Types and Programming Languages
Notes

toshinari tong

March 14, 2023

1 Mathematical Preliminaries

1.1 Sets, Relations, and Functions

A one-place relation on a set S is called a **predicate** on S, written $P(s)$ and regarding it as a function mapping elements of S to truth values.

A relation R on sets S, T is called a **partial function** if $(s, t_1) \in R$ and $(s, t_2) \in R$ implies $t_1 = t_2$.

If $dom(R) = S$, then R is a **total function** (or just function).

A partial function is said to be **defined** on an argument $s \in S$ if $s \in dom(R)$ and undefined otherwise. We write $f(x) \uparrow$ to mean f is undefined on x and $f(x) \downarrow$ to mean f is defined on x. We write $f(x) = fail$ when f returns a failure result (it is defined). Formally, it is actually a function $S \to T \cup \{fail\}$.

Suppose R is a binary relation on a set S and P is a predicate on S. We say that P is **preserved** by R if $s \; R \; s'$ and $P(s)$ implies $P(s')$.

1.2 Ordered Sets

R is **antisymmetric** if $s \; R \; t$ and $t \; R \; s$ implies $s = t$.

A reflexive and transitive relation is a **preorder**, written with symbols like $\leq$ or $\subseteq$. $s < t$ means $s \leq t \wedge s \neq t$.

A preorder that is antisymmetric is called a **partial order**. A partial order is a **total order** if $\forall s, t \in S$ either $s \leq t$ or $t \leq s$.

An element $j \in S$ is a **join** or **least upper bound** of s and t if $s \leq j$ and $t \leq j$ and $(\forall k \in S, s \leq k, t \leq k)\; j \leq k$.

An element $m \in S$ is a **meet** or **greatest lower bound** of s and t if $m \leq s$ and $m \leq t$ and $(\forall n \in S, n \leq s, n \leq t)\; n \leq m$.

The **reflexive closure** is the smallest reflexive relation R' that contains R. (If some other reflexive relation $R \subseteq R''$ then $R' \subseteq R''$)

The **transitive closure** is the smallest transitive relation that contains R, denoted R^+. The **reflexive and transitive closure** of R is denoted R^*.

EXERCISE 2.2.6

Suppose we are given a relation R on a set S. Define $R' = R \cup \{(s, s) \mid s \in S\}$. Show that R' is the reflexive closure of R.

Every reflexive relation that contains R must contain R and $\{(s, s) \mid s \in S\}$.

EXERCISE 2.2.7

Define the following sequence of sets of pairs: $R_0 = R, R_{i+1} = R_i \cup \{(s,u) \mid \exists t\ (s,t) \in R_i \wedge (t,u) \in R_i\}$ Show that $R^+ = \bigcup_i R_i$.

If a transitive relation contains R_i, then it must contain R_{i+1}. By induction, $R^+ = \bigcup_i R_i$.

EXERCISE 2.2.8

Suppose R is a binary relation on a set S and P is a predicate preserved by R. Show that P is also preserved by R^*.

$s\ R^*\ s$ and $P(s)$ implies $P(s)$; $t\ R^*\ v$, $t\ R\ u$, $u\ R\ v$ and $P(t)$ implies $P(v)$.

Suppose we have a preorder $\leq$ on a set S. A **decreasing chain** is a sequence such that $s_{i+1} < s_i$ for every i.

We say that $\leq$ is **well founded** if it contains no infinite decreasing chains.

1.3 Sequences

A **sequence** is written by listing its elements, separated by commas. Commas are used as both the "cons" operation for adding an element to either end of a sequence and as the "append" operation on sequences. The sequence of numbers from 1 to n is abbreviated $1..n$. The empty sequence is written as $\bullet$ or as a blank.

1.4 induction

AXIOM: principle of ordinary induction on natural numbers
Suppose P is a predicate on the natural numbers.
    If $P(0)$
    and, for all i, $P(i) \Longrightarrow P(i+1)$,
    then $P(n)$ holds for all n.

