Types in Programming Languages

Alexey Chentsov

Contents

Introduction

Let us start by considering several systems.

1. Bit strings: computer data (in general sense).
2. S-expressions: recursive structure loosely based on lists (ordered pairs and atoms)
3. λ-expressions.
4. Sets: unstructured collections of abstract elements.

They can be thought as universes. For flexibility, universality and simplicity they are untyped or actually of one type. For instance, the main difference between the objects in the first case is the length of the bit string.

On the other hand from practical standpoint it is convenient to distinguish different sorts of objects due to manipulations that are allowed with them or their behavior. For instance, operations over integers and doubles are performed by different units. Then there should be a *representation*[1] of these entities in universal form. But different kinds of objects might have the same representations. Examples:

1. Integers, floating point numbers, strings, instructions (code) look the same.
2. Both program code and data (in lisp) is represented as S-expression.
3. In pure λ-expressions everything is a function.
4. Sets are a universal mathematical language that allows building very complex mathematical structures.

[1]Ukrainian equivalent: *представлення*

Types are not only the matter of convenience, for naive (untyped) set theory a number of paradoxes, e.g. Russel paradox, arise and one of the ways to avoid them is to introduce two types: proper classes and sets.

So in general the type system is used to organize and systematize the object universe.

For typed universe many constructions lack the meaning. For instance, very few bit strings can represent a meaningful program, neither does S-expression `(1 'three +)`, concatenation of word `'one'` and negative number -0.5 doesn't make sense as well.

Type systems make the programmer more aware of the conventional rules regarding valid manipulation with different entities.

In the case of programming languages type system is used to avoid constructions that don't have meaning. As Luca Cardelli puts it[1]

> Type is a suit of armor that protects the representation from arbitrary or unintended use.

Basically, using types is similar to using units of measurements in physics. Only commensurable quantities might be added, subtracted, compared. It can be used as sanity check for expressions (dimensional homogeneity combined with factor-label method).

There is a loose definition of *type system* [2, p. 1]

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

Type systems are popular and most established lightweight formal methods.

Here we come to instrumental essence of the type system. It allows us to avoid special kind of errors *runtime type errors*.

"Classifying according to kind of values" means that type system may be regarded as calculating approximation to the runtime behaviour. In this respect *type checking*[2] is *conservative*. Type system makes sure that runtime type errors are *absent* but it cannot check if they are *present*. E.g.

`if` <*complex test*> `then 5 else` <*type error*>

Even though the condition might always be true and else branch would be never executed the program is rejected by type system (due to limitations). This situation is similar to banker's algorithm when state is safe only when absence of deadlock could be guaranteed.

[2]Ukrainian equivalent: *перевірка типів*

Dynamic vs static type checking We can talk about *manifest* and *latent* properties or attributes. Attributes known at compile time are called manifest and known at run-time are called latent. For instance, the Lvalue (i.e. the variable) in, say, Java has certain known type T and its Rvalue (i.e. content) has unknown type which is a subtype of T.

Remark. The distinction between manifest and latent attributes is not clear cut and subject to change with evolution of hardware and programming languages. For instance, in C++ expression that previously were determined on execution now can be computed by compile time functions (const-expressions).

Based on this we can do *static* type checking[3] and *dynamic* type checking[4]. Type requirements explicitly placed in the code by programmer are called *type annotations*. These could vary from quite modest (Haskell, ML) to very redundant (C, Pascal). In the case of very extensive type annotations type checking effectively turns into proof checking.

Dynamic type checking[5] requires metainformation about typesystem in runtime (RTTI).

Remark. Luca Cardelli distinguishes *explicitly typed*, *implicitly typed* and *untyped* languages[3]. Type system is the static type system. Untyped languages provide safety with *dynamic checks*. Typed languages use both static tests and dynamic tests (based on the static type system) to provide safety.

The whole type checking can be performed at runtime. However modern programming languages prefer static type checking for two reasons:

- early "availability": mistakes are found before program execution;
- more effective code: dynamic type checking involves extra work in runtime.

On the other hand, static type checking might interfere with sound programming techniques that are incompatible with early binding [1].

Remark. Actually, if CPU were equipped with type determination facilities the performance would have been no longer an issue for dynamic type checking.

[3]Ukrainian equivalent: *статична* перевірка типів
[4]Ukrainian equivalent: *динамічна* перевірка типів
[5]sometimes called *dynamic typing*, arguably a misnomer

Language safety The main motif for type system as formulated by Robin Milner is "Well-typed programs cannot go wrong". It is closely related with notion of language safety. This term is even more contentious than type system. There are several definitions of *language safety*. According to Benjamine Pierce

> A safe language is one that protects its own abstractions [2].

This means it is not necessary to keep in mind all sorts of low-level implementation details, because these abstractions are consistently and completely defined in terms of their own properties [4].

Cardelli distinguished two types of errors. The ones that are detected right away on the spot are called *trapped* and the ones that go unnoticed and then lead to arbitrary behavior called *untrapped*. An example of the latter is stack overflow. Then

> A safe language prevents untrapped run-time errors [3].

Finally, there is a definition

> A safe language is completely defined by its programmer's manual [2].

It means that behavior of any program can be predicted fully based on the language manual. For instance, correctness of some programming techniques of unsafe language might depend on calling convention or stack layout details.

Functions of type system The most important advantage of having a typesystem is code maintainability [3].

Besides error detection, additionally type systems (type annotations) serve other purposes:

- abstraction
- documentation
- efficiency.

It provides better interfaces between the modules (partial contract) which form the basis for more *abstract* design. It *documents* code and it is always up-to-date due to check on every compilation. Statically checked programs have better *efficiency* because of the absence of dynamic checks and optimization based on properties of the types. The latter is mostly related to base types: hard-coding floating point operations, loop unrolling etc. Today, high-performance compilers heavily rely on information gathered by typechecker. One of the strategic goals in this direction is to allow optimizations for user-defined types similar to base types.

Note on supplementary code The textbook [2] is accompanied with implementations of all discussed systems (languages) starting with untyped boolean arithmetic and finishing with system F extended with subtyping. Supplementary systems are provided in OCaml programming language. They can be considered as (formal) language processors. Therefor parser is one of their key components. Choice of programming language is due to author's personal preferences but also due to native support of inductive types in OCaml which simplifies construction and representation of *abstract syntax tree*[6].

Provided systems are designed to operation in batch mode. This means they expect input in the form of user provided file. This input contains expressions ("lines of code") of the given systems. The system then forms the result of expression evaluation on the standard output. For some tasks batch mode is not the most convenient. To experiment with the system the interactive mode is preferred. Also access to the source code allows system extension. Some system are already enriched with the elements of the others. For instance, basic arithmetic is embedded in the most of the system. These extensions implies some tweaks to the parser and maybe evaluation component of the system. Particularly it is appropriate for the varieties of lambda calculus implementation to use interactive mode and add syntactic sugar to allow conventional function definition syntax. This is utilized in this particular course.

[6]Ukrainian equivalent: *абстрактне синтаксичне дерево*

Chapter 1

Simple systems

In order to discuss questions concerning the types in (modern) programming languages one has to present the formal description of systems used. Since immediate exploration of advanced programming constructions could be too much of a leap in complexity we would start from simple systems and gradually move to more modern counterparts. It is important to mention that one of the systems provides the basis for many programming languages and thus plays very important role in theoretical computer science.

1.1 Untyped arithmetics

1.1.1 Language and inductive definitions

Description of the (computing) system begins with the formal description of the language. We have several ways to do this. Most conventional is to use grammars and Backus–Naur form (or its variation).

`t`	::=	*terms:*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
	`0`	*constant zero*
	`succ t`	*successor*
	`pred t`	*predecessor*
	`iszero t`	*zero test*

In the l.h.s. we specify that this defines the *terms* of the language. In the r.h.s. we have *alternatives*. In this construction `t` in the r.h.s. can be

substituted with any correct term of the language. Since it is not the part of the language it is a *meta*-variable. We see that more complex terms are based on simple terms (*subterms*). This is an example of inductive definition. In mathematics this construction is introduced using *inductively defined sets*. Here is the definition

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 $\texttt{t}_1 \in \mathcal{T}$, $\texttt{t}_2 \in \mathcal{T}$, $\texttt{t}_3 \in \mathcal{T}$, then $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}$.

First point gives *base clause* (*basis step*). The second and third points are *inductive clause* (*recursive step*). Inductive definition also has implicit part known as *extremal clause*[1] which states that "only elements satisfying base and inductive clause are in the set" or "nothing else is an element of the set". It guarantees uniqueness of the set and thus the correctness of the definition. Extremal clause is usually omitted.

Preferable way to formulate this construction in theoretical computer science is based on *inference rules*. Inference rule states that if premises are true then the conclusion is too. Inference rule are written down as fractions where numerator is premises and denominator is conclusion of the rule. The rules with empty premises are axioms. This notation is often used in natural deduction systems[2]. The previous definition would take the form

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

$$\frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{succ t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{pred t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{iszero t}_1 \in \mathcal{T}}$$

$$\frac{\texttt{t}_1 \in \mathcal{T} \quad \texttt{t}_2 \in \mathcal{T} \quad \texttt{t}_3 \in \mathcal{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}}$$

Finally, there is a non-inductive way to define the terms.

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= \{\texttt{true}, \texttt{false}, \texttt{0}\} \\ &\cup \{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1 \mid \texttt{t}_1 \in S_i\} \\ &\cup \{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in S_i\}. \end{aligned}$$

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

[1] rarely, *closure clause*

[2] The notation is attributed to Gentzen

Proposition 1.1. *Family* $\{S_i\}_{i \in N}$ *form a cumulative hierarchy, that is*

$$\forall i \in N. S_i \subseteq S_{i+1}.$$

Proposition 1.2. $\mathcal{T} = S$.

Proof. It is enough to prove the characteristic properties of $\mathcal{T}$ for S. First, let us check that conditions 1–3 hold for S. By definition $\{\texttt{true}, \texttt{false}, \texttt{0}\} = S_1 \subseteq S$, i.e. 1 holds. Suppose $\texttt{t} \in S$. Then

$$\texttt{t} \in S_i \Rightarrow \texttt{succ t}, \texttt{pred t}, \texttt{iszero t} \in S_{i+1} \Rightarrow \texttt{succ t}, \texttt{pred t}, \texttt{iszero t} \in S.$$

Finally if $\texttt{t}_{1,2,3} \in S$ then

$$\begin{aligned} \texttt{t}_k \in S_{i_k} &\Rightarrow \texttt{t}_k \in S_m \wedge m = \max\{i_1, i_2, i_3\} \Rightarrow \texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in S_{m+1} \\ &\Rightarrow \texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in S. \end{aligned}$$

Now suppose that S' satisfies conditions 1–3 (of $\mathcal{T}$). Let us prove that $S \subseteq S'$. It is enough to show that $S_i \subseteq S'$ for all i. We will show that by induction. Obviously $S_0 = \emptyset \subseteq S'$. Let $\texttt{t} \in S_{i+1}$ for some i. We have to consider several (general) cases. Case 1: $\texttt{t} \in S_1$. Then $\texttt{t} \in S$ by condition 1. Case 2: $\texttt{t} = \texttt{pred s}$, $\texttt{s} \in S_i$. Then by inductive hypothesis $\texttt{s} \in S'$. As a consequence $\texttt{t} \in S'$ by property 2. Case 3: $\texttt{t} = \texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3$, $\texttt{t}_i \in S_i$. Then by inductive hypothesis $\texttt{t}_i \in S'$. As a consequence $\texttt{t} \in S'$ by property 3. □

This proposition reaffirms that terms of the language $\mathcal{T}$ have limited number of forms: term is either (1) a constant, (2) has the form `succ t`, `pred t`, `iszero t`, or (3) a conditional $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3$.

1.1.2 Induction on terms

The characterization of terms of $\mathcal{T}$ allows special definitions and reasoning on them. First, we introduce *inductive definitions* of functions. Let us define three functions. *consts*(t) is a multiset of constants used in t:

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

where operation $\cup$ in the last line is union of multisets. Size of the term (number of node in abstract syntax tree of $\mathtt{t}$):

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

Depth of the term $\mathtt{t}$:

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

Second, let us consider new inductive methods of reason that suits our inductively defined sets.

Theorem 1.3 (Induction on terms)**.** *Suppose P is a predicate on terms.*
Induction on depth:
If, for each $\mathtt{s}$,
given $P(\mathtt{r})$ for all $\mathtt{r}$ such that $\mathit{depth}(\mathtt{r}) < \mathit{depth}(\mathtt{s})$
we can show $P(\mathtt{s})$,
then $P(\mathtt{s})$ holds for all $\mathtt{s}$.
Induction on size:
If, for each $\mathtt{s}$,
given $P(\mathtt{r})$ for all $\mathtt{r}$ such that $\mathit{size}(\mathtt{r}) < \mathit{size}(\mathtt{s})$ we can show $P(\mathtt{s})$,
then $P(\mathtt{s})$ holds for all $\mathtt{s}$.
Structural induction:
If, for each $\mathtt{s}$,
given $P(\mathtt{r})$ for all immediate subterms $\mathtt{r}$ of $\mathtt{s}$ we can show $P(\mathtt{s})$,
then $P(\mathtt{s})$ holds for all $\mathtt{s}$.

Structural induction was introduced by Burstall (1969) [5]. This technique was further popularized in functional programming. We demonstrate its application on the following

Proposition 1.4. $|consts(\mathtt{t})| \leq size(\mathtt{t})$.

