute `(factorial n)` a new copy of (the body of) `hofact` is unrolled upon each recursive call, as suggested below:

```
factorial n = (fix hofact) n
= hofact (fix hofact) n
= if (n==0) then 1 else n * (fix hofact) (n-1)
= if (n==0) then 1 else n * (hofact (fix hofact)) (n-1)
= ...
```

- Another example:

```
g :: (Int -> Int) -> Int -> Int
g = \f -> \n -> if (n == 0) then 0 else 2+f(n-1)

double :: Int -> Int
double n = (fix g) n

Main> double 5
10
```

3.3.1 Example

- In a call-by-value setting the *fixed-point combinator*[4] is more complex:

$$fix = \lambda f\,.\,(\lambda x\,.\,f\ (\lambda y\,.\,x\ x\ y))\ (\lambda x\,.\,f\ (\lambda y\,.\,x\ x\ y))$$

- To avoid complicated manipulations of Church numerals, in the example given below we work in the language $\lambda_{\mathbf{NB}}$. $\lambda_{\mathbf{NB}}$ is obtained by extending the language λ with the basic types of language **NB**, namely Bool and Nat

- In $\lambda_{\mathbf{NB}}$ we can express the function `double` (implemented above in Haskell) as follows:

$$g = \lambda f\,.\,\lambda n\,.\,\mathsf{if}\ (\mathsf{iszero}\ \ n)\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ (\mathsf{succ}\ (f\ (\mathsf{pred}\ \ n)))$$
$$double = fix\ g$$

- We want to compute $double\ (\mathsf{succ}\ \ 0) = (fix\ g)\ (\mathsf{succ}\ \ 0)$

[4] There is a simpler call-by-name fixed point combinator $Y = \lambda f\,.\,(\lambda x\,.\,f\ (x\ x))\ (\lambda x\,.\,f\ (x\ x))$, but Y diverges in a call-by-value setting.

- In the calculations given below we combine
  - The rules that specify the transition relation for the language **NB** and
  - The call by value beta-reduction rule (E-AppAbs) given in Definition 3.16: [5]

(E-AppAbs) $(\lambda x\,.\,t_{12})\ v_2 \rightarrow [x \mapsto v_2]t_{12}$

where v_2 is a *value*, namely an **NB** value or a lambda-abstraction.

- We can compute as follows:

$double\ (\mathsf{succ}\ \ 0)$

$= (fix\ g)\ (\mathsf{succ}\ \ 0)$

$\rightarrow (\lambda x\,.\,g\ (\lambda y\,.\,x\ x\ y))\ (\lambda x\,.\,g\ (\lambda y\,.\,x\ x\ y))\ (\mathsf{succ}\ 0) \qquad [= h\ h\ (\mathsf{succ}\ 0)]$

$\rightarrow g\ dbl\ (\mathsf{succ}\ 0)$

[where $dbl = \lambda y\,.\,h\ h\ y$, $h = \lambda x\,.\,g\ (\lambda y\,.\,x\ x\ y)$]

$\rightarrow (\lambda n\,.\ \mathsf{if}\ (\mathsf{iszero}\ \ n)\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ \ (\mathsf{succ}\ \ (dbl\ (\mathsf{pred}\ \ n))))\ (\mathsf{succ}\ \ 0)$

$\rightarrow \mathsf{if}\ (\mathsf{iszero}\ \ (\mathsf{succ}\ \ 0))\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ \ (\mathsf{succ}\ \ (dbl\ (\mathsf{pred}\ \ (\mathsf{succ}\ \ 0)))))$

$\rightarrow \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ \ (\mathsf{succ}\ \ (dbl\ (\mathsf{pred}\ \ (\mathsf{succ}\ \ 0)))))$

$\rightarrow \mathsf{succ}\ \ (\mathsf{succ}\ \ (dbl\ (\mathsf{pred}\ \ (\mathsf{succ}\ \ 0))))$

$\rightarrow \mathsf{succ}\ \ (\mathsf{succ}\ \ (dbl\ 0))$

$\rightarrow \mathsf{succ}\ \ (\mathsf{succ}\ \ (h\ h\ 0))$

$[\,h\ h \rightarrow g\ (\lambda y\,.\,h\ h\ y)\ = g\ dbl]$

$\rightarrow \mathsf{succ}\ \ (\mathsf{succ}\ \ (g\ dbl\ 0))$

$\rightarrow \mathsf{succ}\ \ (\mathsf{succ}\ \ ((\lambda n\,.\ \mathsf{if}\ (\mathsf{iszero}\ \ n)\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ \ (\mathsf{succ}\ \ (dbl\ (\mathsf{pred}\ \ n))))\ 0))$

$\rightarrow \mathsf{succ}\ \ (\mathsf{succ}\ \ (\mathsf{if}\ (\mathsf{iszero}\ \ 0)\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ \ (\mathsf{succ}\ \ (dbl\ (\mathsf{pred}\ \ 0)))))$

$\rightarrow \mathsf{succ}\ \ (\mathsf{succ}\ \ (\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{succ}\ \ (\mathsf{succ}\ \ (dbl\ (\mathsf{pred}\ \ 0)))))$

$\rightarrow \mathsf{succ}\ \ (\mathsf{succ}\ \ 0)$

Remark 3.13 *For any argument n*

$$dbl\ n \rightarrow h\ h\ n \rightarrow g\ dbl\ n$$

i.e.

$$dbl\ n \rightarrow^* g\ dbl\ n$$

This reduction produces a new copy of g at each recursive call step. Also, note that

$$dbl = \lambda y\,.\,h\ h\ y = \lambda y\,.\,(h\ h)\ y$$

[5] The substitution operation used in the rule (E-AppAbs) is introduced formally in Definition 3.20.

and

$$fix\ g \to h\ h$$

Remark 3.14 *Although it is possible to "encode" various programming features in the pure lambda calculus (e.g., Booleans, numbers, data structures, recursion) in practice it is more convenient to extend the syntax of the calculus with corresponding constructions.*

3.4 Evaluation relation

- In this section we present the formal definition of the operational semantics of λ
- We recall that the only values in λ are lambda-abstractions and the syntax of λ is given by: $t ::= x \mid \lambda x\,.\,t \mid t\ t$.

Definition 3.15 *[Values in λ]* $v ::= \lambda x\,.\,t$

Definition 3.16 *[Evaluation relation specification for call-by-value pure untyped lambda-calculus λ]*

Computation rule:

(E-AppAbs) $(\lambda x\,.\,t_{12})\ v_2 \to [x \mapsto v_2]t_{12}$

Congruence rules:

$$\textit{(E-App1)}\ \frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$$

$$\textit{(E-App2)}\ \frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}$$

Remark 3.17 *In the call-by-value strategy you evaluate an application $(t_1\ t_2)$ as follows: reduce t_1 to a value (i.e., a lambda abstraction) v_1 using rule (E-App1), reduce t_2 to a value v_2 using (E-App2), and next use rule (E-App) to evaluate the application $(v_1\ v_2)$.*

- Substitution can be expressed by using *alpha-conversion*. Systematically replacing some bound variable names in a (lambda calculus) term we obtain another term that, intuitively, should behave the same. For example, we expect that the terms $(\lambda y\,.\,y)$ and $(\lambda x\,.\,x)$ should behave the same, because they denote the same (identity) function.
- Replacing a bound variable name (occurring in a lambda calculus term) with a *fresh* name is called *alpha-conversion*.
- Two terms are said to be *alpha-equivalent* if one can be obtained from the other by using alpha-conversion operations.

Convention 3.18 *[Alpha-conversion] Alpha-equivalent terms are interchangeable in all contexts (i.e., we work with* alpha-equivalence classes of terms*).*

Definition 3.19 *[Free variables of a λ term] The set $FV(t)$ of free variables of an λ term t can be defined inductively:*

$$FV(x) = \{x\}$$

$$FV(\lambda x\,.\,t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$$

- In definition 3.20, in the equation for $[x \mapsto s](\lambda y\,.\,t_1)$ it is always possible (using alpha-conversion) to choose a bound variable name y such that $y \neq x$ and $y \notin FV(s)$.

Definition 3.20 *[Substitution for λ]*

$$\begin{array}{rcll} [x \mapsto s]x & = & s & \\ [x \mapsto s]y & = & y & \text{if } y \neq x \\ [x \mapsto s](\lambda y\,.\,t_1) & = & \lambda y\,.\,([x \mapsto s]t_1) & \text{if } y \neq x, y \notin FV(s) \\ [x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) & \end{array}.$$

Examples 3.21

- $[x \mapsto y](\lambda y\,.\,x)$ *and* $[x \mapsto y](\lambda z\,.\,x)$ *must yield the same result.*
- $[x \mapsto y](\lambda x\,.\,z)$ *and* $[x \mapsto y](\lambda w\,.\,z)$ *must yield the same result.*

Exercise 3.22 *Recal that $id = \lambda x\,.\,x$ (id is the identity combinator). What does the term*

$$(\lambda y\,.\,y\ (\lambda x\,.\,x))\ (\lambda x\,.\,x)\ id$$

reduce to?

Chapter 4

Nameless Representation of Terms

- Alpha-conversion is convenient for theoretical investigations
- For implementation purposes we need a more concrete representation of lambda-terms
- Nicolas De Bruijn (1972) introduced a concept of *nameless term*
  - Natural numbers are used as pointers to λ binders
  - A natural number k represents "the variable bound by the k-th enclosing λ"
  - De Bruijn terms are invariant with respect to alpha-conversion
  - Each closed ordinary lambda term has a unique de Bruijn representation

4.1 Contexts

- In this chapter we consider again the (pure untyped) lambda-calculus λ. We recall the syntax of ordinary lambda-calculus terms

$$t ::= x \mid \lambda x\,.\,t \mid t\ t$$

  where x is a variable name taken from a countable set $\mathcal{V}$, $(\lambda x\,.\,t)$ is a lambda-abstraction and $(t\ t)$ is an application

- Formally, λ-terms written using De Bruijn indices have the following syntax:[1]

$$t ::= k \mid \lambda\,.\,t \mid t\ t$$

  where $k \in \mathbb{N}$ is a de Bruijn index (represented by a natural number)

- It is convenient to introduce nameless (de Bruijn) terms by means of examples

Examples 4.1

Closed λ-calculus term	*Nameless (de Bruijn) term*
$\lambda x\,.\,x$	$\lambda\,.\,0$
$\lambda x\,.\,\lambda y\,.\,x$	$\lambda\,.\,\lambda\,.\,1$
$\lambda x\,.\,\lambda y\,.\,x\ (y\ x)$	$\lambda\,.\,\lambda\,.\,1\ (0\ 1)$
$\lambda x\,.\,\lambda y\,.\,(x\ y)\ y$	$\lambda\,.\,\lambda\,.\,(1\ 0)\ 0$

[1] A formal definition of the set of de Bruijn terms as a family of sets $\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots\}$, where, for any $n \in \mathbb{N}$, the elements of $\mathcal{T}_n$ are terms with at most n free variables is given in [1], Definition 6.1.2.

Exercise 4.2 *What are the nameless representations of the following combinators (i.e., closed terms):* $c_0, c_1, c_2, \ldots$, $scc, tru, iszro, plus, fix$ *?*

- Lambda terms that contain free variables can be translated to corresponding nameless terms by using the concept of a *naming context*
- A naming context Γ is a sequence $x_n, \ldots, x_0$ of variable names from the set $\mathcal{V}$
- To deal with a term t containing n free variables we need a naming context Γ, such that $length(\Gamma) \geq n$

Example 4.3 *Under* $\Gamma = x, y, z, a, b$ *(i.e.,* $x \mapsto 4, y \mapsto 3, z \mapsto 2, a \mapsto 1, b \mapsto 0$*) we have*

λ calculus term with free variables	*Nameless term*	*Remark*
$x\ (y\ z)$	4 (3 2)	
$\lambda w\,.\,y\ w$	$\lambda\,.\,4\ 0$	$4 = 3 + 1$ *(1 λ-binder)*
$\lambda w\,.\,\lambda a\,.\,x$	$\lambda\,.\,\lambda\,.\,6$	$6 = 4 + 2$ *(2 λ-binders)*

Definition 4.4 *Let* $x_0, \ldots, x_n \in \mathcal{V}$.[2] *A* naming context $\Gamma = x_n, x_{n-1} \ldots, x_1, x_0$ *is a sequence that assigns to each* x_i *the de Bruijn index* i*. Let* $Context$ *with typical variable* Γ *be the set of all naming contexts. We write* $dom(\Gamma) = \{x_n, \ldots, x_0\}$ *to denote the set of variables in the sequence* Γ*.*

- A naming context is a finite (possibly empty) sequence or list of variable names, hence we can put $Context = \mathcal{V}^*$, where $\mathcal{V}$ is the given countable set of variable names. We let Γ, Δ range over the set $Context$.
  - Note that in subsequent chapters contexts will also store type information; the type information will mainly be used by the type checker
- We also introduce a set $VnamesContext$ of *variable names contexts*, which are just lists of variable names.
  - In this chapter the two sets ($VnamesContext$ and $Context$) are equal.
  - However, in subsequent chapters contexts will also store type information and the two sets will behave differently.

Definition 4.5 *We define the set* $VnamesContext$ *of* variable names contexts *as follows:* $(\gamma \in) VnamesContext = \mathcal{V}^*$.[3]

A variable names context $\gamma = x_n, x_{n-1} \ldots, x_1, x_0$ *is a sequence that assigns to each* x_i *the de Bruijn index* i*. We write* $dom(\gamma) = \{x_n, \ldots, x_0\}$ *to denote the set of variable names in the sequence* γ*.*

- In the sequel we present a Haskell implementation of de Bruijn nameless terms

[2] Recall that $\mathcal{V}$ is a given countable set of variable names.

[3] This set $VnamesContext$ of variable names contexts is not used in [1]. In this chapter $VnamesContext = Context$. However, in subsequent chapters the elements of the set $Context$ will also contain type information (not included in $VnamesContext$). We use this set $VnamesContext$ to avoid transmitting type information in the arguments of the auxiliary mappings *removenames* and *restorenames* introduced below. The mappings *removenames* and *restorenames* do not need to handle type information.

- The mathematical definitions of the relevant concepts will be constructed in a series of exercises

- First we implement the syntax of ordinary lambda terms:

```
type X    = String
data Term = TmVar X  | TmAbs X Term | TmApp Term Term

instance Show Term where
   show (TmVar x) = x
   show (TmAbs x term) = "(\\" ++ x ++ "." ++ (show term) ++ ")"
   show (TmApp t1 t2) = "(" ++ (show t1) ++ " " ++ (show t2) ++ ")"
```

- The type `X` is the Haskell implementation of the class $\mathcal{V}$ of variable names, and the implementation of terms is obvious

- Also, we implement the syntax of nameless terms as follows:

```
type K       = Int
data T = Tvar K | Tabs X T | Tapp T T

instance Show T where
   show (Tvar k)     = (show k)
   show (Tabs _ t)   = "(\\" ++ "." ++ (show t) ++ ")"
   show (Tapp t1 t2) = "(" ++ (show t1) ++ " " ++ (show t2) ++ ")"
```

- The type `K` implements the class of de Bruijn indices (numeric, nameless variables)

- The implementation of nameless terms is obvious, apart from one detail

  - A nameless abstraction $\lambda\,.\,t$ is implemented by a term `(Tabs x t)` that contains
    - A nameless term `t` implementing the body of the abstraction
    - Following the ML implementation provided in [1], a term `(Tabs x t)` also contains a variable name `x` (of the type `X`) which is used as explained in remark 4.7.

- Our next aim is to implement a pair of functions, `removenames` and `restorenames`, that can be used to convert an ordinary lambda term to a nameless term and back.

```
type Context       = [X]
type VnamesContext = [X]

toVnamesCtx :: Context -> VnamesContext
toVnamesCtx = id

index :: (Eq a) => a -> [a] -> Int
index x []      = error "element out of context"
index x (x':xs) = if (x' == x) then 0 else 1 + index x xs
```

```
removenames :: Term -> VnamesContext -> T
removenames (TmVar x)     xs = Tvar (index x xs)
removenames (TmAbs x t1)  xs =
   Tabs x (removenames t1 (x:xs))
removenames (TmApp t1 t2) xs =
   Tapp (removenames t1 xs) (removenames t2 xs)
```

- In this chapter, both types `Context` and `VnamesContext` are sequences of variable names.[4] The Haskell list provides a convenient mean to implement this concept.
  - Obviously, the two types `Context` and `VnamesContext` represent the Haskell implementation of the sets *Context* and *VnamesContext*, respectively.
- The function `toVnamesCtx` makes a variable names context from a given context.
  - In this chapter `toVnamesCtx` is just the identity mapping
  - In subsequent chapters we will add type information to variable names, and `toVnamesCtx` will remove the type information attached to each variable name

Remarks 4.6

- *According to the mathematical definition given above (based on [1]), the rightmost variable in the sequence is given the de Bruijn index* 0 *(we count binders from right to left)*
- *In Haskell it is more convenient to access the elements in a list from left to right (starting from the head of the list). Also, it is easier to add elements to the left by using Haskell's cons operator ':'.*
- *Hence, in the Haskell implementation the head of a list is given the de Bruijn index* 0*. Also, we count binders from left to right.*
- *When we consider examples taken from the book [1] we present the elements in a (list implementing a) context in reversed order.*[5]
  - *Note that the Haskell standard module* `Prelude.hs` *contains the library function* `reverse` *which reverses a list.*

- `removenames t xs` takes two arguments, an ordinary lambda term `t` and a sequence of variable names `xs`, and yields the corresponding nameless term of the type `T`
  - We only want to compute `(removenames t xs)` when the set of free variables in `t` is included in the variable names context `xs`

Remark 4.7 *In the second equation of* `removenames`*, the variable name* `x` *is stored in the nameless term that is produced. Using this variable name* `x` *as a* hint *[1], the function* `restorenames` *can yield (back) an ordinary lambda term whose bound variable names are as similar as possible to the bound variable names from the original (ordinary) lambda term.*

[4] In general, some binding (e.g., typing) information can be attached to each variable in a context, as it will be seen in subsequent chapters.

[5] In fact, the ML implementation available from `https://www.cis.upenn.edu/ bcpierce/tapl/` also uses this approach, since in ML it is also more convenient to access the elements of a list from left to right.

- The function `restorenames` produces an ordinary lambda term from a nameless term and a naming context, taking into account the hints for the variable names contained in the nameless term

```
restorenames :: T -> VnamesContext -> Term
restorenames (Tvar k)     xs =
   TmVar (xs !! k)
restorenames (Tabs x t1)  xs =
   let x' = pickFreshName xs x
   in  TmAbs x' (restorenames t1 (x':xs))
restorenames (Tapp t1 t2) xs =
   TmApp (restorenames t1 xs) (restorenames t2 xs)

pickFreshName :: VnamesContext -> X -> X
pickFreshName xs x =
   if isBoundName xs x then pickFreshName xs (x ++ "'") else x

isBoundName :: VnamesContext -> X -> Bool
isBoundName []      x = False
isBoundName (x':xs) x = if (x'==x) then True else isBoundName xs x
```

- The library function `!!` (presented in infix form) takes a variable names context `xs` and a de Bruijn index `k` and yields the `k`-th name in `xs`.[6] It is used in the first equation of `restorenames` to restore a variable name corresponding to `k` from the given variable names context `xs`.

