ROBERT HARPER

Practical Foundations for PROGRAMMING LANGUAGES

Second Edition

Practical Foundations for Programming Languages

Second Edition

Robert Harper
Carnegie Mellon University

CAMBRIDGE
UNIVERSITY PRESS

32 Avenue of the Americas, New York, NY 10013

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107150300

First published 2016

Printed in the United States of America

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging in Publication Data
Names: Harper, Robert, 1957–
Title: Practical foundations for programming languages / Robert Harper, Carnegie Mellon University.
Description: Second edition. | New York NY : Cambridge University Press, 2016. | Includes bibliographical references and index.
Identifiers: LCCN 2015045380 | ISBN 9781107150300 (alk. paper)
Subjects: LCSH: Programming languages (Electronic computers)
Classification: LCC QA76.7 .H377 2016 | DDC 005.13–dc23
LC record available at http://lccn.loc.gov/2015045380

ISBN 978-1-107-15030-0 Hardback

Contents

Preface to the Second Edition

Writing the second edition to a textbook incurs the same risk as building the second version of a software system. It is difficult to make substantive improvements, while avoiding the temptation to overburden and undermine the foundation on which one is building. With the hope of avoiding the second system effect, I have sought to make corrections, revisions, expansions, and deletions that improve the coherence of the development, remove some topics that distract from the main themes, add new topics that were omitted from the first edition, and include exercises for almost every chapter.

The revision removes a number of typographical errors, corrects a few material errors (especially the formulation of the parallel abstract machine and of concurrency in Algol), and improves the writing throughout. Some chapters have been deleted (general pattern matching and polarization, restricted forms of polymorphism), some have been completely rewritten (the chapter on higher kinds), some have been substantially revised (general and parametric inductive definitions, concurrent and distributed Algol), several have been reorganized (to better distinguish partial from total type theories), and a new chapter has been added (on type refinements). Titular attributions on several chapters have been removed, not to diminish credit, but to avoid confusion between the present and the original formulations of several topics. A new system of (pronounceable!) language names has been introduced throughout. The exercises generally seek to expand on the ideas in the main text, and their solutions often involve significant technical ideas that merit study. Routine exercises of the kind one might include in a homework assignment are deliberately few.

My purpose in writing this book is to establish a comprehensive framework for formulating and analyzing a broad range of ideas in programming languages. If language design and programming methodology are to advance from a trade-craft to a rigorous discipline, it is essential that we first get the definitions right. Then, and only then, can there be meaningful analysis and consolidation of ideas. My hope is that I have helped to build such a foundation.

I am grateful to Stephen Brookes, Evan Cavallo, Karl Crary, Jon Sterling, James R. Wilcox and Todd Wilson for their help in critiquing drafts of this edition and for their suggestions for modification and revision. I thank my department head, Frank Pfenning, for his support of my work on the completion of this edition. Thanks also to my editors, Ada Brunstein and Lauren Cowles, for their guidance and assistance. And thanks to Andrew Shulaev for corrections to the draft.

solution could result in injury to a person or loss of property. If you do use this material in such a manner, it is at your own risk. The author and publisher disclaim all liability for direct or consequential damage resulting from its use.

Pittsburgh
July 2015

Preface to the First Edition

Types are the central organizing principle of the theory of programming languages. Language features are manifestations of type structure. The syntax of a language is governed by the constructs that define its types, and its semantics is determined by the interactions among those constructs. The soundness of a language design—the absence of ill-defined programs—follows naturally.

The purpose of this book is to explain this remark. A variety of programming language features are analyzed in the unifying framework of type theory. A language feature is defined by its *statics*, the rules governing the use of the feature in a program, and its *dynamics*, the rules defining how programs using this feature are to be executed. The concept of *safety* emerges as the coherence of the statics and the dynamics of a language.

In this way, we establish a foundation for the study of programming languages. But why these particular methods? The main justification is provided by the book itself. The methods we use are both *precise* and *intuitive*, providing a uniform framework for explaining programming language concepts. Importantly, these methods *scale* to a wide range of programming language concepts, supporting rigorous analysis of their properties. Although it would require another book in itself to justify this assertion, these methods are also *practical* in that they are *directly applicable* to implementation and *uniquely effective* as a basis for mechanized reasoning. No other framework offers as much.

