Programming Languages: Theory and Practice

(Working Draft of August 28, 2002)

Robert Harper
Carnegie Mellon University

Spring Semester, 2002
Preface

This is a collection of lecture notes for Computer Science 15–312 Programming Languages. This course has been taught by the author in the Spring of 1999 and 2000 at Carnegie Mellon University, and by Andrew Appel in the Fall of 1999, 2000, and 2001 at Princeton University. I am grateful to Andrew for his advice and suggestions, and to our students at both Carnegie Mellon and Princeton whose enthusiasm (and patience!) was instrumental in helping to create the course and this text.

What follows is a working draft of a planned book that seeks to strike a careful balance between developing the theoretical foundations of programming languages and explaining the pragmatic issues involved in their design and implementation. Many considerations come into play in the design of a programming language. I seek here to demonstrate the central role of type theory and operational semantics in helping to define a language and to understand its properties.

Comments and suggestions are most welcome. Please send any you may have to me by electronic mail.

Enjoy!
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
<td>iii</td>
</tr>
<tr>
<td>I Preliminaries</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1 Inductive Definitions</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1.1 Informal Overview</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1.1.1 Judgements and Rules</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Rule Induction</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1.1.3 Defining Functions by Rule Induction</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>1.1.4 Admissible and Derivable Rules</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>1.2 A More Rigorous Development</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>1.2.1 Universes</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>1.2.2 Inference Rules</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1.2.3 Rule Induction</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>1.2.4 Admissibility and Derivability</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>1.3 Exercises</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>2 Transition Systems</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>2.1 Transition Systems</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>2.2 Exercises</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>II Defining a Language</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>3 Concrete Syntax</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>3.1 Context-Free Grammars</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>3.2 Ambiguity</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>3.3 Exercises</td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>

**Working Draft** August 28, 2002
## CONTENTS

### 4 First-Order Abstract Syntax
- 4.1 Abstract Syntax Trees ........................................ 25
- 4.2 Structural Induction ............................................ 26
- 4.3 Parsing ............................................................ 27
- 4.4 Exercises ....................................................... 29

### 5 Higher-Order Abstract Syntax
- 5.1 Variables, Binding, and Scope ............................... 31
- 5.2 Higher-Order Terms ............................................. 35
- 5.3 Renaming and Substitution .................................... 36
- 5.4 de Bruijn Indices ............................................... 37
- 5.5 Exercises ....................................................... 39

### 6 Static Semantics
- 6.1 Well-Formed Arithmetic Expressions ....................... 41
- 6.2 Exercises ....................................................... 42

### 7 Dynamic Semantics
- 7.1 Structured Operational Semantics ......................... 43
- 7.2 Evaluation Semantics ......................................... 46
- 7.3 Relating Transition and Evaluation Semantics ........... 47
- 7.4 Exercises ....................................................... 48

### III A Functional Language

#### 8 MinML, A Minimal Functional Language
- 8.1 Abstract Syntax ................................................ 51
- 8.2 Static Semantics ............................................... 52
- 8.3 Properties of Typing .......................................... 54
- 8.4 Dynamic Semantics ............................................ 56
- 8.5 Properties of the Dynamic Semantics ..................... 58
- 8.6 Exercises ....................................................... 58

#### 9 Type Safety for MinML
- 9.1 Defining Type Safety .......................................... 59
- 9.2 Type Safety of MinML ......................................... 60
- 9.3 Run-Time Errors and Safety ................................ 63
## CONTENTS

### IV  An Imperative Language  67

10 TinyC, An Imperative Language  69
   10.1 Syntax  ..................................  70
   10.2 Static Semantics  ...........................  71
   10.3 Dynamic Semantics  .........................  74
   10.4 Block Structure  .............................  78
   10.5 Type Safety  ................................  79
   10.6 Exercises  ..................................  79

### V  Control and Data Flow  81

11 Abstract Machines  83
   11.1 Control Flow  ...............................  84
   11.2 Environments  ..............................  91

12 Continuations  97
   12.1 Informal Overview of Continuations ..........  99
   12.2 Semantics of Continuations .................  102
   12.3 Coroutines  ................................  104

13 Exceptions  109

### VI  Imperative Functional Programming  117

14 Mutable Storage  119
   14.1 References  .................................  119

15 Monads  125
   15.1 Monadic MinML  .........................  126
   15.2 Reifying Effects  .........................  128
   15.3 Exercises  ..................................  129

### VII  Cost Semantics and Parallelism  131

16 Cost Semantics  133
   16.1 Evaluation Semantics  ......................  133
   16.2 Relating Evaluation Semantics to Transition Semantics  ...  134
# CONTENTS

16.3 Cost Semantics .................................................. 135
16.4 Relating Cost Semantics to Transition Semantics ............ 136
16.5 Exercises ......................................................... 137