AXIOM: principle of complete induction on natural numbers
Suppose P is a predicate on the natural numbers.
    If, for each natural number n
    given $P(i)$ for all $i < n$ we can show $P(n)$
    then $P(n)$ holds for all n.

The **lexicographic order** on pairs of natural numbers is defined as follows: $(m,n) \leq (m',n')$ iff $m < m'$ or else $m = m'$ and $n \leq n'$.

AXIOM: principle of complete induction on natural numbers
Suppose P is a predicate on pairs of natural numbers.
    If, for each pair of natural numbers (m,n)
    given $P(m',n')$ for all $(m'n') < (m,n)$ we can show $P(m,n)$
    then $P(m,n)$ holds for all m,n.

The lexicographic induction principle is the basis for proofs by **nested infuction**, where some case of an inductive proof proceeds "by an inner induction".

2 Untyped Arithmetic Expressions

2.1 Introduction

```
t ::=
    true
    false
    if t then t else t
```

```
0
succ t
pred t
iszero t
```

The first line declares we are defining the set of **terms**, and each following line gives one alternative syntactic form for terms.

`t` is called a **metavariable**. It is not a variable of the **object language** - the simple programming language whose syntax we are currently describing - but rather of the **metalanguage** - the notation in which the description is given.

Here are some examples of programs with results:

```
iszero(pred(succ(0));

>>> true
```

For brevity, `succ(succ(succ(0)))` is written as `3`. Results of evaluation are always boolean constants or numbers. Such terms are called **values**.

2.2 Syntax

The set of **terms** is the smallest set $\mathcal{T}$ s.t.

1. $\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq \mathcal{T}$
2. if $\texttt{t}_1 \in \mathcal{T}$, then $\{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1\} \subseteq \mathcal{T}$
3. if $\texttt{t}_1 \in \mathcal{T}$,$\texttt{t}_2 \in \mathcal{T}$, and $\texttt{t}_3 \in \mathcal{T}$, then $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}$

We are defining $\mathcal{T}$ as a set of **trees**, not as a set of strings.

Rules with no premises are often called **axioms**, and the term **inference rule** is used generically to include both axioms and rules with one or more premises.

What we are calling inference rules are actually **rule schemas**, since their premises and conclusions may include metavariables. Each schema represents the infinite set of **concrete rules** that can be obtained.

2.3 Induction on Terms

$$
\begin{array}{ll}
\mathit{Consts}(\texttt{true}) & = \{\texttt{true}\} \\
\mathit{Consts}(\texttt{false}) & = \{\texttt{false}\} \\
\mathit{Consts}(\texttt{0}) & = \{\texttt{0}\} \\
\mathit{Consts}(\texttt{succ t}_1) & = \mathit{Consts}(\texttt{t}_1) \\
\mathit{Consts}(\texttt{pred t}_1) & = \mathit{Consts}(\texttt{t}_1) \\
\mathit{Consts}(\texttt{iszero t}_1) & = \mathit{Consts}(\texttt{t}_1) \\
\mathit{Consts}(\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3) & = \mathit{Consts}(\texttt{t}_1) \cup \mathit{Consts}(\texttt{t}_2) \cup \mathit{Consts}(\texttt{t}_3)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{size}(\texttt{true}) & = 1 \\
\mathit{size}(\texttt{false}) & = 1 \\
\mathit{size}(\texttt{0}) & = 1 \\
\mathit{size}(\texttt{succ t}_1) & = \mathit{size}(\texttt{t}_1) + 1 \\
\mathit{size}(\texttt{pred t}_1) & = \mathit{size}(\texttt{t}_1) + 1 \\
\mathit{size}(\texttt{iszero t}_1) & = \mathit{size}(\texttt{t}_1) + 1 \\
\mathit{size}(\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3) & = \mathit{size}(\texttt{t}_1) + \mathit{size}(\texttt{t}_2) + \mathit{size}(\texttt{t}_3) + 1
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{depth}(\texttt{true}) & = & 1 \\
\mathit{depth}(\texttt{false}) & = & 1 \\
\mathit{depth}(\texttt{0}) & = & 1 \\
\mathit{depth}(\texttt{succ t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{pred t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{iszero t}_1) & = & \mathit{depth}(\texttt{t}_1) + 1 \\
\mathit{depth}(\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3) & = & \max(\mathit{depth}(\texttt{t}_1), \mathit{depth}(\texttt{t}_2), \mathit{depth}(\texttt{t}_3)) + 1
\end{array}
$$