Being a consolidation and distillation of decades of research, this book does not provide an exhaustive account of the history of the ideas that inform it. Suffice it to say that much of the development is not original but rather is largely a reformulation of what has gone before. The notes at the end of each chapter signpost the major developments but are not intended as a complete guide to the literature. For further information and alternative perspectives, the reader is referred to such excellent sources as Constable (1986, 1998), Girard (1989), Martin-Löf (1984), Mitchell (1996), Pierce (2002, 2004), and Reynolds (1998).

The book is divided into parts that are, in the main, independent of one another. Parts I and II, however, provide the foundation for the rest of the book and must therefore be considered prior to all other parts. On first reading, it may be best to skim Part I, and begin in earnest with Part II, returning to Part I for clarification of the logical framework in which the rest of the book is cast.

Numerous people have read and commented on earlier editions of this book and have suggested corrections and improvements to it. I am particularly grateful to Umut Acar, Jesper Louis Andersen, Carlo Angiuli, Andrew Appel, Stephanie Balzer, Eric Bergstrom, Guy E. Blelloch, Iliano Cervesato, Lin Chase, Karl Crary, Rowan Davies, Derek Dreyer, Dan Licata, Zhong Shao, Rob Simmons, and Todd Wilson for their extensive efforts in

1 Abstract Syntax

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

The informal concept of syntax involves 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 can be used. At this level, phrases are *abstract binding trees*, which enrich abstract syntax trees with the concepts of binding and scope.

We will not concern ourselves in this book with concrete syntax but will instead consider pieces of syntax to be finite trees augmented with a means of expressing the binding and scope of identifiers within a syntax tree. To prepare the ground for the rest of the book, we define in this chapter what is a "piece of syntax" in two stages. First, we define abstract syntax trees, or ast's, which capture the hierarchical structure of a piece of syntax, while avoiding commitment to their concrete representation as a string. Second, we augment abstract syntax trees with the means of specifying the binding (declaration) and scope (range of significance) of an identifier. Such enriched forms of abstract syntax are called abstract binding trees, or abt's for short.

Several functions and relations on abt's are defined that give precise meaning to the informal ideas of binding and scope of identifiers. The concepts are infamously difficult to define properly and are the mother lode of bugs for language implementors. Consequently, precise definitions are essential, but they are also fairly technical and take some getting used to. It is probably best to skim this chapter on first reading to get the main ideas, and return to it for clarification as necessary.

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. Ast's 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 can be combined by an *operator*, which has an *arity* specifying the sort of the operator and 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 $s_1, \ldots, s_n$, respectively, into a compound ast of sort s.

The concept of a variable is central and therefore deserves special emphasis. A variable is an *unknown* object drawn from some domain. The unknown can become known by *substitution* of a particular object for all occurrences of a variable in a formula, thereby specializing a general formula to a particular instance. For example, in school algebra variables range over real numbers, and we may form polynomials, such as $x^2 + 2x + 1$, that can be specialized by substitution of, say, 7 for x to obtain $7^2 + (2 \times 7) + 1$, which can be simplified according to the laws of arithmetic to obtain 64, which is $(7 + 1)^2$.

Abstract syntax trees are classified by *sorts* that divide ast's into syntactic categories. For example, familiar programming languages often have a syntactic distinction between expressions and commands; these are two sorts of abstract syntax trees. Variables in abstract syntax trees range over sorts in the sense that only ast's of the specified sort of the variable can be plugged in for that variable. Thus, it would make no sense to replace an expression variable by a command, nor a command variable by an expression, the two being different sorts of things. But the core idea carries over from school mathematics, namely that *a variable is an unknown, or a place-holder, whose meaning is given by substitution.*

As an example, consider a language of arithmetic expressions built from numbers, addition, and multiplication. The abstract syntax of such a language consists of a single sort Exp generated by these operators:

1. An operator $\texttt{num}[n]$ of sort Exp for each $n \in \mathbb{N}$.
2. Two operators, `plus` and `times`, of sort Exp, each with two arguments of sort Exp.