Proof. By structural induction (by induction on $\mathtt{t}$). We have to consider three cases.
Case: $\mathtt{t} \in \{\mathtt{true}, \mathtt{false}, \mathtt{0}\}$
By definition $|consts(\mathtt{t})| = 1 \leq 1 = size(\mathtt{t})$.
Case: $\mathtt{t} = \mathtt{succ}\ \mathtt{t}_1$, $\mathtt{pred}\ \mathtt{t}_1$, or $\mathtt{iszero}\ \mathtt{t}_1$
Then $|consts(\mathtt{succ}\ \mathtt{t}_1)| = |consts(\mathtt{t}_1)| \leq size(\mathtt{t}_1) < size(\mathtt{t}_1) + 1 = size(\mathtt{t})$, where induction hypothesis for $\mathtt{t}_1$ is used.
Case: $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$
Then

$$\begin{aligned} |consts(\mathtt{t})| &= |consts(\mathtt{t}_1) \cup consts(\mathtt{t}_2) \cup consts(\mathtt{t}_3)| \\ &\leq |consts(\mathtt{t}_1)| + |consts(\mathtt{t}_2)| + |consts(\mathtt{t}_3)| \\ &\leq size(\mathtt{t}_1) + size(\mathtt{t}_2) + size(\mathtt{t}_3) \\ &< size(\mathtt{t}) \end{aligned}$$

where inductive hypothesis for $\mathtt{t}_{1,2,3}$ is used. □

So in general technique boils down to proving the property for term based on inductive hypothesis that it holds for "smaller" terms. This requires considering a number of base and inductive cases.

1.1.3 Semantic styles

So far we have only talked about structure of the language expressions. Now we should provide (computational) meaning of the expressions, i.e. language semantics. There are three main approaches to define semantics called *semantic styles*[3].

Axiomatic semantics can be traced back to the work of Floyd and later Hoare. The meaning of the program is defined in terms of assertions and their relationships. There are (logical) rules assigned to each base constructions of the language that specify their effect on pre- and post-conditions. Combining these rules one can reason about the effect of the whole program. Thus the meaning of programs is just what we can prove about them.

[3]Ukrainian equivalent: *семантичні стилі*

$\mathbb{B}$ *(untyped)*

Syntax

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

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

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

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

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

Figure 1.1: Booleans (B)

Denotational semantics In this approach developed by Scott and Strachey each program is interpreted as some mathematical object that belongs to *semantic domain*. Originally the domain was a space that included self-applicable functions and the meaning was obtained as a refinement (sequence of objects converging to result). One can talk in abstract terms about program behavior, e.g. program equivalence.

Operational semantics is defined in terms of *transition relations*. The evaluation of program consists in steps of *state* transformations. Thus evaluation of the terms can be thought as the work of *abstract machine*. This semantics is used throughout the notes.

1.1.4 Evaluation

Let us define the operational semantics for Boolean expressions only. The summary is on figure 1.1. Special subset of terms is separated as *values*[4]. The r.h.s. defines the *evaluation relation* on terms $\longrightarrow$. Some authors rather call it *reduction* relation. Definition is done in the same manner as with terms themselves, that is using inductive definition based on inference rules.

Definition 1.5. The *one-step* evaluation[5] relation $\longrightarrow$ is the smallest binary relation on terms satisfying the three rules in Figure 1.1. When the pair

[4]Ukrainian equivalent: *значення*

[5]Ukrainian equivalent: *один крок* обчислення

$(\mathtt{t}, \mathtt{t}')$ is in the evaluation relation, we say that "the evaluation *statement* (or *judgment*) $\mathtt{t} \longrightarrow \mathtt{t}'$ is *derivable*".

The rules together determine how the terms are evaluated, i.e. evaluation *strategy*[6]. In this particular case evaluation is done from outermost to innermost conditionals, with inner conditionals being evaluated only for the sake of outermost guards. Rules E-IfTrue, E-IfFalse determine the evaluation for terms with fully evaluated guard. Rule E-If extends the evaluation on other terms. Rules E-IfTrue, E-IfFalse are called *computational* rules, E-If – *congruence* rule.

Proof that particular evaluation statement is derivable is usually done using (evaluation) derivation *tree*[7]. It is a structure similar to abstract syntax tree. Its nodes are labeled by inference rules and its root is labeled by evaluation statement derived by the tree.

If we abbreviate

$$\begin{aligned}\mathtt{u} &= \mathtt{if\ true\ then\ false\ else\ false}\\ \mathtt{t} &= \mathtt{if\ u\ then\ t_1\ else\ t_2}\\ \mathtt{s} &= \mathtt{if\ false\ then\ t_1\ else\ t_2}\end{aligned}$$

Then we have derivation tree

$$\dfrac{\dfrac{\dfrac{}{\mathtt{u} \longrightarrow \mathtt{false}}\ \text{E-IfTrue}}{\mathtt{t} \longrightarrow \mathtt{s}}\ \text{E-If}}{\mathtt{if\ t\ then\ t_3\ else\ t_4} \longrightarrow \mathtt{if\ s\ then\ t_3\ else\ t_4}}\ \text{E-If}$$

The derivation tree corresponding to evaluation statement is simply called its *derivation*[8]. The *evaluation* derivation[9] degenerates to list because inference rules in example has at most one premise.

Similarly to induction on terms we can introduce proof technique called induction *on derivations*[10].

Proposition 1.6 (Determinacy of one-step evaluation)**.** *If* $t \longrightarrow t'$ *and* $t \longrightarrow t''$*, then* $t' = t''$*.*

[6]Ukrainian equivalent: *стратегія* обчислення
[7]Ukrainian equivalent: *дерево* виведення
[8]Ukrainian equivalent: *виведення*
[9]Ukrainian equivalent: *виведення (кроку) обчислення*
[10]Ukrainian equivalent: індукція *за виведеннями*

Proof. By induction on derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$.[11] We have to consider cases according to rule applied last (at the root of derivation).
Case: Last rule is E-IfTrue. Then $\mathtt{t} = \mathtt{if\ true\ then\ t'\ else\ t}_1$. Now both E-IfFalse, E-If are not applicable to $\mathtt{t}$ and the only possible conclusion of E-IfTrue is $\mathtt{t} \longrightarrow \mathtt{t}'$. Thus $\mathtt{t}' = \mathtt{t}''$.
Case: Last rule is E-IfFalse. Similar to previous.
Case: Last rule in derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$ is E-If. Then $\mathtt{t}$ has the form $\mathtt{if\ t}_1\ \mathtt{then\ t}_2\ \mathtt{else\ t}_3$, where $\mathtt{t}_1 \longrightarrow \mathtt{t}'_1$ for some $\mathtt{t}'_1$. This excludes $\mathtt{t}_1 = \mathtt{true}$, $\mathtt{t}_1 = \mathtt{false}$. The last rule in derivation of $\mathtt{t} \longrightarrow \mathtt{t}''$ then should be also E-If, i.e. $\mathtt{t}'' = \mathtt{if\ t}''_1\ \mathtt{then\ t}_2\ \mathtt{else\ t}_3$, where $\mathtt{t}_1 \longrightarrow \mathtt{t}''_1$. By induction hypothesis $\mathtt{t}'_1 = \mathtt{t}''_1$, as a result $\mathtt{t}' = \mathtt{t}''$. □

Definition 1.7. A term $\mathtt{t}$ is in *normal form*[12] if no evaluation rule applies to it, i.e. if there is no $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$.

Trivially, following theorem holds.

Theorem 1.8. *Every value is in normal form.*

The converse to it also true.

Theorem 1.9. *If $\mathtt{t}$ is in normal form, then $\mathtt{t}$ is a value.*

Proof. Suppose that $\mathtt{t}$ is not a value. We're going to prove by induction on term structure that $\mathtt{t}$ is not a normal formal. Either $\mathtt{t} = \mathtt{if\ true\ then\ t'\ else\ t''}$. In which case by E-IfTrue $\mathtt{t} \longrightarrow \mathtt{t}'$. Or $\mathtt{t} = \mathtt{if\ false\ then\ t'\ else\ t''}$. Then by rule E-IfTrue $\mathtt{t} \longrightarrow \mathtt{t}''$. Or, finally, $\mathtt{t} = \mathtt{if\ t}_1\ \mathtt{then\ t}_2\ \mathtt{else\ t}_3$, where $\mathtt{t}_1$ is not a value. By inductive hypothesis there is term $\mathtt{t}'_1$ such that $\mathtt{t}_1 \longrightarrow \mathtt{t}'_1$. Then by rule E-If we conclude $\mathtt{t} \longrightarrow \mathtt{if\ t}'_1\ \mathtt{then\ t}_2\ \mathtt{else\ t}_3$. That is in all cases $\mathtt{t}$ is not a normal form. □

Definition 1.10. The *multi-step evaluation* relation $\overset{*}{\longrightarrow}$ is the reflexive, transitive closure of one-step evaluation.

Proposition 1.11 (Uniqueness of multi-step evaluation relation)**.** *If $t \overset{*}{\longrightarrow} u$ and $t \overset{*}{\longrightarrow} u'$, where u and u' are both normal forms, then $u = u'$.*

Proof. It is a corollary of single step evaluation. □

Theorem 1.12 (Termination of evaluation)**.** *For every term $\mathtt{t}$ there is some normal form $\mathtt{t}'$ such that $\mathtt{t} \overset{*}{\longrightarrow} \mathtt{t}'$.*

[11] This proof can be done by induction on term $\mathtt{t}$ and by induction on derivation of $\mathtt{t} \longrightarrow \mathtt{t}''$ as well

[12] Ukrainian equivalent: *нормальна форма*

$\mathbb{B}$ $\mathbb{N}$ *(untyped)* *Extends* **B**

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 $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'} \quad \text{(E-Succ)}$$

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

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'} \quad \text{(E-Pred)}$$

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

$$\texttt{iszero}\ (\texttt{succ}\ nv_1) \longrightarrow \texttt{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero}\ t_1 \longrightarrow \texttt{iszero}\ t_1'} \quad \text{(E-IsZero)}$$

Figure 1.2: Arithmetic expressions (NB)

Exercise 1.13. 1. Suppose we add a new rule

$$\texttt{if true then}\ t_2\ \texttt{else}\ t_3 \longrightarrow t_2 \qquad \text{(E-Funny1)}$$

to the ones in Figure 1.1. Which of the above theorems (3.5.4,7,8,11,12 uniqueness, termination) remain valid?

2. Suppose instead that we add this rule:

$$\frac{t_2 \longrightarrow t_2'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \longrightarrow \texttt{if}\ t_1\ \texttt{then}\ t_2'\ \texttt{else}\ t_3} \qquad \text{(E-Funny2)}$$

Which of the above theorems remain valid? Do any of the proofs need to change?

Let us now extend evaluation relation to numbers. See additional syntactic forms and evaluation rules in Figure 1.2.

There is a subset of terms called *numeric values*, which correspond to natural numbers. There are four computation rules and three congruence rules.

Definition 1.14. A term is *stuck*[13] if it is in normal form but not a value.

[13]Ukrainian equivalent: *застряглий*

$\mathbb{B}$ $\mathbb{N}$ *(untyped)* *alternative*

Evaluation $\boxed{\mathtt{t} \Downarrow \mathtt{v}}$

$$\mathtt{v} \Downarrow \mathtt{v} \quad \text{(B-Value)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{true} \qquad \mathtt{t}_2 \Downarrow \mathtt{v}_2}{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \Downarrow \mathtt{v}_2} \quad \text{(B-IfTrue)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{false} \qquad \mathtt{t}_3 \Downarrow \mathtt{v}_3}{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \Downarrow \mathtt{v}_3} \quad \text{(B-IfFalse)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{nv}_1}{\mathtt{succ}\ \mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1} \quad \text{(B-Succ)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{0}}{\mathtt{pred}\ \mathtt{t}_1 \Downarrow \mathtt{0}} \quad \text{(B-PredZero)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1}{\mathtt{pred}\ \mathtt{t}_1 \Downarrow \mathtt{nv}_1} \quad \text{(B-PredSucc)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{0}}{\mathtt{iszero}\ \mathtt{t}_1 \Downarrow \mathtt{true}} \quad \text{(B-IszeroZero)}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1}{\mathtt{iszero}\ \mathtt{t}_1 \Downarrow \mathtt{false}} \quad \text{(B-IszeroSucc)}$$

Figure 1.3: Big-step semantics rules

Big-step semantics is an alternative way to define term evaluation. Unlike small-step semantics it is aimed to convey whole evaluation process in one step. The evaluation rules for arithmetics are provided in Figure 1.3

1.2 Typed arithmetic

Despite being quite simple the system of arithmetic expressions is enough to demonstrate most important aspects of type usage.

1.2.1 Typing relation

We separate rules for Booleans and for numbers. Details are in the figures 1.4 and 1.5 respectively.

Definition 1.15. The *typing relation*[14] for arithmetic expressions is the smallest binary relation between terms and types satisfying all instances of the rules in Figures 1.4 and 1.5. A term $\mathtt{t}$ is *typable*[15] (or *well-typed*[16]) if there is some T such that $\mathtt{t} : T$.

[14]Ukrainian equivalent: *відношення типізації*
[15]Ukrainian equivalent: *типізовний*
[16]Ukrainian equivalent: *коректно типізований*

$\mathbb{B}$ *(typed)* — *Extends* **B**

New syntactic forms

T	::=		*types:*
		Bool	*type of booleans*

New typing rules — t : T

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{\texttt{t}_1 : \texttt{Bool} \quad \texttt{t}_2 : \texttt{T} \quad \texttt{t}_3 : \texttt{T}}{\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3 : \texttt{T}} \qquad \text{(T-If)}$$

Figure 1.4: Typing rules for booleans (B)

$\mathbb{B}\ \mathbb{N}$ *(typed)* — *Extends* **NB**

New syntactic forms

T	::=	...	*types:*
		Nat	*type of natural numbers*

New typing rules — t : T

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{succ t}_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{pred t}_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{iszero t}_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

Figure 1.5: Typing rules for numbers (NB)

Similar to evaluation relation typing of some term is proved using *typing* derivation[17].