Principles of induction on terms

Suppose P is a predicate on terms.
If, for each term $\texttt{s}$,
    given $P(\texttt{r})$ for all (choose 1 below)
        [$\texttt{r}$ such that $\mathit{depth}(\texttt{r}) < \mathit{depth}(\texttt{s})$]
        [$\texttt{r}$ such that $\mathit{size}(\texttt{r}) < \mathit{size}(\texttt{s})$]
        [immediate subterms $\texttt{r}$ of $\texttt{s}$]
    we can show $P(\texttt{s})$.
then $P(\texttt{s})$ holds for all $\texttt{s}$.

The choice of induction principle is determined by which one leads to a simpler structure for the proof at hand-it is inter-derivable, and commonly structural induction is used wherever possible.

2.4 Semantic Styles

We need a precise definition of how terms are evaluated-the **semantics**:

1. **Operational semantics** specifies the behaviour of a programming language by defining a simple **abstract machine** for it. A **state** of the machine is just a term, and that machine's behaviour is defined by a **transition function** that for each state either gives the next state by performing a step of simplification (**small-step** style / **structural operational semantics**) on the term or declares that the machine has halted. The **meaning** of a term `t` can be taken to be the final state that the machine reaches. It is sometimes useful to give 2 or more different operational semantics for a single language - some more abstract and others closer to the strictires manipulated by an actual interpreter or compiler.

2. **Denotational semantics** takes a more abstract view of taking the meaning of a term to be some mathematical object. Giving denotational semantics for a language consists of finding a collection of **semantic domains** and then defining an **interpretation function** mapping terms into elements of these domains. The search for appropriate semantic domains for modeling various language features is known as **domain theory**. Denotational semantics highlights the essential concepts of the language, can derive powerful laws for reasoning about program behaviours such as laws for proving two programs have exactly the same behaviour, and it is evident that some things are impossible in a language by the properties of the semantic domains.

3. **Axiomatic semantics** is a more direct approach: it takes the laws themselves as the definition of the language. They focus acctention on the process of reasoning about programs, and has given computer science powerful ideas such as **invariants**.

The bête noire of denotational semantics turned out to be the treatment of nondeterminism and concurrency; for axiomatic semantics, it was procedures.

Operational semantics is used exlusively in this book.

2.5 Evaluation

𝔹 *(untyped)*

Syntax

t	::=		*terms:*
		`true`	*constant true*
		`false`	*constant false*
		`if t then t else t`	*conditional*
v	::=		*values:*
		`true`	*true value*
		`false`	*false value*

Evaluation $t \longrightarrow t'$

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-If)}$$

The left-hand column is a grammar defining 2 sets of expressions; the right-hand column defines an **evaluation relation** on terms, written $t \to t'$ and pronounced "t evaluates to t' in one step". The first 2 inference rules have no permises; for the third evaluation rule, it means that a machine in state $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$ can take a step to $\texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3$ if another machine whose state is t_1 can take a step to state t_1'.

What the rules don't say is just as important: The constants `true` and `false` doesn't evaluate to anything; the term `if true then (if false then false else false) else true` doesn't evaluate to `if true then false else true`, as our only choice is to evaluate the outer conditional first. This interplay between rules determines a particular **evaluation strategy**: to evaluate a conditional first evaluate its guard; thus the first 2 rules is sometimes referred to as **computation rules** and the last one as a **congruence rule**.

