Chapter 1

Fundamental Concepts

This book is about the mathematical foundations of programming, with a special attention on computing with infinite objects. How can mathematics help in programming? There are several important roles that mathematical modelling and reasoning plays in computer science.

First of all, mathematics helps us *understand the meaning of programs.* We can use the power of abstraction to interpret data structures and computational processes as mathematical entities. This view allows us to forget about some of the complicated practical details of implementation and to concentrate on the high-level concepts.

A second role of mathematics is in *specification*: stating clearly and precisely what a program must do. Usually the purpose of a program is expressed in common language with all its ambiguities and imprecisions. This opens the risk of the final software being inadequate to the requirements. But if these were not expressed in a clear and precise way, it may be very difficult to point out where the code went wrong. Logical mathematical specification allows us to state with absolute precision what a program must do. We can use the specification as a guide in development and as a test to verify the success of the final software.

A third role for mathematics is in *reasoning* about programs. Using symbolic logic we can prove properties of them. First of all, we can verify, with formal methods, that the software satisfies the specifications developed according to the second point. Further, we can also investigate high-level characteristics of the software and its behaviour and even its computational complexity. This can be done with the help of computers: *proof assistants* are systems that allow a user to write both specifications and programs and verify correctness properties interactively with the systems. If the process is successful it guarantees that the result are certain, eliminating the possibility of any human error.

Finally, the fourth role that mathematics plays in computer science is that of *abstract interpretation.* We see data structures as mathematical objects, to clarify their nature. Data is implemented in specific ways in various computer architecture. By abstracting from the specific details of the implementation, we can concentrate on their essential nature, which doesn't depend on the concrete realization. This nature can be described as part of logical/mathematical reality.

Syntax and Semantics

Two important sides of the use of mathematical modelling in computer science are *syntax* and *semantics*.

Syntax describes the way we write expressions, terms, formulas, programs. It tells us how a programming language is defined precisely: what symbols we use; what are the reserved words and the identifiers; how we combine symbols to form correct expressions and instructions; how we combine instructions to create programs.

Semantics consists in giving the meaning of expressions, terms, formulas, programs. Without it a programming language would consist only of a set of meaningless strings of symbols. Semantics specifies what the symbols stand for and what happens when we combine them according to the rules of the syntax. There are two fundamental ways of giving meaning to a language.

Denotational Semantics interprets expressions and programs as objects in some mathematical space. The meaning of a string of symbols is some abstract entity. The advantage of this approach is that we can use powerful mathematical methods to derive properties of the programs.

Operational Semantics interprets expressions and programs as computations. The meaning of a program consists in what happens when we run it. Operational semantics doesn't move away from syntax to a mathematical reality. Instead it adds a dynamic element to the syntax, stating how syntactic entities are computed.

Arithmetic Expressions

To illustrate the basic functions of syntax and semantics, we use a very simple language of arithmetic expressions. It is extremely limited and not much interesting can be done with it. But it is sufficient to clarify the main notions. Later we'll more to much richer languages.

This language contains natural numbers, 0, 1, 2, et cetera, and the Boolean expressions true and false. we can combine them with simple operations: successor and predecessor for numbers, a test of whether an number is 0, a conditional (if-then-else) constructor.

We give the syntax in two different but equivalent ways: first in Backus-Naur Form (BNF), then by derivation rules. The BNF is simple and compact, while derivation rules are a bit lengthier and involved. However, there are richer languages that cannot be specified by a BNF, so derivation rules will be necessary. They are more flexible and powerful than BNF. For this reasons, we start introducing them immediately in this simple example to make their use clear in view of future more complex languages.

Backus-Naur Form for Arithmetic Expressions:

$$\begin{aligned} \mathit{Expr} ::=\ & \mathsf{true} \mid \mathsf{false} \mid \mathsf{zero} \mid \mathsf{succ}\,\mathit{Expr} \mid \mathsf{pred}\,\mathit{Expr} \\ & \mid \mathsf{isZero}\,\mathit{Expr} \mid \mathsf{if}\,\mathit{Expr}\,\mathsf{then}\,\mathit{Expr}\,\mathsf{else}\,\mathit{Expr} \end{aligned}$$

The meaning of this definition is that *Expr* represents the set of all arithmetic expressions. The left-hand side of the definition gives all the correct ways of defining an expression. There are three constants, true, false and zero, and some recursive constructors that allow us to make new expressions by inserting previously build expressions in the places where *Expr* occurs.

