PART I

Judgments and Rules

1 Syntactic Objects

Programming languages are languages, a means of expressing computations in a form comprehensible to both people and machines. The syntax of a language specifies the means by which various sorts of phrases (expressions, commands, declarations, and so forth) may be combined to form programs. But what sort of thing are these phrases? What is a program made of?

The informal concept of syntax may be seen to involve several distinct concepts. The *surface*, or *concrete*, *syntax* is concerned with how phrases are entered and displayed on a computer. The surface syntax is usually thought of as given by strings of characters from some alphabet (say, ASCII or Unicode). The *structural*, or *abstract*, *syntax* is concerned with the structure of phrases, specifically how they are composed from other phrases. At this level a phrase is a tree, called an *abstract syntax tree*, whose nodes are operators that combine several phrases to form another phrase. The *binding* structure of syntax is concerned with the introduction and use of identifiers: how they are declared and how declared identifiers are to be used. At this level phrases are *abstract binding trees*, which enrich abstract syntax trees with the concepts of binding and scope.

We do not concern ourselves in this book with matters of concrete syntax, but instead work at the level of abstract syntax. To prepare the ground for the rest of the book, this chapter begins by defininining abstract syntax trees and abstract binding trees and some functions and relations associated with them. The definitions are a bit technical, but are absolutely fundamental to what follows. It is probably best to skim this chapter on first reading, returning to it only as the need arises.

1.1 Abstract Syntax Trees

An *abstract syntax tree*, or *ast* for short, is an ordered tree whose leaves are *variables* and whose interior nodes are *operators* whose *arguments* are its children. Abstract syntax trees are classified into a variety of *sorts* corresponding to different forms of syntax. A *variable* stands for an unspecified, or generic, piece of syntax of a specified sort. Ast's may be combined by an *operator*, which has both a sort and an *arity*, a finite sequence of sorts specifying the number and sorts of its arguments. An operator of sort s and arity $s_1, \ldots, s_n$ combines $n \geq 0$ ast's of sort $s_1, \ldots, s_n$, into a compound ast of sort s. As a matter of terminology, a *nullary* operator is one that takes no arguments, a *unary* operator takes one, a *binary* operator takes two, and so forth.

The concept of a variable is central, and therefore deserves special emphasis. As in mathematics a variable is an *unknown* object drawn from some domain, its *range of signficance*. In school mathematics the (often implied) range of significance is the set of real numbers. Here variables range over ast's of a specified sort. Being an unknown, the meaning of a variable is given by *substitution*, the process of "plugging in" an object from the domain for the variable in a formula. So, in school, we might plug in π for x in a polynomial and calculate the result. Here we would plug in an ast of the appropriate sort for a variable in an ast to obtain another ast. The process of substitution is easily understood for ast's, because it amounts to a "physical" replacement of the variable by an ast within another ast. We subsequently consider a generalization of the concept of ast for which the substitution process is somewhat more complex, but the essential idea is the same and bears repeating: *A variable is given meaning by substitution*.

For example, consider a simple language of expressions built from numbers, addition, and multiplication. The abstract syntax of such a language would consist of a single sort Exp and an infinite collection of operators that generate the forms of expression: `num[`n`]` is a nullary operator of sort Exp whenever $n \in \mathbb{N}$; `plus` and `times` are binary operators of sort Exp whose arguments are both of sort Exp. The expression $2 + (3 \times x)$, which involves a variable x, would be represented by the ast

$$\texttt{plus(num[2];times(num[3];}x\texttt{))}$$

of sort Exp, under the assumption that x is also of this sort. Because, say, `num[4]`, is an ast of sort Exp, we may plug it in for x in the preceding ast to obtain the ast

$$\texttt{plus(num[2];times(num[3];num[4]))},$$

which is written informally as $2 + (3 \times 4)$. We may, of course, plug in more complex ast's of sort Exp for x to obtain other ast's as a result.

The tree structure of ast's supports a very useful principle of reasoning, called *structural induction*. Suppose that we wish to prove that some property $\mathcal{P}(a)$ holds for all ast's a of a given sort. To show this, it is enough to consider all the ways in which a may be generated and show that the property holds in each case, under the assumption that it holds for each of its constituent ast's (if any). So, in the case of the sort Exp just described, we must show that

1. the property holds for any variable x of sort Exp; $\mathcal{P}(x)$;
2. the property holds for any number `num[`n`]`: For every $n \in \mathbb{N}$; $\mathcal{P}(\texttt{num[}n\texttt{]})$;
3. assuming the property holds for a_1 and a_2, it holds for `plus(`a_1`;`a_2`)` and `times(`a_1`;`a_2`)`: If $\mathcal{P}(a_1)$ and $\mathcal{P}(a_2)$, then $\mathcal{P}(\texttt{plus(}a_1\texttt{;}a_2\texttt{)})$ and $\mathcal{P}(\texttt{times(}a_1\texttt{;}a_2\texttt{)})$.

Because these cases exhaust all possibilities for the formation of a, we are assured that $\mathcal{P}(a)$ holds for any ast a of sort Exp.

For the sake of precision and to prepare the ground for further developments, precise definitions of the foregoing concepts are now given. Let $\mathcal{S}$ be a finite set of sorts. Let $\{ O_s \}_{s \in \mathcal{S}}$ be a sort-indexed family of *operators* o of sort s with arity $\mathsf{ar}(o) = (s_1, \ldots, s_n)$. Let $\{ \mathcal{X}_s \}_{s \in \mathcal{S}}$

be a sort-indexed family of *variables* x of each sort s. The family $\mathcal{A}[\mathcal{X}] = \{\mathcal{A}[\mathcal{X}]_s\}_{s\in\mathcal{S}}$ of ast's of sort s is defined as follows:

1. A variable of sort s is an ast of sort s: If $x \in \mathcal{X}_s$, then $x \in \mathcal{A}[\mathcal{X}]_s$.
2. Operators combine ast's: If o is an operator of sort s such that $\mathsf{ar}(o) = (s_1, \ldots, s_n)$ and if $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}, \ldots, a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; \ldots; a_n) \in \mathcal{A}[\mathcal{X}]_s$.

It follows from this definition that the principle of structural induction may be used to prove that some property $\mathcal{P}$ holds for every ast. To show that $\mathcal{P}(a)$ holds for every $a \in \mathcal{A}[\mathcal{X}]$, it is enough to show that

1. if $x \in \mathcal{X}_s$, then $\mathcal{P}_s(x)$, and
2. if $o \in \mathcal{O}_s$ and $\mathsf{ar}(o) = (s_1, \ldots, s_n)$, then if $\mathcal{P}_{s_1}(a_1)$ and $\ldots$ and $\mathcal{P}_{s_n}(a_n)$, then $\mathcal{P}_s(o(a_1; \ldots; a_n))$.

For example, it is easy to prove by structural induction that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.

If $\mathcal{X}$ is a sort-indexed family of variables, we write $\mathcal{X}, x$, where x is a variable of sort s such that $x \notin \mathcal{X}_s$, to stand for the family of sets $\mathcal{Y}$ such that $\mathcal{Y}_s = \mathcal{X}_s \cup \{x\}$ and $\mathcal{Y}_{s'} = \mathcal{X}_{s'}$ for all $s' \neq s$. The family $\mathcal{X}, x$, where x is a variable of sort s, is said to be the family obtained by *adjoining* the variable x to the family $\mathcal{X}$.

Variables are given meaning by *substitution*. If x is a variable of sort s, $a \in \mathcal{A}[\mathcal{X}, x]_{s'}$, and $b \in \mathcal{A}[\mathcal{X}]_s$, then $[b/x]a \in \mathcal{A}[\mathcal{X}]_{s'}$ is defined to be the result of substituting b for every occurrence of x in a. The ast a is called the *target* and x is called the *subject* of the substitution. Substitution is defined by the following equations:

1. $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(a_1; \ldots; a_n) = o([b/x]a_1; \ldots; [b/x]a_n)$.

For example, we may check that

$$[\texttt{num[2]}/x]\texttt{plus}(x;\texttt{num[3]}) = \texttt{plus}(\texttt{num[2]};\texttt{num[3]}).$$

We may prove by structural induction that substitution for ast's is well defined.