The expression $2 + (3 \times x)$, which involves a variable, x, would be represented by the ast

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

of sort Exp, under the assumption that x is also of this sort. Because, say, $\texttt{num}[4]$, is an ast of sort Exp, we may plug it in for x in the above ast to obtain the ast

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

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 result.

The tree structure of ast's provides 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 can be generated and show that the property holds in each case under the assumption that it holds for its constituent ast's (if any). So, in the case of the sort Exp just described, we must show

1. The property holds for any variable x of sort Exp: prove that $\mathcal{P}(x)$.
2. The property holds for any number, $\texttt{num}[n]$: for every $n \in \mathbb{N}$, prove that $\mathcal{P}(\texttt{num}[n])$.

3. Assuming that the property holds for a_1 and a_2, prove that it holds for $\texttt{plus}(a_1; a_2)$ and $\texttt{times}(a_1; a_2)$: if $\mathcal{P}(a_1)$ and $\mathcal{P}(a_2)$, then $\mathcal{P}(\texttt{plus}(a_1; a_2))$ and $\mathcal{P}(\texttt{times}(a_1; a_2))$.

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.

It is common to apply the principle of structural induction in a form that takes account of the interpretation of variables as place-holders for ast's of the appropriate sort. Informally, it is often useful to prove a property of an ast involving variables in a form that is conditional on the property holding for the variables. Doing so anticipates that the variables will be replaced with ast's that ought to have the property assumed for them, so that the result of the replacement will have the property as well. This amounts to applying the principle of structural induction to properties $\mathcal{P}(a)$ of the form "if a involves variables $x_1, \ldots, x_k$, and $\mathcal{Q}$ holds of each x_i, then $\mathcal{Q}$ holds of a," so that a proof of $\mathcal{P}(a)$ for all ast's a by structural induction is just a proof that $\mathcal{Q}(a)$ holds for all ast's a under the assumption that $\mathcal{Q}$ holds for its variables. When there are no variables, there are no assumptions, and the proof of $\mathcal{P}$ is a proof that $\mathcal{Q}$ holds for all *closed* ast's. On the other hand, if x is a variable in a, and we replace it by an ast b for which $\mathcal{Q}$ holds, then $\mathcal{Q}$ will hold for the result of replacing x by b in a.

For the sake of precision, we now give precise definitions of these concepts. Let $\mathcal{S}$ be a finite set of sorts. For a given set $\mathcal{S}$ of sorts, an *arity* has the form $(s_1, \ldots, s_n)s$, which specifies the sort $s \in \mathcal{S}$ of an operator taking $n \geq 0$ arguments, each of sort $s_i \in \mathcal{S}$. Let $\mathcal{O} = \{\mathcal{O}_\alpha\}$ be an arity-indexed family of disjoint sets of *operators* $\mathcal{O}_\alpha$ of arity α. If o is an operator of arity $(s_1, \ldots, s_n)s$, we say that o has sort s and has n arguments of sorts $s_1, \ldots, s_n$.

Fix a set $\mathcal{S}$ of sorts and an arity-indexed family $\mathcal{O}$ of sets of operators of each arity. Let $\mathcal{X} = \{\mathcal{X}_s\}_{s \in \mathcal{S}}$ be a sort-indexed family of disjoint finite sets $\mathcal{X}_s$ of *variables* x of sort s. When $\mathcal{X}$ is clear from context, we say that a variable x is of sort s if $x \in \mathcal{X}_s$, and we say that x is *fresh for* $\mathcal{X}$, or just *fresh* when $\mathcal{X}$ is understood, if $x \notin \mathcal{X}_s$ for any sort s. If x is fresh for $\mathcal{X}$ and s is a sort, then $\mathcal{X}, x$ is the family of sets of variables obtained by adding x to $\mathcal{X}_s$. The notation is ambiguous in that the sort s is not explicitly stated but determined from context.