17 Implicit Parallelism .................................................. 139
  17.1 Tuple Parallelism ................................................. 139
  17.2 Work and Depth ................................................ 141
  17.3 Vector Parallelism .............................................. 144

18 A Parallel Abstract Machine ........................................ 149
  18.1 A Simple Parallel Language ..................................... 149
  18.2 A Parallel Abstract Machine .................................... 151
  18.3 Cost Semantics, Revisited ...................................... 153
  18.4 Provable Implementations (Summary) .......................... 154

VIII Data Structures and Abstraction .................................... 157
19 Aggregate Data Structures ........................................... 159
  19.1 Products ........................................................ 159
  19.2 Sums ............................................................ 161
  19.3 Recursive Types ............................................... 162

20 Polymorphism .......................................................... 165
  20.1 Polymorphic MinML ............................................. 166
  20.2 ML-style Type Inference ....................................... 172
  20.3 Parametricity .................................................... 173
    20.3.1 Informal Discussion ...................................... 174
    20.3.2 Relational Parametricity (Optional) .................... 177

21 Data Abstraction ...................................................... 183
  21.1 Existential Types ............................................... 184
    21.1.1 Abstract Syntax ......................................... 184
    21.1.2 Correspondence With ML .................................. 184
    21.1.3 Static Semantics ......................................... 186
    21.1.4 Dynamic Semantics ...................................... 187
    21.1.5 Safety .................................................... 188
  21.2 Representation Independence ................................. 188
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>IX</td>
<td>Laziness 193</td>
</tr>
<tr>
<td>22</td>
<td>Lazy Types 195</td>
</tr>
<tr>
<td>22.1</td>
<td>Lazy Types in MinML 197</td>
</tr>
<tr>
<td>22.1.1</td>
<td>Lazy Lists in an Eager Language 199</td>
</tr>
<tr>
<td>22.1.2</td>
<td>Delayed Evaluation and Lazy Data Structures 205</td>
</tr>
<tr>
<td>23</td>
<td>Lazy Languages 209</td>
</tr>
<tr>
<td>23.0.3</td>
<td>Call-by-Name and Call-by-Need 211</td>
</tr>
<tr>
<td>23.0.4</td>
<td>Strict Types in a Lazy Language 213</td>
</tr>
<tr>
<td>X</td>
<td>Dynamic Typing 215</td>
</tr>
<tr>
<td>24</td>
<td>Dynamic Typing 217</td>
</tr>
<tr>
<td>24.1</td>
<td>Dynamic Typing 219</td>
</tr>
<tr>
<td>24.2</td>
<td>Implementing Dynamic Typing 220</td>
</tr>
<tr>
<td>24.3</td>
<td>Dynamic Typing as Static Typing 222</td>
</tr>
<tr>
<td>25</td>
<td>Featherweight Java 225</td>
</tr>
<tr>
<td>25.1</td>
<td>Abstract Syntax 225</td>
</tr>
<tr>
<td>25.2</td>
<td>Static Semantics 228</td>
</tr>
<tr>
<td>25.3</td>
<td>Dynamic Semantics 230</td>
</tr>
<tr>
<td>25.4</td>
<td>Type Safety 232</td>
</tr>
<tr>
<td>25.5</td>
<td>Acknowledgement 233</td>
</tr>
<tr>
<td>XI</td>
<td>Subtyping and Inheritance 235</td>
</tr>
<tr>
<td>26</td>
<td>Subtyping 237</td>
</tr>
<tr>
<td>26.1</td>
<td>MinML With Subtyping 237</td>
</tr>
<tr>
<td>26.2</td>
<td>Varieties of Subtyping 239</td>
</tr>
<tr>
<td>26.2.1</td>
<td>Arithmetic Subtyping 239</td>
</tr>
<tr>
<td>26.2.2</td>
<td>Function Subtyping 240</td>
</tr>
<tr>
<td>26.2.3</td>
<td>Product and Record Subtyping 242</td>
</tr>
<tr>
<td>26.2.4</td>
<td>Reference Subtyping 243</td>
</tr>
<tr>
<td>26.3</td>
<td>Type Checking With Subtyping 244</td>
</tr>
<tr>
<td>26.4</td>
<td>Implementation of Subtyping 246</td>
</tr>
<tr>
<td>26.4.1</td>
<td>Coercions 246</td>
</tr>
</tbody>
</table>
## Inheritance and Subtyping in Java

27.1 Inheritance Mechanisms in Java ........................................... 251
   27.1.1 Classes and Instances ................................................. 251
   27.1.2 Subclasses .............................................................. 253
   27.1.3 Abstract Classes and Interfaces .................................. 254
27.2 Subtyping in Java ............................................................. 256
   27.2.1 Subtyping ............................................................... 256
   27.2.2 Subsumption ............................................................ 257
   27.2.3 Dynamic Dispatch .................................................... 258
   27.2.4 Casting ................................................................. 259
27.3 Methodology ................................................................. 261