Theorem 1.1. *If $a \in \mathcal{A}[\mathcal{X}, x]$, then for every $b \in \mathcal{A}[\mathcal{X}]$ there exists a unique $c \in \mathcal{A}[\mathcal{X}]$ such that $[b/x]a = c$.*

Proof By structural induction on a. If $a = x$, then $c = b$ by definition; otherwise, if $a = y \neq x$, then $c = y$, also by definition. Otherwise, $a = o(a_1, \ldots, a_n)$, and we have by induction unique $c_1, \ldots, c_n$ such that $[b/x]a_1 = c_1$ and $\ldots$ $[b/x]a_n = c_n$, and so c is $c = o(c_1; \ldots; c_n)$, by definition of substitution. □

In most cases it is possible to enumerate all of the operators that generate the ast's of a sort up front, as we have done in the foregoing examples. However, in some situations this is not possible—certain operators are available only within certain contexts. In such

cases we cannot fix the collection of operators $\mathcal{O}$ in advance, but rather must allow it to be *extensible*. This is achieved by considering families of operators that are indexed by symbolic *parameters* that serve as "names" for the instances. For example, in Chapter 34 we consider a family of nullary operators `cls[`u`]`, where u is a symbolic parameter drawn from the set of *active* parameters. It is essential that distinct parameters determine distinct operators: If u and v are active parameters and $u \neq v$, then `cls[`u`]` and `cls[`v`]` are different operators. Extensibility is achieved by introducing new active parameters. So, if u is not active, then `cls[`u`]` makes no sense, but if u becomes active, then `cls[`u`]` is a nullary operator.

Parameters are easily confused with variables, but they are fundamentally different concepts. As noted earlier, a variable stands for an unknown ast of its sort, but a parameter *does not stand for anything*. It is a purely symbolic identifier whose only significance is whether it is the same as or different from another parameter. Whereas variables are given meaning by substitution, it is not possible, or sensible, to substitute for a parameter. As a consequence, disequality of parameters is preserved by substitution, whereas disequality of variables is not (because the same ast may be substituted for two distinct variables).

To account for the set of active parameters, the set $\mathcal{A}[\mathcal{U};\mathcal{X}]$ is the set of ast's with variables drawn from $\mathcal{X}$ and with parameters drawn from $\mathcal{U}$. Certain operators, such as `cls[`u`]`, are *parameterized* by parameters u of a given sort. The parameters are distinguished from the arguments by the square brackets around them. Instances of such operators are permitted only for parameters drawn from the active set $\mathcal{U}$. So, for example, if $u \in \mathcal{U}$, then `cls[`u`]` is a nullary operator, but if $u \notin \mathcal{U}$, then `cls[`u`]` is not a valid operator. The next section introduces the means of extending $\mathcal{U}$ to make operators available within that context.

1.2 Abstract Binding Trees

Abstract binding trees, or *abt's*, enrich ast's with the means to introduce new variables and parameters, called a *binding*, with a specified range of significance, called its *scope*. The scope of a binding is an abt within which the bound identifier may be used, either as a placeholder (in the case of a variable declaration) or as the index of some operator (in the case of a parameter declaration). Thus the set of active identifiers may be larger within a subtree of an abt than it is within the surrounding tree. Moreover, different subtrees may introduce identifiers with disjoint scopes. The crucial principle is that any use of an identifier should be understood as a reference, or abstract pointer, to its binding. One consequence is that the choice of identifiers is immaterial, so long as we can always associate a unique binding with each use of an identifier.

As a motivating example, consider the expression `let` x `be` a_1 `in` a_2, which introduces a variable x for use within the expression a_2 to stand for the expression a_1. The variable x is bound by the `let` expression for use within a_2; any use of x within a_1 refers to a different variable that happens to have the same name. For example, in the expression

$\texttt{let}\, x\, \texttt{be}\, 7\, \texttt{in}\, x + x$, occurrences of x in the addition refer to the variable introduced by the `let`. However, in the expression $\texttt{let}\, x\, \texttt{be}\, x * x\, \texttt{in}\, x + x$, occurrences of x within the multiplication refer to a variable different from those occurring within the addition. The latter occurrences refer to the binding introduced by the `let`, whereas the former refer to some outer binding not displayed here.

The names of bound variables are immaterial insofar as they determine the same binding. So, for example, the expression $\texttt{let}\, x\, \texttt{be}\, x * x\, \texttt{in}\, x + x$ could just as well have been written as $\texttt{let}\, y\, \texttt{be}\, x * x\, \texttt{in}\, y + y$ without changing its meaning. In the former case the variable x is bound within the addition, and in the latter it is the variable y, but the "pointer structure" remains the same. The expression $\texttt{let}\, x\, \texttt{be}\, y * y\, \texttt{in}\, x + x$ has a different meaning from these two expressions, because now the variable y within the multiplication refers to a different surrounding variable. Renaming of bound variables is constrained to the extent that it must not alter the reference structure of the expression. For example, the expression $\texttt{let}\, x\, \texttt{be}\, 2\, \texttt{in}\, \texttt{let}\, y\, \texttt{be}\, 3\, \texttt{in}\, x + x$ has a different meaning from the expression $\texttt{let}\, y\, \texttt{be}\, 2\, \texttt{in}\, \texttt{let}\, y\, \texttt{be}\, 3\, \texttt{in}\, y + y$, because the y in the expression $y + y$ in the second case refers to the inner declaration, not the outer one, as before.

The concept of an ast may be enriched to account for binding and scope of a variable. These enriched ast's are called abstract binding trees. Abt's generalize ast's by allowing an operator to bind any finite number (possibly zero) of variables in each argument position. An argument to an operator is called an *abstractor* and has the form $x_1, \ldots, x_k\,.\,a$. The sequence of variables $x_1, \ldots, x_k$ is bound within the abt a. (When k is zero, we elide the distinction between $.a$ and a itself.) Written in the form of an abt, the expression $\texttt{let}\, x\, \texttt{be}\, a_1\, \texttt{in}\, a_2$ has the form $\texttt{let}(a_1; x\,.\,a_2)$, which more clearly specifies that the variable x is bound within a_2 and not within a_1. We often write $\vec{x}$ to stand for a finite sequence $x_1, \ldots, x_n$ of distinct variables and write $\vec{x}\,.\,a$ to mean $x_1, \ldots, x_n\,.\,a$.

To account for binding, the arity of an operator is generalized to consist of a finite sequence of *valences*. The length of the sequence determines the number of arguments, and each valence determines the sort of the argument and the number and sorts of the variables that are bound within it. A valence of the form $(s_1, \ldots, s_k)s$ specifies an argument of sort s that binds k variables of sorts $s_1, \ldots, s_k$ within it. We often write $\vec{s}$ for a finite sequence $s_1, \ldots, s_n$ of sorts, and we say that $\vec{x}$ is of sort $\vec{s}$ to mean that the two sequences have the same length and that each x_i is of sort s_i.

Thus, for example, the arity of the operator `let` is (Exp, (Exp)Exp), which indicates that it takes two arguments, described as follows:

1. The first argument is of sort Exp and binds no variables.
2. The second argument is of sort Exp and binds one variable of sort Exp.

The definition expression $\texttt{let}\, x\, \texttt{be}\, 2 + 2\, \texttt{in}\, x \times x$ is represented by the abt

$$\texttt{let(plus(num[2]; num[2]); x.times(x; x))}.$$

Let $\mathcal{O}$ be a sort-indexed family of operators o with arities $\mathsf{ar}(o)$. For a given sort-indexed family $\mathcal{X}$ of variables, the sort-indexed family of abt's $\mathcal{B}[\mathcal{X}]$ is defined similarly to $\mathcal{A}[\mathcal{X}]$,

except that the set of active variables changes for each argument according to which variables are bound within it. A first cut at the definition is as follows:

1. If $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$.
2. If $\mathsf{ar}(o) = ((\vec{s}_1)s_1, \ldots, (\vec{s}_n)s_n)$, and if, for each $1 \leq i \leq n$, $\vec{x}_i$ is of sort $\vec{s}_i$ and $a_i \in \mathcal{B}[\mathcal{X}, \vec{x}_i]_{s_i}$, then $o(\vec{x}_1 . a_1 ; \ldots ; \vec{x}_n . a_n) \in \mathcal{B}[\mathcal{X}]_s$.

The bound variables are adjoined to the set of active variables within each argument, with the sort of each variable determined by the valence of the operator.

This definition is *almost* correct, but fails to properly account for the behavior of bound variables. An abt of the form $\texttt{let}(a_1; x . \texttt{let}(a_2; x . a_3))$ is ill-formed according to this definition, because the first binding adjoins x to $\mathcal{X}$, which implies that the second cannot also adjoin x to $\mathcal{X}, x$ without causing confusion. The solution is to ensure that each of the arguments is well-formed regardless of the choice of bound variable names. This is achieved by altering the second clause of the definition using renaming as follows:[1]

> If $\mathsf{ar}(o) = ((\vec{s}_1)s_1, \ldots, (\vec{s}_n)s_n)$, and if, for each $1 \leq i \leq n$ and for each renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$, where $\vec{x}'_i \notin \mathcal{X}$, we have $\pi_i \cdot a_i \in \mathcal{B}[\mathcal{X}, \vec{x}'_i]$, then $o(\vec{x}_1 . a_1 ; \ldots ; \vec{x}_n . a_n) \in \mathcal{B}[\mathcal{X}]_s$.

The renaming ensures that when we encounter nested binders we avoid collisions. This is called the *freshness condition on binders* because it ensures that all bound variables are "fresh" relative to the surrounding context.

