CHAPTER 0

PRELIMINARIES FOR TOC

0.1 Inference Rules

An *inference rule* is a function from a set of premises to a single conclusion. This is denoted as follows:

$$\frac{\text{premise } 1 \quad \ldots \quad \text{premise } n}{\text{conclusion}}\text{Rule 1}$$

The rule 'Rule 1' states that if all premises $1, 2, \ldots, n$ hold, then the conclusion holds. We can have $n \geq 0$ premises. In particular, if $n = 0$, then the rule is an *axiom*. In formal systems, the inference rules can be thought of as proofs, and the premises the hypotheses of the proof.

We will now go through some inference rules. The Modus Ponens in propositional logic can be denoted as follows:

$$\frac{A \to B \quad A}{B}\text{MP}$$

Similarly, Modus Tollenz is denoted as follows:

$$\frac{A \to B \quad \neg B}{\neg A}\text{MT}$$

We can use the inference rules to define a natural number. In words, the following is a definition for them:

- 0 is a natural number;
- if n is a natural number, then its successor $n+1$ is also a natural number.

The inference rules in this case are given below:

$$\frac{}{0 \text{ nat}}\text{zero} \qquad\qquad \frac{n \text{ nat}}{succ(n) \text{ nat}}\text{succ}$$

Note that the first rule is an axiom.

A *formal system* is a set of inference rules with at least one axiom, e.g. the definition of the natural numbers. Inference rules can be used to define semantics and type systems.

0.2 Mathematical and Structural Induction

We will now look at *mathematical induction*. It turns out that many of the proofs we will see in the course follow a similar structure to induction, called *structural induction*.

We can use mathematical induction to prove a property $P(n)$ holds for all natural numbers $n \in \mathbb{N}$. Its inference rule is:

$$\frac{P(0) \quad P(k) \to P(k+1)\ \forall k \in \mathbb{N}}{P(n)\ \forall n \in \mathbb{N}}\text{MI}$$

That is, if $P(0)$ holds and for all $k \in \mathbb{N}$, $P(k)$ implies $P(k+1)$, then $P(n)$ holds for all $n \in \mathbb{N}$. This follows logically from a domino effect ($P(0)$ implies $P(1)$; $P(1)$ implies $P(2)$; and so on).

We will now illustrate mathematical induction with an example. We prove that for all $n \in \mathbb{N}$,

$$0 + 1 + \cdots + n = \frac{n(n+1)}{2}.$$

If $n = 0$, then we find that

$$0 = \frac{0 \cdot 1}{2},$$

so $P(0)$ holds. Now, assume that for some $k \in \mathbb{N}$, $P(k)$ holds, i.e. we have

$$0 + 1 + \cdots + k = \frac{k(k+1)}{2}.$$

In that case,

$$\begin{aligned} 0 + 1 + \cdots + k + (k+1) &= \frac{k(k+1)}{2} + (k+1) \\ &= \frac{k(k+1) + (k+1) \cdot 2}{2} \\ &= \frac{(k+1)(k+2)}{2} \\ &= \frac{(k+1)((k+1)+1)}{2}. \end{aligned}$$

So, if $P(k)$ holds, then $P(k+1)$ holds. Hence, by mathematical induction, the statement $P(n)$ holds for all $n \in \mathbb{N}$.

We now consider a different version of induction- structural induction. It generalises the concept of mathematical induction, and can be used on different structures we shall define in the course, such as recursive structures.

We will illustrate how to prove something using structural induction. So, assume that X is a recursively-defined structure, and let P be the property we want to show.

- We will first show that $P(x)$ holds for all $x \in X$ that are the base case (e.g. the number 0).
- Now, by assuming the proof holds for a substructure in X (e.g. the number k), we show that the property holds for a bigger structure (e.g. the number $k+1$).

We will see examples following this technique later.