- In the second equation of `restorenames` (the equation for abstraction) the hint `x` is used as explained in remark 4.7 (provided the argument of `restorenames` was computed using `removenames`)

  - The function `pickfreshname` is based on an OCaml implementation available from the web site associated with the book [1]
  - `pickfreshname` takes a variable names context `xs` and a name hint `x`
    - it finds a new name `x'` similar to `x`, such that `x'` is not already in `xs`

- We define some Haskell values of the type `Term` (ordinary lambda terms)

```
t1 :: Term
t1 = TmAbs "m" (TmAbs "n" (TmAbs "s" (TmAbs "z"
        (TmApp (TmApp (TmVar "m") (TmVar "s"))
               (TmApp (TmApp (TmVar "n") (TmVar "s")) (TmVar "z"))))))
```

[6] The function `!!` is a general polymorphic operator (from the standard Haskell library `Prelude.hs`), which returns the element of a list located at a specified index, starting from `0`. For example:

```
Main> [1,2,3,4,5] !!  3
4
Main> [1,2,3,4,5] !!  0
1
```

```
t2 :: Term
t2 = TmApp (TmVar "x") (TmApp (TmVar "y") (TmVar "z"))

t3 :: Term
t3 = TmAbs "w" (TmApp (TmVar "y")(TmVar "w"))

t4 :: Term
t4 = TmAbs "w" (TmAbs "a" (TmVar "x"))
```

- One can perform the following experiments:

```
Main> t1
(\m.(\n.(\s.(\z.((m s) ((n s) z))))))
Main> t2
(x (y z))
Main> t3
(\w.(y w))
Main> t4
(\w.(\a.x))
```

- Next, we define some values of the type `T` (de Bruijn terms), that we obtain by applying `removenames`, with respect to given naming contexts. Recall that the Haskell standard module `Prelude.hs` contains the library function `reverse` which reverses a list.

```
ctx1 :: Context
ctx1 = reverse ["x","y","z","a","b"]

tb1 :: T
tb1 = removenames t1 []

tb2 :: T
tb2 = removenames t2 (toVnamesCtx ctx1)

tb3 :: T
tb3 = removenames t3 (toVnamesCtx ctx1)

tb4 :: T
tb4 = removenames t4 (toVnamesCtx ctx1)
```

- We get

```
Main> tb1
(\.(\.(\.(\.((3 1) ((2 0) 1))))))
Main> tb2
(4 (3 2))
Main> tb3
(\.(4 0))
Main> tb4
(\.(\.6))
```

- We design some auxiliary functions in order to test the properties of `removenames` and `restorenames`
- These functions should have the following properties

  `Main> let xs = toVnamesCtx ctx in removenames (restorenames t xs) xs t`

  for any nameless term `t` (provided the naming context `ctx` is sufficiently large to handle the de Bruijn indices corresponding to free variables), and

  `Main> let xs = toVnamesCtx ctx in restorenames (removenames t xs) xs t'`

  and `t'` is identical with `t` up to renaming of bound variables (i.e. `t` and `t'` are alpha-equivalent), for any ordinary term `t` (provided the set of free variables of `t` is included in the set of variables contained in `ctx`).
- Let

```
tt :: T -> Context -> T
tt t ctx =
   let xs = toVnamesCtx ctx in removenames (restorenames t xs) xs

termterm :: Term -> Context-> Term
termterm t ctx =
   let xs = toVnamesCtx ctx in restorenames (removenames t xs) xs
```

- The following experiments confirm that in our implementation `removenames` and `restorenames` satisfy the above mentioned properties

```
Main> tt tb1 []
(\.(\.(\.(\.((3 1) ((2 0) 1))))))
Main> tt tb2 ctx1
(4 (3 2))
Main> tt tb3 ctx1
(\.(4 0))

Main> tt tb4 ctx1
(\.(\.6))
Main> termterm t1 []
(\m.(\n.(\s.(\z.((m s) ((n s) z))))))
Main> termterm t2 ctx1
(x (y z))
Main> termterm t3 ctx1
(\w.(y w))
Main> termterm t4 ctx1
(\w.(\a'.x))
```

Exercise 4.8 *(Source [1], Exercise 6.1.5) In this exercise you should work with the following definitions. The set of ordinary lambda calculus terms* $(t \in)\mathcal{T}$ *is given by*

$$t ::= x \mid \lambda x\,.\,t \mid t\ t$$

where $(x \in)\mathcal{V}$ *is a given countable set of variable names. The set of nameless (de Bruijn) terms* $(t \in)\mathcal{T}_{\mathcal{B}}$ *is given by:*[7]

$$t ::= k \mid \lambda\,.\,t \mid t\ t$$

where $k \in \mathbb{N}$. *Notice that you should not use 'hints' in the (mathematical) definition of nameless terms.*

Let $(\Gamma \in)Context$ *be the set of naming contexts, with typical variable* Γ *ranging over* $Context$. *A naming context is a finite, possibly empty, sequence of variable names, hence we could put* $Context = \mathcal{V}^*$, *where* $\mathcal{V}$ *is the given countable set of variable names.*

Let $(\gamma \in)VnamesContext = \mathcal{V}^*$ *be the set of variable names contexts, with typical variable* γ *ranging over* $VnamesContext$. *A variable names context is a finite, possibly empty, sequence of variable names. You can proceed under the assumption that the variable names in a variable names context* γ *(or a naming context* Γ*) are pairwise distinct.*

1. *Define a (mathematical) function* $removenames_\gamma(t)$, *based on the above Haskell prototype implementation. The function* $removenames$ *receives two arguments:* γ *(a variable names context) and* t *(an ordinary lambda term), such that* $FV(t) \subseteq dom(\gamma)$ *(i.e. the set of free variables in* t *is a subset of* $dom(\gamma)$*); the function* $removenames$ *computes the (single) representation of* t *as a De Bruijn (nameless) term.*

2. *Define a (mathematical) function* $restorenames_\gamma(t)$, *based on the above Haskell prototype implementation. The function* $restorenames$ *receives as arguments a variable names context* γ *and a nameless term* t*; its yield is an ordinary lambda term. You should not use 'hints' in the mathematical definitions! However, you will have to generate (somehow) variable names corresponding to abstractions occuring in a nameless term. The solution proposed in [1] is to work under the assumption that the set* $\mathcal{V}$ *(of variable names) is ordered. Under this assumption, it is accurate to say "choose the first variable name in* $\mathcal{V}$ *that is not already in* $dom(\gamma)$*" [1].*

   The functions $removenames$ *and* $restorenames$ *must behave as follows:*

   - $removenames_\gamma(restorenames_\gamma(t)) = t$, *for any* $t \in \mathcal{T}_{\mathcal{B}}$, *and*
   - $restorenames_\gamma(removenames_\gamma(t)) = t$, *up to alpha-conversion (Convention 3.18), for any* $t \in \mathcal{T}$

In solving this exercise, we recommend that you use the (mathematical) representation of contexts introduced in Definition 4.4 and Definition 4.5. Namely, a context should be represented as sequence of the form $x_n, x_{n-1}, \ldots, x_1, x_0$, *in which binders are counted from right to left (variable* x_i *corresponds to De Bruijn index* i*)*

Exercise 4.9 *Let* $\gamma = x, y, z, a, b$. *Compute* $restorenames_\gamma(removenames_\gamma(x\ (y\ z)))$ *and* $restorenames_\gamma(removenames_\gamma(\lambda w\,.\,\lambda a\,.\,x))$.

[7] A formal definition of $\mathcal{T}_{\mathcal{B}}$ is given in [1], Definition 6.1.2, page 77, although there the set of nameless terms is (also) named $\mathcal{T}$.

4.2 Shifting and Substitution

- Our final aim in this chapter is to define an operational semantics based on an evaluation relation on de Bruijn (nameless) terms
- We need a substitution operation on nameless terms
- In preparation for the presentation of a substitution operation on nameless terms we introduce a shifting operation
- Substitution is computed by induction on the structure of lambda terms.
- When a lambda-abstraction is encountered, as in $[1 \mapsto t](\lambda\,.\,2)$ (i.e. $[x \mapsto t](\lambda y\,.\,x)$, assuming that 1 is the index of x in the outer context)
  - The de Bruijn indices of the free variables in t must be incremented (so that they point to the same names in the new context)
- This is an (selective incrementation) operation called "shifting". Notice that
  - Only the free identifiers should be shifted
  - Identifiers that are bound should *not* be shifted
    - For example, in $t = 2\ (\lambda\,.\,0)$ (i.e., $t = y\ (\lambda x\,.\,x)$, assuming that 2 is the index of y in the outer context)
      - We must only shift the 2 (0 should *not* be shifted)
- The Haskell implementation of this shifting operation is very simple.
  - The function `(tShift d t)` increments the de Bruijn indices of the free variables in a nameless term `t` with `d`

```
tShift :: Int -> T -> T
tShift d t = aux 0 t
   where aux :: Int -> T -> T
         aux c (Tvar k)     = if k>= c then Tvar (k+d) else Tvar k
         aux c (Tabs x t1)  = Tabs x (aux (c+1) t1)
         aux c (Tapp t1 t2) = Tapp (aux c t1) (aux c t2)
```

- The function `tShift` takes two arguments: a numeric value `d` (which, in general, could be negative; see the definition of the evaluation relation) and a nameless term `t`
  - `tShift` uses an auxiliary mapping `aux`
- The right hand side of the first equation that defines the function `aux` is given by a conditional statement with two alternatives
  - The condition `(k>=c)` is true exactly when the identifier `k` is free, and hence, it is incremented with `d` (`d` is the argument of `tShift`)
  - The `else` branch corresponds to the condition `(k<c)` which is true exactly when the identifier `k` is bound (hence it is not incremented)
- The parameter `c` (of `aux`) controls which variables should be shifted, i.e. incremented

  - The parameter `c` is incremented automatically with `1` when `aux` is called recursively to perform the shifting in the body of an abstraction

- Let

```
ctx' :: Context
ctx' = reverse ["z"]

t' :: Term
t' = TmAbs "x" (TmAbs "y" (TmApp (TmVar "x")
                                 (TmApp (TmVar "y") (TmVar "z"))))

tb' :: T
tb' = removenames t' (toVnamesCtx ctx')

t'' :: Term
t'' = TmAbs "x" (TmApp (TmApp (TmVar "x") (TmVar "z"))
                (TmAbs "y" (TmApp (TmApp (TmVar "y") (TmVar "x"))
                                  (TmVar "z"))))

tb'' :: T
tb'' = removenames t'' (toVnamesCtx ctx')
```

- One can perform the following experiments

```
Main> t'
(\x.(\y.(x (y z))))
Main> t''
(\x.((x z) (\y.((y x) z))))
Main> tb'
(\.(\.(1 (0 2)))
Main> tb''
(\.((0 1) (\.((0 1) 2))))
Main> tShift 2 tb'
(\.(\.(1 (0 4)))
Main> tShift 2 tb''
(\.((0 3) (\.((0 1) 4))))
Main> tShift 100 tb'
(\.(\.(1 (0 102)))
Main> tShift 100 tb''
(\.((0 101) (\.((0 1) 102))))
```

- We also present some experiments with variables `tb1`, `tb2`, `tb3` and `tb4`

  - `tb1` contains no free variables (the result is not modified by the shifting operation)

```
Main> tShift 100 tb1
(\.(\.(\.(\.((3 1) ((2 0) 1))))))
Main> tShift 100 tb2
(104 (103 102))
```

```
Main> tShift 100 tb3
(\.(104 0))
Main> tShift 100 tb4
(\.(\.106))
```

- In the sequel we define the shifting operation formally
  - $\uparrow^d (t)$ is the mathematical function that corresponds to `tShift d t`
  - $\uparrow_c^d (t)$ is the mathematical function that corresponds to `aux c t` (in the mathematical definition d is also transmitted as a parameter of $\uparrow_c^d (t)$)

Definition 4.10 *[Shifting]* $\uparrow_c^d (t)$ *is the d-place shift of a nameless term t above c, given by:*

$$\uparrow_c^d (k) = \begin{cases} k & \text{if } k < c \\ k + d & \text{if } k \geq c \end{cases}$$

$$\uparrow_c^d (\lambda\,.\,t_1) = \lambda\,.\ \uparrow_{c+1}^d (t_1)$$

$$\uparrow_c^d (t_1\ t_2) = (\uparrow_c^d (t_1))\ (\uparrow_c^d (t_2))$$

We put $\uparrow^d (t) = \uparrow_0^d (t)$.

Exercise 4.11 *(Source [1], Exercise 6.2.2) Compute:* $\uparrow^{10} (\lambda\,.\,\lambda\,.\,1\ (0\ 2))$, $\uparrow^{10} (\lambda\,.\,0\ 1\ (\lambda\,.\,0\ 1\ 2))$ *and* $\uparrow^{10} (\lambda\,.\,4\ 0)$.

- Next we construct the Haskell prototype function `(tSubst j s t)` which implements the substitution operation on nameless terms

```
tSubst :: K -> T -> T -> T
tSubst j s (Tvar k)     = if (k == j) then s else (Tvar k)
tSubst j s (Tabs x t1)  = Tabs x (tSubst (j+1) (tShift 1 s) t1)
tSubst j s (Tapp t1 t2) = Tapp (tSubst j s t1) (tSubst j s t2)
```

- The first argument of `tSubst` is a variable `j` (a de Bruijn index), the second and the third arguments are nameless (de Bruijn) terms
- The substitution operation `(tSubst j s t)` is designed to replace free occurrences of `j` in `t` by `s`
- If `t` is a variable `(Tvar k)` then
  - If `k == j` then `tSubst` returns `s`
  - Otherwise `tSubst` returns `(Tvar k)`
- If `t` is an abstraction `(Tabs x t1)`[8] then `tSubst` is called recursively on `t1`
  - The recursive call of `tSubst` on `t1` takes as parameters
    - `(j+1)`, because `tSubst` goes under a lambda abstraction, and

[8] Recall that in a nameless term `(Tabs x 1)`, `x` is just a hint, used by `restorenames` to produce similar names.

  - `(tShift 1 s)`, i.e. the 1-place shift of `s`

- If `t` is an application `(Tapp t1 t2)` then `tSubst` is called recursively upon the two substerms `t1` and `t2`
- We present some experiments to show how `tSubst` works
- We use an auxiliary function, a naming context and some terms

```
termSubst :: Int -> Term -> Term -> Context -> Term
termSubst j s t ctx =
   let xs = (toVnamesCtx ctx)
       s' = removenames s xs
       t' = removenames t xs
   in restorenames (tSubst j s' t') xs

ctx2 :: Context
ctx2 = reverse ["a","b"]

xs2 :: VnamesContext
xs2 = toVnamesCtx ctx2

t5, t6, t7, t8 :: Term
t5 = TmVar "a"
t6 = TmApp (TmVar "a") (TmAbs "z" (TmVar "a"))
t7 = TmApp (TmVar "b") (TmAbs "x" (TmAbs "y" (TmVar "b")))
t8 = TmApp (TmVar "b") (TmAbs "x" (TmVar "b"))
```

- One can perform the following experiments

```
Main> t5
a
Main> t6
(a (\z.a))
Main> t7
(b (\x.(\y.b)))
Main> t8
(b (\x.b))
Main> removenames t5 xs2
1
Main> removenames t6 xs2
(1 (\.2))
Main> removenames t7 xs2
(0 (\.(\.2)))
Main> removenames t8 xs2
(0 (\.1))
Main> tSubst 0 (removenames t5 xs2) (removenames t7 xs2)
(1 (\.(\.3)))
Main> tSubst 0 (removenames t6 xs2) (removenames t8 xs2)
((1 (\.2)) (\.(2 (\.3))))
```

```
Main> termSubst 0 t5 t7 ctx2
(a (\x.(\y.a)))
Main> termSubst 0 t6 t8 ctx2
((a (\z.a)) (\x.(a (\z.a))))
```

- The formal definition of substitution is given in the following

Definition 4.12 *[Substitution]*

$$[j \mapsto s]k = \begin{cases} s & \text{if } k = j \\ k & \text{otherwse} \end{cases}$$

$$[j \mapsto s](\lambda\,.\,t_1) = \lambda\,.\,[j+1 \mapsto\uparrow^1 (s)]t_1$$

$$[j \mapsto s](t_1\ t_2) = ([j \mapsto s]t_1)\ ([j \mapsto s]t_2)$$

Exercise 4.13 *Solve Exercise 6.2.5 from the book [1]*

Exercise 4.14 *Solve Exercise 6.2.8 from the book [1]*

4.3 Evaluation

- Only abstractions are values in the pure untyped lambda calculus

```
isval ::  T -> Context -> Bool
isval (Tabs _ _) ctx = True
isval _          ctx = False
```

- We present a Haskell implementation of the call by value evaluation relation for nameless (de Bruijn) lambda terms

```
eval1 :: T -> Context -> [T]
eval1 (Tapp t1 t2)   ctx
   | (Tabs x t12) <- t1,
     isval t2 ctx        = [ tShift (-1) (tSubst 0 (tShift 1 t2) t12) ]
   | isval t1 ctx        = [ Tapp t1 t2' | t2' <- eval1 t2 ctx ]
   | otherwise           = [ Tapp t1' t2 | t1' <- eval1 t1 ctx ]
eval1 (Tabs _ _)     ctx = []
eval1 (Tvar _)       ctx = []
```

- The rules for application are as usual, except for the rule of beta-reduction (the subcase when `t1` is of the form `(Tabs x t12)`, and `(isval t2)`)
  - In this subcase the substitution operation for nameless terms must be used
- To explain this rule we consider the following example of beta-reduction

  $(\lambda x\,.\,y\ x)\ (\lambda u\,.\,u) \rightarrow y\ (\lambda u\,.\,u)$
  - Notice that the variable x dissapears in this reduction!
- Assuming we use a naming context $\Gamma = y, z$ and a corresponding variable names context $\gamma = y, z$ we have

  - $removenames_\gamma(\lambda x\,.\,y\ x) = \lambda\,.\,2\ 0$
    - Here y is represented by the index 2, because it occurs under a lambda abstraction
  - $removenames_\gamma(\lambda u\,.\,u) = \lambda\,.\,0$
  - $removenames_\gamma(y\ (\lambda u\,.\,u)) = 1\ (\lambda\,.\,0)$
    - Here y is represented by the index 1, because upon beta-reduction it was "released" from a lambda abstraction

- Hence we want to compute as follows

  $(\lambda\,.\,2\ 0)\ (\lambda\,.\,0) \to 1\ (\lambda\,.\,0)$ (not $(\lambda\,.\,2\ 0)\ (\lambda\,.\,0) \to 2\ (\lambda\,.\,0)$!)

  - This is why in the beta-reduction rule for nameless terms we have to apply `tShift (-1)` to the result of substitution

- Also, notice that `t12` occurs in a larger context than `t2` (in the sense that `t12` occurs under a lambda abstraction)
  - Hence, before performing the substitution into `t12` (in the implementation of the beta-reduction rule) we must shift the variables in `t2` up by `1`
    - In the Haskell prototype implementation this shifting operation is expressed by the expression `(tShift 1 t2)`
      - (Note that in the case of beta-reduction `t2` is a *value*)

Remark 4.15 *The negative shifting (`tShift (-1)`) operation (in the beta-reduction rule) cannot give rise to negative indices. Upon substitution, the only occurence of the index `0` is replaced by a term that was shifted (up) by `1`, namely by the term `(tShift 1 t2)`*

- The function `evalT` takes as parameter a nameless term `t` which it reduces to a value
  - For this purpose it calls repeatedly `eval1`

```
evalT :: T -> Context -> T
evalT t ctx =
   case eval1 t ctx of
        [] -> t
        [t'] -> evalT t' ctx
        _    -> error "Nondeterministic evaluation (impossible!)"
```

- The function `evalTerm` can be used to evaluate ordinary lambda-terms
  - It takes two arguments: an ordinary lambda term `t` and a naming context `ctx`
  - It converts `t` to a nameless term
  - It reduces the nameless term to a value by using `evalT`
  - Next it converts the result back to an ordinary lambda term

```
evalTerm :: Term -> Context -> Term
evalTerm term ctx =
   let xs = toVnamesCtx ctx
   in  restorenames (evalT (removenames term xs) ctx) xs
```

- To test this evaluator we define

```
t10 :: Term
t10 = TmApp (TmAbs "x" (TmApp (TmVar "y") (TmVar "x"))) (TmAbs "u" (TmVar "u"))

ctx3 :: Context
ctx3 = reverse ["y","z"]
```

- Also, we use

```
c0 :: Term
c0 = TmAbs "s" (TmAbs "z" (TmVar "z"))

c1 :: Term
c1 = TmAbs "s" (TmAbs "z" (TmApp (TmVar "s") (TmVar "z")))

scc :: Term
scc = TmAbs "n" (TmAbs "s" (TmAbs "z" (TmApp (TmVar "s")
                           (TmApp (TmApp (TmVar "n")(TmVar "s"))
                                  (TmVar "z")))))
```

- One can perform the following experiments:

```
Main> removenames t10 (toVnamesCtx ctx3)
((\.(2 0)) (\.0))
Main> evalT (removenames t10 (toVnamesCtx ctx3)) ctx3
(1 (\.0))
Main> c1
(\s.(\z.(s z)))
Main> c0
(\s.(\z.z))
Main> evalTerm (TmApp scc c0) []
(\s.(\z.(s ((((\s'.(\z'.z')) s) z))))
```

- Notice that in the last experiment the result is behaviorally equivalent with `c1`
  - `(TmApp scc c0)` is the implementation of $(scc\ c_0)$
  - By using mathematical notation, the computation performed in the last experiment is: $(scc\ c_0) \rightarrow^* c_1$
- Finally, we present the evaluation rules for nameless λ-terms

Definition 4.16 *[Evaluation relation specification for call-by-value nameless lambda-terms]*

Computation rule:

(E-AppAbs) $(\lambda\,.\,t_{12})\ v_2\ \rightarrow \uparrow^{-1} ([0 \mapsto \uparrow^1 (v_2)]t_{12})$

Congruence rules:

$$\textit{(E-App1)}\ \frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$$

$$\textit{(E-App2)}\ \frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}$$

Remark 4.17 *As before, the only values are lambda-abstraction (values are denoted by the metavariable v in the above definition)*

Exercise 4.18 *By using the beta-reduction rule given in the above definition prove that*

$$(\lambda\,.\,1\ 0\ 2)\ (\lambda\,.\,0) \to 0\ (\lambda\,.\,0)\ 1$$

Chapter 5

Typed Arithmetic Expressions

- This chapter
  - Presents a very simple type system for the language of arithmetic expressions **NB** (previously considered in chapter 2)
  - Introduces the basic notions and properties related to type systems

5.1 Syntax of arithmetic expressions

- We recall the syntax of **NB** terms (with typical variable t), *values* (with typical variable v) and *numeric values* (with typical variable nv):

  $t ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid 0 \mid \mathsf{succ}\ t \mid \mathsf{pred}\ t \mid \mathsf{iszero}\ t$

  $v ::= \mathsf{true} \mid \mathsf{false} \mid nv$

  $nv ::= 0 \mid \mathsf{succ}\ nv$

- Normally, we expect that the evaluation of a term should yield a value. However, some **NB** expressions do not have a clear meaning.
  - An expression like $\mathsf{succ\ true}$, deserves to be called a *stuck term*: its evaluation cannot produce a value
- By introducing a a type system for **NB** we can avoid evaluating such stuck terms
  - The type system can detect such stuck terms statically (at "compile time", i.e., without performing evaluations)
- In the sequel we introduce various classes of types with typical elements denoted by T, S, R, U, etc.