## Concurrency

28 Concurrent ML .................................................................... 265

## Type Checking

29 Type Checking ................................................................. 269
   29.1 Type Synthesis ............................................................ 269
   29.2 Synthesis and Analysis ................................................ 271
   29.3 Exercises ................................................................. 274
30 Type Reconstruction .......................................................... 275
   30.1 Informal Overview ....................................................... 275
   30.2 Type Reconstruction .................................................... 278
   30.3 Constraint Generation ................................................ 283
   30.4 Solving Constraints ..................................................... 288

## Storage Management

31 Storage Management .......................................................... 297
   31.1 The A Machine ............................................................ 297
   31.2 Garbage Collection ...................................................... 301
Part I

Preliminaries
Chapter 1

Inductive Definitions

Inductive definitions are an indispensable tool in the study of programming languages. In this chapter we will develop the basic framework of inductive definitions, and give some examples of their use.

1.1 Informal Overview

In this section we give an informal overview of inductive definitions, with an emphasis on how they are used in practice.

1.1.1 Judgements and Rules

An inductive definition consists of a collection of inference rules defining one or more judgements. A judgement is an assertion stating that a property holds of some object. For example, the judgement $x \text{ nat}$ might state that $x$ is a natural number, and the judgement $t \text{ tree}$ might state that $t$ is a binary tree.

The inference rules determine the conditions under which a judgement may be inferred, or derived. An inference rule has the form of an implication, stating that a judgement is inferrable whenever some other judgements (possibly none) are inferrable. Rules are written in the form

$$\frac{J_1 \ldots J_n}{J}$$

where $J$ and each $J_i$ ($1 \leq i \leq n$) are judgements. The judgement $J$ is called the conclusion of the rule, and the judgements $J_1, \ldots, J_n$ are its premises. If a rule has no premises (i.e., $n = 0$), the rule is called an axiom.
A rule of this form states that the judgement \( J \) is inferrable, provided that each of the judgements \( J_1, \ldots, J_n \) is inferrable. Thus axioms state that a judgement is inferrable unconditionally, whereas rules with premises state the conditional inferrability of a judgement. For example, the following set of rules, \( R_N \), constitute an inductive definition of the judgement \( x \text{ nat} \):

\[
\text{zero nat} \quad x \text{ nat} \quad \text{succ}(x) \text{ nat}
\]

The first rule states that \( \text{zero} \) is a natural number. The second states that if \( x \) is a natural number, then so is \( \text{succ}(x) \).

Rules may be composed to form a derivation of a judgement \( J \) from premises \( J_1, \ldots, J_n \). A derivation is a tree whose nodes are judgements such that the children of a node are the premises of some rule ending with the judgement at that node. Such a tree is a derivation of a judgement \( J \) from premises \( J_1, \ldots, J_n \) iff the root of the tree is \( J \) and its leaves are the judgements \( J_1, \ldots, J_n \).

Derivation trees are normally depicted as “stacked” inference rules. For example, here is a derivation of the judgement \( \text{succ}(	ext{succ}(\text{zero})) \text{ nat} \):

\[
\text{zero nat} \quad \text{succ(}\text{zero}) \text{ nat} \quad \text{succ(succ(}\text{zero})) \text{ nat}
\]

To take another example, here is an inductive definition of the judgement \( t \text{ tree} \), stating that \( t \) is a binary tree:

\[
\text{empty tree} \quad x \text{ tree} \quad y \text{ tree} \quad \text{node}(x, y) \text{ tree}
\]

Using these rules, we may construct a derivation of the judgement

\[
\text{node(}\text{empty, node(}\text{empty, empty)})) \text{ tree}
\]

as follows:

\[
\text{empty tree} \quad \text{empty tree} \quad \text{node(}\text{empty, empty}) \text{ tree} \quad \text{node(}\text{empty, node(}\text{empty, empty})) \text{ tree}
\]

In practice, we find a derivation of a judgement \( J \) by starting with \( J \) and working “backwards”, looking for a rule ending with \( J \) with premises \( J_1, \ldots, J_n \), then finding derivations of each of the \( J_i \)'s by the same procedure. This process is called goal-directed search; the judgement \( J \) is the goal,
1.1 Informal Overview

and each of the $J_i$’s are subgoals. Note that there may be many rules ending with $J$; if we fail to find a derivation by using one rule, we may have to abandon the attempt, and try another rule instead. If $J$ is, in fact, derivable, then this process will eventually find a derivation, but if not, there is no guarantee that it will terminate! We may, instead, futilely apply rules forever, introducing more sub-goals each time, and never completing the derivation.