The principle of structural induction extends to abt's and is called *structural induction modulo renaming*. It states that to show that $\mathcal{P}(a)[\mathcal{X}]$ holds for every $a \in \mathcal{B}[\mathcal{X}]$, it is enough to show the following:

1. If $x \in \mathcal{X}_s$, then $\mathcal{P}[\mathcal{X}]_s(x)$.
2. For every o of sort s and arity $((\vec{s}_1)s_1, \ldots, (\vec{s}_n)s_n)$, and if for each $1 \leq i \leq n$, we have $\mathcal{P}[\mathcal{X}, \vec{x}'_i]_{s_i}(\pi_i \cdot a_i)$ for every renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$, then $\mathcal{P}[\mathcal{X}]_s(o(\vec{x}_1 . a_1 ; \ldots ; \vec{x}_n . a_n))$.

The renaming in the second condition ensures that the inductive hypothesis holds for *all* fresh choices of bound variable names and not just the ones actually given in the abt.

As an example let us define the judgment $x \in a$, where $a \in \mathcal{B}[\mathcal{X}, x]$, to mean that x *occurs free* in a. Informally, this means that x is bound somewhere outside of a, rather than within a itself. If x is bound within a, then those occurrences of x are different from those occurring outside the binding. The following ensure conditions that this is the case:

1. $x \in x$.
2. $x \in o(\vec{x}_1 . a_1 ; \ldots ; \vec{x}_n . a_n)$ if there exists $1 \leq i \leq n$ such that for every fresh renaming $\pi : \vec{x}_i \leftrightarrow \vec{z}_i$ we have $x \in \pi \cdot a_i$.

The first condition states that x is free in x, but not free in y for any variable y other than x. The second condition states that if x is free in some argument, independent of the choice

[1] The action of a renaming extends to abt's in the obvious way by replacing every occurrence of x by $\pi(x)$, including any occurrences in the variable list of an abstractor as well as within its body.

of bound variable names in that argument, then it is free in the overall abt. This implies, in particular, that x is *not* free in $\texttt{let(zero;}x\,.\,x\texttt{)}$.

The relation $a =_\alpha b$ of *α-equivalence* (so-called for historical reasons), is defined to mean that a and b are identical up to the choice of bound variable names. This relation is defined to be the strongest congruence containing the following two conditions:

1. $x =_\alpha x$.
2. $o(\vec{x}_1\,.\,a_1\,;\,\ldots\,;\vec{x}_n\,.\,a_n) =_\alpha o(\vec{x}'_1\,.\,a'_1\,;\,\ldots\,;\vec{x}'_n\,.\,a'_n)$ if for every $1 \leq i \leq n$, $\pi_i \cdot a_i =_\alpha \pi'_i \cdot a'_i$ for all fresh renamings $\pi_i : \vec{x}_i \leftrightarrow \vec{z}_i$ and $\pi'_i : \vec{x}'_i \leftrightarrow \vec{z}_i$.

The idea is that we rename $\vec{x}_i$ and $\vec{x}'_i$ consistently, avoiding confusion, and check that a_i and a'_i are α-equivalent. If $a =_\alpha b$, then a and b are said to be *α-variants* of each other.

Some care is required in the definition of *substitution* of an abt b of sort s for free occurrences of a variable x of sort s in some abt a of some sort, written $[b/x]a$. Substitution is partially defined by the following conditions:

1. $[b/x]x = b$, and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(\vec{x}_1\,.\,a_1\,;\,\ldots\,;\vec{x}_n\,.\,a_n) = o(\vec{x}_1\,.\,a'_1\,;\,\ldots\,;\vec{x}_n\,.\,a'_n)$, where, for each $1 \leq i \leq n$, we require that $\vec{x}_i \notin b$, and we set $a'_i = [b/x]a_i$ if $x \notin \vec{x}_i$, and $a'_i = a_i$ otherwise.

If x is bound in some argument to an operator, then substitution does not descend into its scope, for to do so would be to confuse two distinct variables. For this reason we must take care to define a'_i in the preceding second condition according to whether $x \in \vec{x}_i$. The requirement that $\vec{x}_i \notin b$ in the second equation is called *capture avoidance*. If some $x_{i,j}$ occurred free in b, then the result of the substitution $[b/x]a_i$ would in general contain $x_{i,j}$ free as well, but then forming $\vec{x}_i\,.\,[b/x]a_i$ would *incur capture* by changing the referent of $x_{i,j}$ to be the jth bound variable of the ith argument. In such cases *substitution is undefined* because we cannot replace x by b in a_i without incurring capture.

One way around this is to alter the definition of substitution so that the bound variables in the result are chosen fresh by substitution. By the principle of structural induction we know inductively that, for any renaming $\pi_i : \vec{x}_i \leftrightarrow \vec{x}'_i$ with $\vec{x}'_i$ fresh, the substitution $[b/x](\pi_i \cdot a_i)$ is well-defined. Hence we may define

$$[b/x]o(\vec{x}_1\,.\,a_1\,;\,\ldots\,;\vec{x}_n\,.\,a_n) = o(\vec{x}'_1\,.\,[b/x](\pi_1 \cdot a_1)\,;\,\ldots\,;\vec{x}'_n\,.\,[b/x](\pi_n \cdot a_n))$$

for some particular choice of fresh bound variable names (any choice will do). There is no longer any need to take care that $x \notin \vec{x}_i$ in each argument, because the freshness condition on binders ensures that this cannot occur, the variable x already being active. Noting that

$$o(\vec{x}_1\,.\,a_1\,;\,\ldots\,;\vec{x}_n\,.\,a_n) =_\alpha o(\vec{x}'_1\,.\,\pi_1 \cdot a_1\,;\,\ldots\,;\vec{x}'_n\,.\,\pi_n \cdot a_n),$$

another way we can avoid undefined substitutions is to first choose an α-variant of the target of the substitution whose binders avoid any free variables in the substituting abt, and then perform substitution without fear of incurring capture. In other words substitution is totally defined on α-equivalence classes of abt's.

To avoid all the bureaucracy of binding, we adopt the following *identification convention* throughout this book:

> *Abstract binding trees are always to be identified up to α-equivalence.*

That is, we implicitly work with α-equivalence classes of abt's, rather than with abt's themselves. We tacitly assert that all operations and relations on abt's respect α-equivalence, so that they are properly defined on α-equivalence classes of abt's. Whenever we examine an abt, we are choosing a representative of its α-equivalence class, and we have no control over how the bound variable names are chosen. Experience shows that any operation or property of interest respects α-equivalence, so there is no obstacle to achieving it. Indeed, we might say that a property or operation is legitimate exactly insofar as it respects α-equivalence!

Parameters, as well as variables, may be bound within an argument of an operator. Such binders introduce a "new" or "fresh" parameter within the scope of the binder wherein it may be used to form further abt's. To allow for parameter declaration, the valence of an argument is generalized to indicate the sorts of the parameters bound within it, as well as the sorts of the variables, by writing $(\vec{s}_1; \vec{s}_2)s$, where $\vec{s}_1$ specifies the sorts of the parameters and $\vec{s}_2$ specifies the sorts of the variables. The sort-indexed family $\mathcal{B}[\mathcal{U}; \mathcal{X}]$ is the set of abt's determined by a fixed set of operators using the parameters $\mathcal{U}$ and the variables $\mathcal{X}$. We rely on naming conventions to distinguish parameters from variables, reserving u and v for parameters and x and y for variables.

1.3 Notes

The concept of abstract syntax has its orgins in the pioneering work of Church, Turing, and Gödel, who first considered the possibility of writing programs that act on representations of programs. Originally programs were represented by natural numbers, using encodings, now called *Gödel numberings*, based on the prime factorization theorem. Any standard text on mathematical logic, such as that by Kleene (1952), contains a thorough account of such representations. The Lisp language (McCarthy, 1965; Allen, 1978) introduced a much more practical and direct representation of syntax as *symbolic expressions*. These ideas were developed further in the language ML (Gordon et al., 1979), which featured a type system capable of expressing abstract syntax trees. The AUTOMATH project (Nederpelt et al., 1994) introduced the idea of using Church's λ-notation (Church, 1941) to account for the binding and scope of variables. These ideas were developed further in LF (Harper et al., 1993).

2 Inductive Definitions

Inductive definitions are an indispensable tool in the study of programming languages. This chapter develops the basic framework of inductive definitions and gives some examples of their use. An inductive definition consists of a set of *rules* for deriving *judgments*, or *assertions*, of a variety of forms. Judgments are statements about one or more syntactic objects of a specified sort. The rules specify necessary and sufficient conditions for the validity of a judgment and hence fully determine its meaning.