Definition 5.1 *[Typing relation for* **NB***] We define the class of* **NB** *types with elements denoted by $T, R, S, ...$ as follows:*

$T ::= \mathsf{Bool} \mid \mathsf{Nat}$

We define a (binary) typing relation for **NB**, *whose elements are pairs* (t, T), *where* t *is an* **NB** *term and* T *is an* **NB** *type. We use the notation* $t : T$ *to express that the term* t *has type* T *(i.e., the pair* (t, T) *is an element of the typing relation). The typing relation for* **NB** *is the smallest binary relation satisfying the following set of rules:*

(T-Zero) $0 : \mathsf{Nat}$

(T-True) $\mathsf{true} : \mathsf{Bool}$

(T-False) $\mathsf{false} : \mathsf{Bool}$

(T-If) $$\frac{t_1 : \mathsf{Bool} \qquad t_2 : T \qquad t_3 : T}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}$$

(T-Succ) $$\frac{t_1 : \mathsf{Nat}}{\mathsf{succ}\ t_1 : \mathsf{Nat}}$$

(T-Pred) $$\frac{t_1 : \mathsf{Nat}}{\mathsf{pred}\ t_1 : \mathsf{Nat}}$$

(T-IsZero) $$\frac{t_1 : \mathsf{Nat}}{\mathsf{iszero}\ t_1 : \mathsf{Bool}}$$

Let t *be an* **NB** *term. If there is some* **NB** *type* T *such that* $t : T$, *we say that* t *is* well-typed.

- By using the above set of rules, for any well-typed term t one can build a derivation tree to infer the type of t

Exercise 5.2 *Let* $t \in \mathbf{NB}$, $t =$ $\mathsf{if}\ \mathsf{iszero}\ (\mathsf{succ}\ 0)\ \mathsf{then}\ \mathsf{succ}\ 0\ \mathsf{else}\ \mathsf{pred}\ 0$. *Study the typing derivation tree given in [1] at page 94, and build the derivation tree that proves* $t : \mathsf{Nat}$.

Exercise 5.3 *(Uniqueness of types for* **NB***; see [1], Theorem 8.2.4) Let* t *be an* **NB** *term. By structural induction on* t, *prove the following:*

- *If* $t : T$ *and* $t : T'$ *(where* T, T' *are* **NB** *types) then* $T = T'$
- *The derivation tree (inference tree) proving that* $t : T$ *is also unique*

(In this proof you may (find convenient to) use the "inversion lemma" given in [1], at page 94; see also Remark 5.10)

Remark 5.4 *Uniqueness of types is easily established for* **NB** *but it may fail for other type systems (see the chapter on subtyping, presented in [1], part III)*

5.2 Haskell implementation of the typing relation

- We implement the syntax of **NB** as in chapter 3.

```
data Term = TmTrue | TmFalse | TmZero | TmIf Term Term Term
   | TmSucc Term | TmPred Term | TmIsZero Term
```

- Also, we implement the class of types for **NB** by using the following `data` declaration.

```
data Ty = TyBool | TyNat

instance Show Ty where
   show TyBool  = "bool"
   show TyNat   = "nat"
```

- In **NB** each well-typed term has a single type (Exercise 5.3). We implement the typing relation for **NB** as a function `typeof`.

```
typeof :: Term -> Ty
typeof TmTrue  = TyBool
typeof TmFalse = TyBool
typeof TmZero  = TyNat
typeof (TmIf t1 t2 t3) = case (typeof t1, typeof t2 == typeof t3) of
                              (TyBool,True) -> typeof t2
                              _             -> error "TmIf: type error"
typeof (TmSucc t1)     = case (typeof t1) of
                              TyNat -> TyNat
                              _     -> error "TmSucc: type error"
typeof (TmPred t1)     = case (typeof t1) of
                              TyNat -> TyNat
                              _     -> error "TmPred: type error"
typeof (TmIsZero t1)   = case (typeof t1) of
                              TyNat -> TyBool
                              _     -> error "TmIsZero: type error"
```

- We use a variable `t` of the type `TmTerm`.

```
t :: Term
t = TmIf (TmIsZero (TmSucc TmZero)) (TmSucc TmZero) (TmPred TmZero)
```

- One can perform the following experiment:

```
Main> typeof t
nat
```

5.3 Basic properties of type systems: safety = progress + preservation

- Computations with well-typed terms should be safe:

  Safety = Progress + Preservation.

- These properties are introduced and established below in the context of **NB**, for which we can prove the following theorems:

Theorem 5.5 *[Progress] In the language* **NB***, if $t : T$ (i.e., if t is a well-typed* **NB** *term that has type T) then either t is a value, or else there is some t' with $t \to t'$.*

Theorem 5.6 *[Preservation] In the language* **NB***, if $t : T$ and $t \to t'$ then $t' : T$.*

Remark 5.7 Safety *(=* progress + preservation*)*[1] *is an exigency imposed upon all type systems studied in [1] (for further explanations see [1], Section 8.3).*

Exercise 5.8 *The proofs of the above theorems are provided in [1], at pages 96 and 97, respectively. Follow carefully the steps of the proofs.*

- *Prove the canonical forms Lemma 5.9 (given in [1], Lemma 8.3.1); you may need the inversion lemma (given in [1], page 94; see also Remark 5.10 given below).*
- *Prove the progress Theorem 5.5 (using Lemma 5.9) and the preservation Theorem 5.6 (in the both cases you can proceed by induction on a derivation of $t : T$)*

Lemma 5.9 *[Canonical forms] If v : Bool then either $v = \mathsf{true}$ or $v = \mathsf{false}$. If v : Nat then v is a numeric term nv (recall that $nv ::= 0 \mid \mathsf{succ}\ nv$).*

Remark 5.10 *A so-called "inversion lemma" is presented in [1], at page 94. The inversion lemma provides a collection of simple facts (that you may need in various proofs, including the proof of Lemma 5.9). For example, the inversion lemma states the following property: if $\mathsf{succ}\ t_1 : R$ then $R = \mathsf{Nat}$ and $t_1 : \mathsf{Nat}$.*

Indeed, the rule (T-Succ)

$$\frac{t_1 : \mathsf{Nat}}{\mathsf{succ}\ t_1 : \mathsf{Nat}}$$

states that if t_1 has type Nat then $\mathsf{succ}\ t_1$ also has type Nat. However, notice that this is the single rule that allows you to infer the type of $\mathsf{succ}\ t_1$. Hence, it also implies that if $\mathsf{succ}\ t_1 : R$ (i.e., if $\mathsf{succ}\ t_1$ is well typed and has type R), then $R = \mathsf{Nat}$ and $t_1 : \mathsf{Nat}$ (i.e., t_1 also has type Nat).

The following properties are also easily established: if $\mathsf{true} : R$ then $R = \mathsf{Bool}$, if $\mathsf{false} : R$ then $R = \mathsf{Bool}$, and if $(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) : R$ then $t_1 : \mathsf{Bool}$, $t_2 : R$ and $t_3 : R$.

[1] *Soundness* is another name used for *safety*.

Chapter 6

Simply Typed Lambda Calculus

- We investigate the well-known simply typed lambda calculus [21, 24].
- First, we present the syntax of the calculus, the typing relation and its properties.
  - The type safety property now requires a *"substitution lemma"* [1]
- The rules that specify the evaluation relation for this calculus are based on the rules given in the previous chapters.

6.1 Syntax and the typing relation

- For ease of presentation, function types are defined over the base type Bool
  - The language can be extended with other base types, such as Nat or Float, without difficulty.
- In this chapter we use the symbol $\lambda^{\rightarrow}_{\mathbf{B}}$ as the name of this language (of simply typed lambda calculus over the base type Bool).

Definition 6.1 *Using BNF, we introduce the class $\Theta^{\rightarrow}_{\mathbf{B}}$ of $\lambda^{\rightarrow}_{\mathbf{B}}$ simple types (over* Bool*) by:*

$$T ::= \mathsf{Bool} \mid T \rightarrow T$$

We let T, S, U range over $\Theta^{\rightarrow}_{\mathbf{B}}$.

Let T_1, T_2 and T_3 be $\lambda^{\rightarrow}_{\mathbf{B}}$ types. Let t_1, t_2 and t_3 be $\lambda^{\rightarrow}_{\mathbf{B}}$ terms (the syntax of $\lambda^{\rightarrow}_{\mathbf{B}}$ terms is introduced in Definition 6.4). The conventions given below are customary in the context of language $\lambda^{\rightarrow}_{\mathbf{B}}$:

- We specify function types using the constructor $\rightarrow$, which associates to the right:

  $T_1 \rightarrow T_2 \rightarrow T_3$ parses as $T_1 \rightarrow (T_2 \rightarrow T_3)$

- On the other hand, the association of the application operation is to the left (a convention already stated in chapter 3):

  $t_1\ t_2\ t_3$ should be parsed as $(t_1\ t_2)\ t_3$

Exercise 6.2 *Explain the meaning of the type expression* $(\mathsf{Bool} \to \mathsf{Bool}) \to (\mathsf{Bool} \to \mathsf{Bool})$. *Is there any difference between* $(\mathsf{Bool} \to \mathsf{Bool}) \to (\mathsf{Bool} \to \mathsf{Bool})$ *and* $(\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool}$*?*

- $\lambda^{\to}_{\mathbf{B}}$ is introduced as an *explicitly typed* language
- A lambda abstraction is now a construction of the form $\lambda x : T \,.\, t$, which expresses the fact that the type of the variable x is constrained to be T;
  - the type of the term t in a lambda abstraction $\lambda x : T \,.\, t$ can be inferred by the type checker under the assumption that the type of the variable x is T.
- The type checker must use a so-called *typing context* (or *typing environment*), which is essentially a function (with finite graph) from variables to types.
- Typing contexts are used to assign types to (free) variables in $\lambda^{\to}_{\mathbf{B}}$ terms.
  - Note that a typing context need only assign values to a finite number of variables, hence, a context describes a *finite* mapping.
  - We use symbols such as Γ or Δ to denote typing contexts. Let $(\Gamma, \Delta \in) \mathit{Context}$ be the set of typing contexts for $\lambda^{\to}_{\mathbf{B}}$.
- Semantically, a typing context $\Gamma \in \mathit{Context}$ is a finite function which maps variables to types.
  - We write $\mathit{dom}(\Gamma)$ for the domain of Γ (a context $\Gamma \in \mathit{Context}$ is a function)
- It is customary to write a typing context Γ as a list $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ of pairs of variables and types, such that the variables $x_1, \ldots, x_n$ are distinct [1].
  - Intuitively, Γ assigns type T_i to variable x_i, $i = 1, \ldots, n$
- The empty typing context is (either omitted or) denoted by the symbol $\emptyset$
- Let $\Gamma \in \mathit{Context}$, $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, be a context. We note the following:
  - The domain of Γ is $\mathit{dom}(\Gamma) = \{x_1, \ldots, x_n\}$
  - The order of the pairs $x_i : T_i$ in such a typing context is not important (see also permutation lemma, mentioned in exercise 6.17)
- The typing relation becomes a set of triples (Γ, t, T), written as $\Gamma \vdash t : T$ (rather than a set of pairs $t : T$, as it was in chapter 5).
  - The empty context is often omitted: we just write $\vdash t : T$ instead of $\emptyset \vdash t : T$.

Remark 6.3 *We write* $\Gamma \vdash t : T$ *to express that the type of term* t *is* T*, assuming that the types of free variables in* t *are bound by* Γ *(i.e.,* $FV(t) \subseteq \mathit{dom}(\Gamma)$*, where the notation* $FV(t)$ *is presented in Definition 6.4).*

- Recall that, in the lambda calculus we can always rename a bound variable by using a fresh new name.
  - This general convention is sometimes called *alpha-conversion* (convention 3.18).

- In particular, alpha-conversion may be needed in rule (T-Abs) of the type checker for $\lambda^{\rightarrow}_{\mathbf{B}}$ (see Definition 6.7) to ensure that the variables of a typing context are (indeed) pairwise distinct.
  - The problem occurs when we want to infer the type of a lambda-abstraction $\lambda x : T\,.\,t$ (in which the variable x is bound) with regard to a typing context Γ which already contains the variable x in its domain, i.e., when $x \in dom(\Gamma)$.
  - In this case, the bound variable name x in the term $\lambda x : T\,.\,t$ is renamed (by means of alpha-conversion) to a new fresh name $x' \notin dom(\Gamma)$ before using rule (T-Abs).

Definition 6.4 *[Syntax of $\lambda^{\rightarrow}_{\mathbf{B}}$ (simply typed lambda-calculus over* Bool*)] We recall the definition of the class ($\Theta^{\rightarrow}_{\mathbf{B}}$) of simple types of $\lambda^{\rightarrow}_{\mathbf{B}}$ given by: $T ::= \mathsf{Bool} \mid T \rightarrow T$. Using a countable set ($x \in$)$\mathcal{V}$ of variable names, we define the class ($t \in$)$\mathcal{T}^{\rightarrow}_{\mathbf{B}}$ of $\lambda^{\rightarrow}_{\mathbf{B}}$* terms *as follows:*

$$t ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid x \mid \lambda x : T\,.\,t \mid t\ t$$

The set of free variables $FV(t)$ *of a $\lambda^{\rightarrow}_{\mathbf{B}}$ term t can be defined by structural induction on t as follows: $FV(\mathsf{true}) = FV(\mathsf{false}) = \emptyset$, $FV(x) = \{x\}$, $FV(\lambda x : T\,.\,t_1) = FV(t_1) \setminus \{x\}$, $FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$ and $FV(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = FV(t_1) \cup FV(t_2) \cup FV(t_3)$.*

- In Haskell we implement the syntax of $\lambda^{\rightarrow}_{\mathbf{B}}$ as follows:
  - The set ($x \in$)$\mathcal{V}$ of *variable names* is implemented by the type `X` in Haskell
  - The set ($t \in$)$\mathcal{T}^{\rightarrow}_{\mathbf{B}}$ of *terms* (Definition 6.4) is implemented by the type `Term`
  - The set ($T \in$)$\Theta^{\rightarrow}_{\mathbf{B}}$ of *simple types* (Definition 6.1) is implemented by the type `Ty`

```
type X    = String
data Ty   = TyBool | TyFun Ty Ty deriving Eq
data Term = TmTrue | TmFalse | TmIf Term Term Term
          | TmVar X  | TmAbs (X,Ty) Term | TmApp Term Term
```

- We also provide `Show` instances for `Ty` and `Term`.

```
instance Show Ty where
   show TyBool             = "bool"
   show (TyFun ty1 ty2)    =
      "(" ++ (show ty1) ++ "->" ++ (show ty2) ++ ")"

instance Show Term where
   show TmTrue             = "true"
   show TmFalse            = "false"
   show (TmIf t1 t2 t3)    =
      "(if " ++ (show t1) ++ " then " ++ (show t2) ++
      " else " ++ (show t3) ++ ")"
   show (TmVar x)          = x
   show (TmAbs (x,ty) term) =
      "(\\" ++ "(" ++ x ++ ":" ++ (show ty) ++ ")" ++ "."
      ++ (show term) ++ ")"
   show (TmApp t1 t2)      =
      "(" ++ (show t1) ++ " " ++ (show t2) ++ ")"
```

- As explained in chapter 4, for implementation purposes it is more convenient to work with the nameless representation of $\lambda^{\rightarrow}_{\mathbf{B}}$ terms
- The class of *contexts* is used both to handle the nameless form of terms, and to associate simple types with (free) variables

Definition 6.5 *The class* $(\Gamma, \Delta \in)Context$ *of* typing contexts *for* $\lambda^{\rightarrow}_{\mathbf{B}}$ *is given by:*[1]

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

- We also use the set $(\gamma \in)VnamesContext = \mathcal{V}^*$ of *variable names contexts* with typical variable γ. We denote the empty sequence (over $\mathcal{V}$) by $\emptyset \in VnamesContext$.
- We define a function $toVnamesCtx : Context \rightarrow VnamesContext$ by

$$toVnamesCtx(\emptyset) = \emptyset$$
$$toVnamesCtx(\Gamma, x : T) = toVnamesCtx(\Gamma), x$$

- The function $toVnamesCtx$ takes as argument a typing context Γ and yields a corresponding variable names context obtained by removing the type information.
- The nameless representation of $\lambda^{\rightarrow}_{\mathbf{B}}$ terms is a mere extension of the representation given in chapter 4
  - In the sequel we only provide the Haskell implementation of $\lambda^{\rightarrow}_{\mathbf{B}}$ nameless terms, leaving the interested reader (the easy task) to infer the formal (mathematical) definitions, based on the explanations provided in chapter 4
  - In the implementation given below the type `K` implements the class of de Bruijn indices (numeric, nameless variables)
  - The type `T` (for which we also provide a `Show` instance) implements the class of $\lambda^{\rightarrow}_{\mathbf{B}}$ nameless terms

```
type K = Int
data T = Ttrue | Tfalse | Tif T T T
       | Tvar K | Tabs (X,Ty) T | Tapp T T

instance Show T where
   show Ttrue           = "true"
   show Tfalse          = "false"
   show (Tif t1 t2 t3)  =
      "(if " ++ (show t1) ++ " then " ++ (show t2) ++
      " else " ++ (show t3) ++ ")"
   show (Tvar k)        = (show k)
```

[1] Following [1], in the formal (mathematical) model, typing contexts are represented as sequences and the elements of a typing context are accessed from right to left (the same convention was used also in the mathematical model given in chapter 4). On the other hand, it is natural to use Haskell lists to implement the concept of a typing context. In the Haskell notation, lists are accessed from left to right (this strategy was also used in the Haskell implementation given in chapter 4). However, recall that a typing context Γ is essentially a function from variables to types; see also permutation Lemma, mentioned in exercise 6.17. Hence the order of elements in a list representing a typing context is not important.

```
   show (Tabs (x,tx) t) =
      "(\\" ++ "(" ++ x ++ ":" ++ (show tx) ++ ")" ++ "."
      ++ (show t) ++ ")"
   show (Tapp t1 t2)    = "(" ++ (show t1) ++ " " ++ (show t2) ++ ")"
```

- We implement the classes of typing contexts `Context` and variable names contexts `VnamesContext` as follows:

```
type Context       = [(X,Ty)]
type VnamesContext = [X]

toVnamesCtx :: Context -> VnamesContext
toVnamesCtx ctx = [ x | (x,_) <- ctx ]
```

- The definitions of functions `removenames` and `restorenames` (with the auxiliary mappings `index`, `pickFreshName` and `isBoundName`) are obtained starting from corresponding definitions presented in chapter 4.

```
removenames :: Term -> VnamesContext -> T
removenames TmTrue              xs = Ttrue
removenames TmFalse             xs = Tfalse
removenames (TmIf t1 t2 t3)     xs =
   Tif (removenames t1 xs) (removenames t2 xs) (removenames t3 xs)
removenames (TmVar x)           xs = Tvar (index x xs)
removenames (TmAbs (x,ty1) t2)  xs =
   Tabs (x,ty1) (removenames t2 (x:xs))
removenames (TmApp t1 t2)       xs =
   Tapp (removenames t1 xs) (removenames t2 xs)

restorenames :: T -> VnamesContext -> Term
restorenames (Ttrue)          xs = TmTrue
restorenames (Tfalse)         xs = TmFalse
restorenames (Tif t1 t2 t3)   xs =
   TmIf (restorenames t1 xs) (restorenames t2 xs) (restorenames t3 xs)
restorenames (Tvar k)         xs = TmVar (xs !! k)
restorenames (Tabs (x,ty) t1) xs =
   let x' = pickFreshName xs x
   in  TmAbs (x',ty) (restorenames t1 (x':xs))
restorenames (Tapp t1 t2)     xs =
   TmApp (restorenames t1 xs) (restorenames t2 xs)

index :: (Eq a) => a -> [a] -> Int
index x []      = error "element out of context"
index x (x':xs) =
   if (x' == x) then 0 else 1 + index x xs
```

```
pickFreshName :: VnamesContext -> X -> X
pickFreshName xs x =
   if isBoundName xs x
   then pickFreshName xs (x ++ "'") else x

isBoundName :: VnamesContext -> X -> Bool
isBoundName []      x = False
isBoundName (x':xs) x =
   if (x'==x) then True else isBoundName xs x
```

- We define a Haskell value `t1` of the type `Term` and a value `ctx1` of the type `Context`, based also on two values of the type `Ty`. Also, we define a value `tb1` of the type `T` (a de Bruijn term).

```
t1 :: Term
t1 = TmAbs ("m",ty2) (TmAbs ("n",ty2) (TmAbs ("s",TyBool)
           (TmAbs ("z",TyBool) (TmApp (TmApp (TmVar "m") (TmVar "s"))
                  (TmApp (TmApp (TmVar "n") (TmVar "s")) (TmVar "z"))))))
t4 :: Term
t4 = TmAbs ("w",TyBool) (TmAbs ("a",TyBool) (TmVar "x"))

ctx1 :: Context
ctx1 = reverse [("x",ty1),("y",ty1),("z",TyBool),("a",TyBool),("b",TyBool)]

ty1,ty2 :: Ty
ty1 = TyFun TyBool TyBool
ty2 = TyFun TyBool (TyFun TyBool TyBool)

tb1 :: T
tb1 = removenames t1 []

tb4 :: T
tb4 = removenames t4 (toVnamesCtx ctx1)
```