Often we give a simultaneous inductive definition of several judgements at once. For example, here is a simultaneous inductive definition of the judgements $t$ tree, stating that $t$ is a variadic tree, and $f$ forest, stating that $f$ is a variadic forest. By “variadic” we mean that the number of children of any given node in a tree varies with each node.

$$
\begin{array}{c}
\text{f forest} \\
\text{node}(f) \text{ tree} \\
\text{nil forest} \\
\text{cons}(t, f) \text{ forest}
\end{array}
$$

1.1.2 Rule Induction

What makes an inductive definition inductive is that the rules are exhaustive in the sense that a judgement is defined to hold iff it can be inferred by these rules. This means that if a judgement $J$ is inferrable from a rule set $R$, then there must be a rule in $R$ ending with $J$ such that each of its premises are also inferrable. For example, if $n$ nat is inferrable according to the rules $R_N$, then either it is inferrable by the first rule, in which case $n = \text{zero}$, or by the second, in which case $n = \text{succ}(m)$ and $m$ nat is itself inferrable. Similarly, if $t$ tree is inferrable according to the rules $R_T$ given above, then either $t = \text{empty}$ or $t = \text{node}(t_1, t_2)$, where $t_1$ tree and $t_2$ tree are both inferrable.

This observation provides the basis for reasoning about derivable judgements by rule induction (also known as induction on derivations). For any set of rules, $R$, to show that a property $P$ holds of every inferrable judgement, it is enough to show that for every rule

$$
\begin{array}{c}
J_1 \\
\vdots \\
J_n \\
\hline
J
\end{array}
$$

in $R$, if $J_1, \ldots, J_n$ all have property $P$, then $J$ also has property $P$. By doing this for every rule in $R$, we cover all the cases, and establish that $P$ holds for every inferrable judgement.

The assumption that $P$ holds for each premise of a rule is called the inductive hypothesis. The proof that $P$ holds for the conclusion, under these
assumptions, is called the \textit{inductive step}. In the case of axioms the inductive hypothesis isvacuous; we must simply establish the conclusion outright, with no further assumptions to help us. If we can carry out the inductive step for each rule in $\mathcal{R}$, we thereby establish that $P$ holds for every inferrable judgement, since the inference must arise by the application of some rule whose premises are derivable (and hence, by inductive hypothesis, have the property $P$).

For example, consider again the rule set $\mathcal{R}_N$. The principle of rule induction for $\mathcal{R}_N$ states that to show $P(n \text{ nat})$, it is enough to show

1. $P(\text{zero nat})$;

2. if $P(n \text{ nat})$, then $P(\text{succ}(n) \text{ nat})$.

This is, of course, the familiar \textit{principle of mathematical induction}.

Similarly, the principle of rule induction for $\mathcal{R}_T$ states that if we are to show that $P(t \text{ tree})$, it is enough to show

1. $P(\text{empty tree})$;

2. if $P(t_1 \text{ tree})$ and $P(t_2 \text{ tree})$, then $P(\text{node}(t_1, t_2) \text{ tree})$.

This is called the \textit{principle of tree induction}, or \textit{induction on the structure of a tree}.

As a notational convenience, when the judgements in question are all of the form $x \ l$ for $x$ an object and $l$ is a property of the object $x$, we often write $P_l(x)$, rather than the more cumbersome $P(x \ l)$. If there is only one form of judgement, $x \ l$, then we often drop the subscript entirely, writing just $P(x)$, rather than $P_l(x)$ or $P(x \ l)$. Thus, instead of writing $P(n \text{ nat})$, we may write $P_{\text{mathsf{nat}}} n$, or just $P(n)$, when it is clear from context that we are working the $\mathcal{R}_N$. Similarly $P(t \text{ tree})$ is often written $P_{\text{tree}}(t)$, or just $P(t)$.

Rule sets that define more than one judgement give rise to proofs by \textit{simultaneous} induction. For example, if we wish to show $P_{\text{tree}}(t)$ for all $t$ such that $t \text{ tree}$ and $P_{\text{forest}}(f)$ for all $f$ such that $f \text{ forest}$, then it is enough to show

1. if $P_{\text{forest}}(f)$, then $P_{\text{tree}}(\text{node}(f))$.

2. $P_{\text{forest}}(\text{nil})$.

3. if $P_{\text{tree}}(t)$ and $P_{\text{forest}}(f)$, then $P_{\text{forest}}(\text{cons}(t, f))$.