Here is the typing derivation for the term `if iszero 0 then 0 else pred 0`:

$$\dfrac{\dfrac{\dfrac{}{\texttt{0 : Nat}}\ \text{T-Zero}}{\texttt{iszero 0 : Bool}}\ \text{T-IsZero} \qquad \dfrac{}{\texttt{0 : Nat}}\ \text{T-Zero} \qquad \dfrac{\dfrac{}{\texttt{0 : Nat}}\ \text{T-Zero}}{\texttt{pred 0 : Nat}}\ \text{T-Pred}}{\texttt{if iszero 0 then 0 else pred 0 : Nat}}\ \text{T-If}$$

Simplest property of the type system is presented by the following lemma.

Lemma 1.16 (Inversion Lemma)**.** *1. If* `true` : R*, then* $R =$ `Bool`*.*

[17] Ukrainian equivalent: *виведення типу*

2. *If* $\texttt{false} : R$, *then* $R = \texttt{Bool}$.

3. *If* $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : R$, *then* $t_1 : \texttt{Bool}$, $t_2 : R$ *and* $t_3 : R$.

4. *If* $0 : R$, *then* $R = \texttt{Nat}$.

5. *If* $\texttt{succ } t_1 : R$, *then* $R = \texttt{Nat}$ *and* $t_1 : \texttt{Nat}$.

6. *If* $\texttt{pred } t_1 : R$, *then* $R = \texttt{Nat}$ *and* $t_1 : \texttt{Nat}$.

7. *If* $\texttt{iszero } t_1 : R$, *then* $R = \texttt{Bool}$ *and* $t_1 : \texttt{Nat}$.

Proof. All points are quite similar. Let us show one of them. If term of the form $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$ is typable, then the last rule of typing derivation should be T-If. Therefore $t_1 : \texttt{Bool}$, $t_2 : R$, $t_3 : R$. □

Theorem 1.17 (Uniqueness of Types)**.** *Each term* t *has at most one type. That is, if* t *is typable, then its type is unique.*

Proof. By induction on typing derivation of $t : T$. The only nontrivial case is T-If. The term $t = \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$ is only typable when the last rule in typing derivation is T-If (inversion lemma). By i.h. type of t_1 and t_2 is unique. Therefore type of t is also unique. □

1.2.2 Type safety

Type safety[18] should allow us to avoid meaningless (wrong) terms. This can be expressed in two properties of type system.

Progress[19]: A well-typed term is not stuck (either it is a value or it can take a step according to the evaluation rules).

Preservation[20]: If a well-typed term takes a step of evaluation, then the resulting term is also well typed.

In order to discuss and prove these properties the following lemma is required.

Lemma 1.18 (Canonical Forms)**.** 1. *If* v *is a value of type* $\texttt{Bool}$*, then* v *is either* $\texttt{true}$ *or* $\texttt{false}$.

2. *If* v *is a value of type* $\texttt{Nat}$*, then* v *is a numeric value according to the grammar.*

[18] Ukrainian equivalent: *типова безпека*

[19] Ukrainian equivalent: *просування*

[20] Ukrainian equivalent: *збереження*

Proof. By inversion lemma numeric values cannot have type `Bool` and vice versa. □

Theorem 1.19 (Progress)**.** *Suppose* $\mathtt{t}$ *is a well-typed term (that is,* $\mathtt{t} : T$ *for some* T*). Then either* $\mathtt{t}$ *is a value or else there is some* $\mathtt{t}'$ *with* $\mathtt{t} \longrightarrow \mathtt{t}'$*.*

Proof. By induction on typing derivation of $\mathtt{t} : T$. We consider only some cases.
Case T-TRUE. $\mathtt{t}$ is a value in this case.
Case T-PRED. The term has form $\mathtt{pred}\ \mathtt{t}_1$ where $\mathtt{t}_1 : \mathtt{Nat}$. If $\mathtt{t}_1$ is a value then by canonical form lemma $\mathtt{t}_1 = \mathtt{nv}_1$. Therefore E-PREDZERO or E-PREDSUCC is applicable to $\mathtt{t}$. Otherwise by i.h. $\mathtt{t}_1 \longrightarrow \mathtt{t}'_1$ then by E-PRED $\mathtt{t} \longrightarrow \mathtt{pred}\ \mathtt{t}'_1$. □

Theorem 1.20 (Preservation (*subject reduction*[21])). *If* $\mathtt{t} : T$ *and* $\mathtt{t} \longrightarrow \mathtt{t}'$*, then* $\mathtt{t}' : T$*.*

1.3 Untyped λ-calculus

Historical background Historically lambda-calculus notation was inspired by the notation used by Russell and Whitehead in Principia Mathematica[6]. Russell used expressions with circumflex like $\hat{x}(\phi x)$. Roughly speaking, this (class abstraction) represents the set denoted $\{x \mid \phi x\}$ in traditional set-builder notation[7]. Also at some point to some extent it was used to express functions $\phi\hat{x}$. For metaphysical reasons Russell abandoned treatment of functions as entities and further discarded the notation in favour of substitutional theory.

Because of typesetting problems with circumflex over letter, Church first used circumflex in front of variable and then for similarity with Λ started using letter λ for abstraction.

1.3.1 Lambda terms

Understanding of free and bound variables. See: wiki.

Let us introduce function $FV(\mathtt{t})$ that gives free variable in term $\mathtt{t}$. By induction on term structure:

$$\begin{aligned} FV(\mathtt{x}) &= \{\mathtt{x}\} \\ FV(\lambda\mathtt{x}.\mathtt{t}) &= FV(\mathtt{t}) \setminus \{\mathtt{x}\} \\ FV(\mathtt{t}_1\ \mathtt{t}_2) &= FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2) \end{aligned}$$

[21] Ukrainian equivalent: *скорочення/редукція підмета*

λ *(α-equivalence part)*

Evaluation $\boxed{\mathtt{t} \overset{\alpha}{\equiv} \mathtt{t}'}$

$$\frac{}{\mathtt{t} \equiv \mathtt{t}} \quad \text{(A-Ref)}$$

$$\frac{\mathtt{t} \equiv \mathtt{t}' \qquad \mathtt{t}' \equiv \mathtt{t}''}{\mathtt{t} \equiv \mathtt{t}''} \quad \text{(A-Trans)}$$

$$\frac{\mathtt{t}_1 \equiv \mathtt{t}_1' \qquad \mathtt{t}_2 \equiv \mathtt{t}_2'}{\mathtt{t}_1\ \mathtt{t}_2 \equiv \mathtt{t}_1'\ \mathtt{t}_2'} \quad \text{(A-App)}$$

$$\frac{\mathtt{y} \notin FV(\mathtt{t})}{\lambda \mathtt{x}.\mathtt{t} \equiv \lambda \mathtt{y}.[\mathtt{x} \mapsto \mathtt{y}]\mathtt{t}} \quad \text{(A-Abs)}$$

Figure 1.6: α-equivalence

Consider two expressions

$$\mathrm{id}(x) = x$$
$$\mathrm{id}(y) = y$$

They can be considered both as calculations and as definition of id. It is obvious that defined function is the same and from first equation we derive $\mathrm{id} = \lambda \mathtt{x}.\mathtt{x}$ while from the second $\mathrm{id} = \lambda \mathtt{y}.\mathtt{y}$. This highlights the property that the names of the bound variables is not important and that bound variables can be changed without change of meaning. This is formalized by the following transformation. We define the relation $\equiv_\alpha$ called α-equivalence. Its inference rules are presented in figure 1.6.

Now we can define substitution rules by induction on term structure:

$$[\mathtt{x} \mapsto \mathtt{t}]\mathtt{x} = \mathtt{t}$$
$$[\mathtt{x} \mapsto \mathtt{t}]\mathtt{y} = \mathtt{y}$$
$$[\mathtt{x} \mapsto \mathtt{t}](\mathtt{t}_1\ \mathtt{t}_2) = [\mathtt{x} \mapsto \mathtt{t}]\mathtt{t}_1\ [\mathtt{x} \mapsto \mathtt{t}]\mathtt{t}_2$$
$$[\mathtt{x} \mapsto \mathtt{t}]\lambda \mathtt{x}.\mathtt{t}_1 = \lambda \mathtt{x}.\mathtt{t}_1$$

Lambda abstraction is trickier. The abstraction by substituted variable shield the term. If the variable is different we have to make sure that substitution avoids *name capturing*, i.e. free variables of term $\mathtt{t}_1$ should not become bound after substitution.

$$\frac{\mathtt{y} \notin FV(\mathtt{t})}{[\mathtt{x} \mapsto \mathtt{t}]\lambda \mathtt{y}.\mathtt{t}_1 = \lambda \mathtt{y}.[\mathtt{x} \mapsto \mathtt{t}]\mathtt{t}_1}$$

Last rule is not restrictive because we can guarantee the premise for term that is α-equivalent to given term $\mathtt{s}$ thus extending substitution to all terms.

Notice that α-equivalence and substitution are defined by *mutual* induction (recursion).

Exercise 1.21. Prove that relation $\overset{\alpha}{\equiv}$ is an equivalence relation.

Exercise 1.22. Prove that following rule

$$[\mathtt{x} \mapsto \mathtt{t}]\lambda\mathtt{x}.\mathtt{t}_1 = \lambda\mathtt{x}.\mathtt{t}_1$$

can be eliminated from the substitution definition.

Exercise 1.23. Prove that function $FV(\mathtt{t})$ is correctly defined w.r.t. relation $\overset{\alpha}{\equiv}$.

Further, as usual in λ-calculus, equality of terms is understood up to α-equivalence.

1.3.2 Beta reduction

The main element of λ-calculus semantics is expression of the kind $(\lambda\mathtt{x}.\mathtt{t})\ \mathtt{t}_2$, which is called *reducible expression*[22] or *redex*[23] for short. The intuition behind the application of function to the given argument is to replace the abstracted variable in the expression of the function with $\mathtt{t}_2$. This is captured by the rule:

$$(\lambda\mathtt{x}.\mathtt{t})\ \mathtt{t}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}$$

Here $[\mathtt{x} \mapsto \mathtt{t}_2]$ is a substitution operator that will be formalized later in the section.

1.3.2.1 Evaluation strategies

Evaluation strategy determines in which order expressions are to be evaluated. Since redexes are in the core of lambda-calculus evaluation we should choose which one to evaluate first. Additional rules guide us through the expression. For lambda-calculus enriched with additional elements they also regulate is it redex reduction or specific rules have the preference in evaluation.

- full beta-reduction
- normal order

[22] Ukrainian equivalent: *скорочуваний вираз, редуковний вираз*
[23] Ukrainian equivalent: *редекс*

- call by name / call by need
- call by value

Mostly last two strategies are implemented in programming languages. Imperative languages use call-by-value. Functional programming language are divided. Some like OCaml or prefer call-by-value, it is called *eager*[24], *strict*[25] or *greedy*[26]. Haskell uses call by need strategy, which is also called *lazy* evaluation[27].

1.3.2.2 Other rules

For some systems additional η-conversion rule is also considered. It written as follows

$$\lambda x.(f\ x) \to f$$

To be valid there is a constraint that does not contain x free. It represents to extensionality. Depending on the direction in which it is applied we could have either η-reduction (direct) or η-expansion (reverse). Addition of this rule allows to perform certain optimization during compilation. But it also means adding complexity in compiler.

1.3.3 Programming in pure λ-calculus

λ-calculus is proven to be Turing-complete. Arithmetic, partial recursive functions could be implemented in it. To be accessible from λ-calculus notions from different mathematical systems require specific representation as λ-terms.

There are "canonical" representation of well-known mathematical constructions in pure λ-calculus. There are booleans, pairs, (Church) numbers. Basic arithmetic operation of addition, multiplication and exponentiation are implemented straightforward. More complex constructions require recursion. It is implemented in λ-calculus using fixed-point combinator. Good account of these implementations is presented in [2].

Exercise 1.24. Implement in pure lambda-calculus iteration without fixed-point combinator.

Exercise 1.25. Implement in enriched lambda-calculus pairs based on embedded booleans.

[24]Ukrainian equivalent: *жадібне*
[25]Ukrainian equivalent: *строге*
[26]Ukrainian equivalent: *жадібне*
[27]Ukrainian equivalent: *лінивi* обчислення

$\longrightarrow$ *(untyped)*

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x.t}$	*abstraction*
		$\mathtt{t\ t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda\mathtt{x.t}$	*abstraction value*

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

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1'\ t_2}} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2'}}{\mathtt{v_1\ t_2} \longrightarrow \mathtt{v_1\ t_2'}} \quad \text{(E-App2)}$$

$$\mathtt{(\lambda x.t_{12})\ v_2} \longrightarrow [\mathtt{x} \mapsto \mathtt{v_2}]\mathtt{t_{12}} \quad \text{(E-AppAbs)}$$

Figure 1.7: Untyped lambda-calculus (λ)

1.3.4 Formal definitions

The summary of the rules is presented in the figure 1.7.

It is easy to see there what strategy is implied here. Notice that λ-abstractions are values. The term representing the function is evaluated first until it is a value. Then its argument is evaluated. Finally, the redex itself is reduced.

1.4 Simply-typed λ-calculus

Next system is known as *simply-typed* lambda calculus[28]. The summary of the rules is presented in the figure 1.8. Its set of types is inductively defined using several base types and *function* type[29] constructor. Type annotations appear for variables in λ-abstractions. Context Γ is introduced. It is needed to assign types to free variables so that open terms could be typed. Typing rules express that function type and its parameter type must agree. Also it specify that function type is assigned to the λ-abstraction.