2.1 Judgments

We start with the notion of a *judgment*, or *assertion*, about a syntactic object. We make use of many forms of judgment, including examples such as these:

n nat	n is a natural number
$n = n_1 + n_2$	n is the sum of n_1 and n_2
τ type	τ is a type
$e : \tau$	expression e has type τ
$e \Downarrow v$	expression e has value v

A judgment states that one (or more) syntactic object has a property or stands in some relation to another. The property or relation itself is called a *judgment form*, and the judgment that an object (or objects) has that property or stands in that relation is said to be an *instance* of that judgment form. A judgment form is also called a *predicate*, and the objects constituting an instance are its *subjects*. We write $a\ \mathsf{J}$ for the judgment asserting that J holds for a. When it is not important to stress the subject of the judgment, we write J to stand for an unspecified judgment. For particular judgment forms, we freely use prefix, infix, or mixfix notation, as illustrated by the preceding examples, in order to enhance readability.

2.2 Inference Rules

An *inductive definition* of a judgment form consists of a collection of *rules* of the form

$$\frac{J_1 \quad \ldots \quad J_k}{J} \tag{2.1}$$

in which J and $J_1, \ldots, J_k$ are all judgments of the form being defined. The judgments above the horizontal line are called the *premises* of the rule, and the judgment below the line is called its *conclusion*. If a rule has no premises (that is, when k is zero), the rule is called an *axiom*; otherwise it is called a *proper rule*.

An inference rule may be read as stating that the premises are *sufficient* for the conclusion: To show J, it is enough to show $J_1, \ldots, J_k$. When k is zero, a rule states that its conclusion holds unconditionally. Bear in mind that there may be, in general, many rules with the same conclusion, each specifying sufficient conditions for the conclusion. Consequently, if the conclusion of a rule holds, then it is not necessary that the premises hold, for it might have been derived by another rule.

For example, the following rules constitute an inductive definition of the judgment a `nat`:

$$\frac{}{\texttt{zero nat}} \tag{2.2a}$$

$$\frac{a\ \texttt{nat}}{\texttt{succ}(a)\ \texttt{nat}}. \tag{2.2b}$$

These rules specify that a `nat` holds whenever either a is `zero` or a is $\texttt{succ}(b)$, where b `nat` for some b. Taking these rules to be exhaustive, it follows that a `nat` iff a is a natural number.

Similarly, the following rules constitute an inductive definition of the judgment a `tree`:

$$\frac{}{\texttt{empty tree}} \tag{2.3a}$$

$$\frac{a_1\ \texttt{tree} \quad a_2\ \texttt{tree}}{\texttt{node}(a_1; a_2)\ \texttt{tree}}. \tag{2.3b}$$

These rules specify that a `tree` holds if either a is `empty` or a is $\texttt{node}(a_1; a_2)$, where a_1 `tree` and a_2 `tree`. Taking these to be exhaustive, these rules state that a is a binary tree, which is to say it is either empty or a node consisting of two children, each of which is also a binary tree.

The judgment $a = b$ `nat` defining the equality of a `nat` and b `nat` is inductively defined by the following rules:

$$\frac{}{\texttt{zero} = \texttt{zero nat}} \tag{2.4a}$$

$$\frac{a = b\ \texttt{nat}}{\texttt{succ}(a) = \texttt{succ}(b)\ \texttt{nat}}. \tag{2.4b}$$

In each of the preceding examples we have made use of a notational convention for specifying an infinite family of rules by a finite number of patterns, or *rule schemes*. For example, Rule (2.2b) is a rule scheme that determines one rule, called an *instance* of the rule scheme, for each choice of object a in the rule. We rely on context to determine whether a rule is stated for a *specific* object a or is instead intended as a rule scheme specifying a rule for *each choice* of objects in the rule.

A collection of rules is considered to define the *strongest* judgment that is *closed under*, or *respects*, those rules. To be closed under the rules simply means that the rules are *sufficient*

to show the validity of a judgment: J holds *if* there is a way to obtain it using the given rules. To be the *strongest* judgment closed under the rules means that the rules are also *necessary*: J holds *only if* there is a way to obtain it by applying the rules. The sufficiency of the rules means that we may show that J holds by *deriving* it by composing rules. Their necessity means that we may reason about it using *rule induction*.

2.3 Derivations

To show that an inductively defined judgment holds, it is enough to exhibit a *derivation* of it. A derivation of a judgment is a finite composition of rules, starting with axioms and ending with that judgment. It may be thought of as a tree in which each node is a rule whose children are derivations of its premises. We sometimes say that a derivation of J is evidence for the validity of an inductively defined judgment J.

We usually depict derivations as trees with the conclusion at the bottom and with the children of a node corresponding to a rule appearing above it as evidence for the premises of that rule. Thus, if

$$\frac{J_1 \quad \ldots \quad J_k}{J}$$

is an inference rule and $\nabla_1, \ldots, \nabla_k$ are derivations of its premises, then

$$\frac{\nabla_1 \quad \ldots \quad \nabla_k}{J}$$

is a derivation of its conclusion. In particular, if $k = 0$, then the node has no children.

For example, this is a derivation of `succ(succ(succ(zero)))` `nat`:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{zero nat}}}{\texttt{succ(zero) nat}}}{\texttt{succ(succ(zero)) nat}}}{\texttt{succ(succ(succ(zero))) nat}}\,. \tag{2.5}$$

Similarly, here is a derivation of a `node(node(empty;empty);empty)` `tree`:

$$\dfrac{\dfrac{\dfrac{}{\texttt{empty tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(empty;empty) tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(node(empty;empty);empty) tree}}\,. \tag{2.6}$$

To show that an inductively defined judgment is derivable we need only find a derivation for it. There are two main methods for finding derivations, called *forward chaining*, or *bottom-up construction*, and *backward chaining*, or *top-down construction*. Forward chaining starts with the axioms and works forward toward the desired conclusion, whereas backward chaining starts with the desired conclusion and works backward toward the axioms.

More precisely, a forward chaining search maintains a set of derivable judgments and continually extends this set by adding to it the conclusion of any rule all of whose premises

are in that set. Initially, the set is empty; the process terminates when the desired judgment occurs in the set. Assuming that all rules are considered at every stage, forward chaining will eventually find a derivation of any derivable judgment, but it is impossible (in general) to decide algorithmically when to stop extending the set and conclude that the desired judgment is not derivable. We may go on and on adding more judgments to the derivable set without ever achieving the intended goal. It is a matter of understanding the global properties of the rules to determine that a given judgment is not derivable.

Forward chaining is undirected in the sense that it does not take account of the end goal when deciding how to proceed at each step. In contrast, backward chaining is goal-directed. Backward chaining search maintains a queue of current goals, judgmnts whose derivations are to be sought. Initially, this set consists solely of the judgment we wish to derive. At each stage, we remove a judgment from the queue and consider all rules whose conclusion is that judgment. For each such rule, we add the premises of that rule to the back of the queue and continue. If there is more than one such rule, this process must be repeated, with the same starting queue, for each candidate rule. The process terminates whenever the queue is empty, all goals having been achieved; any pending consideration of candidate rules along the way may be discarded. As with forward chaining, backward chaining will eventually find a derivation of any derivable judgment, but there is, in general, no algorithmic method for determining in general whether the current goal is derivable. If it is not, we may futilely add more and more judgments to the goal set, never reaching a point at which all goals have been satisfied.

2.4 Rule Induction

Because an inductive definition specifies the *strongest* judgment closed under a collection of rules, we may reason about them by *rule induction*. The principle of rule induction states that to show that a property $\mathcal{P}$ holds of a judgment J whenever J is derivable, it is enough to show that $\mathcal{P}$ is *closed under*, or *respects*, the rules defining J. Writing $\mathcal{P}(J)$ to mean that the property $\mathcal{P}$ holds of the judgment J, we say that $\mathcal{P}$ respects the rule

$$\frac{J_1 \quad \ldots \quad J_k}{J}$$

if $\mathcal{P}(J)$ holds whenever $\mathcal{P}(J_1), \ldots, \mathcal{P}(J_k)$. The assumptions $\mathcal{P}(J_1), \ldots, \mathcal{P}(J_k)$ are called the *inductive hypotheses*, and $\mathcal{P}(J)$ is called the *inductive conclusion* of the inference.

The principle of rule induction is simply the expression of the definition of an inductively defined judgment form as the *strongest* judgment form closed under the rules comprising the definition. This means that the judgment form defined by a set of rules is both (a) closed under those rules and (b) sufficient for any other property also closed under those rules. The former means that a derivation is evidence for the validity of a judgment; the latter means that we may reason about an inductively defined judgment form by rule induction.

When specialized to Rules (2.2), the principle of rule induction states that to show $\mathcal{P}(a\ \mathtt{nat})$ whenever $a\ \mathtt{nat}$, it is enough to show that

1. $\mathcal{P}(\mathtt{zero}\ \mathtt{nat})$, and
2. for every a, if $a\ \mathtt{nat}$ and $\mathcal{P}(a\ \mathtt{nat})$, then $(\mathtt{succ}(a)\ \mathtt{nat}$ and) $\mathcal{P}(\mathtt{succ}(a)\ \mathtt{nat})$.

This is just the familiar principle of *mathematical induction* arising as a special case of rule induction.