An **instance** of an inference rule is obtained by consistently replacing each metavariable by the same term. A rule is **satisfied** by a relation if for each instance of the rule either the conclusion is in the relation or one of the premises is not. The **one-step evaluation** relation $\to$ is the smallest binary relation on terms satisfying the rules in the figure. When the pair (t, t') is in the evaluation relation, we say that "the evaluation **statement** or **judgement** $t \to t'$ is **derivable**". The derivability of a given statement can be justified by exhibiting a **derivation tree** whose leaves are labeled with rules without premises and whos internal nodes are labeled with inference rules.

Determinacy of one-step evaluation If $t \to t'$ and $t \to t''$, then $t' = t''$.

Proof. By induction on a derivation of $t \to t'$. If the last rule used in the derivation of $t \to t'$ is E-IFTRUE, then we know t has the form $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$ where $t_1 = \texttt{true}$. It is obvious that the last rule in the derivation of $t \to t''$ cannot be E-IFFALSE, since we cannot have both $t_1 = \texttt{true}$ and $t_1 = \texttt{false}$. The last rule in the second derivation cannot be E-IF either, since the premise of this rule demands that $t_1 \to t_1'$, but we have observed that `true` does not evaluate to anything. It follows that the last rule can only be E-IFTRUE, and that $t' = t''$. Similarly, if the last rule used in the first derivation is E-IFFALSE, then the last rule in the second derivation must be the same and the result is immediate.

If the last rule used in the derivation of $t \to t'$ is E-IF, then t has the form $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$ where $t_1 \to t_1'$ for some t_1'. By the same reasoning as above, the last rule in the derivation of $t \to t''$ can only be E-IF, then t has the form $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$ where $t_1 \to t_1''$ for some t_1''. Now the induction hypothesis applies, yielding $t_1' = t_1''$, which implies that $t' = t''$.

A term t is in **normal form** if no evaluation rule applies to it.

Theorem every value is in normal form.

Theorem In the current system, if t is in normal form, then t is a value.

Proof. Suppose t is not a value. It is easy to show by structural induction on t that it is not

a normal form.

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

Uniqueness of normal forms If $\mathtt{t} \rightarrow^* \mathtt{u}$ and $\mathtt{t} \rightarrow^* \mathtt{u}'$, where $\mathtt{u}$ and $\mathtt{u}'$ are both normal forms, then $\mathtt{u} = \mathtt{u}'$.

Proof. Corollary of the determinacy of single-step evaluation.

The last property of evaluation we consider is the fact that every term can be evaluated to a value. Most termination proofs in computer science has the same form: we choose some well-founded set S and a function f mapping machine states into S. Next, we show that whenever a machine state $\mathtt{t}$ can take a step to $\mathtt{t}$', $f(\mathtt{t}') < f(\mathtt{t})$. Since S is well-founded, there can be no infinite decreasing chain, and hance no infinite evaluation sequence. f is often called a **termination measure**.

Termination of evaluation For every term $\mathtt{t}$ there is some normal form $\mathtt{t}'$ s.t. $\mathtt{t} \rightarrow^* \mathtt{t}'$.

Proof. Size of the term is a termination measure.

𝔹 ℕ *(untyped)* *Extends* **B** *(3-1)*

New syntactic forms

t	::=	...	*terms:*
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*
v	::=	...	*values:*
		nv	*numeric value*
nv	::=		*numeric values:*
		0	*zero value*
		succ nv	*successor value*