It is easy to check that this induction principle follows from the general principle of rule induction by simply working through the rules $\mathcal{R}_{TF}$, taking account of the notational conventions just mentioned.
1.1 Informal Overview

1.1.3 Defining Functions by Rule Induction

A common use of rule induction is to justify the definition of a function by a set of equations. For example, consider the following recursion equations:

\[
\begin{align*}
\text{\texttt{hgt}_{tree}}(\text{\texttt{empty}}) &= 0 \\
\text{\texttt{hgt}_{tree}}(\text{\texttt{node}}(t_1, t_2)) &= 1 + \max(\text{\texttt{hgt}_{tree}}(t_1), \text{\texttt{hgt}_{tree}}(t_2))
\end{align*}
\]

We prove by rule induction that if \( t \) tree then there exists a unique \( n \geq 0 \) such that \( \text{\texttt{hgt}_{tree}}(t) = n \). In other words, the above equations determine a function, \( \text{\texttt{hgt}} \).

We consider each rule in \( \mathcal{R}_T \) in turn. The first rule, stating that \( \text{\texttt{empty}} \) tree, is covered by the first equation. For the second rule, we may assume that \( \text{\texttt{hgt}_{tree}} \) assigns a unique height to \( t_1 \) and \( t_2 \). But then the second equation assigns a unique height to \( t = \text{\texttt{node}}(t_1, t_2) \).

Similarly, we may prove by simultaneous induction that the following equations define the height of a variadic tree and a variadic forest:

\[
\begin{align*}
\text{\texttt{hgt}_{tree}}(\text{\texttt{node}}(f)) &= 1 + \text{\texttt{hgt}_{forest}}(f) \\
\text{\texttt{hgt}_{forest}}(\text{\texttt{nil}}) &= 0 \\
\text{\texttt{hgt}_{forest}}(\text{\texttt{cons}}(t, f)) &= \max(\text{\texttt{hgt}_{tree}}(t), \text{\texttt{hgt}_{forest}}(f)).
\end{align*}
\]

It is easy to show by simultaneous induction that these equations determine two functions, \( \text{\texttt{hgt}_{tree}} \) and \( \text{\texttt{hgt}_{forest}} \).

1.1.4 Admissible and Derivable Rules

Given an inductive definition consisting of a set of rules \( \mathcal{R} \), there are two senses in which a rule

\[
\begin{array}{ccc}
J_1 & \ldots & J_n \\
\hline \\
J
\end{array}
\]

may be thought of as being redundant.

Such a rule is said to be derivable iff there is a derivation of \( J \) from premises \( J_1, \ldots, J_n \). This means that there is a composition of rules starting with the \( J_i \)’s and ending with \( J \). For example, the following rule is derivable in \( \mathcal{R}_N \):

\[
\begin{array}{c}
n \text{ nat} \\
\hline \\
\text{\texttt{succ}}(\text{\texttt{succ}}(\text{\texttt{succ}}(n))) \text{ nat}
\end{array}
\]
Its derivation is as follows:

\[
\begin{align*}
\frac{n \text{ nat}}{\text{succ}(n) \text{ nat}} \\
\frac{\text{succ}(\text{succ}(n)) \text{ nat}}{\text{succ}(\text{succ}(\text{succ}(n))) \text{ nat}}.
\end{align*}
\]

Such a rule is said to be *admissible* iff its conclusion is derivable from no premises whenever its premises are derivable from no premises. For example, the following rule is admissible in \( R_N \):

\[
\frac{\text{succ}(n) \text{ nat}}{n \text{ nat}}.
\]

First, note that this rule is *not* derivable for any choice of \( n \). For if \( n = \text{zero} \), then the only rule that applies has no premises, and if \( n = \text{succ}(m) \), then the only rule that applies has as premise \( m \text{ nat} \), rather than \( n \text{ nat} \). However, this rule *is* admissible! We may prove this by induction on the derivation of the premise of the rule. For if \( \text{succ}(n) \text{ nat} \) is derivable from no premises, it can only be by second rule, which means that \( n \text{ nat} \) is also derivable, as required.

While this example shows that not every admissible rule is derivable, the converse holds. For a rule to be derivable means precisely that if its premises are derivable, then so is its conclusion!

The distinction between admissible and derivable rules can be hard to grasp at first. One way to gain intuition is to note that if a rule is derivable in a rule set \( R \), then it remains derivable in any rule set \( R' \supseteq R \). This is because the derivation of that rule depends only on what rules are available, and is not sensitive to whether any other rules are also available. In contrast a rule can be admissible in \( R \), but inadmissible in some extension \( R' \supseteq R \)! For example, suppose that we add to \( R_N \) the rule

\[
\frac{\text{succ}(\text{junk}) \text{ nat}}{}.
\]

Now it is no longer the case that the rule

\[
\frac{\text{succ}(n) \text{ nat}}{n \text{ nat}}.
\]

is admissible, because if the premise were derived by the additional rule, there is no way to obtain a derivation of \( \text{junk} \text{ nat} \)!
1.2 A More Rigorous Development

Since admissibility is sensitive to which rules are absent, as well as to which are present, a proof of admissibility almost always proceeds by induction on one or more of its premises. This constitutes an exhaustive analysis of how the premises might have been derived, and concludes that in each case the conclusion must also have been derived. Adding an additional rule requires that we add an additional case to the proof, and there is no assurance (as we have just illustrated) that this will go through.