Similarly, rule induction for Rules (2.3) states that to show $\mathcal{P}(a\ \mathtt{tree})$ whenever $a\ \mathtt{tree}$, it is enough to show that

1. $\mathcal{P}(\mathtt{empty}\ \mathtt{tree})$, and
2. for every a_1 and a_2, if $a_1\ \mathtt{tree}$ and $\mathcal{P}(a_1\ \mathtt{tree})$, and if $a_2\ \mathtt{tree}$ and $\mathcal{P}(a_2\ \mathtt{tree})$, then $(\mathtt{node}(a_1;a_2)\ \mathtt{tree}$ and) $\mathcal{P}(\mathtt{node}(a_1;a_2)\ \mathtt{tree})$.

This is called the principle of *tree induction* and is once again an instance of rule induction.

We may also show by rule induction that the predecessor of a natural number is also a natural number. Although this may seem self-evident, the point of the example is to show how to derive this from first principles.

Lemma 2.1. *If* $\mathtt{succ}(a)\ \mathtt{nat}$*, then* $a\ \mathtt{nat}$.

Proof It suffices to show that the property $\mathcal{P}(a\ \mathtt{nat})$ stating that $a\ \mathtt{nat}$ and that $a = \mathtt{succ}(b)$ implies $b\ \mathtt{nat}$ is closed under Rules (2.2).

Rule (2.2a). Clearly $\mathtt{zero}\ \mathtt{nat}$, and the second condition holds vacuously, because $\mathtt{zero}$ is not of the form $\mathtt{succ}(-)$.

Rule (2.2b). Inductively we know that $a\ \mathtt{nat}$ and that if a is of the form $\mathtt{succ}(b)$, then $b\ \mathtt{nat}$. We are to show that $\mathtt{succ}(a)\ \mathtt{nat}$, which is immediate, and that if $\mathtt{succ}(a)$ is of the form $\mathtt{succ}(b)$, then $b\ \mathtt{nat}$, and we have $b\ \mathtt{nat}$ by the inductive hypothesis.

This completes the proof. □

Using rule induction we may show that equality, as defined by Rules (2.4), is reflexive.

Lemma 2.2. *If* $a\ \mathtt{nat}$*, then* $a = a\ \mathtt{nat}$.

Proof By rule induction on Rules (2.2):

Rule (2.2a). Applying Rule (2.4a), we obtain $\mathtt{zero} = \mathtt{zero}\ \mathtt{nat}$.

Rule (2.2b). Assume that $a = a\ \mathtt{nat}$. It follows that $\mathtt{succ}(a) = \mathtt{succ}(a)\ \mathtt{nat}$ by an application of Rule (2.4b). □

Similarly, we may show that the successor operation is injective.

Lemma 2.3. *If* $\mathtt{succ}(a_1) = \mathtt{succ}(a_2)$ nat, *then* $a_1 = a_2$ nat.

Proof Similar to the proof of Lemma 2.1. □

2.5 Iterated and Simultaneous Inductive Definitions

Inductive definitions are often *iterated*, meaning that one inductive definition builds on top of another. In an iterated inductive definition the premises of a rule

$$\frac{J_1 \quad \ldots \quad J_k}{J}$$

may be instances of either a previously defined judgment form or the judgment form being defined. For example, the following rules define the judgment a list, stating that a is a list of natural numbers:

$$\frac{}{\mathtt{nil}\ \mathsf{list}} \tag{2.7a}$$

$$\frac{a\ \mathsf{nat} \quad b\ \mathsf{list}}{\mathtt{cons}(a;b)\ \mathsf{list}}. \tag{2.7b}$$

The first premise of Rule (2.7b) is an instance of the judgment form a nat, which was defined previously, whereas the premise b list is an instance of the judgment form being defined by these rules.

Frequently two or more judgments are defined at once by a *simultaneous inductive definition*. A simultaneous inductive definition consists of a set of rules for deriving instances of several different judgment forms, any of which may appear as the premise of any rule. Because the rules defining each judgment form may involve any of the others, none of the judgment forms may be taken to be defined prior to the others. Instead we must understand that all of the judgment forms are being defined at once by the entire collection of rules. The judgment forms defined by these rules are, as before, the strongest judgment forms that are closed under the rules. Therefore the principle of proof by rule induction continues to apply, albeit in a form that requires us to prove a property of each of the defined judgment forms simultaneously.

For example, consider the following rules, which constitute a simultaneous inductive definition of the judgments a even, stating that a is an even natural number, and a odd, stating that a is an odd natural number:

$$\frac{}{\mathtt{zero}\ \mathsf{even}} \tag{2.8a}$$

$$\frac{a\ \mathsf{odd}}{\mathtt{succ}(a)\ \mathsf{even}} \tag{2.8b}$$

$$\frac{a\ \mathsf{even}}{\mathtt{succ}(a)\ \mathsf{odd}}. \tag{2.8c}$$

The principle of rule induction for these rules states that to show simultaneously that $\mathcal{P}(a\ \mathsf{even})$ whenever a even and $\mathcal{P}(a\ \mathsf{odd})$ whenever a odd, it is enough to show the following:

1. $\mathcal{P}(\mathtt{zero}\ \mathsf{even})$;
2. if $\mathcal{P}(a\ \mathsf{odd})$, then $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{even})$;
3. if $\mathcal{P}(a\ \mathsf{even})$, then $\mathcal{P}(\mathtt{succ}(a)\ \mathsf{odd})$.

As a simple example, we may use simultaneous rule induction to prove that (1) if a even, then a nat, and (2) if a odd, then a nat. That is, we define the property $\mathcal{P}$ by (1) $\mathcal{P}(a\ \mathsf{even})$ iff a nat, and (2) $\mathcal{P}(a\ \mathsf{odd})$ iff a nat. The principle of rule induction for Rules (2.8) states that it is sufficient to show the following facts:

1. $\mathtt{zero}\ \mathsf{nat}$, which is derivable by Rule (2.2a).
2. If a nat, then $\mathtt{succ}(a)\ \mathsf{nat}$, which is derivable by Rule (2.2b).
3. If a nat, then $\mathtt{succ}(a)\ \mathsf{nat}$, which is also derivable by Rule (2.2b).

2.6 Defining Functions by Rules

A common use of inductive definitions is to define a function by giving an inductive definition of its *graph* relating inputs to outputs and then showing that the relation uniquely determines the outputs for given inputs. For example, we may define the addition function on natural numbers as the relation $\mathsf{sum}(a;b;c)$, with the intended meaning that c is the sum of a and b, as follows:

$$\frac{b\ \mathsf{nat}}{\mathsf{sum}(\mathtt{zero};b;b)} \tag{2.9a}$$

$$\frac{\mathsf{sum}(a;b;c)}{\mathsf{sum}(\mathtt{succ}(a);b;\mathtt{succ}(c))}. \tag{2.9b}$$

The rules define a ternary (three-place) relation $\mathsf{sum}(a;b;c)$ among natural numbers a, b, and c. We may show that c is determined by a and b in this relation.

Theorem 2.4. *For every* a nat *and* b nat*, there exists a unique* c nat *such that* $\mathsf{sum}(a;b;c)$.

Proof The proof decomposes into two parts:

1. (Existence) If a nat and b nat, then there exists c nat such that $\mathsf{sum}(a;b;c)$.
2. (Uniqueness) If $\mathsf{sum}(a;b;c)$, and $\mathsf{sum}(a;b;c')$, then $c = c'$ nat.

For existence, let $\mathcal{P}(a\ \mathtt{nat})$ be the proposition *if* $b\ \mathtt{nat}$ *then there exists* $c\ \mathtt{nat}$ *such that* $\mathsf{sum}(a;b;c)$. We prove that if $a\ \mathtt{nat}$ then $\mathcal{P}(a\ \mathtt{nat})$ by rule induction on Rules (2.2). We have two cases to consider:

Rule (2.2a). We are to show $\mathcal{P}(\mathtt{zero}\ \mathtt{nat})$. Assuming $b\ \mathtt{nat}$ and taking c to be b, we obtain $\mathsf{sum}(\mathtt{zero};b;c)$ by Rule (2.9a).

Rule (2.2b). Assuming $\mathcal{P}(a\ \mathtt{nat})$, we are to show that $\mathcal{P}(\mathtt{succ}(a)\ \mathtt{nat})$. That is, we assume that if $b\ \mathtt{nat}$ then there exists c such that $\mathsf{sum}(a;b;c)$, and we are to show that if $b'\ \mathtt{nat}$, then there exists c' such that $\mathsf{sum}(\mathtt{succ}(a);b';c')$. To this end, suppose that $b'\ \mathtt{nat}$. Then by induction there exists c such that $\mathsf{sum}(a;b';c)$. Taking $c' = \mathtt{succ}(c)$ and applying Rule (2.9b), we obtain $\mathsf{sum}(\mathtt{succ}(a);b';c')$, as required.