- One can perform the following experiments:

```
Main> t1
(\(m:(bool->(bool->bool))).(\(n:(bool->(bool->bool))).(\(s:bool).
(\(z:bool).((m s) ((n s) z))))))
Main> tb1
(\(m:(bool->(bool->bool))).(\(n:(bool->(bool->bool))).(\(s:bool).
(\(z:bool).((3 1) ((2 0) 1))))))
Main> t4
(\(w:bool).(\(a:bool).x))
Main> tb4
(\(w:bool).(\(a:bool).6))
```

- Also, let

```
tt :: T -> Context -> T
tt t ctx =
   let xs = toVnamesCtx ctx
   in  removenames (restorenames t xs) xs

termterm :: Term -> Context-> Term
termterm t ctx =
   let xs = toVnamesCtx ctx
   in  restorenames (removenames t xs) xs
```

- The following experiments confirm that the above implementation of `removenames` and `restorenames` satisfy the properies stated in chapter 4:

```
Main> termterm t1 []
(\(m:(bool->(bool->bool))).(\(n:(bool->(bool->bool))).(\(s:bool).
(\(z:bool).((m s) ((n s) z))))))
Main> termterm t4 ctx1
(\(w:bool).(\(a':bool).x))
Main> tt tb1 []
(\(m:(bool->(bool->bool))).(\(n:(bool->(bool->bool))).(\(s:bool).
(\(z:bool).((3 1) ((2 0) 1))))))
Main> tt tb4 ctx1
(\(w:bool).(\(a':bool).6))
```

- Next, we give the formal definition of the typing relation for $\lambda^{\rightarrow}_{\mathbf{B}}$ (Definition 6.7)
- The elements of the typing relation are triples (Γ, t, T), written as $\Gamma \vdash t : T$
  - We write $\Gamma \vdash t : T$ to express that the type of the term t is T, assuming that the types of (free) variables in t are bound by Γ (Remark 6.3)
- Rules (T-True) and (T-False) are obvious. For rule (T-If) see exercise 6.8
- Rules (T-Var), (T-Abs) and (T-App) are specific to the simply typed lambda calculus
  - In rule (T-Abs), the context $\Gamma, x : T_1$ (of the premise) extends the typing context Γ (of the conclusion) with the assumption $x : T_1$ (the type of x is T_1)
    - It is important to note that, by using *alpha-conversion* (convention 3.18) in rule (T-Abs) we can always choose x such that $x \notin dom(\Gamma)$.
      - Therefore, we can assume that the variable names in the domain of a typing context are always pairwise distinct

Remark 6.6 *Let $\Gamma \in Context$, $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, be a typing context with domain $dom(\Gamma) = \{x_1, \ldots, x_n\}$. Essentially, Γ is a used to represent a function (with finite graph) from variable names to types. This means that the variable names $x_1, \ldots, x_n$ (occurring in the list $\Gamma = x_1 : T_1, \ldots, x_n : T_n$) are assumed to be pairwise distinct. In the following, whenever a pair $x : T$ is appended to a context Γ, by default it is considered that $x \notin dom(\Gamma)$. Intuitively, a pair $x : T$ occurring in a context is a* binding *($x : T$ associates type T to variable name x).*

Definition 6.7 *[Typing relation for $\lambda_{\mathbf{B}}^{\rightarrow}$] The* typing relation *for $\lambda_{\mathbf{B}}^{\rightarrow}$ is the smallest subset of $(Context \times \mathcal{T}_{\mathbf{B}}^{\rightarrow} \times \Theta_{\mathbf{B}}^{\rightarrow})$ satisfying the axioms and rules given below. The elements (Γ, t, T) of the typing relation for $\lambda_{\mathbf{B}}^{\rightarrow}$ are written as $\Gamma \vdash t : T$.*

(T-True) $\Gamma \vdash \mathsf{true} : \mathsf{Bool}$

(T-False) $\Gamma \vdash \mathsf{false} : \mathsf{Bool}$

(T-If) $$\frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}$$

(T-Var) $$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

(T-Abs) $$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, t_2 : T_1 \rightarrow T_2}$$

(T-App) $$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

Exercise 6.8 *Explain the difference between rule (T-If) given in definition 6.7, and rule (T-If) given in definition 5.1 from chapter 5. Let $t \in \mathcal{T}_{\mathbf{B}}^{\rightarrow}$, $t = \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$ (for some $t_1, t_2, t_3 \in \mathcal{T}_{\mathbf{B}}^{\rightarrow}$), where the class $\mathcal{T}_{\mathbf{B}}^{\rightarrow}$ is given in definition 6.4. Give an example of a triple $(\Gamma \vdash t : T)$ that can occur as the conclusion of rule (T-If) given in definition 6.7, using a type T which cannot be used in rule (T-If) given in definition 5.1 from chapter 5.*

Exercise 6.9 *Let $t = ((\lambda x : \mathsf{Bool} \,.\, x)\ (\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}))$, $t \in \mathcal{T}_{\mathbf{B}}^{\rightarrow}$. Intuitively, the type of term t is* Bool*. By using the rules given in definition 6.7, build the derivation tree which proves that $\vdash t : \mathsf{Bool}$ (i.e., $\emptyset \vdash t : \mathsf{Bool}$).*

Exercise 6.10 *Solve exercise 9.2.2 from the book [1].*

- The Haskell implementation of the typing relation for $\lambda_{\mathbf{B}}^{\rightarrow}$ is straightforward (according to exercise 6.13, the typing relation for $\lambda_{\mathbf{B}}^{\rightarrow}$ can be implemented as a function):
  - The function `typeof` computes the type of a (nameless) term with respect to a given typing context
  - For handling variables and abstractions we use two auxiliary mappings `addbinding` and `getTypeFromContext`[2]

```
typeOf :: T -> Context -> Ty
typeOf Ttrue ctx = TyBool
typeOf Tfalse ctx = TyBool
typeOf (Tif t1 t2 t3) ctx =
   case (typeOf t1 ctx) of
     TyBool -> let ty2 = typeOf t2 ctx
                   ty3 = typeOf t3 ctx
```

[2] In the mathematical notation (based on [1]) typing contexts are accessed from right to left (in particular, see definition 6.5, and rule (T-Abs) from definition 6.7). On the other hand, it is natural to use Haskell lists to implement the concept of a typing context. In the Haskell notation, lists are accessed from left to right; see also the Haskell implementation given in chapter 4.

```
                in if (ty2 == ty3) then ty2
                   else error "typeOf -> Tif: arms of conditional
                               have different types"
     _      -> error "typeOf -> Tif: non-boolean condition"
typeOf (Tvar k) ctx = getTypeFromContext k ctx
typeOf (Tabs (x,tyT1) t2) ctx =
   let ctx' = addbinding ctx x tyT1
       tyT2 = typeOf t2 ctx'
   in  TyFun tyT1 tyT2
typeOf (Tapp t1 t2) ctx =
   let { tyT1 = typeOf t1 ctx; tyT2 = typeOf t2 ctx }
   in case tyT1 of
           (TyFun tyT11 tyT12) ->
              if (tyT2 == tyT11) then tyT12
              else error "typeOf -> Tapp: parameter type mismatch"
           _ -> error "typeOf -> Tapp:arrow type expected"

addbinding :: Context -> X -> Ty -> Context
addbinding ctx x ty = (x,ty):ctx

getTypeFromContext :: K -> Context -> Ty
getTypeFromContext k ctx = snd (ctx !! k)
```

- In the mathematical model, when a pair $x : T$ is appended to a context Γ, by default it is considered that $x \notin dom(\Gamma)$; see remark 6.6. Therefore, the reader may wonder why we do not choose a new variable name in the Haskell equation that implements the rule for abstraction (T-Abs)
  - Note that `typeOf` takes as argument a *nameless term*, hence it works with de Bruijn indexes (rather than variable names)
- Also, recall that in the nameless term `(Tabs (x,tyT1) t2)` the variable name `x` is only used as a *hint* by function `restorenames` (as explained in remark 4.7)
- The mapping `(getTypeFromContext k ctx)` takes as arguments a de Bruijn index `k` (rather than a variable name) and a context `ctx`. It uses the mapping `!!` defined in the standard Haskell library `Prelude.hs`.[3]
  - It returns the type corresponding to `k` from `ctx`
- The mapping `addbinding` is used to extend the context upon entry in a nested abstraction

[3]The function `!!` is a polymorphic operator (defined in `Prelude.hs`), which returns the element of a list located at a specified index (starting from `0`). It behaves as illustrated in the following examples:

```
Main> [1,2,3,4,5] !!  3
4
Main> [1,2,3,4,5] !!  0
1
```

- For testing purposes we also define the following functions

```
testTerm :: Term -> Context -> Ty
testTerm term ctx = typeOf (removenames term (toVnamesCtx ctx)) ctx

testT :: T -> Context -> Ty
testT t ctx = typeOf t ctx
```

- Let

```
t' :: Term
t' = TmAbs ("x",ty1) (TmAbs ("y",ty1)
                      (TmApp (TmVar "x") (TmApp (TmVar "y") (TmVar "z"))))
ctx' :: Context
ctx' = reverse [("z",TyBool)]
tb' :: T
tb' = removenames t' (toVnamesCtx ctx')
```

- One can perform the following experiments:

```
Main> testTerm t' ctx'
((bool->bool) -> ((bool->bool) -> bool))
Main> testT tb' ctx'
((bool->bool) -> ((bool->bool) -> bool))
```

6.2 Properties of typing

- Recall that (type) safety = progress + preservation (see section 5.3)
- Some preparative results are needed before we establish type safety for $\lambda^{\rightarrow}_{\mathbf{B}}$ [1]
  - The preservation theorem requires a "substitution lemma" (lemma 6.19)
- The "uniqueness of types" property considered in exercise 6.13 can also be established for language $\lambda^{\rightarrow}_{\mathbf{B}}$

Remark 6.11 *Lemma 9.3.1 given in [1] (page 104) for language $\lambda^{\rightarrow}_{\mathbf{B}}$ is hereafter called the "inversion lemma" for $\lambda^{\rightarrow}_{\mathbf{B}}$ (as in [1]). It is a collection of 6 simple facts, which follow easily from the 6 rules that define the typing relation for $\lambda^{\rightarrow}_{\mathbf{B}}$ (definition 6.7). For example, consider rule (T-Var) given in definition 6.7:*

$$\textit{(T-Var)}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

In this case the "inversion lemma" states the following fact:

- *If* $\Gamma \vdash x : T$ *then* $x : T \in \Gamma$

This is an obvious consequence of the fact that (T-Var) is the single *rule which can be used to infer that* $\Gamma \vdash x : T$.

The clauses of the "inversion lemma" that are implied by rules (T-Abs) and (TApp) are also easily established. By extending each pair $t : T$ to a corresponding triple $\Gamma \vdash t : T$, the last three properties stated in the final part of Remark 5.10 (for language **NB***) can be adapted to the language $\lambda_{\mathbf{B}}^{\rightarrow}$ (introduced in Definition 6.4). For example, in the case of $\lambda_{\mathbf{B}}^{\rightarrow}$ we have: if $\Gamma \vdash \mathsf{true} : T$ then $T = \mathsf{Bool}$. The reader can easily adapt the other two properties given in the final part of Remark 5.10. The "inversion lemma" may be useful in various contexts, in particular it can be used in exercise 6.13.*

Exercise 6.12 *Enunciate and explain the clauses of the "inversion lemma" 9.3.1 (given in [1]), that are implied by rules (T-Abs), (T-App) and (T-Iff) given in definition 6.7.*

In general, several types may be attributed to the same term (e.g., in the presence of subtyping; see [1], part III). However, for the languages **NB** and $\lambda_{\mathbf{B}}^{\rightarrow}$ one can establish the properties (regarding the uniqueness of types) given in exercise 5.3 and exercise 6.13. According to exercise 6.13, the typing relation for $\lambda_{\mathbf{B}}^{\rightarrow}$ is a function (the type inferred for any $\lambda_{\mathbf{B}}^{\rightarrow}$ term t is unique).

Exercise 6.13 *[Uniqueness of types for $\lambda_{\mathbf{B}}^{\rightarrow}$] Let t be an $\lambda_{\mathbf{B}}^{\rightarrow}$ term. Let $\Gamma \in Context, \Gamma = x_1 : T_1, \ldots, x_n : T_n$ be a typing context such that $FV(t) \subseteq dom(\Gamma)$, where $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and $FV(t)$ is the set of free variables occurring in term t (see definition 6.4).*[4] *Proceeding by induction on the depth of a derivation of $\Gamma \vdash t : T$, prove the following:*

(a) If $\Gamma \vdash t : T$ and $\Gamma \vdash t : T'$ (where $T, T' \in \Theta_{\mathbf{B}}^{\rightarrow}$, i.e., T and T' are $\lambda_{\mathbf{B}}^{\rightarrow}$ types) then $T' = T$

(b) The derivation tree (inference tree) proving that $\Gamma \vdash t : T$ (by using the rules given in definition 6.7) is also unique.

As explained in Remark 6.6, you can assume that the variable names $x_1, \ldots, x_n$ occurring in the list representing the context Γ ($\Gamma = x_1 : T_1, \ldots, x_n : T_n$) are pairwise distinct.

- A $\lambda_{\mathbf{B}}^{\rightarrow}$ term t is well-typed w.r.t. some typing context Γ iff there exists a type T and an inference tree (derivation tree) proving that $\Gamma \vdash t : T$. Consider these two classes: inference (derivation) trees and well typed terms.
  - According to exercise 6.13, in the context of $\lambda_{\mathbf{B}}^{\rightarrow}$ the relation (correspondence) between these two classes is of type one-to-one.
- We recall that lambda abstractions are the only values in the pure lambda calculus. In addition, the class of values of $\lambda_{\mathbf{B}}^{\rightarrow}$ includes the values true and false.

Definition 6.14 *[Values in $\lambda_{\mathbf{B}}^{\rightarrow}$]* $\quad v ::= \mathsf{true} \mid \mathsf{false} \mid \lambda x : T \,.\, t$

- A progress theorem can be stated for $\lambda_{\mathbf{B}}^{\rightarrow}$. It is similar to theorem 5.5, given in chapter 5 for language **B**. However, notice that in the context of $\lambda_{\mathbf{B}}^{\rightarrow}$ this property (theorem 6.15) can only be established for *closed terms*.[5]

[4] All free variables of t are in the domain of Γ.

[5] We recall that the notion of a *closed term* is used to denote any lambda term containing no free variables. Variable fun occurring in term $(fun\ (\lambda x \,.\, x))$ is free. Hence $(fun\ (\lambda x \,.\, x))$ is an *open term* (it is not closed).

  - There are *open terms*, like $(fun\ (\lambda x\,.\,x))$, which cannot be reduced any longer (i.e., are normal forms),[6] but are not values ($(fun\ (\lambda x\,.\,x))$ is not a value).

- Let t be a $\lambda^{\rightarrow}_{\mathbf{B}}$ term ($t \in \mathcal{T}^{\rightarrow}_{\mathbf{B}}$). Let T be a $\lambda^{\rightarrow}_{\mathbf{B}}$ type ($T \in \Theta^{\rightarrow}_{\mathbf{B}}$). We recall that $\vdash t : T$ is an abbreviation for $\emptyset \vdash t : T$, where $\emptyset$ is the empty context. The notation $\vdash t : T$ expresses that term t has type T w.r.t. the empty context, i.e., t is a closed and well typed $\lambda^{\rightarrow}_{\mathbf{B}}$ term.

Theorem 6.15 *[Progress] If t is a closed and well-typed $\lambda^{\rightarrow}_{\mathbf{B}}$ term (i.e., there exists some type T such that $\vdash t : T$) then either t is a value or else there is some t' such that $t \rightarrow t'$.*

Study Assignment 6.16 *The proof of theorem 6.15 (progress theorem for $\lambda^{\rightarrow}_{\mathbf{B}}$) can proceed by induction on typing derivations (the proof is provided in [1], page 105).*[7]*. Follow carefully the steps of the proof.*

Exercise 6.17 *Proceeding by induction on typing derivations*

(a) Prove lemma 9.3.6 given in [1] (permutation lemma)

(b) Prove lemma 9.3.7 given in [1] (weakening lemma)

These properties (lemmas) may also be used in the proof of "substitution lemma" 6.19 [1]. For the purpose of this exercise, we recall that all bindings in a typing context must have distinct variable names, i.e., any typing context Γ is a list $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ such that the variable names $x_1, \ldots, x_n$ are pairwise distinct.

- Before we present the "substitution lemma" 6.19 we must adapt the definition of substitution for $\lambda^{\rightarrow}_{\mathbf{B}}$ (starting from definition 3.20).

Definition 6.18 *Substitution over (alpha-equivalence classes) of $\lambda^{\rightarrow}_{\mathbf{B}}$ terms is given by:*

$$
\begin{array}{rcll}
[x \mapsto s]\mathsf{true} & = & \mathsf{true} & \\
[x \mapsto s]\mathsf{false} & = & \mathsf{false} & \\
[x \mapsto s]\,\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{then}\ t_3 & = & \mathsf{if}\ [x \mapsto s]t_1\ \mathsf{then}\ [x \mapsto s]t_2\ \mathsf{else}\ [x \mapsto s]t_3 & \\
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \mathsf{if}\ y \neq x \\
[x \mapsto s](\lambda y : T\,.\,t_1) & = & \lambda y : T\,.\,([x \mapsto s]t_1) & \mathsf{if}\ y \neq x, y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

The mapping $FV(t)$ is given in definition 6.4.

- "Substitution lemma" 6.19 (see [1], lemma 9.3.8) essentially states that the substitution operation (given above, in definition 6.18) maintains types. The type preservation property for $\lambda^{\rightarrow}_{\mathbf{B}}$ (theorem 6.21) can be established by using lemma 6.19.

Lemma 6.19 *If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.*

Study Assignment 6.20 *A detailed proof of lemma 6.19 is provided in [1] (proof of lemma 9.3.8., pages 106—107). Follow carefully the steps of the proof.*

[6] We recall that a term t is a *normal form* if $\neg(\exists t' : t \rightarrow t')$, i.e., no reduction step is possible for t.

[7] The proof uses the "canonical forms" lemma 9.3.4 given in [1], which states that any value v having type Bool is either $v = \mathsf{true}$ or $v = \mathsf{false}$, and any value of a type $T_1 \rightarrow T_2$ is a lambda abstraction $\lambda x : T_1\,.\,t$.

Theorem 6.21 *[Preservation] If* $\Gamma \vdash t : T$ *and* $t \to t'$*, then* $\Gamma \vdash t' : T$*.*

Exercise 6.22 *The proof of preservation theorem 6.21 is provided in [1] (see the proof of theorem 9.3.9, given in appendix A, pages 506–507). The proof uses "substitution lemma" 6.19, and proceeds by induction on the depth of a derivation (inference tree) for* $\Gamma \vdash t : T$*, and by case analysis on the final rule used in the type inference tree. Follow carefully the steps of the proof, and complete the proof for the cases that are not handled in [1], namely cases (T-App) with subcase (E-App2), (T-True), (T-False) and (T-If).*

6.3 Evaluation

- The language $\lambda^{\to}_{\mathbf{B}}$ (simply typed lambda calculus over Bool) is given in definition 6.4.

Definition 6.23 *[Evaluation relation specification for* $\lambda^{\to}_{\mathbf{B}}$ *(call-by-value)]*

(E-IfTrue) $\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \to t_2$

(E-IfFalse) $\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \to t_3$

(E-If) $$\frac{t_1 \to t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \to\ \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}$$

(E-App1) $$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$$

(E-App2) $$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}$$

(E-AppAbs) $(\lambda x : T_{11}\,.\,t_{12})\ v_2 \to [x \mapsto v_2]t_{12}$

Exercise 6.24 *The erasure function (erase) presented in [2], definition 9.5.1, maps* pure simply typed *lambda terms (given by the syntax:* $t ::= x \mid \lambda x : T\,.\,t \mid t\ t$*; see [2], figure 9-1) to* pure untyped *lambda terms (given by the syntax:* $t ::= x \mid \lambda x\,.\,t \mid t\ t$*; see [2], figure 5-3, and definition 3.2 of the document, where this language is named* λ*).*

In this exercise you will define a function $erase_{\mathbf{B}}$*, which extends the erasure operation given in [1] (definition 9.5.1) to handle both lambda terms and values of the basic type* Bool*. For this purpose we consider the language* $\lambda_{\mathbf{B}}$ *given in BNF by the following syntax:* $t ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid x \mid \lambda x\,.\,t \mid t\ t$*.* $\lambda_{\mathbf{B}}$ *extends the the language of pure untyped lambda terms (i.e., language* λ *given in definition 3.2, and [2], figure 5-3) with values of the basic type* Bool*, as used in the language* **B** *given in definition 2.14.*

(a) Define an evaluation relation ($\to$*) for* $\lambda_{\mathbf{B}}$*. For this purpose you should combine the rules given in definitions 2.15 and 3.16, which specify the evaluation relations for* **B** *and* λ*, respectively.*

(b) Let $\lambda^{\to}_{\mathbf{B}}$ *be the simply typed lambda calculus over* Bool*, given in definition 6.4.*[8] *Define a function* $erase_{\mathbf{B}}$*, mapping* $\lambda^{\to}_{\mathbf{B}}$ *(simply typed) terms to corresponding* $\lambda_{\mathbf{B}}$ *(untyped) terms. For example,* $erase_{\mathbf{B}}((\lambda x : \mathsf{Bool}\,.\,x)\ \mathsf{true}) = (\lambda x\,.\,x)\ \mathsf{true}$*.*

[8] The evaluation relation for $\lambda^{\to}_{\mathbf{B}}$ is given in definition 6.23.