Here are some examples of correct expressions:

zero
true
succ zero
succ true   (correct expressions, though meaning unclear)
isZero (succ zero)
if (isZero (pred false)) then zero else (succ (isZero true))

Not all expressions seem to make sense. Some clearly denote a natural number, some other denote a Boolean value, but if we mix them in the wrong order (for example (pred true)) we have a meaningless expression. We'll see at the moment of giving the semantics what we can do with them.

Let us now describe the same language using derivation rules rather than the BNF. A derivation rule is a horizontal line with some expressions written above it and an expression written under it. For example:

$$\frac{e_1 \in \mathit{Expr} \quad e_2 \in \mathit{Expr} \quad e_3 \in \mathit{Expr}}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \in \mathit{Expr}}$$

The rule has three assumptions and a conclusion. It states that if we have already constructed expressions e_1, e_2 and e_3, then we can construct the new expression (if e_1 then e_2 else e_3).

There are rules for the base elements as well. They don't need any assumption, so the space above the line is empty:

$$\frac{}{\mathsf{zero} \in \mathit{Expr}} \qquad \frac{}{\mathsf{true} \in \mathit{Expr}} \qquad \frac{}{\mathsf{false} \in \mathit{Expr}}$$

The rules for the unitary operators have just one assumption:

$$\frac{e \in \mathit{Expr}}{\mathsf{succ}\ e \in \mathit{Expr}} \qquad \frac{e \in \mathit{Expr}}{\mathsf{pred}\ e \in \mathit{Expr}} \qquad \frac{e \in \mathit{Expr}}{\mathsf{isZero}\ e \in \mathit{Expr}}$$

To construct any element of *Expr* we must derive it completely, starting with his subterms. We must build a *derivation tree* in which all assumptions are resolved by using further derivation rules. Here is an example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{zero} \in \mathit{Expr}}}{\mathsf{succ\,zero} \in \mathit{Expr}}}{\mathsf{isZero\,(succ\,zero)} \in \mathit{Expr}} \qquad \dfrac{\dfrac{}{\mathsf{false} \in \mathit{Expr}}}{\mathsf{succ\,false} \in \mathit{Expr}} \qquad \dfrac{\dfrac{}{\mathsf{zero} \in \mathit{Expr}}}{\mathsf{pred\,zero} \in \mathit{Expr}}}{\mathsf{if\,(isZero\,(succ\,zero))\,then\,(succ\,false)\,else\,(pred\,zero)} \in \mathit{Expr}}$$

The derivation has the form of a tree with subterms as nodes and base cases (rules with no assumptions) as leaves. To avoid confusion we use parentheses around subexpressions.

Defining a language using derivation rules is long and boring. BNF is much simpler and more compact. However, derivation rules are useful in more complex languages and for other purposes than just defining syntax. When we go beyond a toy language like *Expr* or a simple imperative programming language, BNF is not flexible enough. We will soon see examples: several system of (dependently) typed λ-calculus can only be defined using derivation rules.

Here are the main areas in which use of derivation rules is essential.

Syntax: Definition of a language. BNF only supports languages with a finite number of *syntactic categories*. We had only one category in our example, the set *Expr*. We could change it a bit by having a category for numerical expressions and a category for Boolean expressions. This would also solve the problem of meaningless terms. However, when we introduce *types*, each of them needs to be a separate category. We cannot give a BNF for all of them. We will see instead how derivation rules make it quite easy to define a typed language.

Manipulation of Expressions: We may want to specify how expressions can be modified in various ways. One example is the definition of substitution. In general defining functions on terms, when they are not obviously recursive, may require giving them as relations generated by derivation rules.

Operational Semantics: Defining the computation rules of a language requires specifying some reduction relation. This is in general defined by derivation rules. Mostly the simple one-step derivation can be given directly, without need of assumption. But there are also *structural rules* that allow us to do a reduction in context: these usually require assumptions.

Logical Systems: Derivation rules were originally introduced in mathematical logic as a way of expressing symbolic reasoning. They are important in defining the specific logical systems that we use to reason about programs.