For uniqueness, we prove that *if* $\mathsf{sum}(a;b;c_1)$*, then if* $\mathsf{sum}(a;b;c_2)$*, then* $c_1 = c_2\ \mathtt{nat}$ by rule induction based on Rules (2.9).

Rule (2.9a). We have $a = \mathtt{zero}$ and $c_1 = b$. By an inner induction on the same rules, we may show that if $\mathsf{sum}(\mathtt{zero};b;c_2)$, then c_2 is b. By Lemma 2.2 we obtain $b = b\ \mathtt{nat}$.

Rule (2.9b). We have that $a = \mathtt{succ}(a')$ and $c_1 = \mathtt{succ}(c_1')$, where $\mathsf{sum}(a';b;c_1')$. By an inner induction on the same rules, we may show that if $\mathsf{sum}(a;b;c_2)$, then $c_2 = \mathtt{succ}(c_2')\ \mathtt{nat}$, where $\mathsf{sum}(a';b;c_2')$. By the outer inductive hypothesis $c_1' = c_2'\ \mathtt{nat}$ and so $c_1 = c_2\ \mathtt{nat}$. □

2.7 Modes

The statement that one (or more) argument of a judgment is (perhaps uniquely) determined by its other arguments is called a *mode specification* for that judgment. For example, we have shown that every two natural numbers have a sum according to Rules (2.9). This fact may be restated as a mode specification by saying that the judgment $\mathsf{sum}(a;b;c)$ has *mode* $(\forall, \forall, \exists)$. The notation arises from the form of the proposition it expresses: *For all* $a\ \mathtt{nat}$ *and for all* $b\ \mathtt{nat}$*, there exists* $c\ \mathtt{nat}$ *such that* $\mathsf{sum}(a;b;c)$. If we wish to further specify that c is *uniquely* determined by a and b, we would say that the judgment $\mathsf{sum}(a;b;c)$ has mode $(\forall, \forall, \exists!)$, corresponding to the proposition that *for all* $a\ \mathtt{nat}$ *and for all* $b\ \mathtt{nat}$*, there exists a unique* $c\ \mathtt{nat}$ *such that* $\mathsf{sum}(a;b;c)$. If we wish to specify only that the sum is unique, *if it exists*, then we would say that the addition judgment has mode $(\forall, \forall, \exists^{\leq 1})$, corresponding to the proposition *for all* $a\ \mathtt{nat}$ *and for all* $b\ \mathtt{nat}$ *there exists at most one* $c\ \mathtt{nat}$ *such that* $\mathsf{sum}(a;b;c)$.

As these examples illustrate, a given judgment may satisfy several different mode specifications. In general the universally quantified arguments are to be thought of as the *inputs* of the judgment, and the existentially quantified arguments are to be thought of as its *outputs*. We usually try to arrange things so that the outputs come after the inputs, but it is not essential that we do so. For example, addition also has the mode $(\forall, \exists^{\leq 1}, \forall)$, stating that the sum and the first addend uniquely determine the second addend, if there is any such addend

at all. Put in other terms, this says that addition of natural numbers has a (partial) inverse, namely subtraction. We could equally well show that addition has mode $(\exists^{\leq 1}, \forall, \forall)$, which is just another way of stating that addition of natural numbers has a partial inverse.

Often there is an intended, or *principal*, mode of a given judgment, which we often foreshadow by our choice of notation. For example, when giving an inductive definition of a function, we often use equations to indicate the intended input and output relationships. For example, we may restate the inductive definition of addition [given by Rules (2.9)] using the following equations:

$$\frac{a\ \mathsf{nat}}{a + \mathtt{zero} = a\ \mathsf{nat}} \tag{2.10a}$$

$$\frac{a + b = c\ \mathsf{nat}}{a + \mathtt{succ}(b) = \mathtt{succ}(c)\ \mathsf{nat}}\,. \tag{2.10b}$$

When using this notation we tacitly incur the obligation to prove that the mode of the judgment is such that the object on the right-hand side of the equations is determined as a function of those on the left. Having done so, we abuse notation, writing $a + b$ for the unique c such that $a + b = c\ \mathsf{nat}$.

2.8 Notes

Aczel (1977) provides a thorough account of the theory of inductive definitions. The formulation given here is strongly influenced by Martin-Löf's development of the logic of judgments (Martin-Löf, 1983, 1987).

3 Hypothetical and General Judgments

A *hypothetical judgment* expresses an entailment between one or more hypotheses and a conclusion. We consider two notions of entailment, called *derivability* and *admissibility*. Both enjoy the same structural properties, but they differ in that derivability is stable under extension with new rules, admissibility is not. A *general judgment* expresses the universality, or generality, of a judgment. There are two forms of general judgment, the *generic* and the *parametric*. The generic judgment expresses generality with respect to all substitution instances for variables in a judgment. The parametric judgment expresses generality with respect to renamings of symbols.

3.1 Hypothetical Judgments

The hypothetical judgment codifies the rules for expressing the validity of a conclusion conditional on the validity of one or more hypothesis. There are two forms of hypothetical judgment that differ according to the sense in which the conclusion is conditional on the hypotheses. One is stable under extension with additional rules, and the other is not.

3.1.1 Derivability

For a given set $\mathcal{R}$ of rules, we define the *derivability* judgment, written $J_1, \ldots, J_k \vdash_{\mathcal{R}} K$, where each J_i and K are basic judgments, to mean that we may derive K from the *expansion* $\mathcal{R}[J_1, \ldots, J_k]$ of the rules $\mathcal{R}$ with the additional axioms

$$\frac{}{J_1} \qquad \cdots \qquad \frac{}{J_k}.$$

We treat the *hypotheses*, or *antecedents*, of the judgment $J_1, \ldots, J_n$ as "temporary axioms" and derive the *conclusion*, or *consequent*, by composing rules in $\mathcal{R}$. Thus evidence for a hypothetical judgment consists of a derivation of the conclusion from the hypotheses using the rules in $\mathcal{R}$.

We use capital Greek letters, frequently Γ or Δ, to stand for a finite collection of basic judgments, and we write $\mathcal{R}[\Gamma]$ for the expansion of $\mathcal{R}$ with an axiom corresponding to each judgment in Γ. The judgment $\Gamma \vdash_{\mathcal{R}} K$ means that K is derivable from rules $\mathcal{R}[\Gamma]$, and the judgment $\vdash_{\mathcal{R}} \Gamma$ means that $\vdash_{\mathcal{R}} J$ for each J in Γ. An equivalent way of defining

$J_1, \ldots, J_n \vdash_{\mathcal{R}} J$ is to say that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{3.1}$$

is *derivable* from $\mathcal{R}$, which means that there is a derivation of J composed of the rules in $\mathcal{R}$ augmented by treating $J_1, \ldots, J_n$ as axioms.

For example, consider the derivability judgment

$$a\ \mathtt{nat} \vdash_{(2.2)} \mathtt{succ(succ}(a)\mathtt{)\ nat} \tag{3.2}$$

relative to Rules (2.2). This judgment is valid for *any* choice of object a, as evidenced by the derivation

$$\dfrac{\dfrac{a\ \mathtt{nat}}{\mathtt{succ}(a)\ \mathtt{nat}}}{\mathtt{succ(succ}(a)\mathtt{)\ nat}} \tag{3.3}$$

that comprises Rules (2.2), starting with a nat as an axiom and ending with succ(succ(a)) nat. Equivalently, the validity of (3.2) may also be expressed by stating that the rule

$$\frac{a\ \mathtt{nat}}{\mathtt{succ(succ}(a)\mathtt{)\ nat}} \tag{3.4}$$

is derivable from Rules (2.2).

It follows directly from the definition of derivability that it is stable under extension with new rules.

Theorem 3.1 (Stability)**.** *If* $\Gamma \vdash_{\mathcal{R}} J$*, then* $\Gamma \vdash_{\mathcal{R}\cup\mathcal{R}'} J$.

Proof Any derivation of J from $\mathcal{R}[\Gamma]$ is also a derivation from $(\mathcal{R} \cup \mathcal{R}')[\Gamma]$, because any rule in $\mathcal{R}$ is also a rule in $\mathcal{R} \cup \mathcal{R}'$. □

Derivability enjoys a number of *structural properties* that follow from its definition, independently of the rules $\mathcal{R}$ in question.

Reflexivity. Every judgment is a consequence of itself: $\Gamma, J \vdash_{\mathcal{R}} J$. Each hypothesis justifies itself as conclusion.

Weakening. If $\Gamma \vdash_{\mathcal{R}} J$, then $\Gamma, K \vdash_{\mathcal{R}} J$. Entailment is not influenced by unexercised options.

Transitivity If $\Gamma, K \vdash_{\mathcal{R}} J$ and $\Gamma \vdash_{\mathcal{R}} K$, then $\Gamma \vdash_{\mathcal{R}} J$. If we replace an axiom by a derivation of it, the result is a derivation of its consequent without that hypothesis.

Reflexivity follows directly from the meaning of derivability. Weakening follows directly from the definition of derivability. Transitivity is proved by rule induction on the first premise.