The family $\mathcal{A}[\mathcal{X}] = \{\mathcal{A}[\mathcal{X}]_s\}_{s \in \mathcal{S}}$ of *abstract syntax trees*, or *ast's*, of sort s is the smallest family satisfying the following conditions:

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 arity $(s_1, \ldots, s_n)s$, 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* can be used to prove that some property $\mathcal{P}$ holds of every ast. To show $\mathcal{P}(a)$ holds for every $a \in \mathcal{A}[\mathcal{X}]$, it is enough to show:

1. If $x \in \mathcal{X}_s$, then $\mathcal{P}_s(x)$.
2. If o has arity $(s_1, \ldots, s_n)s$ and $\mathcal{P}_{s_1}(a_1)$ 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 $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$ whenever $\mathcal{X} \subseteq \mathcal{Y}$.

Variables are given meaning by *substitution*. If $a \in \mathcal{A}[\mathcal{X}, x]_{s'}$, and $b \in \mathcal{A}[\mathcal{X}]_s$, then $[b/x]a \in \mathcal{A}[\mathcal{X}]_{s'}$ is 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 on 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. □

1.2 Abstract Binding Trees

Abstract binding trees, or *abt's*, enrich ast's with the means to introduce new variables and symbols, 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 can be used, either as a place-holder (in the case of a variable declaration) or as the index of some operator (in the case of a symbol declaration). Thus, the set of active identifiers can 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 $\texttt{let}\, x\, \texttt{be}\, a_1\, \texttt{in}\, a_2$, 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`. On the other hand, in the expression $\texttt{let}\, x\, \texttt{be}\, x * x\, \texttt{in}\, x + x$, occurrences of x within the multiplication refer to a different variable than those occurring within the addition. The

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 $\rho : \vec{x}_i \leftrightarrow \vec{z}_i$ we have $x \in \widehat{\rho}(a_i)$.

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, independently of the choice of bound variable names in that argument, then it is free in the overall abt.

The relation $a =_\alpha b$ of *α-equivalence* (so-called for historical reasons) means that a and b are identical up to the choice of bound variable names. The α-equivalence relation is 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 $1 \leq i \leq n$, $\widehat{\rho_i}(a_i) =_\alpha \widehat{\rho'_i}(a'_i)$ for all fresh renamings $\rho_i : \vec{x}_i \leftrightarrow \vec{z}_i$ and $\rho'_i : \vec{x}'_i \leftrightarrow \vec{z}_i$.

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 *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.

The definition of $[b/x]a$ is quite delicate and merits careful consideration.

One trouble spot for substitution is to notice that if x is bound by an abstractor within a, then x does not occur free within the abstractor and hence is unchanged by substitution. For example, $[b/x]\texttt{let}(a_1; x.a_2) = \texttt{let}([b/x]a_1; x.a_2)$, there being no free occurrences of x in $x.a_2$. Another trouble spot is the *capture* of a free variable of b during substitution. For example, if $y \in b$ and $x \neq y$, then $[b/x]\texttt{let}(a_1; y.a_2)$ is undefined, rather than being $\texttt{let}([b/x]a_1; y.[b/x]a_2)$, as one might at first suspect. For example, provided that $x \neq y$, $[y/x]\texttt{let}(\texttt{num}[0]; y.\texttt{plus}(x; y))$ is undefined, not $\texttt{let}(\texttt{num}[0]; y.\texttt{plus}(y; y))$, which confuses two different variables named y.

Although capture avoidance is an essential characteristic of substitution, it is, in a sense, merely a technical nuisance. If the names of bound variables have no significance, then capture can always be avoided by first renaming the bound variables in a to avoid any free variables in b. In the foregoing example, if we rename the bound variable y to y' to obtain $a' \triangleq \texttt{let}(\texttt{num}[0]; y'.\texttt{plus}(x; y'))$, then $[b/x]a'$ *is* defined and is equal to $\texttt{let}(\texttt{num}[0]; y'.\texttt{plus}(b; y'))$. The price for avoiding capture in this way is that substitution is only determined up to α-equivalence, and so we may no longer think of substitution as a function but only as a proper relation.