1.4.1 Properties of $\lambda_{\rightarrow}$

Lemma 1.26 (Canonical form)**.** *1. If v is a value of type `Bool`, then v is either `true` or `false`.*

2. If v is a value of type $T_1 \rightarrow T_2$, then $v = \lambda x : T_1.t_2$.

[28]Ukrainian equivalent: лямбда-числення *з простою типізацією*

[29]Ukrainian equivalent: *"стрілочний"* тип

$\to$ *(typed)* — *Based on* λ

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda \mathtt{x}:\mathtt{T}.\mathtt{t}$	*abstraction*
		$\mathtt{t}\ \mathtt{t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda \mathtt{x}:\mathtt{T}.\mathtt{t}$	*abstraction value*
$\mathtt{T}$	::=		*types:*
		$\mathtt{T}\to\mathtt{T}$	*type of functions*
Γ	::=		*contexts:*
		$\emptyset$	*empty context*
		$\Gamma, \mathtt{x}:\mathtt{T}$	*term variable binding*

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-App2)}$$

$$(\lambda \mathtt{x}:\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-AppAbs)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

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

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

Figure 1.8: Pure simply typed lambda-calculus ($\lambda_\to$)

Proposition 1.27 (Progress)**.** *Suppose* $\mathtt{t}$ *is a closed, well-typed term (that is,* $\vdash \mathtt{t} : \mathtt{T}$ *for some* $\mathtt{T}$*). Then either* $\mathtt{t}$ *is a value or else there is some* $\mathtt{t}'$ *with* $\mathtt{t} \longrightarrow \mathtt{t}'$*.*

Proof. By induction on typing derivation of $\vdash \mathtt{t} : \mathtt{T}$. The proof has two parts: for pure λ-calculus and for booleans. Proof for booleans remain the same. Here we consider proof for pure $\lambda_\to$. It is obvious that $\mathtt{t}$ is not a variable, so case T-Var is vacuously true. If the last rule is T-Abs then $\mathtt{t}$ is immediately a value. Thus only case of application should be considered.
Case T-App: $\mathtt{t}$ has a form $\mathtt{t}_2\ \mathtt{t}_1$ with $\vdash \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}$ and $\vdash \mathtt{t}_1 : \mathtt{T}_1$.
Then by inductive hypothesis for $\vdash \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}$ either $\mathtt{t}_2$ is a value or evaluates in one step to $\mathtt{t}_2'$. If $\mathtt{t}_2 \longrightarrow \mathtt{t}_2'$ then by E-App1 $\mathtt{t} \longrightarrow \mathtt{t}_2'\ \mathtt{t}_1$. Otherwise by canonical form lemma $\mathtt{t}_2$ should have a form $\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_3$, i.e. $\mathtt{t}$ is a redex. By inductive hypothesis for $\vdash \mathtt{t}_1 : \mathtt{T}_1$ either $\mathtt{t}_1$ is a value or $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$. In former case E-AppAbs applies to $\mathtt{t}$, and in the latter E-App2 does. □

Lemma 1.28 (Preservation of types under substitution). *If* $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$ *and* $\Gamma \vdash \mathtt{s} : \mathtt{S}$ *then* $\Gamma \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{t} : \mathtt{T}$.

Proof. By induction on typing derivation of $\Gamma \vdash \mathtt{t} : \mathtt{T}$.
Case T-Var.

> Subcase 1. $\mathtt{t} = \mathtt{y}$, $\Gamma \vdash \mathtt{y} : \mathtt{T}$, $\mathtt{y}{:}\mathtt{T} \in \Gamma$. Then $[\mathtt{x}\mapsto\mathtt{s}]\mathtt{y} = \mathtt{y}$, and respectively $\Gamma \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{y} : \mathtt{T}$.
> Subcase 2. $\mathtt{t} = \mathtt{x}$, $\mathtt{T} = \mathtt{S}$. Then $[\mathtt{x}\mapsto\mathtt{s}]\mathtt{x} = \mathtt{s}$, and $\Gamma \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{x} : \mathtt{T}$.

Case T-Abs. $\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T}_1.\mathtt{t}_2$, $\mathtt{T} = \mathtt{T}_1 \to \mathtt{T}_2$. $\Gamma, \mathtt{x}{:}\mathtt{S}, \mathtt{y}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2$. Due to α-equivalence we can assume that $\mathtt{y} \notin \{\mathtt{x}\} \cup FV(\mathtt{s})$. Therefore $[\mathtt{x}\mapsto\mathtt{s}]\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T}_1.[\mathtt{x}\mapsto\mathtt{s}]\mathtt{t}_2$. Reordering last two elements of typing context for $\mathtt{t}_2$ we obtain $\Gamma, \mathtt{y}{:}\mathtt{T}_1, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_2 : \mathtt{T}_2$. Also we can weaken the typing of $\mathtt{s}$: $\Gamma, \mathtt{y}{:}\mathtt{T}_1 \vdash \mathtt{s} : \mathtt{S}$. By inductive hypothesis $\mathtt{y}{:}\mathtt{T}_1, \Gamma \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{t}_2 : \mathtt{T}_2$. Then by rule T-Abs $\Gamma \vdash \lambda\mathtt{y}{:}\mathtt{T}_1.[\mathtt{x}\mapsto\mathtt{s}]\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2$.
Case T-App. $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$, $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_1{:}\mathtt{T}_2 \to \mathtt{T}$. $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_2{:}\mathtt{T}_2$.
Then by inductive hypothesis $\Gamma \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{t}_2 : \mathtt{T}_2$ and $\Gamma \vdash [\mathtt{x}\mapsto\mathtt{s}]\mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}$.
By T-App, $\Gamma \vdash [\mathtt{x}\mapsto\mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) : \mathtt{T}$ □

Proposition 1.29 (Preservation). *If* $\Gamma \vdash \mathtt{t} : \mathtt{T}$ *and* $\mathtt{t} \longrightarrow \mathtt{t}'$, *then* $\Gamma \vdash \mathtt{t}' : \mathtt{T}$.

Proof. By induction on typing derivation of $\Gamma \vdash \mathtt{t} : \mathtt{T}$.
Cases T-Var, T-Abs are vacuously true because their conclusion term is a normal form.
Case T-App: $\mathtt{t}$ has a form $\mathtt{t}_2\ \mathtt{t}_1$ with $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}$ and $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$.
We consider subcases according to last rule of evaluation derivation

> Subcase E-App1: $\mathtt{t}_2 \longrightarrow \mathtt{t}_2'$
> Then $\mathtt{t}'$ is $\mathtt{t}_2'\ \mathtt{t}_1$. By inductive hypothesis $\Gamma \vdash \mathtt{t}_2' : \mathtt{T}_1 \to \mathtt{T}$ and $\mathtt{t}' : \mathtt{T}$ by rule T-App.
> Subcase E-App2: $\mathtt{t}_2 = \mathtt{v}_2$ is a value and $\mathtt{t}_1 \longrightarrow \mathtt{t}_1'$
> Then $\mathtt{t}'$ is $\mathtt{v}_2\ \mathtt{t}_1'$. By inductive hypothesis $\Gamma \vdash \mathtt{t}_1' : \mathtt{T}_1$. Therefore by rule T-App $\Gamma \vdash \mathtt{t}' : \mathtt{T}_2$.
> Subcase E-AppAbs: $(\lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_3)\ \mathtt{v}_1$ as $\mathtt{t}$
> This is a redex that evaluates to $\mathtt{t}' = [\mathtt{x}\mapsto\mathtt{v}_1]\mathtt{t}_3$. Since $\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_3 : \mathtt{T}$ and $\Gamma \vdash \mathtt{v}_1 : \mathtt{T}_1$ then by substitution lemma $\Gamma \vdash [\mathtt{x}\mapsto\mathtt{v}_1]\mathtt{t}_3 : \mathtt{T}$.
>
> □

1.4.2 Type reconstruction problem

It has already been shown that it is typical to start with untyped system or language. It is easier to implement and it is more efficient in program evalu-

ation. Then in order to provide safety the language is augmented with types. In this setting question arise if it is possible to take program from original system (without types) and construct from it typed version with identical runtime behavior. This problem is called *type reconstruction*[30] problem. Besides being peculiar theoretical problem it has straightforward application: construction tool can assist programmer in identifying types of variables. This is especially viable when such types are quite involved, which is common for modern libraries, typical example are iterator types in C++. In this more local sense the problem of automatic type detection of expression based on its context is referred to as *type inference*[31]. This allows us to use advantages in safety of typed system and to avoid type annotations that seem redundant or get rid of them completely.

Type reconstruction can be considered as the inverse process to *type erasure*[32]. Like type reconstruction type erasure connects typed and untyped language. It is the process of removing information about types from the program before executing it. The most important property of type erasure is that it commutes with evaluation. But it also establishes close relationship between untyped and typed languages.

In the simplest case of typed arithmetic there are no type annotations at all therefore type erasure is identity. Semantics for typed arithmetic is implicitly preserved because it is based on the same evaluation rules. Therefore typed arithmetic term are evaluated as expected. Typed reconstruction in this case is type-checking that is finding out if the term is typable.

Let us consider simply-typed lambda calculus $\lambda_{\to}$. Erasure of a simply-typed term boils down to erasing type annotations. This can be expressed inductively as follows

$$\begin{aligned} \texttt{erase}(x) &= x \\ \texttt{erase}(\lambda x : T.\ t) &= \lambda x.\ \texttt{erase}(t) \\ \texttt{erase}(t_1\ t_2) &= \texttt{erase}(t_1)\ \texttt{erase}(t_2) \end{aligned}$$

Since evaluation rules for simply-typed lambda terms and untyped lambda-terms are basically the same (up to type annotations), it is to see following proposition.

Proposition 1.30 (Erasure commutes with evaluation)**.** *Following holds*

- *If* $t \to t'$ *then* $\texttt{erase}(t) \to \texttt{erase}(t')$

[30]Ukrainian equivalent: *відновлення типів*
[31]Ukrainian equivalent: *виведення типів*
[32]Ukrainian equivalent: *стирання типів*

- *If* $\texttt{erase}(t) \to m'$ *then there is* $\lambda_\to$*-term* t' *such that* $t \to t'$ *and* $m' = \texttt{erase}(t')$.

Now we can refine the notion of type reconstruction problem

Definition 1.31. Term m of untyped lambda-calculus is *typable* in $\lambda_\to$ if there are t, T, Γ such that $\Gamma \vdash t : T$, $\texttt{erase}(t) = m$.

This concepts play important role in following chapter when we consider how polymorphic constructions can be added to the system.

Chapter 2

Polymorphism

2.1 Introduction

In conventional languages functions are monomorphic. Still there are examples of symbols that has multiple meaning. Christopher Strachey called this symbols *polymorphic*[1]:

> We expect compiling system to interpret correctly ambiguous symbols (such as +) which mean different things according to the types of their operands. We call ambiguous operators of this sort **polymorphic** as they have several forms depending on their arguments.[8]

Strachey divided such symbols into two categories. Some coincide by chance, other work uniformly. First situation was called *ad hoc* polymorphism[2]. In this case polymorphic symbols work on several different types but have unrelated behaviour. Such polymorphic symbols have meaning only for limited number of cases. There is no single systematic way of determining the type of the result from the type of the arguments[8]. The second is called *parametric* polymorphism[3]. Parametric is regular and should work for any types in uniform way.

The intrinsic problem of polymorphism choosing correct form of the symbol from the context. The choice is complicated by the fact that several forms can be accepted for some contexts. Therefore

Luca Cardelli and Peter Wegner further refined the classification of polymorphism [1]. This classification is represented in figure 2.1. One can see

[1]Ukrainian equivalent: *поліморфні, "багатоформні"*
[2]Ukrainian equivalent: *ad hoc* поліморфізм
[3]Ukrainian equivalent: *параметричний* поліморфізм

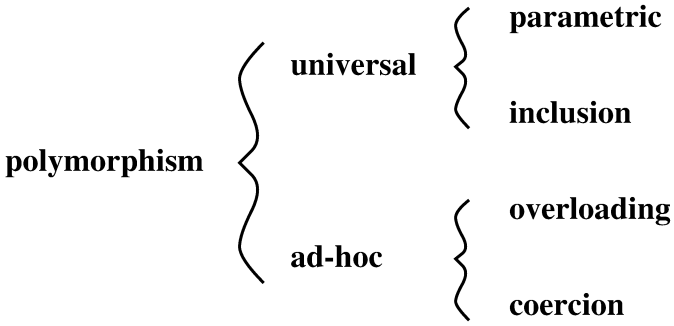


Figure 2.1: Polymorphism classification[1]

that parametric polymorphism when different types are treated uniformly is generalized to *universal* polymorphism[4], while ad hoc polymorphism is further subdivided into overloading and *coercion*[5]. Let us briefly consider each of the refined categories.

2.1.1 Ad hoc polymorphism

Overloading[6] is the ability to define several objects (functions) with the same name. It (degeneration) can be (and sometimes is) split by source code preprocessing stage. When used properly it is helpful tool that facilitates writing of the better code. Coercion is instead a semantic operation which is needed to convert an argument to the type expected by a function, in a situation which would otherwise result in a type error[1]. Strachey called such functions transfer functions. This could work either statically when application to arguments is generated at compile time, or decision about coercion specifics is made dynamically dictated by run-time evaluation of the arguments. The overloading and coercion can be combined in several ways. Typically overloading covers the limited list of most important cases, while intermediate cases are reduced to this list via coercion.