3.1.2 Admissibility

Admissibility, written as $\Gamma \models_{\mathcal{R}} J$, is a weaker form of hypothetical judgment stating that $\vdash_{\mathcal{R}} \Gamma$ implies $\vdash_{\mathcal{R}} J$. That is, the conclusion J is derivable from rules $\mathcal{R}$ whenever the assumptions Γ are all derivable from rules $\mathcal{R}$. In particular if any of the hypotheses are *not* derivable relative to $\mathcal{R}$, then the judgment is vacuously true. An equivalent way to define the judgment $J_1, \ldots, J_n \models_{\mathcal{R}} J$ is to state that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{3.5}$$

is *admissible* relative to the rules in $\mathcal{R}$. This means that given any derivations of $J_1, \ldots, J_n$ using the rules in $\mathcal{R}$, we may construct a derivation of J using the rules in $\mathcal{R}$.

For example, the admissibility judgment

$$\mathtt{succ}(a)\ \mathtt{nat} \models_{(2.2)} a\ \mathtt{nat} \tag{3.6}$$

is valid, because any derivation of $\mathtt{succ}(a)\ \mathtt{nat}$ from Rules (2.2) must contain a subderivation of $a\ \mathtt{nat}$ from the same rules, which justifies the conclusion. The validity of (3.6) may equivalently be expressed by stating that the rule

$$\frac{\mathtt{succ}(a)\ \mathtt{nat}}{a\ \mathtt{nat}} \tag{3.7}$$

is admissible for Rules (2.2).

In contrast to derivability, the admissibility judgment is *not* stable under extension to the rules. For example, if we enrich Rules (2.2) with the axiom

$$\frac{}{\mathtt{succ(junk)\ nat}} \tag{3.8}$$

(where `junk` is some object for which `junk nat` is not derivable), then the admissibility (3.6) is *invalid*. This is because Rule (3.8) has no premises and there is no composition of rules deriving `junk nat`. Admissibility is as sensitive to which rules are *absent* from an inductive definition as it is to which rules are *present* in it.

The structural properties of derivability ensure that derivability is stronger than admissibility.

Theorem 3.2. *If* $\Gamma \vdash_{\mathcal{R}} J$, *then* $\Gamma \models_{\mathcal{R}} J$.

Proof Repeated application of the transitivity of derivability shows that if $\Gamma \vdash_{\mathcal{R}} J$ and $\vdash_{\mathcal{R}} \Gamma$, then $\vdash_{\mathcal{R}} J$. □

To see that the converse fails, observe that there is no composition of rules such that

$$\mathtt{succ(junk)\ nat} \vdash_{(2.2)} \mathtt{junk\ nat},$$

yet the admissibility judgment

$$\mathtt{succ(junk)\ nat} \models_{(2.2)} \mathtt{junk\ nat}$$

holds vacuously.

Evidence for admissibility may be thought of as a mathematical function transforming derivations $\nabla_1, \ldots, \nabla_n$ of the hypotheses into a derivation ∇ of the consequent. Therefore the admissibility judgment enjoys the same structural properties as derivability, and hence is a form of hypothetical judgment:

Reflexivity. If J is derivable from the original rules then J is derivable from the original rules: $J \models_{\mathcal{R}} J$.

Weakening. If J is derivable, from the original rules, assuming that each of the judgments in Γ are derivable from these rules, then J must also be derivable, assuming that Γ and also K are derivable from the original rules: If $\Gamma \models_{\mathcal{R}} J$, then $\Gamma, K \models_{\mathcal{R}} J$.

Transitivity. If $\Gamma, K \models_{\mathcal{R}} J$ and $\Gamma \models_{\mathcal{R}} K$, then $\Gamma \models_{\mathcal{R}} J$. If the judgments in Γ are derivable, so is K, by assumption, and hence so are the judgments in Γ, K, and hence so is J.

Theorem 3.3. *The admissibility judgment $\Gamma \models_{\mathcal{R}} J$ enjoys the structural properties of entailment.*

Proof Follows immediately from the definition of admissibility as stating that if the hypotheses are derivable relative to $\mathcal{R}$, then so is the conclusion. □

If a rule r is admissible with respect to a rule set $\mathcal{R}$ then $\vdash_{\mathcal{R},r} J$ is equivalent to $\vdash_{\mathcal{R}} J$. For if $\vdash_{\mathcal{R}} J$, then obviously $\vdash_{\mathcal{R},r} J$, by simply disregarding r. Conversely, if $\vdash_{\mathcal{R},r} J$, then we may replace any use of r by its expansion in terms of the rules in $\mathcal{R}$. It follows by rule induction on $\mathcal{R}, r$ that every derivation from the expanded set of rules $\mathcal{R}, r$ may be transformed into a derivation from $\mathcal{R}$ alone. Consequently, if we wish to show that $\mathcal{P}(J)$ whenever $\vdash_{\mathcal{R},r} J$, it is sufficient to show that $\mathcal{P}$ is closed under the rules $\mathcal{R}$ alone. That is, we need only consider the rules $\mathcal{R}$ in a proof by rule induction to derive $\mathcal{P}(J)$.

3.2 Hypothetical Inductive Definitions

It is useful to enrich the concept of an inductive definition to permit rules with derivability judgments as premises and conclusions. Doing so permits us to introduce *local hypotheses* that apply only in the derivation of a particular premise and also allows us to constrain inferences based on the *global hypotheses* in effect at the point where the rule is applied.

A *hypothetical inductive definition* consists of a collection of *hypothetical rules* of the following form:

$$\frac{\Gamma\ \Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma\ \Gamma_n \vdash J_n}{\Gamma \vdash J}. \tag{3.9}$$

The hypotheses Γ are the global hypotheses of the rule, and the hypotheses Γ_i are the local hypotheses of the ith premise of the rule. Informally, this rule states that J is a derivable consequence of Γ whenever each J_i is a derivable consequence of Γ, augmented with the additional hypotheses Γ_i. Thus one way to show that J is derivable from Γ is to show, in

turn, that each J_i is derivable from $\Gamma\ \Gamma_i$. The derivation of each premise involves a "context switch" in which we extend the global hypotheses with the local hypotheses of that premise, establishing a new set of global hypotheses for use within that derivation.

In most cases a rule is stated for *all* choices of global context, in which case it is said to be *uniform*. A uniform rule may be given in the *implicit* form

$$\frac{\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma_n \vdash J_n}{J} \tag{3.10}$$

which stands for the collection of all rules of the form (3.9) in which the global hypotheses have been made explicit.

A hypothetical inductive definition is to be regarded as an ordinary inductive definition of a *formal derivability judgment* $\Gamma \vdash J$ consisting of a finite set of basic judgments Γ and a basic judgment J. A collection of hypothetical rules $\mathcal{R}$ defines the strongest formal derivability judgment that is *structural* and *closed* under rules $\mathcal{R}$. Structurality means that the formal derivability judgment must be closed under the following rules:

$$\frac{}{\Gamma, J \vdash J} \tag{3.11a}$$

$$\frac{\Gamma \vdash J}{\Gamma, K \vdash J} \tag{3.11b}$$

$$\frac{\Gamma \vdash K \quad \Gamma, K \vdash J}{\Gamma \vdash J}. \tag{3.11c}$$

These rules ensure that formal derivability behaves like a hypothetical judgment. By a slight abuse of notation we write $\Gamma \vdash_{\mathcal{R}} J$ to indicate that $\Gamma \vdash J$ is derivable from rules $\mathcal{R}$.

The principle of *hypothetical rule induction* is just the principle of rule induction applied to the formal hypothetical judgment. So to show that $\mathcal{P}(\Gamma \vdash J)$ whenever $\Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under both the rules of $\mathcal{R}$ and under the structural rules. Thus, for each rule of the form (3.10), whether structural or in $\mathcal{R}$, we must show that

$$\text{if } \mathcal{P}(\Gamma\ \Gamma_1 \vdash J_1) \text{ and } \ldots \text{ and } \mathcal{P}(\Gamma\ \Gamma_n \vdash J_n), \text{ then } \mathcal{P}(\Gamma \vdash J).$$

This is just a restatement of the principle of rule induction given in Chapter 2, specialized to the formal derivability judgment $\Gamma \vdash J$.

In practice we usually dispense with the structural rules by the method described in Subsection 3.1.2. By proving that the structural rules are admissible, any proof by rule induction may restrict attention to the rules in $\mathcal{R}$ alone. If all of the rules of a hypothetical inductive definition are uniform, structural rules (3.11b) and (3.11c) are readily seen to be admissible. Usually, Rule (3.11a) must be postulated explictly as a rule, rather than be shown to be admissible on the basis of the other rules.