1.2 A More Rigorous Development

In this section we will give a more rigorous account of inductive definitions of a subset of a given set. This will include as a special case the foregoing treatment of inductive definitions of judgements, and will make clear the mathematical underpinnings of the principle of rule induction.

1.2.1 Universes

We will consider inductive definitions of subsets of some fixed universe of objects. In principle we may consider inductive definitions over any set of objects we like, but in practice we confine ourselves to sets of finitary objects, which can be put into one-to-one correspondence with the natural numbers. Given such a correspondence, it suffices to make all inductive definitions over the set of natural numbers. However, doing so requires that we explicitly define the encoding of each object of interest as a natural number, called its Gödel number. To avoid this complication we take a more liberal approach in which we admit inductive definitions over any specified set of objects.

For example, we will make use of the set of (finite) strings over a given alphabet as a universe for inductive definitions. Let \( \Sigma \) be a countable set of symbols, or letters, or characters. For example, \( \Sigma \) might be the set of ASCII or UniCode characters. The set of strings over \( \Sigma \), written \( \Sigma^* \), consists of the finite sequences of symbols from \( \Sigma \). We write \( s_1 s_2 \) for the concatenation of the string \( s_1 \) followed by the string \( s_2 \), write \( \varepsilon \) for the null string, and treat every \( a \in \Sigma \) as string of length 1.

Another example is the set of (first-order) terms over a given set of operators. Let \( O \) be a countable set of operators, and let \( \alpha : O \to \mathbb{N} \) be an assignment of arities to each of the operators. An operator \( o \in O \) of arity \( n \) (i.e., for which \( \alpha(o) = n \)) is said to be \( n \)-ary; the 0-ary operators are called constants. The set \( T \) of ast's, or terms, consists of all expressions of
the form $o(t_1, \ldots, t_n)$, where $o$ is an $n$-ary operator, and $t_1, \ldots, t_n$ are themselves ast’s. Such a term may be depicted as an ordered tree with root labelled by the operator $o$, and with $n$ children corresponding to the terms $t_1, \ldots, t_n$.

We often work with combinations of these basic universes. For example, we may consider inductive subsets of $T \times T$, the set of ordered pairs of ast’s, and so forth. Generally we will leave implicit the exact choice of the universe for a particular inductive definition.

### 1.2.2 Inference Rules

An inductive definition of a subset of a universe $\mathcal{U}$ consists of a collection of rules over $\mathcal{U}$. A rule over $\mathcal{U}$ has the form

$$
\frac{x_1 \ldots x_n}{x}
$$

where $x \in \mathcal{U}$ and each $x_i \in \mathcal{U}$ ($1 \leq i \leq n$). Thus a rule consists of a finite subset of $\mathcal{U}$ and an element of $\mathcal{U}$. The element $x$ is called the conclusion of the rule; the elements $x_1, \ldots, x_n$ are called the premises of the rule. A rule set is, quite obviously, a set of rules.

A subset $A \subseteq \mathcal{U}$ is closed under $R$, or $R$-closed, iff $x \in A$ whenever

$$
\frac{x_1 \ldots x_n}{x}
$$

is a rule in $R$ and each $x_i \in A$ for every $1 \leq i \leq n$.

The subset $I = I(R)$ inductively defined by $R$ is given by the equation

$$
I(R) = \bigcap \{ A \subseteq \mathcal{U} \mid A \text{ is } R\text{-closed} \}.
$$

As we shall see shortly, this is the smallest set closed under $R$.

For example, here is a set, $R_P$, of rules for deriving strings that are, as we shall prove later, are palindromes:

$$
\varepsilon \quad a \quad s \quad a \quad s
$$

The set of rules $R_P$ just given has $2 \times |\Sigma| + 1$ rules, where $|\Sigma|$ is the cardinality of the alphabet $\Sigma$. In particular, if $\Sigma$ is infinite, then there are infinitely many rules! Since we cannot expect to write down infinitely many rules, we need some means of defining large (or even infinite) rule sets. Here we have specified these using rule schemes. A rule scheme is a rule
1.2 A More Rigorous Development

involving one or more parameters ranging over a specified set (by default, the universe). For example, the third rule above is a rule scheme with two parameters, \( a \) and \( s \). The rule scheme determines one rule for each possible choice of character \( a \in \Sigma \) and \( s \in \Sigma^* \).