0.3 Syntax

To specify the syntax of nested phrases, e.g. expressions and commands, we need a *context-free grammar*. The grammar of a language is a set of rules that specify how the phrases are formed. Each rule specifies how a phrase can be formed from symbols (e.g. keywords and punctuation) and simpler phrases.

A (context-free) grammar consists of:

- a set of *terminal symbols*;
- a set of *non-terminal symbols* (representing a phrase);
- a *sentence symbol* (a non-terminal symbol representing a sentence); and
- a set of *production rules* (how phrases are composed from terminal symbols and sub-phrases).

We will use the *Backus Naur Form* (BNF) notation to express a grammar. It is denoted

$$S ::= \alpha \mid \beta \mid \gamma.$$

Here, the rule S states that a term of type S can be either α, β or γ (each of which is a sequence of terminal or non-terminal symbols).

We will illustrate the grammar using a simple programming language called *Expr*. Here,

- the terminal symbols are the natural numbers $0, 1, \ldots,$ $\mathit{true}, \mathit{false}, +, -, =$;
- the non-terminal symbols are: exp and val;
- the sentence symbol is exp;
- the production rules are:

$$\begin{aligned} \mathit{exp} &::= \mathit{val} \mid \mathit{exp} + \mathit{val} \mid \mathit{exp} - \mathit{val} \mid \mathit{exp} == \mathit{val} \\ \mathit{val} &::= 0 \mid 1 \mid \cdots \mid \mathit{true} \mid \mathit{false}. \end{aligned}$$

The grammar above defines the formal syntax of a language for the language *Expr*. In particular, we can write mathematical expressions in this language. The grammar defines the *smallest* set of entities containing natural numbers and booleans, and is closed under the production rules.

0.4 Semantics

Semantics studies the meaning of a programming language (PL) or a calculus. In particular, it gives a non-ambiguous meaning to a program. There are 2 types of semantics:

- *Static semantics* (called type system) assigns types to language terms;
- *Dynamic semantics* gives meaning to programs in a language based on their behaviour during execution- this is independent of the compiler.

In practice, PLs have informal (but not imprecise) or semi-formal semantics. We typically only find formal semantics for calculi.

There are 3 types of dynamic semantics:

- *Operational semantics* defines how a program is executed, one step at a time, to produce a result/observable behaviour.
- *Denotational semantics* defines the meaning of a program as a mathematical object, e.g. treating them as a function.
- *Axiomatic semantics* defines a logic for pre- and post-conditions, and a proof system; the meaning of a program is what can be proved about its behaviour.

We will mostly consider operational semantics.

Reduction rules

For the language *Expr*, we can define the operational semantics by formalising its evaluations. We write $exp \to exp'$ to denote that exp reduces to exp' by one-step evaluation. Reductions are only possible if they can be derived from the set of reduction rules. The reduction rules will be given by a set of inference rules.

The meaning of an expression exp is calculated by reducing it:

$$exp \to exp_1 \to exp_2 \to \dots$$

until we reach a *stuck expression*. A stuck expression is either:

- a value (representing the meaning of the original expression). In *Expr*, a value is produced by $val ::= \bullet$; or
- a non-value which cannot be reduced any further (e.g. $1 + true$, representing a run-time error).

The use of inference rules following the structure of the syntax to define reductions is known as *Structural Operational Semantics*.

In the language *Expr*, we want $+$, $-$ and $=$ to represent their mathematical operations, i.e.

- $1 + 1 \to 2$;
- $3 - 2 \to 1$; and

- $2 == 2 \to \mathit{true}$.