The general form of a derivation rule is

$$\frac{A_1 \quad A_2 \quad \cdots \quad A_n}{B}$$

where A_1, A_2, A_n are the *assumptions* and B is the *conclusion.* Assumptions and conclusions are *judgments*, some sentences stating a fact. In our example a judgment is something in the form $e \in \mathit{Expr}$ states that e is a correct arithmetic expression.

The rule says that if we have already derived A_1, A_2, A_n, then we can derive B. All there judgments can contain *meta-variables*, identifiers that can be replaced by concrete expressions. In our examples, e, e_1, e_2, e_3 are meta-variables. In concrete derivations, the meta-variables are instantiated by expressions.

A correct judgment is a statement that has been derived by giving a full derivation tree whose leaves are instances of rules without assumptions.

Semantics

We have defined the syntax of arithmetic expressions. That is, we specified what strings of symbols are correct terms of the language *Expr*. Now we want to define the semantics. That is, we describe precisely what the expression mean. There are two basic kind of semantics. According to *denotational* semantics, the meaning of an expression is an object in some mathematical space. According to *operational* semantics, the meaning of an expression is the value that it computes when we execute it.

Denotational Semantics

Arithmetic expressions describe both integer numbers from zero and Boolean values. Therefore we choose two mathematical domains to interpret them, the set $\mathbb{N} = \{0, 1, 3, \ldots\}$ of natural numbers and the set $\mathbb{B} = \{\mathsf{True}, \mathsf{False}\}$ of truth values.

We interpret every term of *Expr* as an element of either of these sets. For example the term $(\mathsf{succ}\,(\mathsf{succ}\,(\mathsf{succ}\,\mathsf{zero})))$ denotes the number 3 and the term $(\mathsf{isZero}\,(\mathsf{succ}\,\mathsf{zero}))$ denotes the truth value False. However, some terms have a meaningless combination of the operations for both domains, for example $(\mathsf{succ}\,\mathsf{true})$. We leave the denotation of these meaningless terms undefined.

Formally, we use *semantic brackets* $[\![-]\!]$ around an expression to indicate its denotation, for example $[\![\mathsf{succ}\,(\mathsf{succ}\,(\mathsf{succ}\,\mathsf{zero}))]\!] = 3$ and $[\![\mathsf{isZero}\,(\mathsf{succ}\,\mathsf{zero})]\!] = \mathsf{False}$. For meaningless terms we use the *bottom* symbol $\bot$, so $[\![\mathsf{succ}\,\mathsf{true}]\!] = \bot$ simply means that $\mathsf{succ}\,\mathsf{true}$ is meaningless. The symbol $\bot$ itself is not any mathematical object. (However, in a more advanced kind of denotational semantics, called *domain theory*, we define mathematical spaces in which $\bot$ is an actual element.)

The interpretation is defined by *structural recursion* on the syntax: we define the meaning of an expression in terms of the meaning of its components. For every rule that generates terms, we assume that we already know the interpretation of the assumptions and we specify how to combine them to obtain the interpretation of the conclusion.

$$
\begin{array}{l}
[\![\mathsf{zero}]\!] = 0 \\
[\![\mathsf{false}]\!] = \mathsf{False} \\
[\![\mathsf{true}]\!] = \mathsf{True} \\
[\![\mathsf{succ}\, e]\!] = [\![e]\!] + 1 \quad \text{if } [\![e]\!] \in \mathbb{N} \\
[\![\mathsf{pred}\, e]\!] = \begin{cases} 0 & \text{if } [\![e]\!] = 0 \\ n-1 & \text{if } [\![e]\!] = n > 0 \end{cases} \\
[\![\mathsf{isZero}\, e]\!] = \begin{cases} \mathsf{True} & \text{if } [\![e]\!] = 0 \\ \mathsf{False} & \text{if } [\![e]\!] = n > 0 \end{cases} \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] = \begin{cases} [\![e_2]\!] & \text{if } [\![e_1]\!] = \mathsf{True} \\ [\![e_3]\!] & \text{if } [\![e_1]\!] = \mathsf{False} \end{cases}
\end{array}
$$

The expressions that don't fall into these cases are left undefined, that is $[\![e]\!] = \bot$. Also, the same side condition that we imposed for succ, that $[\![e]\!] \in \mathbb{N}$, must hold for pred and isZero; if not, the expression is meaningless. Similarly, in the last case, $[\![e]\!]$ must be in $\mathbb{B}$. We also impose that $[\![e_2]\!]$ and $[\![e_3]\!]$ are in the same set, either both in $\mathbb{N}$ or both in $\mathbb{B}$.