New evaluation rules

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{succ}\ \mathtt{t}_1 \longrightarrow \mathtt{succ}\ \mathtt{t}_1'} \quad \text{(E-Succ)}$$

$$\mathtt{pred}\ 0 \longrightarrow 0 \quad \text{(E-PredZero)}$$

$$\mathtt{pred}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{nv}_1 \quad \text{(E-PredSucc)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{pred}\ \mathtt{t}_1 \longrightarrow \mathtt{pred}\ \mathtt{t}_1'} \quad \text{(E-Pred)}$$

$$\mathtt{iszero}\ 0 \longrightarrow \mathtt{true} \quad \text{(E-IszeroZero)}$$

$$\mathtt{iszero}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{iszero}\ \mathtt{t}_1 \longrightarrow \mathtt{iszero}\ \mathtt{t}_1'} \quad \text{(E-IsZero)}$$

$\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

You cannot use E-PREDSUCC to evaluate `pred(succ(pred 0))` to `pred 0`.

EXERCISE 3.5.14

Show that Determinacy of one-step evaluation is also valid for evaluation relation on arithmetic expressions.

Last is E-SUCC: unique form, second derivation must be E-SUCC as well, induction hypothesis;
Last is E-PREDZERO: cannot be E-PRED because `0` doesn't evaluate to anything, second derivation must be E-PREDZERO;
Last is E-PREDSUCC: cannot be E-PRED because `succ` $\mathtt{nv}_1$ doesn't evaluate, second derivation must be E-PREDSUCC;
Last is E-PRED: unique form, second derivation must be E-PRED, induction hypothesis;
Last is E-ISZEROZERO: cannot be E-ISZERO because ..., second derivation must be same;
Last is E-ISZEROSUCC: cannot be E-ISZERO because ..., second derivation must be same;
Last is E-ISZERO: unique form, second derivation must be same, induction hypothesis.

A closed term is **stuck** if it is in normal form but not a value (e.g. `succ false`)

EXERCISE 3.5.16

EXERCISE [RECOMMENDED, ⋆⋆⋆]: A different way of formalizing meaningless states of the abstract machine is to introduce a new term called `wrong` and augment the operational semantics with rules that explicitly generate `wrong` in all the situations where the present semantics gets stuck. To do this in detail, we introduce two new syntactic categories

`badnat ::=`	*non-numeric normal forms:*
`wrong`	*run-time error*
`true`	*constant true*
`false`	*constant false*
`badbool ::=`	*non-boolean normal forms:*
`wrong`	*run-time error*
`nv`	*numeric value*

and we augment the evaluation relation with the following rules:

$$\texttt{if badbool then } t_1 \texttt{ else } t_2 \longrightarrow \texttt{wrong} \qquad \text{(E-IF-WRONG)}$$

$$\texttt{succ badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-SUCC-WRONG)}$$

$$\texttt{pred badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-PRED-WRONG)}$$

$$\texttt{iszero badnat} \longrightarrow \texttt{wrong} \qquad \text{(E-ISZERO-WRONG)}$$

Show that these two treatments of run-time errors agree.

Evaluates to `wrong` in the second treatment $\iff$ stuck in the first treatment.

EXERCISE 3.5.17

An alternative style of operational semantics is **big-step** semantics, formulating the notion of "this term evaluates to that final value", written $t \Downarrow v$.

$$v \Downarrow v \qquad \text{(B-VALUE)}$$

$$\frac{t_1 \Downarrow \texttt{true} \quad t_2 \Downarrow v_2}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_2} \qquad \text{(B-IFTRUE)}$$

$$\frac{t_1 \Downarrow \texttt{false} \quad t_3 \Downarrow v_3}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_3} \qquad \text{(B-IFFALSE)}$$

$$\frac{t_1 \Downarrow nv_1}{\texttt{succ } t_1 \Downarrow \texttt{succ } nv_1} \qquad \text{(B-SUCC)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{pred } t_1 \Downarrow 0} \qquad \text{(B-PREDZERO)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{pred } t_1 \Downarrow nv_1} \qquad \text{(B-PREDSUCC)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{iszero } t_1 \Downarrow \texttt{true}} \qquad \text{(B-ISZEROZERO)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{iszero } t_1 \Downarrow \texttt{false}} \qquad \text{(B-ISZEROSUCC)}$$

Show that $t \to^* v$ iff $t \Downarrow v$.