(c) Intuitively, evaluation should not be influencd by such an erasure operation, because type information is not used during evaluation. This intuition is expressed as follows in [1]: "evaluation commutes with erasure". Read theorem 9.5.2 given in [1], which formalizes this intuition for the erasure mapping (erase) given in [1], definition 9.5.1. Next, formulate and prove a similar theorem for the mapping $erase_{\mathbf{B}}$.

6.3.1 A Haskell implementation of the evaluation relation

- The rules given in definiton 6.23 are well suited for theoretical purposes
- We design a Haskell implementation of $\lambda^{\rightarrow}_{\mathbf{B}}$
  - Following the approach presented in chapter 4, the implementation works with *nameless terms*
  - We only provide the Haskell implementation of the relevant concepts
    - The interested reader can easily recover the mathematical definitions
- We adopt a more terse style of the presentation
  - The Haskell functions given below are an easy adaptation of corresponding functions presented in chapter 4
- The functions `tSubst` and `tShift` implement the substitution and shifting operations, respectively, for $\lambda^{\rightarrow}_{\mathbf{B}}$ (simply typed lambda calculus with base type Bool)
  - The corresponding definitions for the untyped calculus λ are given in section 4.2

```
tSubst :: K -> T -> T -> T
tSubst j s Ttrue            = Ttrue
tSubst j s Tfalse           = Tfalse
tSubst j s (Tif t1 t2 t3)   =
   Tif (tSubst j s t1) (tSubst j s t2) (tSubst j s t3)
tSubst j s (Tvar k)         =
   if (k == j) then s else (Tvar k)
tSubst j s (Tabs (x,ty) t1) =
   Tabs (x,ty) (tSubst (j+1) (tShift 1 s) t1)
tSubst j s (Tapp t1 t2)     =
   Tapp (tSubst j s t1) (tSubst j s t2)

tShift :: Int -> T -> T
tShift d t = aux 0 t
   where aux :: Int -> T -> T
         aux c Ttrue            = Ttrue
         aux c Tfalse           = Tfalse
         aux c (Tif t1 t2 t3)   = Tif (aux c t1) (aux c t2) (aux c t3)
         aux c (Tvar k)         = if k>= c then Tvar (k+d) else Tvar k
         aux c (Tabs (x,ty) t1) = Tabs (x,ty) (aux (c+1) t1)
         aux c (Tapp t1 t2)     = Tapp (aux c t1) (aux c t2)
```

- For example, the value `tb'::T` was introduced previously in section 6.1. One can perform the following experiments:

```
Main> tb'
(\(x:(bool->bool)).(\(y:(bool->bool)).(1 (0 2))))
Main> tShift 100 tb'
(\(x:(bool->bool)).(\(y:(bool->bool)).(1 (0 102))))
```

- Let also

```
t100, t101 ::  Term
t100 = TmVar "a"
t101 =
   TmApp (TmVar "b")
         (TmApp (TmAbs ("x",TyBool)
                (TmApp (TmVar "b") (TmVar "x")))
                (TmVar "y"))
tb100 = removenames t100 (toVnamesCtx ctx100)
tb101 = removenames t101 (ToVnamesCtx ctx100)

ctx100 :: Context
ctx100 = reverse [("a",ty1),("b",ty1),("y",TyBool)]

termSubst :: Int -> Term -> Term -> Context -> Term
termSubst j s t ctx =
   let xs = toVnamesCtx ctx
       s' = removenames s xs
       t' = removenames t xs
   in restorenames (tSubst j s' t') xs
```

- You can test these functions as follows:

```
Main> tb100
2
Main> tb101
(1 ((\(x:bool).(2 0)) 0))
Main> tShift 100 tb101
(101 ((\(x:bool).(102 0)) 100))
Main> tSubst 1 tb101 tb100
2
Main> tSubst 1 tb100 tb101
(2 ((\(x:bool).(3 0)) 0))
Main> termSubst 1 t101 t100 ctx100
a
Main> termSubst 1 t100 t101 ctx100
(a ((\(x:bool).(a x)) y))
```

- Finally, we can implement the evaluation function `eval1`.
  - Contexts are not needed here (are maintained for compatibility with models presented in [1], where various extensions are investigated)

  - In $\lambda^{\rightarrow}_{\mathbf{B}}$, only booleans and abstractions are values.

- Functions `eval1` and `evalT` are designed starting from corresponding functions presented in chapter 4.

```
isval ::  T -> Context -> Bool
isval Ttrue      ctx = True
isval Tfalse     ctx = True
isval (Tabs _ _) ctx = True
isval _          ctx = False

eval1 :: T -> Context -> [T]
eval1 (Tif Ttrue t2 t3)              ctx = [t2]
eval1 (Tif Tfalse t2 t3)             ctx = [t3]
eval1 (Tif t1 t2 t3)                 ctx =
      [ Tif t1' t2 t3 | t1' <- eval1 t1 ctx ]
eval1 (Tapp t1 t2)                   ctx
   | (Tabs (x,_) t12) <- t1, isval t2 ctx =
      [ tShift (-1) (tSubst 0 (tShift 1 t2) t12) ]
   | isval t1 ctx                        =
      [ Tapp t1 t2' | t2' <- eval1 t2 ctx ]
   | otherwise                           =
      [ Tapp t1' t2 | t1' <- eval1 t1 ctx ]
eval1 (Tabs _ _)                     ctx = []
eval1 (Ttrue)                        ctx = []
eval1 (Tfalse)                       ctx = []
eval1 (Tvar _)                       ctx = []

evalT :: T -> Context -> T
evalT t ctx =
   case eval1 t ctx of
        [] -> t
        [t'] -> evalT t' ctx
        _    -> error "Nondeterministic evaluation (impossible!)"

evalTerm :: Term -> Context -> Term
evalTerm term ctx =
   let xs = toVnamesCtx ctx
   in  restorenames (evalT (removenames term xs) ctx) xs
```

- Let

```
xapply, xnot :: Term
xapply = TmAbs ("f",TyFun TyBool TyBool)
               (TmAbs ("x",TyBool)
                      (TmApp (TmVar "f") (TmVar "x")))
xnot = TmAbs ("b",TyBool) (TmIf (TmVar "b") TmFalse TmTrue)
```

- `xapply` and `xnot` implement the $\lambda^{\rightarrow}_{\mathbf{B}}$ terms $\lambda f : \mathsf{Bool} \rightarrow \mathsf{Bool}\,.\,(\lambda x : \mathsf{Bool}\,.\,(f\ x))$, and $\lambda b : \mathsf{Bool}\,.\,(\,\mathsf{if}\ b\ \mathsf{then\ false\ else\ true})$, respectively. You can perform the following experiments:

```
Main> evalTerm (TmApp xnot TmTrue) []
false
Main> evalTerm (TmApp xnot TmFalse) []
true
Main> evalTerm (TmApp (TmApp xapply xnot) TmTrue) []
false
Main> evalTerm (TmApp (TmApp xapply xnot) TmFalse) []
true
```

- Let also

```
tyT4 :: Ty
tyT4 = TyFun (TyFun TyBool TyBool) TyBool

ctx1000 :: Context
ctx1000 = reverse [("y",tyT4),("z",TyBool)]

t1000 :: Term
t1000 = TmApp (TmAbs ("x",TyFun TyBool TyBool)
                 (TmApp (TmVar "y") (TmVar "x")))
                    (TmAbs ("u",TyBool) (TmVar "u"))
```

- One can perform the following experiments

```
Main> t1000
((\(x:(bool->bool)).(y x)) (\(u:bool).u))
Main> removenames t1000 (toVnamesCtx ctx1000)
((\(x:(bool->bool)).(2 0)) (\(u:bool).0))
Main> typeOf (removenames t1000 (toVnamesCtx ctx1000)) ctx1000
bool
Main> evalT (removenames t1000 (toVnamesCtx ctx1000)) ctx1000
(1 (\(u:bool).0)
Main> evalTerm t1000 ctx1000
(y (\(u:bool).u))
```