Here is the calculation of the denotation of a complex expression:

$$[\![\mathsf{if}\ (\mathsf{if}\ (\mathsf{isZero}\ (\mathsf{pred\ zero}))\ \mathsf{then\ false\ else\ true})\ \mathsf{then}\ (\mathsf{succ}\ (\mathsf{pred\ zero}))\ \mathsf{else\ zero}]\!] = ?$$

Since it is a bit involved, we start calculating the denotation of its component parts first.

$$
\begin{array}{l}
[\![\mathsf{pred\ zero}]\!] = 0 \qquad \text{because } [\![\mathsf{zero}]\!] = 0, \\
\text{therefore } [\![\mathsf{isZero}\ (\mathsf{pred\ zero})]\!] = \mathsf{True};
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{if}\ (\mathsf{isZero}\ (\mathsf{pred\ zero}))\ \mathsf{then\ false\ else\ true}]\!] \\
= [\![\mathsf{false}]\!] \quad \text{because } [\![\mathsf{isZero}\ (\mathsf{pred\ zero})]\!] = \mathsf{True} \\
= \mathsf{False};
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{if}\ (\mathsf{if}\ (\mathsf{isZero}\ (\mathsf{pred\ zero}))\ \mathsf{then\ false\ else\ true})\ \mathsf{then}\ (\mathsf{succ}\ (\mathsf{pred\ zero}))\ \mathsf{else\ zero}]\!] \\
= [\![\mathsf{zero}]\!] \quad \text{because } [\![\mathsf{if}\ (\mathsf{isZero}\ (\mathsf{pred\ zero}))\ \mathsf{then\ false\ else\ true}]\!] = \mathsf{False} \\
= 0.
\end{array}
$$

Operational Semantics

A different way to explain the meaning of expressions consists in specifying their computations. A complex term can be simplified by applying some *reduction* rules. Repeatedly reducing a term may eventually result in a *value*. This reduction of expressions to values is the *operational semantics* of a language.

We denote the reduction relation by a squiggly arrow $\leadsto$. When we write $e_1 \leadsto e_2$, we mean that we can do one step of simplification in e_1 and obtain e_2. For example $\mathsf{pred}\ (\mathsf{succ\ zero}) \leadsto \mathsf{zero}$ or $\mathsf{isZero}\ (\mathsf{succ\ zero}) \leadsto \mathsf{true}$. Sometimes several steps of reduction are possible. We use $\leadsto^*$ to denote several steps of $\leadsto$.

There are two kinds of rules for $\leadsto$: the actual simplification rules, that specify how terms of a certain form reduce, and the *structural rules*, that say that reduction can be done in context, that is, we can reduce a subterm inside a bigger term.

The simplification rules for *Expr* are:

$$\begin{array}{l}
\mathsf{if\,true\,then}\,e_2\,\mathsf{else}\,e_3 \rightsquigarrow e_2\\
\mathsf{if\,false\,then}\,e_2\,\mathsf{else}\,e_3 \rightsquigarrow e_3\\
\mathsf{pred}\,(\mathsf{succ}\,e) \rightsquigarrow e\\
\mathsf{pred\,zero} \rightsquigarrow \mathsf{zero}\\
\mathsf{isZero\,zero} \rightsquigarrow \mathsf{true}\\
\mathsf{isZero}\,(\mathsf{succ}\,e) \rightsquigarrow \mathsf{false}.
\end{array}$$

The structural rules are formulated as derivation rules. (The simplification rules can also be given as derivation rules with no assumptions.) For each of the expression constructors, we state that if one of its argument contains a reducible term, then we can perform the reduction inside the constructor. There are many structural rules, one for each argument of each constructor. The general shape is the same for all of them. Here are just a few of them to give the idea of the general pattern:

$$\frac{e \rightsquigarrow e'}{\mathsf{succ}\,e \rightsquigarrow \mathsf{succ}\,e'}$$

This rule states that if a term e reduces to e', then we can perform this reduction step in the argument of the $\succ$ constructor. For example $\mathsf{succ}\,(\mathsf{isZero\,zero}) \rightsquigarrow \mathsf{succ\,true}$. Never mind that this term is itself meaningless. Reductions are purely syntactic operations, they don't worry whether the expressions are sensible.