To do so, we define reduction rules, starting with + and −:

$$\frac{\text{if } \mathit{val}_1 \text{ and } \mathit{val}_2 \text{ are integer literals and } \mathit{val}_3 \text{ is their sum}^*}{\mathit{val}_1 + \mathit{val}_2 \to \mathit{val}_3}\text{R-Sum}$$
$$\frac{\text{if } \mathit{val}_1 \text{ and } \mathit{val}_2 \text{ are integer literals and } \mathit{val}_3 \text{ is their difference}^*}{\mathit{val}_1 - \mathit{val}_2 \to \mathit{val}_3}\text{R-Diff.}$$

We denote by $*$ the side-conditions. These are not premises, but we require them for the inference rule to be applied on top of the premises. So, the rules above are axioms. We can similarly define the rule for $=$:

$$\frac{\text{if } \mathit{val}_1 \text{ and } \mathit{val}_2 \text{ are equal integer literals}^*}{\mathit{val}_1 == \mathit{val}_2 \to \mathit{true}}\text{R-Eq}_1$$
$$\frac{\text{if } \mathit{val}_1 \text{ and } \mathit{val}_2 \text{ are equal integer literals}^*}{\mathit{val}_1 == \mathit{val}_2 \to \mathit{false}}\text{R-Eq}_2.$$

The inference rules can also be extended to expressions in general.

$$\frac{\mathit{exp} \to \mathit{exp}'}{\mathit{exp} + \mathit{val} \to \mathit{exp}' + \mathit{val}}\text{R-expSum}$$
$$\frac{\mathit{exp} \to \mathit{exp}'}{\mathit{exp} - \mathit{val} \to \mathit{exp}' - \mathit{val}}\text{R-expDiff}$$
$$\frac{\mathit{exp} \to \mathit{exp}'}{\mathit{exp} == \mathit{val} \to \mathit{exp}' == \mathit{val}}\text{R-expEq.}$$

Type System

From the reduction rules given, for any operational semantics for *Expr*, we would either get a value (e.g. $1+5-2$) or stuck expressions (e.g. $1+\mathit{true}$). To avoid these run-time errors, we enhance the language with static semantics- a *type system*. This ensures that ill-formed expressions are ruled out.

We will now define a type system for *Expr*. We do so using inference rules. First, we define the syntax of types:

$$T ::= \mathit{bool} \mid \mathit{int}.$$

A *type judgment* for *Expr* is of the form

$$\vdash \mathit{expr} : T.$$

This denotes that the expression *expr* is of type T, with respect to the typing context (which is empty in this case). In general, the typing context is non-empty, and of the form:

$$\Gamma \vdash \mathit{term} : \mathit{type}.$$

Here, Γ is a *typing constraint*, meaning that it has some type assumptions, e.g. for variables. This means that under Γ, the *term* has the given *type*.

The type system for *Expr* is given with the following rules:

$$\frac{}{\vdash true\colon bool}\text{T-True}$$

$$\frac{}{\vdash false\colon bool}\text{T-False}$$

$$\frac{\text{if } val \text{ is an integer literal}^*}{\vdash val\colon int}\text{T-Int}$$

$$\frac{\vdash exp\colon int \qquad \vdash val\colon int}{\vdash exp + val\colon int}\text{T-Sum}$$

$$\frac{\vdash exp\colon int \qquad \vdash val\colon int}{\vdash exp - val\colon int}\text{T-Diff}$$

$$\frac{\vdash exp\colon int \qquad \vdash val\colon int}{\vdash exp == val\colon bool}\text{T-Eq.}$$

A type system for a PL gives us guarantees about programs during runtime. In particular, they ensure that unwanted type errors are ruled out. This intuition is formalised in the *type safety* property, which tells us that well-typed programs don't go wrong. In particular, there are 2 theorems that make up the type safety property:

- *type preservation*- if a term is well-typed and reduces, then it reduces to a well-typed term;
- *progress*- if a term is well-typed, then it either reduces or is a value.

Note that this doesn't guarantee termination of the reduction process.

In summary, the *Expr* language has been defined using:

- *syntax*- using a grammar in BNF;
- *dynamic semantics*- meaning of terms of the language (as behaviour); and
- *type system*- assigning types to terms of the language.