Appendix A

A Haskell Implementation of Simply Typed Lambda-calculus Extended with Simple Features and References

```
type X             = String
type K             = Int
type VnamesContext = [X]
type Context       = [(X,Ty)]
type Label         = String
type Location      = Int

type Store       = [T]
type StoreTyping = [Ty]

data L a = L Label a deriving Eq

data Ty = TyUnit | TyNat | TyBool | TyFun Ty Ty
        | TyTuple [Ty]
        | TyRecord [L Ty]
        | TyVariant [L Ty]
        | TyList Ty
        | TyRef Ty
          deriving Eq

instance Show Ty where
   show TyBool = "bool"
   show TyNat  = "nat"
   show TyUnit = "unit"
   show (TyFun ty1 ty2) = "(" ++ (show ty1) ++ " -> " ++ (show ty2) ++ ")"
   show (TyTuple tys) = "{" ++ (showlist tys) ++ "}"
   show (TyRecord ltys) = "{" ++ (showlist ltys) ++ "}"
   show (TyVariant ltys) = "<" ++ (showlist ltys) ++ ">"
```

```
   show (TyList ty) = "(" ++ "List " ++ (show ty) ++ ")"
   show (TyRef ty) = "(" ++ "Ref " ++ (show ty) ++ ")"

data Term = TmUnit | TmTrue | TmFalse
   | TmZero | TmSucc Term | TmPred Term
   | TmIsZero Term
   | TmNot Term | TmAnd Term Term | TmIf Term Term Term
   | TmVar X  | TmAbs (X,Ty) Term | TmApp Term Term
   | TmSeq Term Term | TmLet X Term Term
   | TmFix Term | TmLetrec (X,Ty) Term Term
   | TmTuple [Term] | TmTupleProj Term Int
   | TmRecord [L Term] | TmRecordProj Term Label
   | TmVariantTag (L Term) Ty
   | TmVariantCase Term [L (VariantCase Term)]
   | TmNil Ty | TmCons Ty Term Term | TmIsNil Ty Term
   | TmHead Ty Term | TmTail Ty Term
   | TmRef Term | TmDeref Term | TmAssign Term Term
   | TmLoc Location
   | TmToUnit Term

data VariantCase a = VariantCase X a

instance (Show a) => Show (VariantCase a) where
   show (VariantCase x t) = (show x) ++ ">" ++ "=>" ++ (show t)

instance (Show a) => Show (L a) where
   show (L l a) = l ++ "=" ++ (show a)

instance Show Term where
   show TmUnit    = "unit"
   show TmTrue    = "true"
   show TmFalse   = "false"
   show TmZero    = "0"
   show (TmSucc t) = "(succ " ++ (show t) ++ ")"
   show (TmPred t) = "(pred " ++ (show t) ++ ")"
   show (TmIsZero t) = "(iszero " ++ (show t) ++ ")"
   show (TmNot t) = "(not " ++ (show t) ++ ")"
   show (TmAnd t1 t2) = "(" ++ (show t1) ++ " and " ++ (show t2) ++ ")"
   show (TmIf t1 t2 t3) = "(if " ++ (show t1) ++ " then " ++ (show t2) ++
                          " else " ++ (show t3) ++ ")"
   show (TmVar x) = x
   show (TmAbs (x,ty) t) = "(\\" ++ "(" ++ x ++ ":" ++ (show ty) ++ ")" ++
                           "." ++ (show t) ++ ")"
   show (TmApp t1 t2) = "(" ++ (show t1) ++ " " ++ (show t2) ++ ")"
   show (TmSeq t1 t2) = "(" ++ (show t1) ++ ";" ++ (show t2) ++ ")"
   show (TmLet x t1 t2) = "(let " ++ x ++ "=" ++ (show t1) ++
                          " in " ++ (show t2) ++ ")"
   show (TmFix t) = "(fix " ++ (show t) ++ ")"
```

```
   show (TmLetrec (x,ty) t1 t2) = "(letrec " ++ x ++ ":" ++ (show ty) ++
                                  " = " ++ (show t1)
                                  ++ " in " ++ (show t2) ++ ")"
   show (TmTuple ts) = "{" ++ (showlist ts) ++ "}"
   show (TmTupleProj t i) = "(" ++ (show t) ++ "." ++ (show i) ++ ")"
   show (TmRecord lts) = "{" ++ (showlist lts) ++ "}"
   show (TmRecordProj t l) = "(" ++ (show t) ++ "." ++ l ++ ")"
   show (TmVariantTag (L l t) ty) = "<" ++ l ++ "=" ++ (show t) ++ ">" ++
                                    " as " ++ (show ty)
   show (TmVariantCase t lxts) = "case " ++ "(" ++ (show t) ++ ")" ++
                                 " of " ++ "<" ++ (showlist lxts) ++ ">"
   show (TmNil ty) = "(nil" ++ "[" ++ (show ty) ++ "])"
   show (TmCons ty t1 t2) =
      "(cons" ++ "[" ++ (show ty) ++ "]" ++ " " ++ (show t1) ++
      " " ++ (show t2) ++ ")"
   show (TmIsNil ty t) =
      "(isnil" ++ "[" ++ (show ty) ++ "]" ++ " " ++ (show t) ++ ")"
   show (TmHead ty t) =
      "(head" ++ "[" ++ (show ty) ++ "]" ++ " " ++ (show t) ++ ")"
   show (TmTail ty t) =
      "(tail" ++ "[" ++ (show ty) ++ "]" ++ " " ++ (show t) ++ ")"
   show (TmRef t) =
      "(ref" ++ " " ++ (show t) ++ ")"
   show (TmDeref t) =
      "(!" ++ " " ++ (show t) ++ ")"
   show (TmAssign t1 t2) =
      "(" ++ (show t1) ++ " := " ++ (show t2) ++ ")"
   show (TmLoc l) =
      "(location" ++ " " ++ (show l) ++ ")"
   show (TmToUnit t) =
      "(tounit" ++ " " ++ (show t) ++ ")"

{- Nameless terms -}
data T = Tunit | Ttrue | Tfalse | Tzero | Tsucc T | Tpred T | Tiszero T
   | Tnot T | Tand T T | Tif T T T
   | Tvar K | Tabs (X,Ty) T | Tapp T T | Tseq T T | Tlet X T T
   | Tfix T
   | Ttuple [T] | Ttupleproj T Int
   | Trecord [L T] | Trecordproj T Label
   | Tvarianttag (L T) Ty | Tvariantcase T [L (VariantCase T)]
   | Tnil Ty | Tcons Ty T T
   | Tisnil Ty T | Thead Ty T | Ttail Ty T
   | Tref T | Tderef T | Tassign T T | Tloc Location
   | Ttounit T

instance Show T where
   show Tunit    = "unit"
   show Ttrue    = "true"
```

```
   show Tfalse   = "false"
   show Tzero    = "0"
   show (Tsucc t) = "(succ " ++ (show t) ++ ")"
   show (Tpred t) = "(pred " ++ (show t) ++ ")"
   show (Tiszero t) = "(iszero " ++ (show t) ++ ")"
   show (Tnot t) = "(not " ++ (show t) ++ ")"
   show (Tand t1 t2) = "(" ++ (show t1) ++ " and " ++ (show t2) ++ ")"
   show (Tif t1 t2 t3) = "(if " ++ (show t1) ++ " then " ++ (show t2) ++
                         " else " ++ (show t3) ++ ")"
   show (Tvar k)     = (show k)
   show (Tabs (x,ty) t)  = "(\\" ++ "(" ++ x ++ ":" ++ (show ty) ++ ")" ++
                           "." ++ (show t) ++ ")"
   show (Tapp t1 t2) = "(" ++ (show t1) ++ " " ++ (show t2) ++ ")"
   show (Tseq t1 t2) = "(" ++ (show t1) ++ ";" ++ (show t2) ++ ")"
   show (Tlet x t1 t2) = "(let " ++ x ++ "=" ++ (show t1) ++
                         " in " ++ (show t2) ++ ")"
   show (Tfix t) = "(fix " ++ (show t) ++ ")"
   show (Ttuple ts) = "{" ++ (showlist ts) ++ "}"
   show (Ttupleproj t i) = "(" ++ (show t) ++ "." ++ (show i) ++ ")"
   show (Trecord lts) = "{" ++ (showlist lts) ++ "}"
   show (Trecordproj t l) = "(" ++ (show t) ++ "." ++ l ++ ")"
   show (Tvarianttag (L l t) ty) = "<" ++ l ++ "=" ++ (show t) ++ ">" ++
                                   " as " ++ (show ty)
   show (Tvariantcase t lxts) = "case " ++ "(" ++ (show t) ++ ")" ++
                                " of " ++ "<" ++ (showlist lxts) ++ ">"
   show (Tnil ty) = "(nil" ++ "[" ++ (show ty) ++ "])"
   show (Tcons ty t1 t2) =
      "(cons" ++ "[" ++ (show ty) ++ "]" ++ " " ++
      (show t1) ++ " " ++ (show t2) ++ ")"
   show (Tisnil ty t) =
      "(isnil" ++ "[" ++ (show ty) ++ "]" ++ " " ++ (show t) ++ ")"
   show (Thead ty t) =
      "(head" ++ "[" ++ (show ty) ++ "]" ++ " " ++ (show t) ++ ")"
   show (Ttail ty t) =
      "(tail" ++ "[" ++ (show ty) ++ "]" ++ " " ++ (show t) ++ ")"
   show (Tref t) =
      "(ref" ++ " " ++ (show t) ++ ")"
   show (Tderef t) =
      "(!" ++ " " ++ (show t) ++ ")"
   show (Tassign t1 t2) =
      "(" ++ (show t1) ++ " := " ++ (show t2) ++ ")"
   show (Tloc l) =
      "(location" ++ " " ++ (show l) ++ ")"
   show (Ttounit t) =
      "(tounit" ++ " " ++ (show t) ++ ")"

showlist :: (Show a) => [a] -> String
showlist [] = ""
```

```
showlist [x] = show x
showlist (x:xs) = (show x) ++ "," ++ (showlist xs)

index :: (Eq a) => a -> [a] -> Int
index x []      = error "element out of context"
index x (x':xs) = if (x' == x) then 0 else 1 + index x xs

removenames :: Term -> VnamesContext -> T
removenames TmUnit    xs = Tunit
removenames TmTrue    xs = Ttrue
removenames TmFalse   xs = Tfalse
removenames TmZero    xs = Tzero
removenames (TmSucc t) xs = Tsucc (removenames t xs)
removenames (TmPred t) xs = Tpred (removenames t xs)
removenames (TmIsZero t) xs = Tiszero (removenames t xs)
removenames (TmNot t) xs = Tnot (removenames t xs)
removenames (TmAnd t1 t2) xs = Tand (removenames t1 xs) (removenames t2 xs)
removenames (TmIf t1 t2 t3) xs =
   Tif (removenames t1 xs) (removenames t2 xs) (removenames t3 xs)
removenames (TmVar x)     xs = Tvar (index x xs)
removenames (TmAbs (x,ty1) t2)  xs =
   Tabs (x,ty1) (removenames t2 (x:xs))
removenames (TmApp t1 t2) xs =
   Tapp (removenames t1 xs) (removenames t2 xs)
removenames (TmSeq t1 t2) xs =
   Tseq (removenames t1 xs) (removenames t2 xs)
removenames (TmLet x t1 t2) xs =
   let t1' = removenames t1 xs
   in Tlet x t1' (removenames t2 (x:xs))
removenames (TmFix t) xs = Tfix (removenames t xs)
removenames (TmLetrec (x,ty) t1 t2) xs =
   removenames (TmLet x (TmFix (TmAbs (x,ty) t1)) t2) xs
removenames (TmTuple ts) xs =
   Ttuple [ removenames t xs | t <- ts ]
removenames (TmTupleProj t i) xs =
   Ttupleproj (removenames t xs) i
removenames (TmRecord lts) xs =
   Trecord [ L l (removenames t xs) | L l t <- lts ]
removenames (TmRecordProj t i) xs =
   Trecordproj (removenames t xs) i
removenames (TmVariantTag (L l t) ty) xs =
   Tvarianttag (L l (removenames t xs)) ty
removenames (TmVariantCase t0 lxts)  xs =
   Tvariantcase (removenames t0 xs)
                [ L l (VariantCase x (removenames t (x:xs)))
                | L l (VariantCase x t) <- lxts ]
removenames (TmNil ty) xs = Tnil ty
removenames (TmCons ty t1 t2) xs =
```

```
   Tcons ty (removenames t1 xs) (removenames t2 xs)
removenames (TmIsNil ty t) xs =
   Tisnil ty (removenames t xs)
removenames (TmHead ty t) xs =
   Thead ty (removenames t xs)
removenames (TmTail ty t) xs =
   Ttail ty (removenames t xs)
removenames (TmRef t) xs =
   Tref (removenames t xs)
removenames (TmDeref t) xs =
   Tderef (removenames t xs)
removenames (TmAssign t1 t2) xs =
   Tassign (removenames t1 xs) (removenames t2 xs)
removenames (TmLoc l) xs = Tloc l
removenames (TmToUnit t) xs =
   Ttounit (removenames t xs)

pickFreshName :: VnamesContext -> X -> X
pickFreshName xs x =
   if isBoundName xs x then pickFreshName xs (x ++ "'")
   else x

isBoundName :: VnamesContext -> X -> Bool
isBoundName []      x = False
isBoundName (x':xs) x = if (x'==x) then True else isBoundName xs x

restorenames :: T -> VnamesContext -> Term
restorenames (Tunit)        xs = TmUnit
restorenames (Ttrue)        xs = TmTrue
restorenames (Tfalse)       xs = TmFalse
restorenames (Tzero)        xs = TmZero
restorenames (Tsucc t)      xs = TmSucc (restorenames t xs)
restorenames (Tpred t)      xs = TmPred (restorenames t xs)
restorenames (Tiszero t)    xs = TmIsZero (restorenames t xs)
restorenames (Tnot t)       xs = TmNot (restorenames t xs)
restorenames (Tand t1 t2)   xs =
   TmAnd (restorenames t1 xs) (restorenames t2 xs)
restorenames (Tif t1 t2 t3) xs =
   TmIf (restorenames t1 xs) (restorenames t2 xs) (restorenames t3 xs)
restorenames (Tvar k)     xs   = TmVar (xs !! k)
restorenames (Tabs (x,ty) t1) xs =
   let x' = pickFreshName xs x
   in  TmAbs (x',ty) (restorenames t1 (x':xs))
restorenames (Tapp t1 t2) xs =
   TmApp (restorenames t1 xs) (restorenames t2 xs)
restorenames (Tseq t1 t2) xs =
   TmSeq (restorenames t1 xs) (restorenames t2 xs)
restorenames (Tlet x t1 t2) xs =
```

```
   let x' = pickFreshName xs x
   in  TmLet x' (restorenames t1 xs) (restorenames t2 (x':xs))
restorenames (Tfix t) xs = TmFix (restorenames t xs)
restorenames (Ttuple ts) xs = TmTuple [ restorenames t xs | t <- ts ]
restorenames (Ttupleproj t i) xs = TmTupleProj (restorenames t xs) i
restorenames (Trecord lts) xs =
   TmRecord [ L l (restorenames t xs) | L l t <- lts ]
restorenames (Trecordproj t l) xs =
   TmRecordProj (restorenames t xs) l
restorenames (Tvarianttag (L l t) ty) xs =
   TmVariantTag (L l (restorenames t xs)) ty
restorenames (Tvariantcase t0 lxts) xs =
   TmVariantCase (restorenames t0 xs)
                 [ let x' = pickFreshName xs x
                   in  L l (VariantCase x' (restorenames t (x':xs)))
                 | L l (VariantCase x t) <- lxts ]
restorenames (Tnil ty) xs = TmNil ty
restorenames (Tcons ty t1 t2) xs =
   TmCons ty (restorenames t1 xs) (restorenames t2 xs)
restorenames (Tisnil ty t) xs =
   TmIsNil ty (restorenames t xs)
restorenames (Thead ty t) xs =
   TmHead ty (restorenames t xs)
restorenames (Ttail ty t) xs =
   TmTail ty (restorenames t xs)
restorenames (Tref t) xs =
   TmRef (restorenames t xs)
restorenames (Tderef t) xs =
   TmDeref (restorenames t xs)
restorenames (Tassign t1 t2) xs =
   TmAssign (restorenames t1 xs) (restorenames t2 xs)
restorenames (Tloc l) xs = TmLoc l
restorenames (Ttounit t) xs =
   TmToUnit (restorenames t xs)

toVnamesCtx :: Context -> VnamesContext
toVnamesCtx ctx = [ x | (x,_) <- ctx ]

tt :: T -> Context -> T
tt t ctx =
   let xs = toVnamesCtx ctx
   in  removenames (restorenames t xs) xs

termterm :: Term -> Context-> Term
termterm t ctx =
   let xs = toVnamesCtx ctx
   in  restorenames (removenames t xs) xs
```

```
-- Increments the de Bruijn indices of the free variables in a term
tShift :: Int -> T -> T
tShift d t = aux 0 t
   where aux :: Int -> T -> T
         aux c Tunit          = Tunit
         aux c Ttrue          = Ttrue
         aux c Tfalse         = Tfalse
         aux c Tzero          = Tzero
         aux c (Tsucc t)      = Tsucc (aux c t)
         aux c (Tpred t)      = Tpred (aux c t)
         aux c (Tiszero t)    = Tiszero (aux c t)
         aux c (Tnot t1)      = Tnot (aux c t1)
         aux c (Tand t1 t2)   = Tand (aux c t1) (aux c t2)
         aux c (Tif t1 t2 t3) = Tif (aux c t1) (aux c t2) (aux c t3)
         aux c (Tvar k)       = if k>= c then Tvar (k+d) else Tvar k
         aux c (Tabs (x,ty1) t2) = Tabs (x,ty1) (aux (c+1) t2)
         aux c (Tapp t1 t2)   = Tapp (aux c t1) (aux c t2)
         aux c (Tseq t1 t2)   = Tseq (aux c t1) (aux c t2)
         aux c (Tlet x t1 t2) = Tlet x (aux c t1) (aux (c+1) t2)
         aux c (Tfix t)       = Tfix (aux c t)
         aux c (Ttuple ts)    = Ttuple [ aux c t | t <- ts ]
         aux c (Ttupleproj t i) = Ttupleproj (aux c t) i
         aux c (Trecord lts) = Trecord [ L l (aux c t) | L l t <- lts ]
         aux c (Trecordproj t i) = Trecordproj (aux c t) i
         aux c (Tvarianttag (L l t) ty) = Tvarianttag (L l (aux c t)) ty
         aux c (Tvariantcase t0 lxts)   =
            Tvariantcase (aux c t0) [ L l (VariantCase x (aux (c+1) t))
                                    | L l (VariantCase x t) <- lxts ]
         aux c (Tnil ty) = Tnil ty
         aux c (Tcons ty t1 t2) = Tcons ty (aux c t1) (aux c t2)
         aux c (Tisnil ty t) = Tisnil ty (aux c t)
         aux c (Thead ty t) = Thead ty (aux c t)
         aux c (Ttail ty t) = Ttail ty (aux c t)
         aux c (Tref t) = Tref (aux c t)
         aux c (Tderef t) = Tderef (aux c t)
         aux c (Tassign t1 t2) = Tassign (aux c t1) (aux c t2)
         aux c (Tloc l) = Tloc l
         aux c (Ttounit t) = Ttounit (aux c t)

-- Implements the substitution of a term s
-- for variable number j in a term t (written [j -> s]t)
tSubst :: K -> T -> T -> T
tSubst j s Tunit        = Tunit
tSubst j s Ttrue        = Ttrue
tSubst j s Tfalse       = Tfalse
tSubst j s Tzero        = Tzero
tSubst j s (Tsucc t1)   = Tsucc (tSubst j s t1)
tSubst j s (Tpred t1)   = Tpred (tSubst j s t1)
```

```
tSubst j s (Tiszero t1) = Tiszero (tSubst j s t1)
tSubst j s (Tnot t1)    = Tnot (tSubst j s t1)
tSubst j s (Tand t1 t2) = Tand (tSubst j s t1) (tSubst j s t2)
tSubst j s (Tif t1 t2 t3) =
   Tif (tSubst j s t1) (tSubst j s t2) (tSubst j s t3)
tSubst j s (Tvar k)     = if (k == j) then s else (Tvar k)
tSubst j s (Tabs (x,ty) t1)  = Tabs (x,ty) (tSubst (j+1) (tShift 1 s) t1)
tSubst j s (Tapp t1 t2) = Tapp (tSubst j s t1) (tSubst j s t2)
tSubst j s (Tseq t1 t2) = Tseq (tSubst j s t1) (tSubst j s t2)
tSubst j s (Tlet x t1 t2)  =
   Tlet x (tSubst j s t1) (tSubst (j+1) (tShift 1 s) t2)
tSubst j s (Tfix t)  = Tfix (tSubst j s t)
tSubst j s (Ttuple ts)  = Ttuple [ tSubst j s t | t <- ts ]
tSubst j s (Ttupleproj t i)  = Ttupleproj (tSubst j s t) i
tSubst j s (Trecord lts)  = Trecord [ L l (tSubst j s t) | L l t <- lts ]
tSubst j s (Trecordproj t i)  = Trecordproj (tSubst j s t) i
tSubst j s (Tvarianttag (L l t) ty) = Tvarianttag (L l (tSubst j s t)) ty
tSubst j s (Tvariantcase t0 lxts) =
   Tvariantcase (tSubst j s t0)
                [ L l (VariantCase x (tSubst (j+1) (tShift 1 s) t))
                | L l (VariantCase x t) <- lxts ]
tSubst j s (Tnil ty) = Tnil ty
tSubst j s (Tcons ty t1 t2) =
   Tcons ty (tSubst j s t1) (tSubst j s t2)
tSubst j s (Tisnil ty t) = Tisnil ty (tSubst j s t)
tSubst j s (Thead ty t) = Thead ty (tSubst j s t)
tSubst j s (Ttail ty t) = Ttail ty (tSubst j s t)
tSubst j s (Tref t) = Tref (tSubst j s t)
tSubst j s (Tderef t) = Tderef (tSubst j s t)
tSubst j s (Tassign t1 t2) =
   Tassign (tSubst j s t1) (tSubst j s t2)
tSubst j s (Tloc l) = Tloc l
tSubst j s (Ttounit t) = Ttounit (tSubst j s t)

t100, t101 ::  Term
t100 = TmVar "a"
t101 = TmApp (TmVar "b")
         (TmApp (TmAbs ("x",TyBool)
    (TmApp (TmVar "b") (TmVar "x")))
                (TmVar "y"))
tb100 = removenames t100 (toVnamesCtx ctx100)
tb101 = removenames t101 (toVnamesCtx ctx100)

ctx100 :: Context
ctx100 = reverse [("a",ty1),("b",ty1),("y",TyBool)]

ty1 = TyFun TyBool TyBool
ty2 = TyFun TyBool (TyFun TyBool TyBool)
```

```
ty3 = TyFun (TyFun TyBool TyBool) (TyFun TyBool TyBool)
tyT4 :: Ty
tyT4 = TyFun (TyFun TyBool TyBool) TyBool
ty5 = TyFun (TyFun TyBool TyBool) (TyFun TyBool TyBool)

ty7 = TyFun (TyFun (TyFun TyBool TyBool) TyBool) (TyFun TyBool TyBool)

{-
Main> testTerm t1 ctx1 st0
((bool->(bool->bool))->((bool->(bool->bool))->(bool->(bool->bool))))
Main> testTerm t2 ctx1 st0
bool
Main> testTerm t3 ctx1 st0
(bool->bool)
Main> testTerm t4 ctx1 st0
(bool->(bool->(bool->bool)))
-}

t1 :: Term
t1 = TmAbs ("m",ty2) (TmAbs ("n",ty2) (TmAbs ("s",TyBool) (TmAbs ("z",TyBool)
         (TmApp (TmApp (TmVar "m") (TmVar "s"))
                (TmApp (TmApp (TmVar "n") (TmVar "s")) (TmVar "z"))))))

ctx1 :: Context
ctx1 = reverse [("x",ty1),("y",ty1),("z",TyBool),("a",TyBool),("b",TyBool)]

t2 :: Term
t2 = TmApp (TmVar "x") (TmApp (TmVar "y") (TmVar "z"))

t3 :: Term
t3 = TmAbs ("w",TyBool) (TmApp (TmVar "y")(TmVar "w"))

t4 :: Term
t4 = TmAbs ("w",TyBool) (TmAbs ("a",TyBool) (TmVar "x"))

tb1 :: T
tb1 = removenames t1 []

tb2 :: T
tb2 = removenames t2 (toVnamesCtx ctx1)
{- t2 = Tapp (Tvar 4) (Tapp (Tvar 3) (Tvar 2)) -}

tb3 :: T
tb3 = removenames t3 (toVnamesCtx ctx1)
{- t3 = Tabs (Tapp (Tvar 4) (Tvar 0)) -}

tb4 :: T
tb4 = removenames t4 (toVnamesCtx ctx1)
```

```
{- t4 = Tabs (Tabs (Tvar 6)) -}

{- Experiments from the book
Main> termterm t1 ctx1
(\(m:(bool -> (bool -> bool))).(\(n:(bool -> (bool -> bool))).
(\(s:bool).(\(z':bool).((m s)((n s) z'))))))
Main> termterm t2 ctx1
(x (y z))
Main> termterm t3 ctx1
(\(w:bool).(y w))
Main> termterm t4 ctx1
(\(w:bool).(\(a':bool).x))
Main> tt tb1 ctx1
(\(m:(bool -> (bool -> bool))).(\(n:(bool -> (bool -> bool))).
(\(s:bool).(\(z':bool).((3 1)((2 0) 1))))))
Main> tt tb2 ctx1
(4 (3 2))
Main> tt tb3 ctx1
(\(w:bool).(4 0))
Main> tt tb4 ctx1
(\(w:bool).(\(a':bool).6))
Main> tShift 100 tb1
(\(m:(bool -> (bool -> bool))).(\(n:(bool -> (bool -> bool))).
(\(s:bool).(\(z:bool).((3 1)((2 0) 1))))))
Main> tShift 100 tb2
(104 (103 102))
Main> tShift 100 tb3
(\(w:bool).(104 0))
Main> tShift 100 tb4
(\(w:bool).(\(a:bool).106))
-}

{- Experiments with typeOf
Main> testTerm t' ctx' st0
((bool->bool) -> ((bool->bool) -> bool))
Main> testT tb' ctx'st0
((bool->bool) -> ((bool->bool) -> bool))
-}

ctx' :: Context
ctx' = reverse [("z",TyBool)]

t' :: Term
t' = TmAbs ("x",ty1) (TmAbs ("y",ty1)
                            (TmApp (TmVar "x") (TmApp (TmVar "y") (TmVar "z"))))

tb' :: T
tb' = removenames t' (toVnamesCtx ctx')
```

```
-- t'' is NOT well typed
t'' :: Term
t'' = TmAbs ("x",ty1) (TmApp (TmApp (TmVar "x") (TmVar "z"))
                (TmAbs ("y",ty3)
                       (TmApp (TmApp (TmVar "y") (TmVar "x")) (TmVar "z"))))

tb'' :: T
tb'' = removenames t'' (toVnamesCtx ctx')

{- Experiments from the book
Main> t'
(\(x:(bool->bool)).(\(y:(bool->bool)).(x (y z))))
Main> t''
(\(x:(bool->bool)).((x z) (\(y:((bool->bool)->(bool->bool))).((y x) z))))
Main> tb'
(\(x:(bool->bool)).(\(y:(bool->bool)).(1 (0 2))))
Main> tb''
(\(x:(bool->bool)).((0 1) (\(y:((bool->bool)->(bool->bool))).((0 1) 2))))
Main> tShift 2 tb'
(\(x:(bool->bool)).(\(y:(bool->bool)).(1 (0 4))))
Main> tShift 2 tb''
(\(x:(bool->bool)).((0 3) (\(y:((bool->bool)->(bool->bool))).((0 1) 4))))
Main> tShift 100 tb'
(\(x:(bool->bool)).(\(y:(bool->bool)).(1 (0 102))))
Main> tShift 100 tb''
(\(x:(bool->bool)).((0 101) (\(y:((bool->bool)->(bool->bool))).((0 1) 102))))
-}

{-
Main> testTerm t5 ctx2 st0
bool
-}

t5, t6, t7, t8 :: Term
t5 = TmVar "a"
{- t6,t7 and t8 are not well-typed -}
t6 = TmApp (TmVar "a") (TmAbs ("z",ty1) (TmVar "a"))
t7 = TmApp (TmVar "b") (TmAbs ("x",ty1) (TmAbs ("y",ty1) (TmVar "b")))
t8 = TmApp (TmVar "b") (TmAbs ("x",ty1) (TmVar "b"))

ctx2 :: Context
ctx2 = reverse [("a",TyBool),("b",TyBool)]

xs2 :: VnamesContext
xs2 = toVnamesCtx ctx2

termSubst :: Int -> Term -> Term -> Context -> Term
```

```
termSubst j s t ctx =
   let xs = toVnamesCtx ctx
       s' = removenames s xs
       t' = removenames t xs
   in restorenames (tSubst j s' t') xs

{- Experiments from the book
Main> termSubst 0 t5 t7 ctx2
(a (\(x:(bool->bool)).(\(y:(bool->bool)).a)))
Main> termSubst 0 t6 t8 ctx2
((a (\(z:(bool->bool)).a)) (\(x:(bool->bool)).(a (\(z:(bool->bool)).a))))
Main> tSubst 0 (removenames t5 xs2) (removenames t7 xs2)
(1 (\(x:(bool->bool)).(\(y:(bool->bool)).3)))
Main> tSubst 0 (removenames t6 xs2) (removenames t8 xs2)
((1 (\(z:(bool->bool)).2)) (\(x:(bool->bool)).(2 (\(z:(bool->bool)).3))))
Main> tb100
2
Main> tb101
(1 ((\(x:bool).(2 0)) 0))
Main> tShift 100 tb101
(101 ((\(x:bool).(102 0)) 100))
Main> tSubst 1 tb101 tb100
2
Main> tSubst 1 tb100 tb101
(2 ((\(x:bool).(3 0)) 0))
Main> termSubst 1 t101 t100 ctx100
a
Main> termSubst 1 t100 t101 ctx100
(a ((\(x:bool).(a x)) y))
-}

-- Context not needed here, but needed later
isval ::  T -> Context -> Bool
isval Tunit                    ctx = True
isval Ttrue                    ctx = True
isval Tfalse                   ctx = True
isval Tzero                    ctx = True
isval (Tsucc nv)               ctx = isnumericval nv ctx
isval (Tabs _ _)               ctx = True
isval (Ttuple vs)              ctx = and [ isval v ctx | v <- vs ]
isval (Trecord lvs)            ctx = and [ isval v ctx | L _ v <- lvs ]
isval (Tvarianttag (L l v) ty) ctx = isval v ctx
isval (Tnil ty)                ctx = True
isval (Tcons ty v1 v2)         ctx = and [isval v1 ctx,isval v2 ctx]
isval (Tloc l)                 ctx = True
isval (Ttounit v)              ctx = isval v ctx
isval _                        ctx = False
```

```
isnumericval :: T -> Context -> Bool
isnumericval Tzero       ctx = True
isnumericval (Tsucc nv)  ctx = isnumericval nv ctx
isnumericval _           ctx = False

-- Context not needed here, but needed later
eval1 :: (T,Store) -> Context -> [(T,Store)]
eval1 (Tif Ttrue t2 t3,s)               ctx = [(t2,s)]
eval1 (Tif Tfalse t2 t3,s)              ctx = [(t3,s)]
eval1 (Tif t1 t2 t3,s)                  ctx =
      [ (Tif t1' t2 t3,s') | (t1',s') <- eval1 (t1,s) ctx ]
eval1 (Tapp t1 t2,s)                    ctx
   | (Tabs (x,_) t12) <- t1, isval t2 ctx   =
      [ (tShift (-1) (tSubst 0 (tShift 1 t2) t12),s) ]
   | isval t1 ctx                           =
      [ (Tapp t1 t2',s') | (t2',s') <- eval1 (t2,s) ctx ]
   | otherwise                              =
      [ (Tapp t1' t2,s') | (t1',s') <- eval1 (t1,s) ctx ]
eval1 (Tseq Tunit t2,s)                 ctx =
      [ (t2,s) ]
eval1 (Tseq t1 t2,s)                    ctx =
      [ (Tseq t1' t2,s') | (t1',s') <- eval1 (t1,s) ctx ]
eval1 (Tlet x t1 t2,s)                  ctx
   | isval t1 ctx                           =
      [ (tShift (-1) (tSubst 0 (tShift 1 t1) t2),s) ]
   | otherwise                              =
      [ (Tlet x t1' t2,s') | (t1',s') <- eval1 (t1,s) ctx ]
eval1 (Tfix t,s)                        ctx
   | (Tabs (x,ty1) t2) <- t                 =
      [ (tShift (-1) (tSubst 0 (tShift 1 (Tfix (Tabs (x,ty1) t2))) t2),s) ]
   | otherwise                              =
      [ (Tfix t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tabs _ _,s)                      ctx = []
eval1 (Tunit,s)                         ctx = []
eval1 (Ttrue,s)                         ctx = []
eval1 (Tfalse,s)                        ctx = []
eval1 (Tzero,s)                         ctx = []
eval1 (Tnot Ttrue,s)                    ctx = [ (Tfalse,s) ]
eval1 (Tnot Tfalse,s)                   ctx = [ (Ttrue,s) ]
eval1 (Tnot t,s)                        ctx =
      [ (Tnot t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tand Tfalse t2,s)                ctx = [ (Tfalse,s) ]
eval1 (Tand Ttrue Ttrue,s)              ctx = [ (Ttrue,s) ]
eval1 (Tand Ttrue Tfalse,s)             ctx = [ (Tfalse,s) ]
eval1 (Tand Ttrue t2,s)                 ctx =
      [ (Tand Ttrue t2',s') | (t2',s') <- eval1 (t2,s) ctx ]
eval1 (Tand t1 t2,s)                    ctx =
      [ (Tand t1' t2,s') | (t1',s') <- eval1 (t1,s) ctx ]
```

```
eval1 (Tsucc t,s)                       ctx
   | isnumericval t ctx                     = []
   | otherwise                              =
    [ (Tsucc t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tpred Tzero,s)                   ctx = [ (Tzero,s) ]
eval1 (Tpred t,s)                       ctx
   | (Tsucc nv) <- t,
     isnumericval nv ctx                    = [ (nv,s) ]
   | otherwise                              =
      [ (Tpred t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tiszero Tzero,s)                 ctx = [ (Ttrue,s) ]
eval1 (Tiszero t,s)                     ctx
   | (Tsucc nv) <- t                        = [ (Tfalse,s) ]
   | otherwise                              =
      [ (Tiszero t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tvar _,s)                        ctx = []
eval1 (Ttupleproj t i,s)                ctx
   | (Ttuple ts) <- t,
     (and [ isval t ctx | t <- ts ])        = [ (ts !! (i-1),s) ]
   | otherwise                              =
      [ (Ttupleproj t' i,s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Ttuple ts,s)                     ctx =
   case vsprefix ts (\t -> isval t ctx) of
        (vs,[])    -> []
        (vs,t:ts') ->
            [ (Ttuple (vs ++ (t':ts')),s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Trecordproj t l,s)               ctx
   | (Trecord lts) <- t,
     (and [ isval t ctx | L _ t <- lts ])   = [ (selectl lts l,s) ]
   | otherwise                              =
      [ (Trecordproj t' l,s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Trecord lts,s)                   ctx =