3.3 General Judgments

General judgments codify the rules for handling variables in a judgment. As in mathematics in general, a variable is treated as an *unknown* ranging over a specified collection of

objects. A *generic* judgment expresses that a judgment holds for any choice of objects replacing designated variables in the judgment. Another form of general judgment codifies the handling of parameters. A *parametric* judgment expresses generality over any choice of fresh renamings of designated parameters of a judgment. To keep track of the active variables and parameters in a derivation, we write $\Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J$ to indicate that J is derivable from Γ according to rules $\mathcal{R}$, with objects consisting of abt's over parameters $\mathcal{U}$ and variables $\mathcal{X}$.

Generic derivability judgment is defined by

$$\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \forall \pi : \vec{x} \leftrightarrow \vec{x}'\ \pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J,$$

where the quantification is restricted to variables $\vec{x}'$ not already active in $\mathcal{X}$. Evidence for generic derivability consists of a *generic derivation* ∇ involving the variables $\vec{x}$ such that for every fresh renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$, the derivation $\pi \cdot \nabla$ is evidence for $\pi \cdot \Gamma \vdash^{\mathcal{X},\vec{x}'}_{\mathcal{R}} \pi \cdot J$. The renaming ensures that the variables in the generic judgment are fresh (not already declared in $\mathcal{X}$) and that the meaning of the judgment is not dependent on the choice of variable names.

For example, the generic derivation ∇

$$\dfrac{\dfrac{\dfrac{}{x\ \mathtt{nat}}}{\mathtt{succ}(x)\ \mathtt{nat}}}{\mathtt{succ}(\mathtt{succ}(x))\ \mathtt{nat}}$$

is evidence for the judgment

$$x \mid x\ \mathtt{nat} \vdash^{\mathcal{X}}_{(2.2)} \mathtt{succ}(\mathtt{succ}(x))\ \mathtt{nat}.$$

This is because every fresh renaming of x to y in ∇ results in a valid derivation of the corresponding renaming of the indicated judgment.

The generic derivability judgment enjoys the following *structural properties* governing the behavior of variables:

Proliferation. If $\vec{x} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$, then $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$.

Renaming. If $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$, then $\vec{x}, x' \mid [x \leftrightarrow x'] \cdot \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [x \leftrightarrow x'] \cdot J$ for any $x' \notin \mathcal{X}, \vec{x}$.

Substitution. If $\vec{x}, x \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$ and $a \in \mathcal{B}[\mathcal{X}, \vec{x}]$, then $\vec{x} \mid [a/x]\Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [a/x]J$.

(It is left implicit in the principle of substitution that sorts are to be respected in that the substituting object must be of the same sort as the variable that is being substituted.) Proliferation is guaranteed by the interpretation of rule schemes as ranging over all expansions of the universe. Renaming is built into the meaning of the generic judgment. Substitution holds as long as the rules themselves are closed under substitution. This need not be the case, but in practice this requirement is usually met.

Parametric derivability is defined analogously to generic derivability, albeit by generalizing over parameters rather than over variables. Parametric derivability is defined by

$$\vec{u}; \vec{x} \mid \Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \forall \rho : \vec{u} \leftrightarrow \vec{u}'\ \forall \pi : \vec{x} \leftrightarrow \vec{x}'\ \rho \cdot \pi \cdot \Gamma \vdash^{\mathcal{U},\vec{u}';\mathcal{X},\vec{x}'}_{\mathcal{R}} \rho \cdot \pi \cdot J.$$

Evidence for parametric derivability consists of a derivation ∇ involving the parameters $\vec{u}$ and variables $\vec{x}$, each of whose fresh renamings is a derivation of the corresponding renaming of the underlying hypothetical judgment.

Recalling from Chapter 1 that parameters admit disequality, we cannot expect any substitution principle for parameters to hold of a parametric derivability. It does, however, validate the structural properties of proliferation and renaming because the presence of additional parameters does not affect the formation of an abt, and parametric derivability is defined to respect all fresh renamings of parameters.

3.4 Generic Inductive Definitions

A *generic inductive definition* admits generic hypothetical judgments in the premises of rules, with the effect of augmenting the variables, as well as the rules, within those premises. A *generic rule* has the form

$$\frac{\vec{x}\,\vec{x}_1 \mid \Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \vec{x}\,\vec{x}_n \mid \Gamma\,\Gamma_n \vdash J_n}{\vec{x} \mid \Gamma \vdash J}. \tag{3.12}$$

The variables $\vec{x}$ are the *global variables* of the inference, and, for each $1 \leq i \leq n$, the variables $\vec{x}_i$ are the *local variables* of the ith premise. In most cases a rule is stated for *all* choices of global variables and global hypotheses. Such rules may be given in *implicit form*:

$$\frac{\vec{x}_1 \mid \Gamma_1 \vdash J_1 \quad \ldots \quad \vec{x}_n \mid \Gamma_n \vdash J_n}{J}. \tag{3.13}$$

A generic inductive definition is just an ordinary inductive definition of a family of *formal generic judgments* of the form $\vec{x} \mid \Gamma \vdash J$. Formal generic judgments are identified up to the renaming of variables, so that the latter judgment is treated as identical to the judgment $\vec{x}' \mid \pi \cdot \Gamma \vdash \pi \cdot J$ for any renaming $\pi : \vec{x} \leftrightarrow \vec{x}'$. If $\mathcal{R}$ is a collection of generic rules, we write $\vec{x} \mid \Gamma \vdash_{\mathcal{R}} J$ to mean that the formal generic judgment $\vec{x} \mid \Gamma \vdash J$ is derivable from rules $\mathcal{R}$.

When specialized to a collection of generic rules, the principle of rule induction states that to show $\mathcal{P}(\vec{x} \mid \Gamma \vdash J)$ whenever $\vec{x} \mid \Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under the rules $\mathcal{R}$. Specifically, for each rule in $\mathcal{R}$ of the form (3.12), we must show that

$$\text{if } \mathcal{P}(\vec{x}\,\vec{x}_1 \mid \Gamma\,\Gamma_1 \vdash J_1) \ \ldots \ \mathcal{P}(\vec{x}\,\vec{x}_n \mid \Gamma\,\Gamma_n \vdash J_n) \text{ then } \mathcal{P}(\vec{x} \mid \Gamma \vdash J).$$

By the identification convention (stated in Chapter 1) the property $\mathcal{P}$ must respect renamings of the variables in a formal generic judgment.

To ensure that the formal generic judgment behaves like a generic judgment, we must always ensure that the following *structural rules* are admissible:

$$\frac{}{\vec{x} \mid \Gamma, J \vdash J} \tag{3.14a}$$

$$\frac{\vec{x} \mid \Gamma \vdash J}{\vec{x} \mid \Gamma, J' \vdash J} \tag{3.14b}$$

$$\frac{\vec{x} \mid \Gamma \vdash J}{\vec{x}, x \mid \Gamma \vdash J} \tag{3.14c}$$

$$\frac{\vec{x}, x' \mid [x \leftrightarrow x'] \cdot \Gamma \vdash [x \leftrightarrow x'] \cdot J}{\vec{x}, x \mid \Gamma \vdash J} \tag{3.14d}$$

$$\frac{\vec{x} \mid \Gamma \vdash J \quad \vec{x} \mid \Gamma, J \vdash J'}{\vec{x} \mid \Gamma \vdash J'} \tag{3.14e}$$

$$\frac{\vec{x}, x \mid \Gamma \vdash J \quad a \in \mathcal{B}[\vec{x}]}{\vec{x} \mid [a/x]\Gamma \vdash [a/x]J}. \tag{3.14f}$$

The admissibility of Rule (3.14a) is, in practice, ensured by explicitly including it. The admissibility of Rules (3.14b) and (3.14c) is ensured if each of the generic rules is uniform, as we may assimilate the additional parameter x to the global parameters and the additional hypothesis J to the global hypotheses. The admissibility of Rule (3.14d) is ensured by the identification convention for the formal generic judgment. Rule (3.14f) must be verified explicitly for each inductive definition.

The concept of a generic inductive definition extends to parametric judgments as well. Briefly, rules are defined on formal parametric judgments of the form $\vec{u}; \vec{x} \mid \Gamma \vdash J$, with parameters $\vec{u}$ as well as variables $\vec{x}$. Such formal judgments are identified up to the renaming of their variables and their parameters to ensure that the meaning is independent of the choice of variable and parameter names.

3.5 Notes

The concepts of entailment and generality are fundamental to logic and programming languages. The formulation given here builds on the work of Martin-Löf (1983, 1987) and Avron (1991). Hypothetical and general reasoning are consolidated into a single concept in the AUTOMATH languages (Nederpelt et al., 1994) and in the LF Logical Framework (Harper et al., 1993). These systems permit arbitrarily nested combinations of hypothetical and general judgments, whereas the present account considers only general hypothetical judgments over basic judgment forms.

The failure to distinguish parameters from variables is the source of many errors in language design. The crucial distinction is that whereas it makes sense to distinguish cases based on whether two parameters are the same or distinct, it makes no sense to do so for variables, because disequality is not preserved by substitution. Adhering carefully to this distinction avoids much confusion and complication in language design (see, for example, Chapter 41).