Monomorphic languages constrain object to one particular behavior which is too restrictive for expressive power. To lift this limitations polymorphism should be used. Therefore polymorphism is present in older programming languages. But they view polymorphic constructions as exceptions. They usually allow it in the following cases:

1. overloading

[4]Ukrainian equivalent: *універсальний* поліморфізм
[5]Ukrainian equivalent: *неявне приведення типів*
[6]Ukrainian equivalent: *довизначення, перевантаження*

2. coercion

3. subtyping (sub/super ranges)

4. value sharing, like `nil`

The last two can be viewed as a special cases of subtyping and parametric polymorphism respectively.

Ad hoc polymorphism is also called *apparent* polymorphism[7] because it disappears in the close range. Indeed for overloading several (possibly unrelated) monomorphic functions are named the same, and for coercion type of function result is fixed and independent of the argument type.

2.1.2 Subtype polymorphism

Languages with *subtype* polymorphism[8] or subtyping provide the notion of *subtype*[9]. Type S is a subtype of T, denoted as $S <: T$, if terms of type S can be safely treated (be used in place) as terms of type T. This way any term belongs to multiple types at once and language, in fact, supports polymorphism. Many popular programming languages provide this kind of polymorphism. Often subtyping relation is based on the inheritance relation. For this reason subtyping is sometimes called *inheritance* polymorphism[10]. Due to requirement in some languages to include in subtype all methods of the supertype it is also called *inclusion* polymorphism[11]. Most type systems equipped with subtyping have this relation to be a preorder.

One of the major differences of subtyping from parametric polymorphism is that code executed might depend on the particular type. This could add flexibility. At the same time, subtyping has intrinsic limitations. This is known as binary method problem[9].

Assume we want to manipulate in the program with the monoid algebraic structures. For this reason we introduce corresponding abstract class

```
interface Monoid {
  Monoid idElt();
  Monoid binop(Monoid y);
}
```

Now for particular examples of monoids we provide concrete classes. For instance:

[7]Ukrainian equivalent: *удаваний* поліморфізм
[8]Ukrainian equivalent: *підтиповий* поліморфізм
[9]Ukrainian equivalent: *підтип*
[10]Ukrainian equivalent: *спадковий* поліморфізм
[11]Ukrainian equivalent: *включний* поліморфізм

```
class IntAddMonoid implements Monoid {
  public IntAddMonoid(int x) { n = x; }
  public IntAddMonoid binop}(IntAddMonoid other)
    { return new IntAddMonoid(n + other.n); }

  public IntAddMonoid idElt()
    { return new IntAddMonoid(0); }

  public int n;
};

class DoubleAddMonoid implements Monoid {
  public DoubleAddMonoid(double x) { n = x; }

  public DoubleAddMonoid binop(DoubleAddMonoid other)
    { return new DoubleAddMonoid(n + other.n); }

  public DoubleAddMonoid idElt()
    { return new DoubleAddMonoid(0.0); }


  public double n;
};
```

Method `binop` signature is chosen according to method intended usage. Unfortunately this attempt fails with the following error message:

```
error:  IntAddMonoid is not abstract and does not override abstract method
binop(Monoid) in Monoid
```

On the other hand if we try a workaround using type down cast:

```
class IntAddMonoid implements Monoid {
  public IntAddMonoid(int x) { n = x; }
  public IntAddMonoid binop(Monoid other)
    { return new IntAddMonoid(n + ((IntAddMonoid)other).n); }

  public IntAddMonoid id_elt()
    { return new IntAddMonoid(0); }
  int n;
};
```

Then it results in the following runtime error as soon as we pass non-`IntAddMonoid` instance as an argument to `binop`.

This situation is the consequence of subtyping rules. We want to implement the method of the abstract class therefore the type of the method in the subclass has to be compatible with type of this method in abstract

class. We come to the question: when $T' \to S' <: T \to S$? The requirements of subtyping imply that $S' <: S$ and $T <: T'$ in this case. From the standpoint of order theory function type constructor "$\_ \to \_$" is monotone (order-preserving) by the second argument and antitone (order-reversing) by the first argument. Therefore argument type is called *contravariant*[12], while result type *covariant*[13]. Both of these terms stem from the category theory, where (among other examples) ordered sets are considered as categories and maps between them as functors.

In general any operator on types (*type constructor*) can be considered. For instance, type constructor for arrays:

$$\texttt{Array}\colon \mathscr{T} \to \mathscr{T}.$$

2.1.3 Introduction to parametric polymorphism

Parametric polymorphism is considered the purest form of polymorphism. As was mentioned earlier in this case function should work uniformly for any given types. This literally means that the same code is used independently of the argument types. Canonical example of this is the identity function[14]:

```
- fun id x = x;
val id = fn : 'a -> 'a
- id 1;
val it = 1 : int
- id 1.0;
val it = 1.0 : real
```

Because of the parametric polymorphism universality special measures and approaches should be taken. In order to pass arguments of various types uniformity of types representation and use of certain calling convention might be required. Also to accomplish anything nontrivial "hands-off" approach is utilized[10]. Let us demonstrate it with the example of function `map`:

```
- fun map f [] = []
-   | map f (x::ls) = (f x)::map f ls;
val map = fn : ('a -> 'b) -> 'a list -> 'b list

- fun dub x = x + x;
val dub = fn : int -> int
```

[12]Ukrainian equivalent: *контраваріантність*
[13]Ukrainian equivalent: *коваріантність*
[14]implemented in ML

```
- map dub [1,2,3];
val it = [2, 4, 6] : int list
```

You can see that this function makes no assumptions about type parameters. It relies on list structure using pattern matching. When it needs to touch elements of the list it uses extra parameter (function `f`) to accomplish this.

Some typesystems also support setting bounds to type parameters used in polymorphic construction. Usually these bounds are provided in the form of subtyping constraints. This case of parametric polymorphism is known as *bounded* polymorphism[15]. It is supported in Haskell with type classes, in C# and Java with generics, in Scala with either virtual types or generics. This approach is further developed in *generic programming*[16], where data types are grouped using concepts. Concepts are defined in terms of requirements to the types that adhere to the concepts. There are two types of requirements syntactic (like having a certain signature), semantic (datatype satisfies certain axioms).

Finally, there is a question how freely polymorphic constructions can be used. This means whether operations with them are restricted or polymorphic entities can be used freely in all kinds of operations. Such entities got the name of *first-class citizens*[17][8]. Consider the following small example of imaginary conversation with system similar to ML

```
- fun f g = (g 1, g 1.0)
- f id
```

First line defines function that takes polymorphic function as an argument which is further applied to arguments of two different types. Then function `f` is applied for identity. For this construction to be valid it should be possible to pass polymorphic functions as parameters. This is possible when system considers such functions as first-class citizens. Therefore polymorphism that allows such things is often called *first-class* polymorphism[18]. In reality given example fails with following output:

```
stdIn:1.12-1.24 Error: operator and operand do not agree
[overload - bad instantiation]
  operator domain: 'Z[INT]
  operand:         real
  in expression:
    g 1.0
```

[15]Ukrainian equivalent: поліморфізм *з обмеженнями*
[16]Ukrainian equivalent: *параметризоване програмування*
[17]Ukrainian equivalent: *об'єкти першого класу*
[18]Ukrainian equivalent: поліморфізм *першого класу*

Reasons for this would be discussed in the following subsections.

2.2 Let-polymorphism

In this subsection we consider one variant of parametric polymorphism in detail. We would target the both sides of the question, i.e. formal specification of the system in the terms already used in the notes as well as implementation as practical algorithm.

We start with simply-typed lambda calculus $\lambda_{\rightarrow}$ and through type reconstruction problem gradually accommodate it to "polymorphic environment". The major departure from system $\lambda_{\rightarrow}$ is extension of basic types with type variables (parameters): Types:

$$T ::= BasicTypes \quad | \quad X_i \quad | \quad T \rightarrow T$$

Next step is define manipulations with type variables:

Definition 2.1. *Type substitutions* σ is a finite mapping from type variables to types.

For example, $\sigma = [X \mapsto U, Y \mapsto T]$. Application to terms and contexts respectively:

$$\begin{aligned}
\sigma(X) &= \begin{cases} T & \text{if } (X \mapsto T) \in \sigma \\ X & \text{else } (X \notin \text{dom}\sigma) \end{cases} \\
\sigma(\text{Bool}) &= \text{Bool} \\
\sigma(T_1 \rightarrow T_2) &= \sigma T_1 \rightarrow \sigma T_2 \\
\sigma(x_1 : T_1, \ldots, x_n : T_n) &= (x_1 : \sigma T_1, \ldots, x_n : \sigma T_n)
\end{aligned}$$

Two questions arise[2]:

1. "Are *all* substitution instances of t well typed?" That is, for every σ, do we have $\sigma\Gamma \vdash \sigma t : T$ for some T?

2. "Is *some* substitution instance of t well typed?" That is, can we find a σ such that $\sigma\Gamma \vdash \sigma t : T$ for some T?

The former is connected to parametric polymorphism, while the latter is related to type reconstruction problem. Usually one is interested in combination of the two, i.e. finding most general substitution that make term well-typed.

The main challenge here is how to combine together fragments that use unrelated type parameters into one term. This is typical situation with `let`-expression where some term is denoted by some name and then used in other term, possibly multiple times. This might require binding type parameters depending on the context where the term occurs. The following approach can be used to analyze this situation.

Term

Definition 2.2. Let Γ be a context and t a term. A *solution* for (Γ, t) is a pair (σ, T) such that $\sigma\Gamma \vdash \sigma t : T$.

For example, let $\Gamma = f{:}X,\ a{:}Y$ and $t = f\ a$. Then $([X \mapsto \mathrm{Nat}{\to}\mathrm{Nat}, Y \mapsto \mathrm{Nat}], \mathrm{Nat})$ is a solution for (Γ, t).

The problem with this definition is that it is declarative and therefore quite impractical. To fix this alternative definition is used[2].

Definition 2.3. A *constraint set* C is a set of equations $\{S_i = T_i\}_{i \in \{1..n\}}$.

Definition 2.4. A substitution σ is said to *unify* an equation $S = T$ if the substitution instances σS and σT are identical. We say that σ *unifies* (or *satisfies*) C if it unifies every equation in C.

Definition 2.5. The *constraint typing relation* $\Gamma \vdash t : T \mid_\chi C$ is defined by the rules in next figure.

Informally, $\Gamma \vdash t : T \mid_\chi C$ can be read "term t has type T under assumptions Γ whenever constraints C are satisfied".

The idea behind these rules is to make the term is typable independently of how its subterms are mixed together but balance it with the set of equations that will make sure that the different parts of the term agree. The rule CT-App stands out for it introduces new type parameter. All rules require that there is no name capturing for type variables introduced.

Definition 2.6. Suppose that $\Gamma \vdash t : S \mid C$. A *solution* for (Γ, t, S, C) is a pair (σ, T) such that σ satisfies C and $\sigma S = T$.

The definition 2.6 provides us with additional information in convenient form. With its help it is much easier to obtain the answer whether the necessary type substitution for the given term exists (i.e. its type can be reconstructed).

But first the certain equivalence of definitions 2.2 and 2.6 should be established. This could be presented in the form of two theorems[2]:

Theorem 2.7 (Soundness of constraint typing)**.** *Suppose that* $\Gamma \vdash t : S \mid C$. *If* (σ, T) *is a solution for* (Γ, t, S, C)*, then it is also a solution for* (Γ, t).

$$\frac{\mathtt{x{:}T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \;\; |_{\emptyset}\; \{\}} \quad \text{(CT-Var)}$$

$$\frac{\Gamma, \mathtt{x{:}T_1} \vdash \mathtt{t_2} : \mathtt{T_2} \;\; |_{\mathcal{X}}\; C}{\Gamma \vdash \lambda \mathtt{x{:}T_1}.\mathtt{t_2} : \mathtt{T_1 \to T_2} \;\; |_{\mathcal{X}}\; C} \quad \text{(CT-Abs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t_1} : \mathtt{T_1} \;\; |_{\mathcal{X}_1}\; C_1 \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_2} \;\; |_{\mathcal{X}_2}\; C_2 \\ \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(\mathtt{T_2}) = \mathcal{X}_2 \cap FV(\mathtt{T_1}) = \emptyset \\ \mathtt{X} \notin \mathcal{X}_1, \mathcal{X}_2, \mathtt{T_1}, \mathtt{T_2}, C_1, C_2, \Gamma, \mathtt{t_1}, \text{ or } \mathtt{t_2} \\ C' = C_1 \cup C_2 \cup \{\mathtt{T_1} = \mathtt{T_2 \to X}\}\end{array}}{\Gamma \vdash \mathtt{t_1\ t_2} : \mathtt{X} \;\; |_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{\mathtt{X}\}}\; C'} \quad \text{(CT-App)}$$

$$\Gamma \vdash \mathtt{0} : \mathtt{Nat} \;\; |_{\emptyset}\; \{\} \quad \text{(CT-Zero)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T} \;\; |_{\mathcal{X}}\; C \qquad C' = C \cup \{\mathtt{T} = \mathtt{Nat}\}}{\Gamma \vdash \mathtt{succ\ t_1} : \mathtt{Nat} \;\; |_{\mathcal{X}}\; C'} \quad \text{(CT-Succ)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T} \;\; |_{\mathcal{X}}\; C \qquad C' = C \cup \{\mathtt{T} = \mathtt{Nat}\}}{\Gamma \vdash \mathtt{pred\ t_1} : \mathtt{Nat} \;\; |_{\mathcal{X}}\; C'} \quad \text{(CT-Pred)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T} \;\; |_{\mathcal{X}}\; C \qquad C' = C \cup \{\mathtt{T} = \mathtt{Nat}\}}{\Gamma \vdash \mathtt{iszero\ t_1} : \mathtt{Bool} \;\; |_{\mathcal{X}}\; C'} \quad \text{(CT-IsZero)}$$

$$\Gamma \vdash \mathtt{true} : \mathtt{Bool} \;\; |_{\emptyset}\; \{\} \quad \text{(CT-True)}$$

$$\Gamma \vdash \mathtt{false} : \mathtt{Bool} \;\; |_{\emptyset}\; \{\} \quad \text{(CT-False)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t_1} : \mathtt{T_1} \;\; |_{\mathcal{X}_1}\; C_1 \\ \Gamma \vdash \mathtt{t_2} : \mathtt{T_2} \;\; |_{\mathcal{X}_2}\; C_2 \qquad \Gamma \vdash \mathtt{t_3} : \mathtt{T_3} \;\; |_{\mathcal{X}_3}\; C_3 \\ \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{\mathtt{T_1} = \mathtt{Bool}, \mathtt{T_2} = \mathtt{T_3}\}\end{array}}{\Gamma \vdash \mathtt{if\ t_1\ then\ t_2\ else\ t_3} : \mathtt{T_2} \;\; |_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3}\; C'} \quad \text{(CT-If)}$$

Figure 2.2: Constraint typing rules

Theorem 2.8 (Completeness of constraint typing)**.** *Suppose that* $\Gamma \vdash t : S \;|_{\chi}\; C$*. If* (σ, T) *is a solution for* (Γ, t) *and* $\mathrm{dom}(\sigma) \cap \chi = \varnothing$*, then there is some solution* (σ', T) *for* (Γ, t, S, C) *such that* $\sigma' \setminus \chi = \sigma$*.*

This means that if solution in one sense exists then there is solution in another sense, and either can be obtained from another.