Since the $\mathsf{if}-\mathsf{then}-\mathsf{else}-$ constructor has three argument, we have three structural rules for reduction:

$$\frac{e_1 \rightsquigarrow e_1'}{\mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3 \rightsquigarrow \mathsf{if}\,e_1'\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3}$$

$$\frac{e_2 \rightsquigarrow e_2'}{\mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3 \rightsquigarrow \mathsf{if}\,e_1\,\mathsf{then}\,e_2'\,\mathsf{else}\,e_3}$$

$$\frac{e_3 \rightsquigarrow e_3'}{\mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3 \rightsquigarrow \mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3'}$$

The subterm of an expressions to which the reduction rule is applied are called the *redex*, the result of reducing it is called the *reductum*. For example, in the reduction

$$\begin{array}{l}
\mathsf{if}\,(\mathsf{isZero\,zero})\,\mathsf{then}\,(\mathsf{succ\,zero})\,\mathsf{else}\,(\mathsf{pred\,zero})\\
\quad\rightsquigarrow \mathsf{if\,true\,then}\,(\mathsf{succ\,zero})\,\mathsf{else}\,(\mathsf{pred\,zero})
\end{array}$$

the redex is $(\mathsf{isZero\,zero})$ and the reductum is true.

There may be several redexes in an expression. We have the choice of which one to reduce. For example, the previous expressions can be reduced with this alternative step (redex underlined):

$$\begin{array}{l}
\mathsf{if}\,(\mathsf{isZero\,zero})\,\mathsf{then}\,(\mathsf{succ\,zero})\,\mathsf{else}\,(\underline{\mathsf{pred\,zero}})\\
\quad\rightsquigarrow \mathsf{if}\,(\mathsf{isZero\,zero})\,\mathsf{then}\,(\mathsf{succ\,zero})\,\mathsf{else\,zero}.
\end{array}$$

Now we defined the one step reduction relation by rules. The many-step reduction relation $\rightsquigarrow^*$ consists in applying several one step reduction consecutively, in any of the allowed order. For example, we can reduce the example term that we used earlier to illustrate denotational semantics (at each step, the active redex is underlined):

$$\begin{array}{l}
\mathsf{if}\ (\mathsf{if}\ (\mathsf{isZero}\ (\underline{\mathsf{pred\,zero}}))\ \mathsf{then\,false\,else\,true})\ \mathsf{then}\ (\mathsf{succ}\ (\mathsf{pred\,zero}))\ \mathsf{else\,zero} \\
\rightsquigarrow \mathsf{if}\ (\mathsf{if}\ (\mathsf{isZero\,zero})\ \mathsf{then\,false\,else\,true})\ \mathsf{then}\ (\mathsf{succ}\ (\underline{\mathsf{pred\,zero}}))\ \mathsf{else\,zero} \\
\rightsquigarrow \mathsf{if}\ (\mathsf{if}\ (\underline{\mathsf{isZero\,zero}})\ \mathsf{then\,false\,else\,true})\ \mathsf{then}\ (\mathsf{succ\,zero})\ \mathsf{else\,zero} \\
\rightsquigarrow \mathsf{if}\ (\underline{\mathsf{if\,true\,then\,false\,else\,true}})\ \mathsf{then\ \ else}\ (\mathsf{succ\,zero})\mathsf{zero} \\
\rightsquigarrow \underline{\mathsf{if\,false\,then}\ (\mathsf{succ\,zero})\ \mathsf{else\,zero}} \\
\rightsquigarrow \mathsf{zero}
\end{array}$$

After performing reduction steps as far as possible, we obtain a term that doesn't contain any redexes any more.

Definition 1 *A term is in* normal form *if it contains no redexes.*

A value *is a term belonging to one of the following languages, respectively of* numerals *and of* Boolean values*:*

$$\begin{array}{l}
nv ::= \mathsf{zero} \mid \mathsf{succ}\ nv \\
bv ::= \mathsf{true} \mid \mathsf{false}
\end{array}$$

For a natural number n, we use the notation $\overline{n}$ for the corresponding numeral: $\overline{0} = \mathsf{zero}$, $\overline{1} = \mathsf{succ\,zero}$, $\overline{2} = \mathsf{succ}\ (\mathsf{succ\,zero})$ and so on.