Apply structural induction on B-rules to show $t \Downarrow v$ implies $t \to^* v$;

$$t \to^* t \qquad \frac{t \to t'}{t \to^* t'} \qquad \frac{t \to^* t' \quad t' \to^* t''}{t \to^* t''}$$

idk how to do the other way

3 An ML Implementation of Arithmetic Expressions

```
type term =
TmTrue of info
| TmFalse of info
| TmIf of info * term * term * term
| TmZero of info
| TmSucc of info * term
| TmPred of info * term
| TmIsZero of info * term
(*Each abstract syntax tree node is annotated with a value of type info,
*which describes what character position in which source file the node originated.*)
let rec isnumericalval t = match t with
TmZero(_) -> true
| TmSucc(_,t1) -> isnumericval t1
| _ -> false
(*_ matches anything, info needs to be matched; rec indicates it is recursive*)
let rec isval t = match t with
TmTrue(_) -> true
| TmFalse(_) -> true
| t when isnumericval t -> true
| _ -> false
exception NoRuleApplies
let rec eval1 t = match t with
TmIf(_,TmTrue(_),t2,t3) -> t2
| TmIf(_,TmFalse(_),t2,t3) -> t3
| TmIf(fi,t1,t2,t3) -> let t1' = eval1 t1 in TmIf(fi,t1',t2,t3)
| TmSucc(fi,t1) -> let t1' = eval1 t1 in TmSucc(fi,t1')
| TmPred(_,TmZero(_)) -> TmZero(dummyinfo)
| TmPred(_,TmSucc(_,nv1)) when (isnumericval nv1) -> nv1
| TmPred(fi,t1) -> let t1' = eval1 t1 in TmPred(fi,t1')
| TmIsZero(_,TmZero(_)) -> TmTrue(dummyinfo)
| TmIsZero(_,TmSucc(_,nv1)) when (isnumericval nv1) -> TmFalse(dummyinfo)
| TmIsZero(fi,t1) -> let t1' = eval1 t1 in TmIsZero(fi,t1')
| _ -> raise NoRuleApplies
(*let ...  in ...  is a way to create local variables in the second expression*)
let rec eval t =
try let t' = eval1 t in eval t'
with NoRuleApplies -> t
```

4 The Untyped Lambda-Calculus

4.1 Basics

t ::=	terms:
x	variable
λx.t	abstraction
t t	application

The **concrete syntax** or surface syntax of the language refers to the strings of characers; the **abstract syntax** refers to an internal representation of programs as labeled trees (ASTs). We avoid writing too many parentheses: application associates to the left and bodies of abstractions extend as far right as possible. λx.λy.x y x stands for the same tree as λx.(λy.((x y) x)).

Another subtlety is the use of metavariable for terms t,s,u, metavariable for variables x,y,z and object-language variables x,y,z.

An occurence of the variable x is said to be **bound** by abstraction λx.t when it occurs in the body t. (λx is a **binder** whose scope is t.) An occurence of x is **free** if it is not bound by an enclosing abstraction. A term with no free variables is **closed**; closed terms are also called **combinators**. The simplest combinator is identity id = λx.x;

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

where $[x \mapsto t_2]t_{12}$ means "the term obtained by replacing all free occurences of x in t_{12} by t_2". Following Church, a term of the form $(\lambda x.t_{12})t_2 \rightarrow$ [x is called a **redex** (reducible expression) and the operation of rewriting a redex according to the above rule is **beta-reduction**.