   case vsprefix lts (\(L l t) -> isval t ctx) of
        (lvs,[])         -> []
        (lvs,L l t:lts') ->
            [ (Trecord (lvs ++ (L l t':lts')),s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tvarianttag (L l t) ty,s)        ctx
   | isval t ctx                        = []
   | otherwise                            =
      [ (Tvarianttag (L l t') ty,s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tvariantcase t0 lxts,s)          ctx
   | (Tvarianttag (L lj vj) ty) <- t0,
     isval vj ctx                           =
          let VariantCase xj tj = selectl lxts lj
          in  [ (tShift (-1) (tSubst 0 (tShift 1 vj) tj),s) ]
   | otherwise                              =
          [ (Tvariantcase t0' lxts,s') | (t0',s') <- eval1 (t0,s) ctx ]
eval1 (Tnil ty,s)                       ctx = []
```

```
eval1 (Tcons ty t1 t2,s)               ctx
   | isval t1 ctx, isval t2 ctx            = []
   | isval t1 ctx                          =
      [ (Tcons ty t1 t2',s') | (t2',s') <- eval1 (t2,s) ctx ]
   | otherwise                             =
      [ (Tcons ty t1' t2,s') | (t1',s') <- eval1 (t1,s) ctx ]
eval1 (Tisnil ty t,s)                  ctx
   | (Tnil _) <- t                         = [ (Ttrue,s) ]
   | (Tcons _ v1 v2) <- t,
     isval v1 ctx, isval v2 ctx            = [ (Tfalse,s) ]
   | otherwise                             =
      [ (Tisnil ty t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Thead ty t,s)                   ctx
   | (Tcons _ v1 v2) <- t,
     isval v1 ctx, isval v2 ctx            = [ (v1,s) ]
   | otherwise                             =
      [ (Thead ty t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Ttail ty t,s)                   ctx
   | (Tcons _ v1 v2) <- t,
     isval v1 ctx, isval v2 ctx            = [ (v2,s) ]
   | otherwise                             =
      [ (Ttail ty t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tref t,s)                       ctx
   | isval t ctx                           = let newlocation = length s
                                             in  [(Tloc newlocation,s++[t])]
   | otherwise                             =
      [ (Tref t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tderef t,s)                     ctx
   | (Tloc l) <- t                         = [(getvalfromstore s l,s)]
   | isval t ctx                           = error "eval1: no rules apply"
   | otherwise                             =
      [ (Tderef t',s') | (t',s') <- eval1 (t,s) ctx ]
eval1 (Tassign t1 t2,s)                ctx
   | (Tloc l) <- t1, isval t2 ctx          = [(Tunit,updatestore s l t2)]
   | isval t1 ctx, isval t2 ctx            = error "eval1: no rules apply"
   | isval t1 ctx                          =
      [ (Tassign t1 t2',s') | (t2',s') <- eval1 (t2,s) ctx ]
   | otherwise                             =
      [ (Tassign t1' t2,s') | (t1',s') <- eval1 (t1,s) ctx ]
eval1 (Ttounit t,s)                    ctx
   | isval t ctx                           = [ (Tunit,s) ]
   | otherwise                             =
      [ (Ttounit t',s') | (t',s') <- eval1 (t,s) ctx ]

updatestore :: Store -> Location -> T -> Store
updatestore []     l v = error "updatestore: bad location"
updatestore (v0:s) l v =
   if (l<0) then error "updatestore: bad location"
```

```
   else if (l==0) then v:s else v0:updatestore s (l-1) v

getvalfromstore :: Store -> Location -> T
getvalfromstore s l = s !! l

--------------
selectl :: [L a] -> Label -> a
selectl []          l =
   error ("selectl -> no record field labeled with " ++ l)
selectl (L l' x:ls) l =
   if (l' == l) then x else selectl ls l

vsprefix :: [a] -> (a -> Bool) -> ([a],[a])
vsprefix [] f = ([],[])
vsprefix (x:xs) f =
   if (f x) then (let (xs1,xs2) = vsprefix xs f in (x:xs1,xs2))
   else ([],x:xs)

-- Context not needed here, but needed later
evalT :: (T,Store) -> Context -> (T,Store)
evalT (t,s) ctx =
   case eval1 (t,s) ctx of
        [] -> (t,s)
        [(t',s')] -> evalT (t',s') ctx
        _    -> error "Nondeterministic evaluation (impossible!)"

evalTerm :: (Term,Store) -> Context -> (Term,Store)
evalTerm (term,s) ctx =
   let xs = toVnamesCtx ctx
       (t',s') = evalT (removenames term xs,s) ctx
   in (restorenames t' xs,s')

ctx0 :: Context
ctx0 = []

s0 :: Store
s0 = []

st0 :: StoreTyping
st0 = []

{- Experiments with typeOf
Main> testT tb5 ctx5 st0
bool
-}

tb5 :: T
tb5 = Tapp (Tabs ("x",ty1) (Tapp (Tapp (Tvar 1)(Tvar 0)) (Tvar 2)))
```

```
          (Tabs ("y",TyBool) (Tvar 0))

ctx5 :: Context
ctx5 = [("w",ty3),("u",TyBool)]

{- Experiments with typeOf
Main> testTerm tstscc0 [] st0
((bool->bool)->(bool->bool))
Main> testTerm tstscc1 [] st0
((bool->bool)->(bool->bool))
Main> testTerm tstscc2 [] st0
((bool->bool)->(bool->bool))
Main> testTerm tstscc3 [] st0
((bool->bool)->(bool->bool))
Main> testTerm tstscc4 [] st0
((bool->bool)->(bool->bool))
Main> testTerm tstscc5 [] st0
((bool->bool)->(bool->bool))
-}

c0 :: Term
c0 = TmAbs ("s",ty1) (TmAbs ("z",TyBool) (TmVar "z"))

c1 :: Term
c1 = TmAbs ("s",ty1) (TmAbs ("z",TyBool) (TmApp (TmVar "s") (TmVar "z")))

c2 :: Term
c2 = TmAbs ("s",ty1) (TmAbs ("z",TyBool)
              (TmApp (TmVar "s") (TmApp (TmVar "s") (TmVar "z"))))

c3 :: Term
c3 = TmAbs ("s",ty1)
       (TmAbs ("z",TyBool)
          (TmApp (TmVar "s") (TmApp (TmVar "s")
                 (TmApp (TmVar "s") (TmVar "z")))))

c4 :: Term
c4 = TmAbs ("s",ty1) (TmAbs ("z",TyBool)
                           (TmApp (TmVar "s") (TmApp (TmVar "s")
                             (TmApp (TmVar "s") (TmApp (TmVar "s") (TmVar "z"))))))

c5 :: Term
c5 = TmAbs ("s",ty1) (TmAbs ("z",TyBool)
                        (TmApp (TmVar "s") (TmApp (TmVar "s")
                           (TmApp (TmVar "s") (TmApp (TmVar "s")
                             (TmApp (TmVar "s") (TmVar "z")))))))

scc :: Term
```

```
scc = TmAbs ("n",ty5) (TmAbs ("s",ty1)
             (TmAbs ("z",TyBool)
                  (TmApp (TmVar "s")
                      (TmApp (TmApp (TmVar "n")(TmVar "s")) (TmVar "z")))))

tstscc0 :: Term
tstscc0 = TmApp scc c0

tstscc1 :: Term
tstscc1 = TmApp scc c1

tstscc2 :: Term
tstscc2 = TmApp scc c2

tstscc3 :: Term
tstscc3 = TmApp scc c3

tstscc4 :: Term
tstscc4 = TmApp scc c4

tstscc5 :: Term
tstscc5 = TmApp scc c5

t10 :: Term
t10 = TmApp (TmAbs ("x",TyFun TyBool TyBool)
               (TmApp (TmVar "y") (TmVar "x")))
                (TmAbs ("u",TyBool) (TmVar "u"))

ctx4 :: Context
ctx4 = [("u",TyBool)]

{- Experiments with typeOf
Main> testTerm t10 ctx3 st0
bool
Main> testTerm t11 ctx4 st0
bool
Main> testTerm t12 ctx3 st0
(bool -> bool)
Main> testTerm t13 ctx3 st0
((bool -> bool) -> bool)
-}

t11 :: Term
t11 = TmApp (TmAbs ("x",TyBool) (TmVar "x")) (TmVar "u")

t12 :: Term
t12 = TmAbs ("x",TyBool) (TmVar "x")
```

```
t13 :: Term
t13 = TmAbs ("x",TyFun TyBool TyBool) (TmApp (TmVar "y") (TmVar "x"))

ctx3 :: Context
ctx3 = reverse [("y",tyT4),("z",TyBool)]

{- Experiments from the book
Main> removenames t10 (toVnamesCtx ctx3)
((\(x:(bool->bool)).(2 0)) (\(u:bool).0))
Main> evalT (removenames t10 (toVnamesCtx ctx3),s0) ctx3
((1 (\(u:bool).0)),[])
Main> c3
(\(s:(bool->bool)).(\(z:bool).(s (s (s z)))))
Main> c2
(\(s:(bool->bool)).(\(z:bool).(s (s z))))
Main> c1
(\(s:(bool->bool)).(\(z:bool).(s z)))
Main> c0
(\(s:(bool->bool)).(\(z:bool).z))
Main> evalTerm (TmApp scc c0,s0) []
((\(s:(bool->bool)).(\(z:bool).
(s ((((\(s':(bool->bool)).(\(z':bool).z')) s) z)))),[])
-- Notice that the resulted term is behaviorally equivalent to
-- (\s.(\z.(s z))) = c1
-- and obviously, (TmApp scc c0) is the implementation of (scc c0),
-- i.e. scc c0 ->* c1
-}

xapply, xnot :: Term
xapply = TmAbs ("f",TyFun TyBool TyBool)
               (TmAbs ("x",TyBool) (TmApp (TmVar "f") (TmVar "x")))

xnot = TmAbs ("b",TyBool) (TmIf (TmVar "b") TmFalse TmTrue)

testt, testf :: Term
testt = TmApp (TmApp xapply xnot) TmTrue
testf = TmApp (TmApp xapply xnot) TmFalse

testnott, testnotf :: Term
testnott = TmApp xnot TmTrue
testnotf = TmApp xnot TmFalse

exet, exef :: (Term,Store)
exet = evalTerm (testt,s0) []
exef = evalTerm (testf,s0) []

exent, exenf :: (Term,Store)
exent = evalTerm (testnott,s0) []
```

```
exenf = evalTerm (testnott,s0) []

-- Type checking

typeOf :: T -> Context -> StoreTyping -> Ty
typeOf Ttrue ctx st = TyBool
typeOf Tfalse ctx st = TyBool
typeOf Tzero ctx st = TyNat
typeOf (Tsucc t1) ctx st =
   case typeOf t1 ctx st of
        TyNat -> TyNat
        _     -> error "typeOf -> Tsucc: non-nat argument"
typeOf (Tpred t1) ctx st =
   case typeOf t1 ctx st of
        TyNat -> TyNat
        _     -> error "typeOf -> Tpred: non-nat argument"
typeOf (Tiszero t1) ctx st =
   case typeOf t1 ctx st of
        TyNat -> TyBool
        _     -> error "typeOf -> Tiszero: non-nat argument"
typeOf (Tnot t1) ctx st =
   case typeOf t1 ctx st of
        TyBool -> TyBool
        _      -> error "typeOf -> Tnot: non-boolean argument"
typeOf (Tand t1 t2) ctx st =
   case (typeOf t1 ctx st,typeOf t2 ctx st) of
        (TyBool,TyBool) -> TyBool
        _               -> error "typeOf -> Tand: non-boolean argument(s)"
typeOf (Tif t1 t2 t3) ctx st =
   case (typeOf t1 ctx st) of
        TyBool -> let { ty2 = typeOf t2 ctx st; ty3 = typeOf t3 ctx st }
                  in if (ty2 == ty3) then ty2
                     else error ("typeOf -> Tif: arms of conditional have
                                  different types" ++
                                 " \n type of then branch = " ++ (show ty2) ++
                                 " \n type of else branch = " ++ (show ty3))
        _      -> error "typeOf -> Tif: non-boolean condition"
typeOf (Tvar k) ctx st = getTypeFromContext k ctx
typeOf (Tabs (x,tyT1) t2) ctx st =
   let ctx' = addbinding ctx x tyT1
       tyT2 = typeOf t2 ctx' st
   in  TyFun tyT1 tyT2
typeOf (Tapp t1 t2) ctx st =
   let tyT1 = typeOf t1 ctx st
       tyT2 = typeOf t2 ctx st
   in case tyT1 of
           (TyFun tyT11 tyT12) ->
              if (tyT2 == tyT11) then tyT12
```

```
              else error "typeOf -> Tapp: parameter type mismatch"
           _ -> error "typeOf -> Tapp:arrow type expected"
typeOf (Tunit) ctx st = TyUnit
typeOf (Tseq t1 t2) ctx st =
   case (typeOf t1 ctx st) of
        TyUnit -> typeOf t2 ctx st
        _      -> error "typeOf -> Tseq: Unit expected as type of first arg"
typeOf (Tlet x t1 t2) ctx st =
   let tyT1 = typeOf t1 ctx st
       ctx' = addbinding ctx x tyT1
   in typeOf t2 ctx' st
typeOf (Tfix t) ctx st =
   case typeOf t ctx st of
       (TyFun ty1 ty2) -> if (ty1 == ty2) then ty1 else error "typeOf -> Tfix"
       _               -> error "typeOf -> Tfix"
typeOf (Ttuple ts) ctx st = TyTuple [ typeOf t ctx st | t <- ts ]
typeOf (Ttupleproj t i) ctx st =
   case (typeOf t ctx st) of
        (TyTuple lts) -> if (i > length lts)
                         then error "typeOf -> Trecordproj: index too large"
                         else lts !! (i-1)
        _             -> error "typeOf -> Trecordproj: Trecord expected
                                  as type of first arg"
typeOf (Trecord lts) ctx st =
   TyRecord [ L l (typeOf t ctx st) | L l t <- lts ]
typeOf (Trecordproj t l) ctx st =
   case (typeOf t ctx st) of
        (TyRecord lts) -> selectl lts l
        _              -> error "typeOf -> Trecordproj: TyRecord expected
                                  as type of first arg"
typeOf (Tvarianttag (L lj tj) tyT) ctx st =
   case tyT of
        (TyVariant ltys) -> if (typeOf tj ctx st == selectl ltys lj) then tyT
                            else error "typeOf -> Tvarianttag: type misfit"
        _                -> error "typeOf -> Tvarianttag: TyVariant type
                                   expected as second arg"
typeOf (Tvariantcase t0 lxts) ctx st =
   case (typeOf t0 ctx st) of
        (TyVariant ltys) ->
           let (tyT:ltys') = [ typeOf ti (addbinding ctx xi (selectl ltys li)) st
                             | L li (VariantCase xi ti) <- lxts ]
           in  if (and [ tyT' == tyT| tyT' <- ltys' ]) then tyT
               else error "typeOf -> Tvariantcase: type misfit"
        _                -> error "typeOf -> Tvariantcase: TyVariant type
                                   expected as type of first arg"
typeOf (Tnil ty)  ctx st = TyList ty
typeOf (Tcons ty1 t1 t2) ctx st =
   if (typeOf t1 ctx st == ty1 && typeOf t2 ctx st == TyList ty1)
```

```
   then TyList ty1
   else error "typeOf -> Tcons: type misfit"
typeOf (Tisnil ty t) ctx st =
   if (typeOf t ctx st == TyList ty) then TyBool
   else error "typeOf -> Tisnil: type misfit"
typeOf (Thead ty t) ctx st =
   if (typeOf t ctx st == TyList ty) then ty
   else error "typeOf -> Thead: type misfit"
typeOf (Ttail ty t) ctx st =
   if (typeOf t ctx st == TyList ty) then TyList ty
   else error "typeOf -> Thead: type misfit"
typeOf (Tloc l) ctx st =
   TyRef (getTypeFromStoreTyping l st)
typeOf (Tref t) ctx st =
   TyRef (typeOf t ctx st)
typeOf (Tderef t) ctx st =
   case typeOf t ctx st of
        (TyRef ty) -> ty
        _          -> error "typeOf -> Tderef: reference expected "
typeOf (Tassign t1 t2) ctx st =
   case (typeOf t1 ctx st,typeOf t2 ctx st) of
        (TyRef ty1,ty2) -> if (ty1==ty2) then TyUnit
                           else error "typeOf -> tassign: type misfit"
        _               -> error "typeOf -> tassign: type misfit"
typeOf (Ttounit t) ctx st =
   let ty = typeOf t ctx in TyUnit

addbinding :: Context -> X -> Ty -> Context
addbinding ctx x ty = (x,ty):ctx

getTypeFromContext :: K -> Context -> Ty
getTypeFromContext k ctx = snd (ctx !! k)

getTypeFromStoreTyping :: Location -> StoreTyping -> Ty
getTypeFromStoreTyping l st = st !! l

testTerm :: Term -> Context -> StoreTyping -> Ty
testTerm term ctx st =
   typeOf (removenames term (toVnamesCtx ctx)) ctx st

testT :: T -> Context -> StoreTyping -> Ty
testT t ctx st = typeOf t ctx st

ctx1000 :: Context
ctx1000 = reverse [("y",tyT4),("z",TyBool)]

t1000 :: Term
t1000 = TmApp (TmAbs ("x",TyFun TyBool TyBool)
```

```
                (TmApp (TmVar "y") (TmVar "x")))
                   (TmAbs ("u",TyBool) (TmVar "u"))
{-
Main> evalTerm (t1000,s0) ctx1000
((y (\(u:bool).u)),[])
-}

tmnr :: Int -> Term
tmnr 0 = TmZero
tmnr n = TmSucc (tmnr (n-1))

tmladd :: Term
tmladd = TmAbs ("m",TyNat)
               (TmAbs ("n",TyNat)
                  (TmIf (TmIsZero (TmVar "m"))
                        (TmVar "n")
                        (TmSucc
                           (TmApp (TmApp (TmVar "add") (TmPred (TmVar "m")))
                                  (TmVar "n")))))

tmlmul :: Term
tmlmul = TmAbs ("m",TyNat)
               (TmAbs ("n",TyNat)
                  (TmIf
                     (TmApp (TmApp (TmVar "eqnat") (TmVar "m")) (tmnr 1))
                     (TmVar "n")
                     (TmApp (TmApp (TmVar "add") (TmVar "n"))
                            (TmApp (TmApp (TmVar "mul") (TmPred (TmVar "m")))
                                   (TmVar "n"))))))

tmleqnat :: Term
tmleqnat = TmAbs ("m",TyNat)
               (TmAbs ("n",TyNat)
                  (TmIf (TmAnd (TmIsZero (TmVar "m")) (TmIsZero (TmVar "n")))
                  TmTrue
                  (TmIf (TmIsZero (TmVar "m")) TmFalse
        (TmIf (TmIsZero (TmVar "n")) TmFalse
                              (TmApp
                                 (TmApp (TmVar "eqnat") (TmPred (TmVar "m")))
                                 (TmPred (TmVar "n")))
                        )
                  )))

tmadd :: Term
tmadd =
   TmLetrec ("add",TyFun TyNat (TyFun TyNat TyNat)) tmladd
            (TmApp (TmApp (TmVar "add") (tmnr 5)) (tmnr 5))
```

```
tmmul :: Term
tmmul =
   TmLetrec ("add",TyFun TyNat (TyFun TyNat TyNat)) tmladd
   (TmLetrec ("eqnat",TyFun TyNat (TyFun TyNat TyBool)) tmleqnat
            (TmLetrec ("mul",TyFun TyNat (TyFun TyNat TyNat)) tmlmul
                      (TmApp (TmApp (TmVar "mul") (tmnr 5)) (tmnr 5))))

{-
Main> test tmadd
type = nat
value = (...,[])
[value = (10 = 5+5,[])]
Main> test tmmul
type = nat
value = (...,[])
[value = (25 = 5*5,[])]
Main> test tmfact
type = nat
value = (...,[])
[value = (120 = 5!,[])]
Main> test tmfib
type = nat
value = (...,[])
[value = (21 = fib 8,[])]
Main> test tmrecord
type = {1=nat,2=nat,3=nat,4=nat}
value = (...,[])
[value = ({1=1,2=2,3=6,4=24},[])]
Main> test tmcase1
type = nat
value = ((succ 0),[])
Main> test tmcase2
type = nat
value = (0,[])
Main> test tmcase3
type = nat
value = (...,[])
[value = (10,[])]
Main> test tmtag
type = <N=nat,B=bool>
value = (<B=false> as <N=nat,B=bool>,[])
Main> test tmlist
type = (List nat)
[value = ((cons[nat] 1 (cons[nat] 5 (cons[nat] 3 (nil[nat])))),[])]
Main> test tmappend
type = (List nat)
[value = ((cons[nat] 1 (cons[nat] 2 (cons[nat] 3
```

```
(cons[nat] 4 (cons[nat] 5 (cons[nat] 6 (cons[nat] 7 (nil[nat])))))),[])]
-}

tmlfact :: Term
tmlfact =
   TmAbs ("n",TyNat)
         (TmIf (TmIsZero (TmVar "n")) (tmnr 1)
               (TmApp (TmApp (TmVar "mul") (TmVar "n"))
                      (TmApp (TmVar "fact") (TmPred (TmVar "n")))))

tmfact :: Term
tmfact =
   TmLetrec ("add",TyFun TyNat (TyFun TyNat TyNat)) tmladd
   (TmLetrec ("eqnat",TyFun TyNat (TyFun TyNat TyBool)) tmleqnat
   (TmLetrec ("mul",TyFun TyNat (TyFun TyNat TyNat)) tmlmul
            (TmLetrec ("fact",TyFun TyNat TyNat) tmlfact
                      (TmApp (TmVar "fact") (tmnr 5)))))

tmlfib :: Term
tmlfib =
   TmAbs ("n",TyNat)
         (TmIf (TmIsZero (TmVar "n")) TmZero
         (TmIf (TmApp
                   (TmApp (TmVar "eqnat") (TmVar "n")) (tmnr 1)) (tmnr 1)
               (TmApp (TmApp (TmVar "add")
                         (TmApp (TmVar "fib") (TmPred (TmVar "n"))))
                           (TmApp (TmVar "fib")
                                  (TmPred (TmPred (TmVar "n")))))))

tmfib :: Term
tmfib =
   TmLetrec ("add",TyFun TyNat (TyFun TyNat TyNat)) tmladd
   (TmLetrec ("eqnat",TyFun TyNat (TyFun TyNat TyBool)) tmleqnat
            (TmLetrec ("fib",TyFun TyNat TyNat) tmlfib
                      (TmApp (TmVar "fib") (tmnr 8))))

tmfactn :: Int -> Term
tmfactn n = TmLetrec ("fact",TyFun TyNat TyNat) tmlfact
                     (TmApp (TmVar "fact") (tmnr n))

tmrecord :: Term
tmrecord =
   TmLetrec ("add",TyFun TyNat (TyFun TyNat TyNat)) tmladd
   (TmLetrec ("eqnat",TyFun TyNat (TyFun TyNat TyBool)) tmleqnat
   (TmLetrec ("mul",TyFun TyNat (TyFun TyNat TyNat)) tmlmul
   (TmRecord [L "1" (tmfactn 1), L "2" (tmfactn 2),
              L "3" (tmfactn 3), L "4" (tmfactn 4)])))
```

```
tmcase1 :: Term
tmcase1 = TmVariantCase
             (TmVariantTag (L "N" TmZero)
                           (TyVariant [L "N" TyNat,L "B" TyBool]))
                   [L "N" (VariantCase "x1" (TmSucc (TmVar "x1"))),
                    L "B" (VariantCase "x2" TmZero)]

tmcase2 :: Term
tmcase2 = TmVariantCase
             (TmVariantTag (L "B" TmTrue)
                 (TyVariant [L "N" TyNat,L "B" TyBool]))
                    [L "N" (VariantCase "x1" (TmSucc (TmVar "x1"))),
                     L "B" (VariantCase "x2" TmZero)]

tmcase3 :: Term
tmcase3 = TmVariantCase
             (TmVariantTag (L "N2" (TmTuple [tmnr 5,tmnr 10]))
                 (TyVariant [L "B" TyBool,L "N2" (TyTuple [TyNat,TyNat])]))
                    [L "N2" (VariantCase "x1" (TmTupleProj (TmVar "x1") 2)),
                     L "B" (VariantCase "x2" TmZero)]

tmtag :: Term
tmtag = TmVariantTag (L "B" TmFalse) (TyVariant [L "N" TyNat,L "B" TyBool])

tmlist :: Term
tmlist =
   TmCons TyNat (tmnr 1)
                (TmCons TyNat (tmnr 5)
                              (TmCons TyNat (tmnr 3) (TmNil TyNat)))

tmxs, tmys :: Term
tmys =
   TmCons TyNat (tmnr 5)
                (TmCons TyNat (tmnr 6)
                              (TmCons TyNat (tmnr 7) (TmNil TyNat)))
tmxs =
   TmCons TyNat (tmnr 1)
                (TmCons TyNat (tmnr 2)
                              (TmCons TyNat (tmnr 3)
                                  (TmCons TyNat (tmnr 4) (TmNil TyNat))))

tmlappend :: Term
tmlappend =
   TmAbs ("xs",TyList TyNat)
     (TmAbs ("ys",TyList TyNat)
         (TmIf (TmIsNil TyNat (TmVar "xs")) (TmVar "ys")
               (TmCons TyNat (TmHead TyNat (TmVar "xs"))
                  (TmApp (TmApp (TmVar "append") (TmTail TyNat (TmVar "xs")))
```

```
                        (TmVar "ys"))
              )))

tmappend :: Term
tmappend =
 TmLetrec ("append",TyFun (TyList TyNat) (TyFun (TyList TyNat) (TyList TyNat)))
          tmlappend
          (TmApp (TmApp (TmVar "append") tmxs) tmys)

tmrefbasics0 :: Term
tmrefbasics0 =
   TmLet "r" (TmRef (tmnr 5)) (TmDeref (TmVar "r"))

{-
Main> test tmrefbasics0
type  = nat
value = (5,[5])
-}

tmrefbasics1 :: Term
tmrefbasics1 =
   TmLet "r" (TmRef (tmnr 5))
             (TmSeq (TmAssign (TmVar "r") (tmnr 7)) (TmDeref (TmVar "r")))

{-
Main> test tmrefbasics1
type  = nat
value = (7,[7])
-}

tmrefbasics2 :: Term
tmrefbasics2 =
   TmLet "r" (TmRef (tmnr 5))
             (TmSeq (TmAssign (TmVar "r") (tmnr 7))
                    (TmSucc (TmSucc (TmSucc (TmDeref (TmVar "r"))))))

{-
Main> test tmrefbasics2
type  = nat
value = (10,[7])
-}

tmrefbasics3 :: Term
tmrefbasics3 =
   TmLet "r" (TmRef (tmnr 5))
             (TmSeq (TmAssign (TmVar "r") (TmSucc (TmDeref (TmVar "r"))))
             (TmSeq (TmAssign (TmVar "r") (TmSucc (TmDeref (TmVar "r"))))
             (TmSeq (TmAssign (TmVar "r") (TmSucc (TmDeref (TmVar "r")))))
```

```
             (TmSeq (TmAssign (TmVar "r") (TmSucc (TmDeref (TmVar "r"))))
             (TmSeq (TmAssign (TmVar "r") (TmSucc (TmDeref (TmVar "r"))))
                    (TmDeref (TmVar "r")))))))

{-
Main> test tmrefbasics3
type  = nat
value = (10,[10])
-}

tmrefbasics4 :: Term
tmrefbasics4 =
   TmLet "r" (TmRef (tmnr 100))
             (TmLet "s" (TmVar "r")
             (TmSeq (TmAssign (TmVar "s") (tmnr 5)) (TmDeref (TmVar "r"))))

{-
Main> test tmrefbasics4
type  = nat
value = (5,[5])
-}

tmrefbasics5 :: Term
tmrefbasics5 =
   TmLet "r" (TmRef (tmnr 10))
             (TmLet "s" (TmVar "r")
             (TmSeq (TmAssign (TmVar "s") (tmnr 5))
                    (TmPred (TmPred (TmDeref (TmVar "r"))))))

{-
Main> test tmrefbasics5
type  = nat
value = (3,[5])
-}

tmrefshared1 :: Term
tmrefshared1 =
   TmLet "c" (TmRef (tmnr 0))
   (TmLet "incc" (TmAbs ("x",TyUnit)
                   (TmSeq
                     (TmAssign (TmVar "c") (TmSucc (TmDeref (TmVar "c"))))
                     (TmDeref (TmVar "c"))))
   (TmLet "decc" (TmAbs ("x",TyUnit)
                   (TmSeq
                      (TmAssign (TmVar "c") (TmPred (TmDeref (TmVar "c"))))
                      (TmDeref (TmVar "c"))))
   (TmApp (TmVar "incc") TmUnit)))
```

```
{-
Main> test tmrefshared1
type  = nat
value = (1,[1])
-}

tmrefshared2 :: Term
tmrefshared2 =
   TmLet "c" (TmRef (tmnr 0))
   (TmLet "incc"
      (TmAbs ("x",TyUnit)
             (TmSeq (TmAssign (TmVar "c") (TmSucc (TmDeref (TmVar "c"))))
                    (TmDeref (TmVar "c"))))
   (TmLet "decc"
      (TmAbs ("x",TyUnit)
             (TmSeq (TmAssign (TmVar "c") (TmPred (TmDeref (TmVar "c"))))
                    (TmDeref (TmVar "c"))))
   (TmSeq (TmToUnit (TmApp (TmVar "incc") TmUnit))
          (TmApp (TmVar "decc") TmUnit))))

{-
Main> test tmrefshared2
type  = nat
value = (0,[0])
-}

tmrefshared3 :: Term
tmrefshared3 =
   TmLet "c" (TmRef (tmnr 0))
   (TmLet "incc"
      (TmAbs ("x",TyUnit)
             (TmSeq (TmAssign (TmVar "c") (TmSucc (TmDeref (TmVar "c"))))
                    (TmDeref (TmVar "c"))))
   (TmLet "decc"
      (TmAbs ("x",TyUnit)
             (TmSeq (TmAssign (TmVar "c") (TmPred (TmDeref (TmVar "c"))))
                    (TmDeref (TmVar "c"))))
   (TmSeq (TmToUnit (TmApp (TmVar "incc") TmUnit))
   (TmSeq (TmToUnit (TmApp (TmVar "incc") TmUnit))
   (TmSeq (TmToUnit (TmApp (TmVar "incc") TmUnit))
          (TmApp (TmVar "decc") TmUnit))))))

{-
Main> test tmrefshared3
type  = nat
value = (2,[2])
-}
```

```
tmrefshared4 :: Term
tmrefshared4 =
   TmLet "c" (TmRef (tmnr 0))
   (TmLet "incc"
      (TmAbs ("x",TyUnit)
             (TmSeq (TmAssign (TmVar "c") (TmSucc (TmDeref (TmVar "c"))))
                    (TmDeref (TmVar "c"))))
   (TmLet "decc"
      (TmAbs ("x",TyUnit)
             (TmSeq (TmAssign (TmVar "c") (TmPred (TmDeref (TmVar "c"))))
                    (TmDeref (TmVar "c"))))
   (TmLet "o" (TmRecord [L "i" (TmVar "incc"),L "d" (TmVar "decc")])
   (TmSeq (TmToUnit (TmApp (TmRecordProj (TmVar "o") "i") TmUnit))
   (TmSeq (TmToUnit (TmApp (TmRecordProj (TmVar "o") "i") TmUnit))
   (TmSeq (TmToUnit (TmApp (TmRecordProj (TmVar "o") "i") TmUnit))
   (TmSeq (TmToUnit (TmApp (TmRecordProj (TmVar "o") "i") TmUnit))
          (TmApp (TmRecordProj (TmVar "o") "d") TmUnit)))))))))

{-
Main> test tmrefshared4
type  = nat
value = (3,[3])
-}

tmrefarray :: Term
tmrefarray =
   TmLetrec ("add",TyFun TyNat (TyFun TyNat TyNat)) tmladd
   (TmLetrec ("eqnat",TyFun TyNat (TyFun TyNat TyBool)) tmleqnat
   (TmLet "newarray"
       (TmAbs ("",TyUnit) (TmRef (TmAbs ("n",TyNat) TmZero)))
   (TmLet "lookup"
       (TmAbs ("a",TyRef (TyFun TyNat TyNat))
              (TmAbs ("n",TyNat)
                     (TmApp (TmDeref (TmVar "a")) (TmVar "n"))))
   (TmLet "update"
       (TmAbs ("a",TyRef (TyFun TyNat TyNat))
         (TmAbs ("m",TyNat)
           (TmAbs ("v",TyNat)
              (TmLet "oldf" (TmDeref (TmVar "a"))
                     (TmAssign (TmVar "a")
                     (TmAbs ("n",TyNat)
                        (TmIf (TmApp
                                 (TmApp (TmVar "eqnat") (TmVar "m"))
                                        (TmVar "n"))
                              (TmVar "v")
                              (TmApp (TmVar "oldf") (TmVar "n"))))
                             )))))
   (TmLet "array" (TmApp (TmVar "newarray") TmUnit)
```

```
          (TmSeq
              (TmApp (TmApp (TmApp (TmVar "update")
                        (TmVar "array")) (tmnr 1)) (tmnr 3))
          (TmSeq (TmApp (TmApp (TmApp (TmVar "update")
                        (TmVar "array")) (tmnr 2)) (tmnr 4))
 (TmApp (TmApp (TmVar "add")
                        (TmApp (TmApp (TmVar "lookup") (TmVar "array"))
                                 (tmnr 1)))
                        (TmApp (TmApp (TmVar "lookup") (TmVar "array"))
                                 (tmnr 2))
                  ))))))))

{-
Main> test tmrefarray
type = nat
value = (7=3+4=array(1)+array(2),[...])
-}

test :: Term -> IO ()
test t = do putStrLn ("type  = " ++ (show (typeOf (removenames t []) ctx0 st0)));
            putStrLn ("value = " ++ (show (evalTerm (t,s0) ctx0)));
```