Procedure that decides the constrain set is call *unification algorithm.* It is presented in the figure 2.3.

Unification algorithm has properties summarized in the next proposition.

Proposition 2.9. *The algorithm* ***unify*** *always terminates*

1. ***unify***(C) *halts, either by failing or by returning a substitution, for all* C*;*
2. *if* ***unify***$(C) = \sigma$ *, then* σ *is a unifier for* C*;*
3. *if* δ *is a unifier for* C*, then* ***unify***$(C) = \sigma$ *with* $\sigma \sqsubseteq \delta$*.*

This reasoning can be adopted in the type system in the form of adjusted typing rules and more flexible in the for of algorithm. Rules are presented

$$
\begin{array}{lll}
\mathit{unify}(C) & = & \text{if } C = \emptyset, \text{ then } [\,] \\
& & \text{else let } \{\mathsf{S} = \mathsf{T}\} \cup C' = C \text{ in} \\
& & \quad \text{if } \mathsf{S} = \mathsf{T} \\
& & \quad\quad \text{then } \mathit{unify}(C') \\
& & \quad \text{else if } \mathsf{S} = \mathsf{X} \text{ and } \mathsf{X} \notin \mathit{FV}(\mathsf{T}) \\
& & \quad\quad \text{then } \mathit{unify}([\mathsf{X} \mapsto \mathsf{T}]C') \circ [\mathsf{X} \mapsto \mathsf{T}] \\
& & \quad \text{else if } \mathsf{T} = \mathsf{X} \text{ and } \mathsf{X} \notin \mathit{FV}(\mathsf{S}) \\
& & \quad\quad \text{then } \mathit{unify}([\mathsf{X} \mapsto \mathsf{S}]C') \circ [\mathsf{X} \mapsto \mathsf{S}] \\
& & \quad \text{else if } \mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2 \text{ and } \mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2 \\
& & \quad\quad \text{then } \mathit{unify}(C' \cup \{\mathsf{S}_1 = \mathsf{T}_1, \mathsf{S}_2 = \mathsf{T}_2\}) \\
& & \quad \text{else} \\
& & \quad\quad \mathit{fail}
\end{array}
$$

Figure 2.3: Unification algorithm

below

$$\frac{X \notin \chi \qquad \Gamma,\ x : X \vdash t_1 : T \mid_\chi C}{\Gamma \vdash \lambda x.t_1 : X \to T \mid_{\chi \cup \{X\}} C} \text{ (CT-AbsInf)}$$

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \mid_\chi C}{\Gamma \vdash \texttt{let } x = t_1 \texttt{ in } t_2 : T_2 \mid_\chi C} \text{ (CT-LetPoly)}$$

Here the rule (CT-LetPoly) is used instead of conventional

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{let } x = t_1 \texttt{ in } t_2 : T_2} \text{ (T-LetPoly)}$$

It makes sure that each occurrence of t_1 would end up with individual copy of type variables. But there is a trade-off type checking complexity becomes exponential in the size of term if `let`-expression are nested in declaration part.

Rule-based type checking of expression $\texttt{let } x = t_1 \texttt{ in } t_2$ can be replaced with the following algorithm [2]:

1. We use the constraint typing rules to calculate a type S_1 and a set C_1 of associated constraints for the right-hand side t_1.

2. We use unification to find a most general solution σ to the constraints C_1 and apply σ to S_1 (and Γ) to obtain t_1's principal type T_1.

3. We generalize any variables remaining in T_1. If $X_1 \dots X_n$ are the remaining variables, we write $\forall X_1 \dots X_n.T_1$ for the principal *type scheme*[19] of t_1.

4. We extend the context to record the type scheme $\forall X_1 \dots X_n.T_1$ for the bound variable x, and start type checking the body t_2.

5. Each time we encounter an occurrence of x in t_2, we look up its type scheme $\forall X_1 \dots X_n.T_1$. We now generate fresh type variables $Y_1 \dots Y_n$ and use them to instantiate the type scheme, yielding $[X_1 \mapsto Y_1, \dots, X_n \mapsto Y_n]T_1$, which we use as the type of x.

It is quite clear from the rule set of the system and presented algorithm that type schemes are only allowed in the head of `let`-expression. This means that for arguments of the lambda-abstraction only monotypes are allowed. This explains why earlier example produces an error.

The type system discussed in this subsection is known as Hindley–Milner or HM type system. It is implemented in ML family of programming languages. Modern programming language use HM system with slight modification for type inference, their type systems might be more expressive than HM system in certain aspects. For instance, OCaml includes algebraic data types. Haskell programming language implements typesystem for which HM system can be considered a restriction. It will be discussed in the next subsection.

2.3 First-class polymorphism

The next typesystem is called System F. It provides first-class polymorphism. Therefore it is called *polymorphic* lambda calculus[20]. Its main feature is introduction of universal data types. Because of quantification of the type parameters it is also known as *second-order* lambda calculus[21].

As usual we start with simpler system and extend it to new one. Additional type form appears:

$$T ::= X \quad | \quad \boxed{\forall X.T} \dots$$

There are two new operations on terms:

$\boxed{\lambda X.t}$	type abstraction
$\boxed{t\ [T]}$	type application

Overall system is summarized in figure 2.4

[19] also called polytypes

[20] Ukrainian equivalent: *поліморфне* лямбда-числення

[21] Ukrainian equivalent: *2-го порядку* лямбда-числення

$\rightarrow$ $\forall$ *Based on* $\lambda_{\rightarrow}$

Syntax

t	::=		*terms:*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
		λX.t	*type abstraction*
		t [T]	*type application*
v	::=		*values:*
		λx:T.t	*abstraction value*
		λX.t	*type abstraction value*
T	::=		*types:*
		X	*type variable*
		T$\rightarrow$T	*type of functions*
		$\forall$X.T	*universal type*
Γ	::=		*contexts:*
		$\emptyset$	*empty context*
		Γ, x:T	*term variable binding*
		Γ, X	*type variable binding*

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TApp)}$$

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TappTabs)}$$

Typing $\boxed{\Gamma \vdash t : T}$

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

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

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

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TApp)}$$

Figure 2.4: System F

Typing rule (T-TAbs) introduces quantification of type variables. Evaluation rules are basically intact. Type safety theorems can be proven[2]. They retain the same form as for system $\lambda_{\rightarrow}$.

Theorem 2.10 (preservation)**.** *If* $\Gamma \vdash t : T$ *and* $t \rightarrow t'$*, then* $\Gamma \vdash t' : T$

Theorem 2.11 (progress)**.** *If* t *is a closed, well-typed term (that is,* $\vdash t : T$ *for some* T*). Then either* t *is a value or else there is some* t' *with* $t \rightarrow t'$*.*

System F can also be shown to have a normalization property, i.e. that typable terms are normalizing.

Now we are able to fully explain the earlier example regarding first-class polymorphism. Construction used can be presented in system F with the following well-typed term:

$$\begin{aligned}&\mathrm{id} \equiv \Lambda t.\ \lambda x : t.\ x\\&f \equiv \lambda g : \forall t.t \to t.\ (g\ [\mathrm{int}]\ 1, g\ [\mathrm{real}]\ 1.0)\end{aligned}$$

This can be implemented as the following Haskell code

```
f :: (forall t. t → t) → (Integer, Double)
f g = (g (1 :: Integer), g 1.0)
res = f id
```

Type reconstruction problem can be formulated for system F. The erasure for system F is defined as follows:

$$\begin{array}{lcl}\mathtt{erase}(x) & = & x\\ \mathtt{erase}(\lambda x : T.\ t) & = & \lambda x.\ \mathtt{erase}(t)\\ \mathtt{erase}(t_1\ t_2) & = & \mathtt{erase}(t_1)\ \mathtt{erase}(t_2)\\ \mathtt{erase}(\lambda X.t_2) & = & \mathtt{erase}(t_2)\\ \mathtt{erase}(t_1\ [T_2]) & = & \mathtt{erase}(t_1)\end{array}$$

Decidability for type reconstruction problem for system F was one of long standing problems in type theory. Finally the following was shown:

Theorem 2.12 (Well, 1994)**.** *t is undecidable whether, given a closed term m of the untyped lambda-calculus, there is some well-typed term t in System F such that $\mathtt{erase}(t) = m$.*

Therefore, unfortunately, in general case type annotations have to be provided for terms in System F.

Given a type with quantifiers consider the path from the root to quantifier when the type is drawn as a tree.

Definition 2.13. The *rank* of a quantifier in the type is the number of passes to the left of arrows in this path.

For example, in type term $\forall T.(\forall X.X) \to T$ first quantifier has rank 0 and second has rank 1.

Definition 2.14. Type is called of *rank* $k+1$ if its greatest rank of quantifiers is k.

Definition 2.15. Polymorphism is called *rank-n*[22], where n is a number, if it allows at most rank-n types. Rank-1 polymorphism is also called *prenex*. Polymorphism is called *arbitrary-rank* (*higher-rank*) if it allows any System F types, that is rank-n types, for any n.

Prenex polymorphism is called so because all type terms can be writtern down in prenex form[23]. That is exactly the form used in `let`-polymorphism for type schemes or polytypes.

Definition 2.16. Polymorphism is called *predicative*[24] if type variables range only over quantifier-free types, i.e. monotypes. Polymorphism is called *impredicative*[25] if type variables are allowed to range over quantified types including the very type in which they occur.

[22]Ukrainian equivalent: *рангу n*
[23]when all quantifiers are in front
[24]Ukrainian equivalent: *предикативний*
[25]Ukrainian equivalent: *непредикативний*

Chapter 3

Selected type-based programming techniques

3.1 Elements of generic programming

Generic programming is a style of programming that strives to present algorithm and data structures in most abstract form without loosing the efficiency. The central notion of generic programming is generic algorithm, which is procedural schemata parameterized by data types and independent of their specific representaion[11]. This way it is possible to group solutions of the similar problems compactly and effectively.

Key ideas of generic programming are following.

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide suffi-

ciently precise characterizations of the domain for which each algorithm is the most efficient.

3.1.1 Basic notions of generic programming

Like many other paradigms generic programming could be viewed at the following levels: general ideas, principle and notions of it, mechanisms in particular programming language that facilitate programming in this style and specific systems (libraries) that are built according to it.

The notion of *concepts*[1] is introduced. Typically they are compositions of requirements that should be satisfied by type parameters used in generic algorithm declaration. It was mentioned earlier that requirements could be syntactic and semantic. If syntactic requirements are met by types it is said that the concept is *satisfied*. If additionally semantic requirements are met then the concept is *modeled*. Logic of the algorithm relies on concepts used. Concepts can be unary, but can involve multiple type parameters as well. If extension of concept C' is included in extension of concept C then C' is said to *refine*[2] concept C. Concepts organized by refinement relation form the taxonomy. Authors of generic programming draw parallels between generic programming and mathematical construction like algebras and algebraic structures [12, 13].

Means through which programming language provides support for generic programming constructions are called *generics*[3] [10]. Usually they realize some kind of parametric polymorphism.

3.1.2 Example of algorithm generalization

One usually doesn't start with generic version. Generic algorithms are obtained from concrete cases by gradually lifting level of abstraction. In this process concepts are taking shape. Sometimes wrong generalization would be made, then extending algorithm to another case would lead to adjusted version.

For many reasons C++ became the de facto standard programming language for generic programming. Therefore code samples in this section would be presented in C++.