- **full beta-reduction**: any redex may be reduced ay any time; except this strategy in all of them the evaluation relation is a partial function: t evaluates in one step to at most one term t'.
- **normal order**: the leftmost, outermost redex is always reduced first
- **call by name**: same as normal order, but allow no reductions inside abstractions
- **call by need**: used by Haskell; instead of re-evaluating an argument each time it is used, overwrites all occurences of the argument with its value the first time it is evaluated, avoiding subsequent re-evaluation. but this demands a reduction relation on abstract syntax graphs (why?) rather than syntax trees
- **call by value**: used by most languages; only outermost redexes whose right-hand side has already been reduced to a value are reduced

  $$\text{id}(\text{id}(\lambda z.\text{id } z)) \rightarrow \text{id}(\lambda z.\text{id } z) \rightarrow \lambda z.\text{id } z$$

  (lambda-abstractions are values) The call-by-value strategy is **strict** in the sense that the arguments to functions are always evaluated whether or not they are used by the body of the function. **Non-strict**, or lazy strategies such as call-by-name and call-by-need evaluate only the arguments that are actually used.

The choice of evaluation strategy actually makes little difference when discussing type systems.

4.2 Programming in the Lambda-Calculus

Multi-argument functions: use **higher-order functions** that yield functions as results: f = λx.λy.s; f v w $\rightarrow^*$ [y ↦ w][x ↦ v]s This transformation is called **currying** in honor of Haskell Curry.

Church Booleans:
tru = λt.λf.t;
fls = λt.λf.f;
It basically chooses between first and second input.
test = λl.λm.λn.l m n;

`test tru v w` $\to$ `v` and `test fls v w` $\to$ `w`
`and` $= \lambda$`b.`λ`c.b c fls;`
`and tru tru` $\to$ `tru` <u>`tru`</u> `fls` $\to$ `tru`
`and tru fls` $\to$ `tru` <u>`fls`</u> `fls` $\to$ `fls`
`and fls tru` $\to$ `fls tru` <u>`fls`</u> $\to$ `fls`
`and fls fls` $\to$ `fls fls` <u>`fls`</u> $\to$ `fls`

EXERCISE 5.2.1

Define `or` and `not` functions.

`or` $= \lambda$`b.`λ`c.b tru c;`
`or tru tru` $\to$ `tru` <u>`tru`</u> `tru` $\to$ `tru`
`or tru fls` $\to$ `tru` <u>`tru`</u> `fls` $\to$ `tru`
`or fls tru` $\to$ `fls tru` <u>`tru`</u> $\to$ `tru`
`or fls fls` $\to$ `fls tru` <u>`fls`</u> $\to$ `fls`
`not` $= \lambda$`b.b fls tru;`

Using booleans, we can encode pairs of values as terms:
`pair` $= \lambda$`f.`λ`s.`λ`b.b f s;`
`fst` $= \lambda$`p.p tru;`
`snd` $= \lambda$`p.p fls;`
`fst(pair v w)` $\to^*$ `v`

Define the Church numerals as:
$c_0 = \lambda s.\lambda z.z;$
$c_1 = \lambda s.\lambda z.s\ z;$
$c_2 = \lambda s.\lambda z.s\ (s\ z);$
$c_3 = \lambda s.\lambda z.s\ (s\ (s\ z));$
`etc.`
`scc` $= \lambda$`n.`λ`s.`λ`z.s (n s z);`
`scc` $c_n \to \lambda s.\lambda z.s\ (c_n\ s\ z) \to \lambda s.\lambda z.s\ ((\lambda s_n.\lambda z_n.s_n\ (...))\ \ s\ z)$

EXERCISE 5.2.2

Find another way to define the successor function.

`scc` $= \lambda$`n.`λ`s.`λ`z.n s (s z);`

`plus` $= \lambda$`m.`λ`n.`λ`s.`λ`z.m s (n s z);`
`times` $= \lambda$`m.`λ`n.m (plus n)` c_0`;`

EXERCISE 5.2.3

Is it possible to define multiplication on Church numerals without using `plus`?

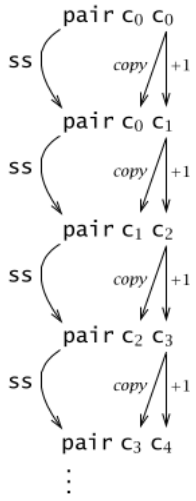
i dont know

EXERCISE 5.2.4

Define a term for raising one number to the power of another.