Bibliography

[1] B. Pierce, *Types and Programming Languages,* MIT Press, 2002 (resources for the book, including OCaml prototype implementations of evaluators and type checkers, are available at `http://www.cis.upenn.edu/~bcpierce/tapl/`)

[2] B. Pierce, editor, *Advanced Topics in Types and Programming Languages,* MIT Press, 2005.

[3] B. Pierce, et al, *Software Foundations,* Electronic textbook, 2019. (available at `http://www.cis.upenn.edu/~bcpierce/sf`)

[4] F. Piessen, Formal Systems and their Applications, Course nr: B-KUL-H04H8B, Catholic University of Leuven, 2014 (available from `http://www.cis.upenn.edu/~bcpierce/tapl/`)

[5] J. Hillston. *A compositional approach to performance modeling,* Cambridge Univ. Press, 1996 Available from `http://www.dcs.ed.ac.uk/pepa/papers`.

[6] P. Hudak, J. Peterson and J.H. Fasel, A gentle introduction to Haskell 98, 2000 (Tutorial available from `www.haskell.org`).

[7] D. Friedman, D.T. Christiansen, *The Little Typer,* MIT Press, 2018.

[8] M. Odersky, L. Spoon, B. Venners, *Programming in Scala, third edition,* Artima, 2016.

[9] R. Milner, *Communicating and mobile systems: the π-calculus,* Cambridge University Press, 1999.

[10] S.L. Peyton Jones, et al. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language, *SIGPLAN Notices* 27(5): 1-, 1992.

[11] PRISM web page, `http://www.prismmodelchecker.org/`.

[12] Y. Falcone, K. Havelund, G. Reger, A Tutorial on Runtime Verification, *Engineering Dependable Software Systems* 2013, pages 141–175, 2013.

[13] S. Banach, Sur les operations dans les ensembles abstraits et leur application aux equations integrales, *Fund. Math.* 3:133-181, 1922.

[14] H.P. Barendregt. *The Lambda Calculus*, North Holland, 1984.

[15] A. Bejleri, R. Hu, N. Yoshida, Session-Based Programming for Parallel Algorithms: Expressiveness and Performance, *Proc. PLACES 2009*, pages 17-29, 2009.

[16] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, N. Yoshida, Monitoring Networks through Multiparty Session Types, *Proc. FMOODS/FORTE 2013*, pages 50-65, 2013.

[17] R. Bruni, I. Lanese, H.Melgratti, E. Tuosto, Multiparty sessions in SOC, 2007.

[18] L. Cardelli, R. Mardare, Stochastic Pi-Calculus Revisited, 2010.

[19] M. Charalambides, P. Dinges, G. Agha, Parameterized Concurrent Multi-Party Session Types, *Proc. FOCLASA 2012*, pages 16-30, 2012.

[20] F. Chen, G. Rosu, MOP: An Efficient and Generic Runtime Verification Framework, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 569-588, 2007.

[21] A. Church, A formulation of the simple theory of types, *Journal of Symbolic Logic*, 5:56-68, 1940.

[22] A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, 1941.

[23] F. Ciocchetta, J. Hillston, Bio-PEPA: a Framework for the Modelling and Analysis of Biochemical Networks, *Theoretical Computer Science* 410 (33-34), pp. 3065-3084, 2009.

[24] H.B. Curry, R.Feys, W. Craig. *Combinatory logic,* vol. I, North Holland, 1958 (second edition, 1968).

[25] H.B. Curry, J.R. Hindley, J.P. Seldin. *Combinatory logic,* vol. II, North Holland, 1972.

[26] V. Forejt, M. Kwiatkowska, G. Norman and D. Parker. Automated Verification Techniques for Probabilistic Systems. In M. Bernardo and V. Issarny (editors), Formal Methods for Eternal Networked Software Systems (SFM'11), volume 6659 of *Lecture Notes in Computer Science*, pages 53-113, Springer. June 2011. [Tutorial paper on probabilistic model checking, focusing on verification techniques for MDPs, accompanied by case studies and examples for PRISM.]

[27] K. Honda, V.T. Vasconcelos, M. Kubo, Language Primitives and Type Discipline for Structured Communication-based Programming, 1999.

[28] K. Honda, N. Yoshida, M. Carbone, Multiparty Asynchronous Session Types, *Proc. POPL 2008*, pages 273-284, 2008.

[29] R. Hu, N. Yoshida, K. Honda, Session-Based Distributed Programming in Java, *Proc. ECOOP 2008*, pages 516-541, 2008.

[30] R. Hu, N. Yoshida, K. Honda, Type-Safe Communication in Java with Session Types, 2007.

[31] S. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, A. Z. Caldeira, Modular Session Types for Distributed Object-Oriented Programming, *Proc. POPL 2010*, pages 299-312, 2010.

[32] A. Igarashi, N. Kobayashi, A Generic Type System for the Pi-Calculus, *Proc. POPL 2009*.

[33] M. Kwiatkowska, G. Norman and D. Parker. Stochastic Model Checking. In M. Bernardo and J. Hillston (editors), Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07), volume 4486 of *Lecture Notes in Computer Science* (Tutorial Volume), pages 220-270, Springer. June 2007. [Tutorial paper covering probabilistic model checking of DTMCs/CTMCs and PRISM.]

[34] M. Kwiatkowska, G. Norman, J. Sproston and F. Wang. Symbolic Model Checking for Probabilistic Timed Automata. *Information and Computation*, 205(7), pages 1027-1077. July 2007.

[35] G. Norman, D. Parker and J. Sproston. Model Checking for Probabilistic Timed Automata. *Formal Methods in System Design*, 43(2), pages 164-190, Springer. September 2013. [Survey/tutorial paper on probabilistic timed automata and techniques for their verification, and two illustrative case studies.]

[36] G. Plotkin. Call-by-value, call-by-name, and the λ-calculus, *Theoretical Computer Science*, 1:125-159, 1975.

[37] C. Priami, Stochastic π-Calculus, *Computer Journal*, 38(7), 1995.

[38] G. Rosu et al, Publications Related to the K Framework, available from `http://www.kframework.org/index.php/Grigore_Rosu`.

[39] D. Scott, Domains for denotational semantics, In *Automata, languages and programming*, vol. 140 of Lecture Notes in Mathematics, pages 577-613. Springer-Verlag, 1982.

[40] N. Yoshida, P.-M. Denielou, A. Bejleri, R. Hu, Dependent Session Types for Evolving Multiparty Communication Topologies 2009.

[41] Haskell web page: `www.haskell.org`.

[42] Scala Programming Language: `https://www.scala-lang.org/`

[43] Project "BETTY: Behavioural Types for Reliable Large-Scale Software Systems", ICT COST Action IC1201 (2012-2016), `http://www.cost.eu/domains_actions/ict/Actions/IC1201`. `http://w3.cost.eu/fileadmin/domain_ files/ICT/Action_IC1201/mou/IC1201-e.pdf`.

[44] PEPA, `http://www.dcs.ed.ac.uk/pepa/`

[45] Bio-PEPA,
`http://www.biopepa.org/`

[46] Session Java,
`http://code.google.com/p/sessionj/`

[47] Scribble,
`http://www.jboss.org/scribble`
`http://scribble.github.com/`
`https://confluence.oceanobservatories.org/display/CIDev/Scribble+Core+Development`

[48] Run time verification, `https://runtimeverification.com/`,
RV-Match, `https://runtimeverification.com/match/`,
RV-Predict, `https://runtimeverification.com/predict/`,
RV-Monitor, `https://runtimeverification.com/monitor/`