A simultaneous inductive definition of one or more judgements can be considered a single inductive definition of a subset of a suitable universe by a simple “labelling” device. A simultaneous inductive definition of the judgements \( x_1 l_1, \ldots, x_n l_n \), where each \( x_i \) ranges over a universe \( \mathcal{U} \), may be thought of as a simple inductive definition of a subset of the disjoint union of \( n \) copies of \( \mathcal{U} \), namely

\[
\mathcal{U} \times \{ l_1, \ldots, l_n \} = \{ x l_i \mid x \in \mathcal{U}, \ 1 \leq i \leq n \},
\]

where we write \( x l_i \) for the ordered pair \((x, l_i)\). The rules defining these judgements emerge as rules over this enlarged universe. Thus the rules \( \mathcal{R}_{T,F} \) given above may be seen as defining an inductive subset of the universe \( T \times \{ \text{tree, forest} \} \).

1.2.3 Rule Induction

As we mentioned earlier, the set \( I(\mathcal{R}) \) is the least set closed under \( \mathcal{R} \).

Theorem 1

Let \( \mathcal{R} \) be a rule set over \( \mathcal{U} \), and let \( I = I(\mathcal{R}) \).

1. \( I \) is \( \mathcal{R} \)-closed.

2. If \( A \) is \( \mathcal{R} \)-closed, then \( I \subseteq A \).

Proof:

1. Suppose that

\[
\frac{x_1 \ldots x_n}{x}
\]

is a rule in \( \mathcal{R} \), and that

\[
X = \{ x_1, \ldots, x_n \} \subseteq I.
\]

Since \( I \) is the intersection of all \( \mathcal{R} \)-closed sets, \( X \subseteq A \) for each \( \mathcal{R} \)-closed set \( A \). But then \( x \in A \) for each such \( A \), by the definition of \( \mathcal{R} \)-closure, and hence \( x \) is an element of their intersection, \( I \).

2. If \( A \) is \( \mathcal{R} \)-closed, then it is among the sets in the intersection defining \( I \). So \( I \subseteq A \).
The importance of this theorem is that it licenses the principle of *proof by rule induction* for a rule set $\mathcal{R}$:

To show that $I(\mathcal{R}) \subseteq X$, it suffices to show that $X$ is $\mathcal{R}$-closed.

That is, if we wish to show that $x \in X$ for every $x \in I(\mathcal{R})$, it is enough to show that $X$ is closed under the rules $\mathcal{R}$.

Returning to the inductively defined set $P$ above, suppose we wish to show that every $s \in P$ is in fact a palindrome. That is, we wish to show that

$$P \subseteq \{ s \in \Sigma^* \mid s = s^R \}.$$  

For this to hold, it is enough to show that the set of palindromes is closed under the rules $\mathcal{R}$. We consider each rule in turn, showing that if the premises are palindromes, then so is the conclusion.

1. $\varepsilon = \varepsilon^R$, so $\varepsilon$ is a palindrome.

2. $a = a^R$ for every $a \in \Sigma$, so $a$ is a palindrome.

3. Assume that $s = s^R$. Observe that

$$((a \, s \, a)^R = a \, s^R \, a$$

This completes the proof.\(^1\)

The *parity* of a palindrome is either 0 or 1, according to whether its length is either even or odd. Now that we know that the set of palindromes is inductively defined by the rules given earlier, we may define the parity function by the following equations:

$$parity(\varepsilon) = 0$$
$$parity(a) = 1$$
$$parity(a \, s \, a) = parity(s)$$

Notice that we include one clause of the function definition for each rule defining the domain of the function.

Why does this define a function? We must prove that if $s$ is a palindrome, then there exists a unique $x \in \{ 0, 1 \}$ such that $parity(s) = x$. This

\(^1\)You might also like to prove that every palindrome is a member of $P$. This can be achieved by (strong) induction over the length of palindrome $s$.
may be proved by rule induction by showing that the property \( P(s) \) given by the formula

\[
\exists! \ x \in \{0, 1\} \ \text{parity}(s) = x
\]

is closed under the rules \( \mathcal{R}_P \). The first two rules, for the null string and the single-letter strings, are covered by the first two clauses of the definition of \( \text{parity} \). For the third rule, we assume that \( \text{parity} \) is well-defined for \( s \) (i.e., there exists a unique \( x \) such that \( \text{parity}(s) = x \)). But then it follows directly from the third clause of the definition of \( \text{parity} \) that it is uniquely defined for \( asa \).