All values are normal forms, but there are also normal forms that are not values, for example $\mathsf{isZero}\ (\mathsf{pred\,true})$. This happens because these terms are meaningless. When we define languages with variables, we will also have normal forms containing free variables, which are not values because of the indeterminacy of the values of the variables.

We have seen that a term can be reduced in several ways. If it contains more than one redex, we can choose which one to reduce first. The final result of the computation, that is, the normal form that we obtain at the end, does not depend on the order of reduction.

Theorem 2 (Confluence) *Given an expression $e \in Expr$; if there exist two expressions $e_1, e_2 \in Expr$ such that $e \rightsquigarrow e_1$ and $e \rightsquigarrow e_2$, then there exist a common reduct $e' \in Expr$ such that $e_1 \rightsquigarrow^* e'$ and $e_2 \rightsquigarrow^* e'$.*

Proof. In the case that the redexes that were reduced in e_1 and e_2 are independent, that is, they occur in separated places inside e, then it is enough to perform the other reduction in each of the two:

$$\begin{tikzcd}
 & e = \cdots r_1 \cdots r_2 \cdots \arrow[dl, "r_1 \rightsquigarrow r_1'"'] \arrow[dr, "r_2 \rightsquigarrow r_2'"] & \\
e_1 = \cdots r_1' \cdots r_2 \cdots \arrow[dr, dotted, "r_1 \rightsquigarrow r_1'"] & & e_2 = \cdots r_1 \cdots r_2' \cdots \arrow[dl, dotted, "r_1 \rightsquigarrow r_1'"'] \\
 & e' = \cdots r_1' \cdots r_2' \cdots &
\end{tikzcd}$$

But it can also happen that the redexes are contained inside each other. For example in the term

$$\underline{\mathsf{if\ true\ then\ }(\underline{\mathsf{pred\ zero}})\mathsf{\ else\ zero}}$$

we have two redexes inside each other. In this particular example it is still true that after reducing one, the other will still be present, so we can adopt the same strategy as for when they are separated.

However, it is also possible that the reduction of one redex makes the other disappear. This is what happens in the following redex:

$$e = \underline{\mathsf{if\ true\ then\ zero\ else\ }(\underline{\mathsf{pred\ (succ\ zero)}})}.$$

When we reduce each redex separately, we get:

$$e_1 = \mathsf{zero}, \quad e_2 = \mathsf{if\ true\ then\ zero\ else\ zero}.$$

We can still perform the reduction of the first redex in e_2, obtaining zero, but the second redex has disappeared from e_1, so there is nothing to do. Clearly, we still have a common reduct zero, and this will be the true in similar cases of overlapping redexes.

This is the reason we formulated the theorem using many-step reduction: $e_1 \rightsquigarrow^* e'$ and $e_2 \rightsquigarrow^* e'$. Specifically, one of the two many-step reductions could be empty. $\square$

While the proof of the confluence theorem is straightforward in this case, it becomes more complicated for richer languages. In particular, if the language has reduction rules that can duplicate some terms, then one reduction step may generate many copies of an existing redex. We will need more sophisticated proof techniques to show that confluence still holds.

If we keep reducing some redexes in a term, we will eventually get rid of them all and arrive at a normal form.

Theorem 3 (Normal Form) *For every expression $e \in Expr$ there exist a (unique) normal form $e \in Expr$ such that $e \rightsquigarrow^* t$.*

Proof. Notice that every reduction rule makes the term smaller. Therefore it is necessary that eventually the reduction process will terminate. This is a proof by induction on the length of the term. □

In more complex systems, reduction steps do not necessarily decrease the size of the term. In some cases they can increase it in an explosive way. Therefore a simple proof by induction on the length of the term will not work any more and we will have to invent more sophisticated proof techniques.

Soundness and Completeness

We have defined two radically different ways to give meaning to expressions: denotational semantics interpret them as elements of a mathematical space, operational semantics interpret them as computation processes that lead to a normal form.