Let us consider two following normal function

```
int sum(int* array, int n) {
```

[1]Ukrainian equivalent: *концепти*
[2]Ukrainian equivalent: *уточнює*
[3]Ukrainian equivalent: *джженерики*

```
    int result = 0;
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}

float sum(float* array, int n) {
    float result = 0;
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}
```

Though the work they do is for different types they are very similar. We can generalize them to arguments of numeric types:

```
template<typename T>
T sum(T* array, int n) {
    T result = 0;
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}
```

This version does not take into account following case

```
std::string concatenate(std::string* array, int n) {
    std::string result = "";
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}
```

In order to do it variable initialization should be corrected:

```
template<typename T>
T sum(T* array, int n) {
    T result = T();
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}
```

Now we want to allow other types of collections:

```
template<typename Container, typename T>
T sum(const Container& array, int n) {
    T result = T();
```

```
    for (int i = 0; i < n; ++i)
        result = result + array[i];
    return result;
}
```

This generalization was in wrong direction for it is not suitable for next case (lists)

```
template<typename T>
struct node {
    node<T>* next;
    T data;
};

template<typename T>
T sum(node<T> *first, node<T> *last) {
    int s = T();
    for (; first != last; first = first->next)
        s = s + first->data;
    return s;
}
```

Here we see that use of array index operator in function template contradicts to linked list approach because list elements are not randomly accessed. In order to cover this case we need to reconsider they way how the collection is traversed for earlier cases. We come to the following function

```
template<typename T>
T sum(T* array, int n) {
    T result = T();
    for (T* current = array; current != array + n;
                                          ++current)
        result = result + *current;
    return result;
}
```

Then the generalization of iteration through elements would shape as follows[4]

```
template<typename I, typename T>
T sum(I start, I end, T init) {
    for (I current = start; current != end;
                            current = next(current))
        init = init + get(current);
    return init;
```

[4]actually in C++ dereference and increment operators could retain

```
}
```

During specialization (application of algorithm to lists) compiler would be looking for the following definitions:

```
template<typename T>
struct node {
    node<T>* next;
    T data;
};

template<typename T>
node<T>* next(node<T>* n) { return n->next; }

template<typename T>
T get(node<T>* n) { return n->data; }
```

They provide adaptation of general schemata to the linked-lists.

Based on the last version of algorithm we derive several requirements to the type parameters

- `T` *must have an additive operator* `+`
- `T` *must have an assignment operator*
- `I` *must have an inequality operator* `!=`
- `I` *must have a copy constructor*
- `I` *must have an operation* `next()` *that moves to the next value in the sequence*
- `I` *must have an operation* `get()` *that returns the current value (of type* `T`*).*

For these requirements properly named concepts could be introduced. They are given in table 3.1. Then combining information for this generic algorithms the iterator concept can be formed. It is presented in table 3.2. Then we consider another generic algorithm

```
// Requirements:  Iterator<I, T>
template<typename I, typename T>
int distance(I start, I end) {
    int i = 0;
    for (; start != end; ++start)
        ++i;
    return i;
}
```

Concept	Requirement
`CopyConstructible<T>`	`T` must have a copy constructor.
`Assignable<T>`	`T` must have an assignment operator.
`Addable<T>`	`T` must have an additive operator +.
`EqualityComparable<T>`	`T` must have operators ==, != comparing two Ts and returning a bool.

Table 3.1: Named concepts

Concept	Requirement

`Iterator<I, T>`	`EqualityComparable<I>`
	`CopyConstructible<I>`
	`Assignable<I>`
	I has operation `next()` that moves to the next value in the sequence
	I has operation `get()` that returns the current value of `T`

Table 3.2: First version of iterator concept

And it is obvious that second type parameter is redundant for it. Therefore we add requirement for associated to obtain update algorithm.

```
template<typename I>
typename iterator_traits<I>::value_type
sum(I start, I end, typename iterator_traits<I>::value_type init) {
    for (I current = start; current != end; current = next(current))
        init = init + get(current);
    return init;
}
```

For language supporting concepts algorithm would be similar to the next.

```
template<InputIterator Iter>
```

Concept	Requirement

`Iterator<I>`	`EqualityComparable<I>`
	`CopyConstructible<I>`
	`Assignable<I>`
	`value_type` is an associated type
	`next`, `get` returns `value_type`

Table 3.3: Updated version of iterator concept

```
          where Addable<Iter::value_type>
                && Assignable<Iter::value_type>
Iter::value_type
sum(Iter first, Iter last, Iter::value_type init) {
    for (; first != last; first = next(first))
        init = init + get(first);
    return init;
}
```

Finally we can write algorithm generalized also by binary operation to obtain.

```
template<InputIterator Iter,
         BinaryOperation<Iter::value_type, Iter::value_type> Op>
         where Assignable<Iter::value_type, Op::result_type>
Iter::value_type
accumulate(Iter first, Iter last, Iter::value_type init, Op op) {
    for (; first != last; first = next(first))
        init = op(init, get(first));
    return init;
}
```

Up to C++20 version of the language this features were purely experimental, therefore unstandardized and used only by geeks. Concepts were included in C++20 standard. They are defined in terms of constant expressions. Their syntax is similar to presented but slightly different. This concerns constrained template parameters, namely how n-ary concepts are referred to. Naming conventions used for them are different also. New feature in standard library use concepts heavily.

3.1.2.1 Generic programming in STL

Projects like standard template library (STL) were testing ground for generic programming. STL library is still one of the base case-studies of generic programming techniques. Many fruitful ideas and generic algorithm are implemented in this system. Concise account of it from generic programming perspective is presented in [10].

3.2 Datatype-generic programming

In generic programming we try to achieve more possibilities for parametrization without expanding possibilities for uncaught errors. *Datatype-generic programming*[5] (DGP), on the other hand, is a special approach to genericity

[5]Ukrainian equivalent: *програмування параметризоване за класами алгебр*

based on parametrization by *shape* (polytypism). DGP is quite similar to generic programming techniques. The prefix 'Data' is used to highlight the differences from them. Extended account on DGP can be found in [14] and [15].

3.2.1 Roy–Floyd-Warshall algorithm

Let us start with motivational example.

Reachability problem In the following code solution for reachability in the graph problem is considered. It is known as Warshall's algorithm. It should be mentioned that it find routes for all pairs of vertices. It starts from adjacency matrix of initial graph and finishes with same matrix for its transitive closure.

```
for each k, 1 ≤ k ≤ N
do for each pair (i, j), 1 ≤ i, j ≤ N
   do a_ij := a_ij ∨ (a_ik ∧ a_kj)
   end_for
end_for
```

In order to talk about correctness of this algorithm its pre- and post-conditions should be written down. Without this information algorithm definition is not fully meaningful. Therefore, precond:

- a matrix $N \times N$ of Bool
- a_{ij} = true if there an edge (i, j).

Postcond:

- a_{ij} = true if there path $[i, j]$.

Least-cost path problem Similar to the previous algorithm this one looks for shortest path for all pairs of vertices. It is known as Floyd's algorithm.

```
for each k, 1 ≤ k ≤ N
do for each pair (i, j), 1 ≤ i, j ≤ N
   do a_ij := a_ij ↓ (a_ik + a_kj)
   end_for
end_for
```

where $x \downarrow y$ is a minimum of x and y. Once again following information is provided for completeness. Precond:

- a matrix $N \times N$ of $\mathrm{Int} \cup \{\infty\}$.
- a_{ij} is a cost passing edge (i, j).
- $a_{ij} = \infty$ if there is no edge.

Postcond:

- a_{ij} is the least cost passing from i to j.

Bottleneck problem The last example is maximin paths. The solution is presented below

```
for each k, 1 ≤ k ≤ N
do for each pair (i, j), 1 ≤ i, j ≤ N
   do a_ij := a_ij ↑ (a_ik ↓ a_kj)
   end_for
end_for
```

where additionally $x \uparrow y$ is a maximum of x and y. Precond:

- a matrix $N \times N$ of $\mathrm{Int} \cup \{\infty\}$.
- a_{ij} is a least height of bridge on edge (i, j).
- $a_{ij} = 0$ if there is no edge.

Postcond:

- a_{ij} is the max possible height of vehicle to pass from i to j.

Generic path-algorithm (transitive closure algorithm) All three algorithms have similar shapes. The only difference are initial values and the concrete binary operations used in each of them. If other algorithms are also taken into account we achieve abstract (general) form of the algorithm.

```
for each k, 1 ≤ k ≤ N
do for each pair (i, j), 1 ≤ i, j ≤ N
   do a_ij := a_ij ⊕ (a_ik ⊗ a*_kk ⊗ a_kj)
   end_for
end_for
```

It is possible to talk about applicability (executability) and correctness of the algorithm. Assume that the type of the matrix elements is A. Then the algorithm applies when this type supports operations used in algorithm. This situation is studied in abstract algebra. The list of operations supported

and their *arities*[6] are called signature. Hence for applicability type of the matrix elements A is a carrier of algebra with signature $(\oplus, \otimes, ^*)$. Once again we specify pre- and post-conditions relying to some extent on mentioned operations. Precond:

- a matrix $N \times N$ of A.
- $a_{ij} = \sum\{e : e \text{ is an edge from } i \text{ to } j : \texttt{label}\, e\}$.

Postcond:

- $a_{ij} = \sum\{p : p \text{ is path from } i \text{ to } j : \texttt{weight}\, p\}$
  $\texttt{weight}\, [] = 1$
  $\texttt{weight}(e : p) = \texttt{label}\, e \otimes \texttt{weight}\, p$

Now for the algorithm to make sense certain requirements to algebraic operations should be met. Algorithm is correct if $(A, \oplus, \otimes, ^*, 0, 1)$ is a *regular* algebra[7], which means following.

- $(0, 1, \oplus, \otimes)$ form semiring with unit structure;
- $\oplus$ should be idempotent
  - partial order on A: $a \leq b \Leftrightarrow a \oplus b = b$;
- * – least fixed-point operator for $\otimes$:

$$1 \oplus a \otimes a^* \leq a^* \geq a^* \otimes a \oplus 1$$
$$a \otimes x \leq x \rightarrow a^* \otimes x \leq x$$
$$x \otimes a \leq x \rightarrow x \otimes a^* \leq x$$

In abstract algebra these requirements are usually called axioms. Some information on this matter is provided in the next subsection.

3.2.2 Algebras & Morphisms

Good account of algebras is provided in [16]. Here only the basic notions are recalled. (Concrete) Algebra is a set A (called) with a number of operations on it. Operations f is a function of type $A^k \rightarrow A$, k is operation arity. There

[6]Ukrainian equivalent: *арності*
[7]Ukrainian equivalent: *регулярна* алгебра

are prefix, infix, suffix forms of notation. In mathematics and programming binary operations are usually written infix.

Here are a few algebra examples[8]. Most of them are self-explaining. Symbol $+\!\!+$ denotes word concatenation.

- $(N, 0, (+))$ with $0 :: 1 \to N$, $(+) :: N \times N \to N$
- $(N, 0, (\uparrow))$ with $0 :: 1 \to N$, $(\uparrow) :: N \times N \to N$
- $(R, 1, (\times))$ with $1 :: 1 \to R$, $(\times) :: R \times R \to R$
- $(B, \top, (\equiv))$ with $\top :: 1 \to B$, $(\equiv) :: B \times B \to B$
- $(B, \bot, (\vee))$ with $\bot :: 1 \to B$, $(\vee) :: B \times B \to B$
- $(B, \top, (\wedge))$ with $\top :: 1 \to B$, $(\wedge) :: B \times B \to B$
- $(A^*, \varepsilon, (+\!\!+))$ with $\varepsilon :: 1 \to A^*$, $(+\!\!+) :: A^* \times A^* \to A^*$

Datatypes in Haskell One of the features of the programming languages is the possibility to define self referential types. Therefore such types are called *inductive* types[9]. Algebras arise in Haskell naturally from type definition using `data` keyword. Consider the following example in Haskell.

```
data Nat = zero | succ Nat
```

First word of alternatives defined in the type are called value constructors[10] The following conversation shows their type:

```
*Main> :t succ
succ :: Nat -> Nat
```

This is exactly the type of unary operation in the algebra with the carrier `Nat`. This way data type and value constructors form an algebra

- $(\texttt{Nat}, \texttt{zero}, \texttt{succ})$ with $\texttt{zero} :: 1 \to \texttt{Nat}$, $(\texttt{succ}) :: \texttt{Nat} \to \texttt{Nat}$

Therefore such types are also called *algebraic*[11].

[8] by Haskell convention operators are parenthesized

[9] Ukrainian equivalent: *індуктивні* типи

[10] For illustration their first letters are lowercased

[11] Ukrainian equivalent: *алгебраїчні*

Algebra morphisms Algebra *morphisms*[12] are the algebraic structure-preserving maps between two algebras. They depend on the class of algebras. For each class of algebras we have specific type of morphisms. Both source and target algebra should belong to the same class.

Here is an example. The algebra morphisms $\exp : (N, 0, (+)) \to (R, 1, (\times))$ is defined in the following way (exponentiation):

$$\begin{aligned} \exp 0 \qquad &= 1 \\ \exp(x + y) &= (\exp x) \times (\exp y) \end{aligned}$$

Another example is the function $\texttt{length} : (A^*, \varepsilon, (+\!\!+)) \to (N, 0, (+))$ between strings and numbers

$$\begin{aligned} \texttt{length}\, \varepsilon \qquad &= 0 \\ \texttt{length}(s +\!\!+ q) &= \texttt{length}\, s + \texttt{length}\, q \end{aligned}$$

Defining function on inductive datatypes Functions can be defined on inductive types. For this *pattern matching*[13] is used where for each kind of values of the type certain processing specified. Consider example in Haskell:

```
data Nat = zero | succ Nat
-- pattern matching
pred zero = zero
pred (succ x) = x
even zero = True
even (succ x) = not (even x)
```

This way morphisms between algebras can be constructed.

Algebras of some class with their morphisms form a category. Another level of abstraction can be form using this categories.

Types parameterization also fits this scheme. Consider the notion of type constructors in Haskell. These entities:

- create new types from existing;
- thus have type parameters.

The `List` defined below gives an example of type constructor.

```
data List a = nil | cons a (List a)
```

Action of constructor `List` can be extended to functions

[12] also homomorphisms

[13] Ukrainian equivalent: *зіставлення зі зразком*

```
map_List f =  h where
  h nil = nil
  h (cons x xs) = cons (f x) (h xs)
```

3.2.3 Type constructors and functors

Type constructors like `List` enjoy good property. Name they preserve that categorical structure.