To restore the functional character of substitution, it is sufficient to adopt the *identification convention*, which is stated as follows:

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

That is, α-equivalent abt's are regarded as identical. Substitution can be extended to α-equivalence classes of abt's to avoid capture by choosing representatives of the equivalence classes of b and a in such a way that substitution is defined, then forming the equivalence class of the result. Any two choices of representatives for which substitution is defined gives α-equivalent results, so that substitution becomes a well-defined total function. *We will adopt the identification convention for abt's throughout this book.*

It will often be necessary to consider languages whose abstract syntax cannot be specified by a fixed set of operators but rather requires that the available operators be sensitive to the context in which they occur. For our purposes, it will suffice to consider a set of *symbolic parameters*, or *symbols*, that index families of operators so that as the set of symbols varies, so does the set of operators. An *indexed operator* o is a family of operators indexed by symbols u, so that $o[u]$ is an operator when u is an available symbol. If $\mathcal{U}$ is a finite set of symbols, then $\mathcal{B}[\mathcal{U};\mathcal{X}]$ is the sort-indexed family of abt's that are generated by operators and variables as before, admitting all indexed operator instances by symbols $u \in \mathcal{U}$. Whereas a variable is a place-holder that stands for an unknown abt of its sort, a symbol *does not stand for anything*, and is not, itself, an abt. The only significance of symbol is whether it is the same as or differs from another symbol; the operator instances $o[u]$ and $o[u']$ are the same exactly when u is u' and are the same symbol.

The set of symbols is extended by introducing a *new*, or *fresh*, symbol within a scope using the abstractor $u.a$, which binds the symbol u within the abt a. An abstracted symbol is "new" in the same sense as for an abstracted variable: the name of the bound symbol can be varied at will provided that no conflicts arise. This renaming property ensures that an abstracted symbol is distinct from all others in scope. The only difference between symbols and variables is that the only operation on symbols is renaming; there is no notion of substitution for a symbol.

Finally, a word about notation: to help improve the readability we often "group" and "stage" the arguments to an operator, using round brackets and braces to show grouping, and generally regarding stages to progress from right to left. All arguments in a group are considered to occur at the same stage, though their order is significant, and successive groups are considered to occur in sequential stages. Staging and grouping is often a helpful mnemonic device, but has no fundamental significance. For example, the abt $o\{a_1;a_2\}(a_3;x.a_4)$ is the same as the abt $o(a_1;a_2;a_3;x.a_4)$, as would be any other order-preserving grouping or staging of its arguments.

1.3 Notes

The concept of abstract syntax has its origins in the pioneering work of Church, Turing, and Gödel, who first considered 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 Kleene (1952), has 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).

The concept of abstract binding trees presented here was inspired by the system of notation developed in the NuPRL Project, which is described in Constable (1986) and from Martin-Löf's system of arities, which is described in Nordstrom et al. (1990). Their enrichment with symbol binders is influenced by Pitts and Stark (1993).

Exercises

1.1. Prove by structural induction on abstract syntax trees that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.

1.2. Prove by structural induction modulo renaming on abstract binding trees that if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{B}[\mathcal{X}] \subseteq \mathcal{B}[\mathcal{Y}]$.

1.3. Show that if $a =_\alpha a'$ and $b =_\alpha b'$ and both $[b/x]a$ and $[b'/x]a'$ are defined, then $[b/x]a =_\alpha [b'/x]a'$.

1.4. Bound variables can be seen as the formal analogs of pronouns in natural languages. The binding occurrence of a variable at an abstractor fixes a "fresh" pronoun for use within its body that refers unambiguously to that variable (in contrast to English, in which the referent of a pronoun can often be ambiguous). This observation suggests an alternative representation of abt's, called *abstract binding graphs*, or *abg's* for short, as directed graphs constructed as follows:

(a) Free variables are atomic nodes with no outgoing edges.

(b) Operators with n arguments are n-ary nodes, with one outgoing edge directed at each of their children.