We want to show that these two interpretation of our language agree with each other in some sense. We have seen an example with the term

$$\mathsf{if}\ (\mathsf{if}\ (\mathsf{isZero}\ (\mathsf{pred}\ \mathsf{zero}))\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true})\ \mathsf{then}\ (\mathsf{succ}\ (\mathsf{pred}\ \mathsf{zero}))\ \mathsf{else}\ \mathsf{zero}$$

We calculated its denotation to be the number 0. We computed its normal form to be the numeral zero. This is good news: the two interpretations agree.

We want to show that this always happens. First of all, the reduction process doesn't change the denotation of a term.

Theorem 4 (Soundness) *For every expression $e \in Expr$, if $e \leadsto e'$ for some other $e' \in Expr$, then $[\![e]\!] = [\![e']\!]$. (For meaningless terms this means that they are both $\bot$.)*

Proof. **TO DO** [By induction on the structure of the term.] □

If we reduce an expression all the way to normal form, the denotation will be preserved, so if the normal form is a value, the denotation must be the corresponding object in the mathematical space.

Corollary 5 *If $e \leadsto^* v$ for some value v, then $[\![e]\!] = [\![v]\!]$.*

The vice versa is also true: the denotation of an expression (when it exists) will correspond exactly to a value to which the expression reduces.

Theorem 6 (Completeness) *For every expression $e \in Expr$, we have:*

- *If $[\![e]\!] = n$ for some $n \in \mathbb{N}$, then $e \leadsto^* \overline{n}$;*
- *If $[\![e]\!] = \mathsf{True}$, then $e \leadsto^* \mathsf{true}$;*
- *If $[\![e]\!] = \mathsf{False}$, then $e \leadsto^* \mathsf{false}$;*
- *If $[\![e]\!] = \bot$ then there is no value v such that $e \leadsto^* v$.*

Proof. **TO DO** [By induction on the structure of e and by definition of $[\![-]\!]$ and $\rightsquigarrow$.] $\square$

The combination of the soundness and completeness theorems tell us that there is complete agreement between the definitional and operational semantics. This also suggests a clever method to compute the normal form of a term (when it exists): First find the result of $[\![e]\!]$ and then take the syntactic value corresponding to it. We may succinctly write: $e \rightsquigarrow^* \overline{[\![e]\!]}$ (with the additional definitions $\overline{\mathsf{True}} = \mathsf{true}$ and $\overline{\mathsf{False}} = \mathsf{false}$). This process is called *Normalization by Evaluation*.

Reduction Strategies

We have noticed that certain terms can contain many redexes and therefore the order of reduction steps is not unique. Although the confluence property ensures that any reduction sequence will produce the same normal form, there are issues of efficiency that make certain reduction strategies more convenient than others. Also, for other languages in which there is no guarantee that a normal form exists, the choice of reduction strategy may make the difference between a terminating computation and an infinite one.

Let's take as example the following term:

$$e = \mathsf{if}\ (\underline{\mathsf{isZero\ zero}})\ \mathsf{then}\ (\underline{\mathsf{pred}\ (\mathsf{succ\ zero})})\ \mathsf{else}\ (\mathsf{succ}\ (\underline{\mathsf{pred\ zero}})).$$

It contains three redexes, which we can choose to reduce in any possible order.

Two of the most important reduction strategies are called *Eager Evaluation* and *Lazy Evaluation*. They differ in the order in which they reduce the arguments of a function. In e there are three arguments, the *if* test condition, the *then* branch and the *else* branch.

In eager evaluation we reduce all arguments of a function to normal form before reducing the main expression:

$$e \rightsquigarrow^* \mathsf{if\ true\ then\ zero\ else\ succ\ zero} \rightsquigarrow \mathsf{zero}.$$

In lazy evaluation, we only reduce the arguments that are needed to make a computation step in the main expression:

$$e \rightsquigarrow \mathsf{if\ true\ then}\ (\mathsf{pred}\ (\mathsf{succ\ zero}))\ \mathsf{else}\ (\mathsf{succ}\ (\mathsf{pred\ zero})) \rightsquigarrow \mathsf{pred}\ (\mathsf{succ\ zero}) \rightsquigarrow \mathsf{zero}.$$

Notice that we saved one reduction step: we didn't have to reduce the *else* branch of the expression.