In order to talk about it we need to introduce couple notions first.

Definition 3.1. A *category* C consists of

- collection of *objects* $\operatorname{Obj} C$
- collection of *morphisms* (*arrows*) $\operatorname{Mor} C$

such that

- each $f \in \operatorname{Mor} C$ is assigned a type $f : a \to b$, where $a, b \in \operatorname{Obj}$
- for $f : a \to b$, $g : b \to c$ the *composition* is defined $f \fatsemi g : a \to c$
- for all $f : a \to b$, $g : b \to c$; $h : c \to d$

$$f \fatsemi (g \fatsemi h) = (f \fatsemi g) \fatsemi h$$

- for each $a \in \operatorname{Obj} C$ there is the *identity* $1_a : a \to a$ such that for $f : a \to b$

$$f \fatsemi 1_b = f = 1_a \fatsemi f$$

Type constructors generalize to the notion of functor. A functor $F : C \to D$ is a map between categories that preserves structures

$$\begin{aligned} F &: \operatorname{Obj} C \to \operatorname{Obj} D \\ F &: C(a, b) \to D(Fa, Fb) \end{aligned}$$

such that

$$\begin{aligned} F1_a &= 1_{Fa} \\ F(f \fatsemi g) &= Ff \fatsemi Fg. \end{aligned}$$

n-ary functors Using product categories functors can be further generalized to n-ary functors. Functor $F\colon C_1 \times \cdots \times C_n \to D$ is a map

$$F\colon \prod_{i=1}^{n} \operatorname{Obj} C_i \to \operatorname{Obj} D$$
$$F\colon \prod_{i=1}^{n} C_i(a_i, b_i) \to D(Fa_1 \ldots a_n, Fb_1 \ldots b_n)$$

such that

$$\begin{aligned} F1_{a_1} \ldots 1_{a_n} &= 1_{Fa_1 \ldots a_n} \\ F(f_1 \fatsemi g_1 \ldots f_n \fatsemi g_n) &= Ff_1 \ldots f_n \fatsemi Fg_1 \ldots g_n. \end{aligned}$$

In Haskell programming language there is the category $\mathcal{H}_{ASK}$. Object and morphisms are Haskell types[14] and functions respectively:

$$\begin{aligned} \operatorname{Obj}_{\mathcal{H}_{ASK}} &= \{T \mid T - \text{is a type}\} \\ \mathcal{H}_{ASK}(T, S) &= \{f \mid f - \text{is a function of type } T \to S\} \end{aligned}$$

Composition defined as usual function composition:

```
f ∘ g = h where h x = f (g x)
```

$$g \fatsemi f = f \circ g$$

3.2.4 Polynomial Functors

Polynomial functors[15] play important role because most algebraic data type definitions in Haskell can be derived using them. The reason for this is that basic building blocks like alternatives and multiple parameters in constructors translate to well-known categorical notions of sum and product. And also constants (nullary data constructor) obviously translate to constant functors. Finally, flexibility with type parameters is achieved using extraction functors.

3.2.4.1 Const and extraction functors

Let $a \in \operatorname{Obj} C$. Define constant functor a^{K} as follows.

$$a^{\mathrm{K}}\colon x_1 \ldots x_n \mapsto a$$
$$a^{\mathrm{K}}\colon f_1 \ldots f_n \mapsto 1_a$$

Then $a^{\mathrm{K}}\colon C^n \to C$ is an n-functor. Here is an example of code for the case when $n = k = 2$.

[14] due to limitations some subset of types is used

[15] Ukrainian equivalent: *поліноміальний* функтори

```
data Ex22 a b = In22 b
map22 f g (In22 x) = In22 (g x)
```

Define the family of extraction functors $\mathrm{Ex}_k^n \colon C_1 \times \cdots \times C_n \to C_k$ such that

$$\mathrm{Ex}_k^n \colon x_1 \ldots x_n \mapsto x_k$$
$$\mathrm{Ex}_k^n \colon f_1 \ldots f_n \mapsto f_k$$

Particularly $I_C = \mathrm{Ex}_1^1 \colon C \to C$ (identity functor):

$$I_C x = x$$
$$I_C f = f$$

3.2.4.2 Sum functor

Sum functor captures the notion of alternatives. Consider the following fragment in Haskell[16].

```
data a + b = inl a | inr b

f + g = h where
    h (inl x) = inl (f x)
    h (inr x) = inr (g x)

f ▽ g = h where
    h (inl x) = f x
    h (inr x) = g x
```

This can be represented in, so called, pointfree form or style:

$$h = f + g \Leftrightarrow \begin{cases} \texttt{inl} \fatsemi h = f \fatsemi \texttt{inl} \\ \texttt{inr} \fatsemi h = g \fatsemi \texttt{inr} \end{cases}$$
$$h = f \mathbin{▽} g \Leftrightarrow \begin{cases} \texttt{inl} \fatsemi h = f \\ \texttt{inr} \fatsemi h = g \end{cases}$$

This can be understood as the definition of the function as the unique solution of the corresponding equation. In category theory construction with these properties is called coproduct. Specifically in category theory respective equations are typically represented as commutative diagrams.

[16] To define infix type constructors `TypeOperators` option is required here

This can be used to prove identities by calculating. If we have to prove something for $f \triangledown g \fatsemi h$. Let us just try to express it as "junk" itself:

$$f \triangledown g \fatsemi h = \alpha \triangledown \beta \Leftrightarrow \begin{cases} \texttt{inl} \fatsemi f \triangledown g \fatsemi h = \alpha \\ \texttt{inr} \fatsemi f \triangledown g \fatsemi h = \beta \end{cases} \Leftrightarrow \begin{cases} \alpha = f \fatsemi h \\ \beta = g \fatsemi h \end{cases}$$

That is $f \triangledown g \fatsemi h = (f \fatsemi h) \triangledown (g \fatsemi h)$

3.2.4.3 Product functor

For type products representation we can use either tuples or infix variant of type constructor. Here is the code fragment of the former.

```
type a × b =  (a, b)

exl:: a × b →  a
exl (x, y) = x
exr (x, y) = y
(f × g) (x, y) =  (f x, g y)
(f △ g) x =  (f x, g x)
```

The last two identities can be rewritten in pointfree form as follows.

$$h = f \times g \Leftrightarrow \begin{cases} h \fatsemi \texttt{exl} = \texttt{exl} \fatsemi f \\ h \fatsemi \texttt{exr} = \texttt{exr} \fatsemi g \end{cases}$$

$$h = f \triangledown g \Leftrightarrow \begin{cases} h \fatsemi \texttt{exl} = f \\ h \fatsemi \texttt{exr} = g \end{cases}$$

3.2.4.4 Combining functors

For

$$\begin{aligned} &F_i \colon C_1 \times \cdots \times C_n \to D_i, && i = \overline{1,m} \\ &G \colon D_1 \times \cdots \times D_m \to E \end{aligned}$$

there is functor $H \colon C_1 \times \cdots \times C_n \to E$ such that

$$\begin{aligned} Ha_1 \ldots a_n &= G\ (F_1 a_1 \ldots a_n) \ldots (F_m a_1 \ldots a_n) \\ Hf_1 \ldots f_n &= G\ (F_1 f_1 \ldots f_n) \ldots (F_m f_1 \ldots f_n) \end{aligned}$$

i.e.

$$H\__1 \cdots \__n = G\ (F_1 \__1 \cdots \__n) \ldots (F_n \__1 \cdots \__n)$$

3.2.4.5 Polynomial functor definition and example

Definition 3.2. A *polynomial functor* is a functor obtained by combination of extract and constant functors.

Example. Let's get back to the list example

```
data List a = nil | cons a (List a)
```

Its *pattern* functor[17] is the binary polynomial functor L defined on objects as follows.

$$L\ a\ z = 1 + a \times z$$

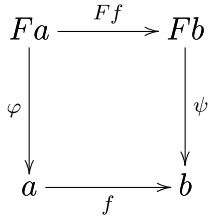
with obvious definition on morphisms.

3.2.5 Functor algebras

Let F be an endofunctor on category C, i.e. $F\colon C \to C$.

Definition 3.3. An *F-algebra* is a pair (a, φ), such that $\varphi\colon Fa \to a$.

Definition 3.4. An *F-algebra morphism* $f\colon (a, \varphi) \to (b, \psi)$ is a morphism $f\colon a \to b$ of category C, such that $f \circ \varphi = \beta \circ Ff$. Equivalently following diagram



commutes.

We can compose F-algebra morphisms using morphism composition is C.

Proposition 3.5. *F-algebra with F-algebra morphisms form a category. Let's denote it $(F : C)$.*

Example. Let Square : **Set** $\to$ **Set** be a doubling functor $x \mapsto x \times x$. Then (Square, **Set**) is a category of magmas, i.e. (unconstrained) algebras with single binary operation.

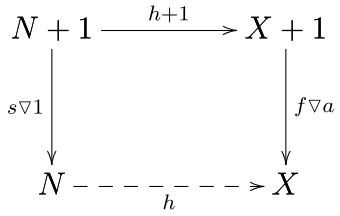
Example. In general, if F : **Set** $\to$ **Set** is a polynomial functor then $(F, \mathbf{Set})$ is a category of universal algebras with signature determined by functor F.

[17] Ukrainian equivalent: *каркасний* функтор

Definition 3.6. An initial object of category $(F : C)$ is called *initial F-algebra*, denoted (μ_F, in_F).

Definition 3.7. Unique morphism from initial F-algebra to any F-algebra (a, ϕ) is called *catamorphism*, denoted $(\!|\phi|\!)$.

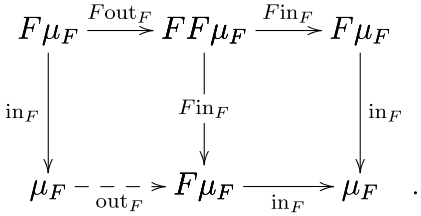
Example. Consider algebra with unary operation f and constant e. To it corresponds the polynomial functor $F\colon x \mapsto 1 + x$. The initial algebra in this case is $(N, 1 \triangledown s)$. This can be demostrated by commutative diagram



which is the categorical reformulation of recursive definition of h.

Lemma 3.8 (Plotkin)**.** *An initial F algebra is an isomorphism.*

Proof. Let (μ_F, in_F) be an initial F-algebra. Consider the morphism $F(\mathrm{in}_F)$. Clear it is an F-algebra $(F\mu_F, F(\mathrm{in}_F))$. We denote by $\mathrm{out}_F\colon F\mu_F \to \mu_F$ a catamorphism for this algebra. Let's prove that out_F is an inverse of $\mathrm{in_F}$. Consider following commutative diagram



Outer square defines catamorphism $(\!|\mathrm{in}_F|\!)$. By universal property $(\!|\mathrm{in}_F|\!) = \mathrm{id}_{\mu_F}$. Then immediately from left square:

$$\mathrm{out}_F \circ \mathrm{in}_F = F(\mathrm{in}_F \circ \mathrm{out}_F) = \mathrm{id}_{F\mu_F}$$

□

This construction has remarkable property. If the functor F is parameterized by additional parameters, i.e. it is an n-functor, and it satisfies certain conditions then initial F-algebra μ_F is also functorial. Many inductive types (e.g. in Haskell) can be formalized this way. This is the case for types

like lists. It also explains uniqueness of recursive definitions on inductive datatypes.

The dual construction to initial algebras are final co-algebras, final objects of category of co-algebras. Using them coinductive datatypes such as streams are formalized.

Bibliography

[1] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[2] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[3] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97, pages 2277–2308. CRC Press, 2004.

[4] Niklaus Wirth. A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3):32–39, 2008.

[5] R. M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[6] Kevin Klement. Russell's 1903–05 anticipation of the lambda calculus. *History and Philosophy of Logic*, 24:15–37, 2003.

[7] Bernard Linsky. The notation in principia mathematica. In Edward N. Zalta, editor, *Stanford Encyclopedia of Philosophy*. 2011.

[8] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000. Reprinted from lecture notes dated 1968, with a foreword [17].

[9] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[10] Jeremy G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, August 2005.

[11] David Musser and Alexander A. Stepanov. Generic programming. In *Symbolic and algebraic computation: ISSAC '88*, pages 13–25. Springer, 1988.

[12] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.

[13] Alexander A. Stepanov and Daniel E. Rose. *From Mathematics to Generic Programming*. Addison-Wesley Professional, 1st edition, 2014.

[14] Roland Carl Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.

[15] Jeremy Gibbons. Datatype-generic programming. In *Proceedings of the 2006 International Conference on Datatype-generic Programming*, volume 4719 of *LNCS*, pages 1–71, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag Berlin Heidelberg, 2012.

[17] Peter D. Mosses. A foreword to 'Fundamental concepts in programming languages'. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000.

Glossary

abstract syntax tree абстрактне синтаксичне дерево. 7, 11, *see* parse tree
algebra алгебра.
  carrier носій. 52
  regular регулярна. 52
aritie арність. 52

coercion неявне приведення типів. 30
concept концепт. 44, 49
contravariant контраваріантність. 33
covariant коваріантність. 33

datatype-generic programming програмування параметризоване за класами алгебр. 49
derivation виведення. 14
  evaluation виведення (кроку) обчислення. 14
  induction on індукція за виведеннями. 14
  tree дерево. 14
  typing виведення типу. 18

evaluation обчислення.
  eager жадібне. 23
  greedy жадібне. 23
  lazy ліниві. 23
  one-step один крок. 13, 14
  strategy стратегія. 14, 22
  strict строге. 23

first-class citizen об'єкт першого класу. 34
functor функтор.
  pattern каркасний. 59
  polynomial поліноміальний. 56