### 1.2.4 Admissibility and Derivability

Using this machinery we may shed additional light on admissibility and derivability of inference rules. Fix a rule set \( \mathcal{R} \) over some universe \( U \). A rule

\[
\frac{x_1 \ldots x_n}{x}
\]

is derivable iff \( x \in I(\mathcal{R} \cup \{ x_1, \ldots, x_n \}) \). That is, we take \( x_1, \ldots, x_n \) as new axioms, and ask whether \( x \) is derivable according to this expansion of the rule set. The same rule is admissible iff \( x \in I(\mathcal{R}) \) whenever \( x_i \in I(\mathcal{R}) \) for each \( 1 \leq i \leq n \). That is, we check whether \( x \) is in the set inductively defined by \( \mathcal{R} \), whenever the \( x_i \)'s are in that same set.

For example, consider the rule set \( \mathcal{R}_P \) defining the palindromic strings. It is easy to see that the rule

\[
\frac{s}{aba}
\]

is derivable, since if we add \( s \) as a new axiom, then we can apply the third rule of \( \mathcal{R}_P \) to this axiom twice to obtain a derivation of \( asaba \). On the other hand, the rule

\[
\frac{asa}{s}
\]

is admissible, since if \( asa \in I(\mathcal{R}) \), then so we must also have \( s \in I(\mathcal{R}) \). This is easily proved by rule induction, showing that the set

\[
\{ t \mid t = asa \text{ and } s \in I(\mathcal{R}) \}
\]

is \( \mathcal{R} \)-closed.

**Working Draft**

**August 28, 2002**
1.3 Exercises

1. Give (simultaneous) inductive definitions of various languages.

2. Prove properties of these languages, including well-definedness of certain functions over them.

3. Fill in missing proofs.
Chapter 2

Transition Systems

Transition systems are fundamental to the study of programming languages. They are used to describe the execution behavior of programs by defining an abstract computing device with a set, $S$, of states that are related by a transition relation, $\rightarrow$. The transition relation describes how the state of the machine evolves during execution.

2.1 Transition Systems

A transition system consists of a set $S$ of states, a subset $I \subseteq S$ of initial states, a subset $F \subseteq S$ of final states, and a binary transition relation $\rightarrow \subseteq S \times S$. We write $s \rightarrow s'$ to indicate that $(s, s') \in \rightarrow$. It is convenient to require that $s \nrightarrow$ in the case that $s \in F$.

An execution sequence is a sequence of states $s_0, \ldots, s_n$ such that $s_0 \in I$, and $s_i \rightarrow s_{i+1}$ for every $0 \leq i < n$. An execution sequence is maximal iff $s_n \nrightarrow$; it is complete iff it is maximal and, in addition, $s_n \in F$. Thus every complete execution sequence is maximal, but maximal sequences are not necessarily complete.

A state $s \in S$ for which there is no $s' \in S$ such that $s \rightarrow s'$ is said to be stuck. We require that all final states are stuck: if $s \in F$, then $s \nrightarrow$. But not all stuck states are final; these correspond to “run-time errors”, states for which there is no well-defined next state.

A transition system is deterministic iff for every $s \in S$ there exists at most one $s' \in S$ such that $s \rightarrow s'$. Most of the transition systems we will consider in this book are deterministic, the notable exceptions being those used to model concurrency.
The reflexive, transitive closure, \(\mapsto^*\), of the transition relation \(\mapsto\) is inductively defined by the following rules:

\[
\begin{align*}
\frac{}{s \mapsto^* s} & \quad s \mapsto s' \quad s' \mapsto^* s'' \quad \frac{}{s \mapsto^* s''}
\end{align*}
\]

It is easy to prove by rule induction that \(\mapsto^*\) is indeed reflexive and transitive.

The complete transition relation, \(\mapsto^1\), is the restriction to \(\mapsto^*\) to \(S \times F\). That is, \(s \mapsto^1 s'\) iff \(s \mapsto^* s'\) and \(s' \in F\).

The multistep transition relation, \(\mapsto^n\), is defined by induction on \(n \geq 0\) as follows:

\[
\begin{align*}
\frac{}{s \mapsto^0 s} & \quad s \mapsto s' \quad s' \mapsto^n s'' \quad \frac{}{s \mapsto^{n+1} s''}
\end{align*}
\]

It is easy to show that \(s \mapsto^* s'\) iff \(s \mapsto^n s'\) for some \(n \geq 0\).

Since the multistep transition is inductively defined, we may prove that \(P(e, e')\) holds whenever \(e \mapsto^* e'\) by showing

1. \(P(e, e)\).
2. if \(e \mapsto e'\) and \(P(e', e'')\), then \(P(e, e'')\).

The first requirement is to show that \(P\) is reflexive. The second is often described as showing that \(P\) is closed under head expansion, or closed under reverse evaluation.

### 2.2 Exercises

1. Prove that \(s \mapsto^* s'\) iff there exists \(n \geq 0\) such that \(s \mapsto^n s'\).