`power` $= \lambda$`m.`λ`n.n (times m)` c_0`;`

`iszro` $= \lambda$`n.m (`λ`x.fls) tru;`



zz = pair c_0 c_0;
ss = λp.pair (snd p) (plus c_1 (snd p));
prd = λm.fst (m ss zz);

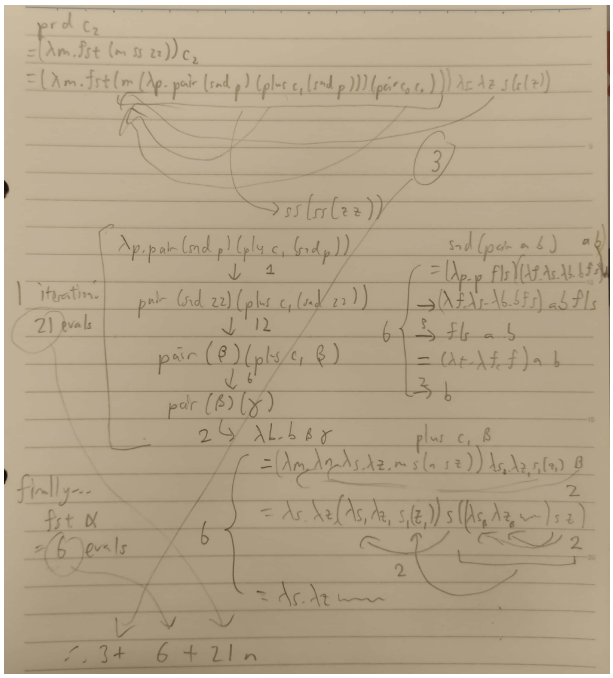
EXERCISE 5.2.5

Use `prd` to define a subtraction function.

sub = λm.λn.n prd m

EXERCISE 5.2.6

Approximately how many steps of evaluation are required to calculate `prd` c_n?



$21n + 9$

EXERCISE 5.2.7

Write a function `equal` that tests two numbers for equality and returns a Church boolean.

λm.λn.and (n prd m (λa.fls) tru) (m prd n (λa.fls) tru)

EXERCISE 5.2.8

A list can be represented in the lambda-calculus by its `fold` function. For example, the list `[x,y,z]` becomes a function that takes two arguments `c` and `n` and returns `c x (c y (c z n)))`. What would the representation of `nil` be? `n`.

Write a function `cons` that takes an element `h` and a list `t` and returns a list formed by prepending `h` to `t`. `cons` = λh.λt.λc.λn.c h (t c n);

Write `isnil` and `head` functions, each taking a list parameter. `isnil` = λt.t (λa.λb.fls) tru;
`head` = λt.t tru fls

Finally, write a `tail` function for this representation of lists.
`tail` = λt.

Enriching the calculus: we will use symbol λ for the pure lambda-calculus and λ**NB** for the enriched system with booleans and arithmetic expressions from section 2.

```
realbool = λb.b true false;
churchbool = λb.if b then tru else fls;
realeq = λm.λn.(equal m n) true false;
realnat = λm.m (λx.succ x) 0;
```

The reason primitive boolean and numbers come in handy is because of evaluation order. For example, `scc` c_1 doesn't actually evaluate to c_2:

`scc` $c_1 \rightarrow$ λs.λz.s ((λs'.λz'.s' z') s z))

By the rules of call-by-value evaluation we cannot reduce the redex. It is **behaviourally equivalent** to c_2, but the leftover computation makes it difficult to check if our functions are behaving the way we expect it to.

`realnat (times` c_2 c_2`)` $\rightarrow$ 4

This conversion has the effect of supplying the two extra arguments that `times` c_2 c_2 is waiting for, forcing all of the latent computation in its body.