(c) Abstractors are nodes with one edge directed to the scope of the abstracted variable.

(d) Bound variables are back edges directed at the abstractor that introduced it.

Notice that ast's, thought of as abt's with no abstractors, are *acyclic* directed graphs (more precisely, variadic trees), whereas general abt's can be *cyclic*. Draw a few examples of abg's corresponding to the example abt's given in this chapter. Give a precise definition of the sort-indexed family $\mathcal{G}[\mathcal{X}]$ of abstract binding graphs. What representation would you use for bound variables (back edges)?

2 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. 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 abstract binding trees of some 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 an abstract binding tree. We shall make use of many forms of judgment, including examples such as these:

n nat	n is a natural number
$n_1 + n_2 = n$	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 abstract binding trees have a property or stand in some relation to one another. The property or relation itself is called a *judgment form*, and the judgment that an object or objects have that property or stand 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 J or J a, for the judgment asserting that J holds of the abt a. Correspondingly, we sometimes notate the judgment form J by $-$ J, or J $-$, using a dash to indicate the absence of an argument to J. When it is not important to stress the subject of the judgment, we write J to stand for an unspecified judgment, that is, an instance of some judgment form. For particular judgment forms, we freely use prefix, infix, or mix-fix notation, as illustrated by the above 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 can 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 form an inductive definition of the judgment form $-$ 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 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 form $-$ 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 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 form a is b expresses the equality of two abt's a and b such that a nat and b nat is inductively defined by the following rules:

$$\frac{}{\texttt{zero is zero}} \tag{2.4a}$$

$$\frac{a\ \texttt{is}\ b}{\texttt{succ}(a)\ \texttt{is succ}(b)} \tag{2.4b}$$

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 will 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.

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 $\mathcal{P}(a)$ whenever a $\mathtt{nat}$, it is enough to show:

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

The sufficiency of these conditions is the familiar principle of *mathematical induction*.

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

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

The sufficiency of these conditions is called the principle of *tree 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 $\mathcal{P}(a)$ 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. □

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 is a.

Proof By rule induction on rules (2.2):

Rule (2.2a) Applying rule (2.4a) we obtain $\mathtt{zero}$ is $\mathtt{zero}$.

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

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

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

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 form $-$ list, which states 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 can 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:

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

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

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

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

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

As an example, we may use simultaneous rule induction to prove that (1) if a even, then either a is zero or a is succ(b) with b odd, and (2) if a odd, then a is succ(b) with b even. We define $\mathcal{P}(a)$ to hold iff a is zero or a is succ(b) for some b with b odd, and define $\mathcal{Q}(b)$ to hold iff b is succ(a) for some a with a even. The desired result follows by rule induction, because we can prove the following facts:

1. $\mathcal{P}(\mathtt{zero})$, which holds because zero is zero.
2. If $\mathcal{Q}(b)$, then succ(b) is succ(b') for some b' with $\mathcal{Q}(b')$. Take b' to be b and apply the inductive assumption.
3. If $\mathcal{P}(a)$, then succ(a) is succ(a') for some a' with $\mathcal{P}(a')$. Take a' to be a and apply the inductive assumption.

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 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 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* 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 sum(a;b;c).
2. (Uniqueness) If sum(a;b;c), and sum(a;b;c'), then c is c'.

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

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

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

For uniqueness, we prove that *if* sum$(a;b;c_1)$*, then if* sum$(a;b;c_2)$*, then* c_1 is c_2 by rule induction based on rules (2.9).

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

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

2.7 Notes

Aczel (1977) provides a thorough account of the theory of inductive definitions on which the present account is based. A significant difference is that we consider inductive definitions of judgments over abt's as defined in Chapter 1, rather than with natural numbers. The emphasis on judgments is inspired by Martin-Löf's logic of judgments (Martin-Löf, 1983, 1987).

Exercises

2.1. Give an inductive definition of the judgment max$(m;n;p)$, where m nat, n nat, and p nat, with the meaning that p is the larger of m and n. Prove that every m and n are related to a unique p by this judgment.

2.2. Consider the following rules, which define the judgment hgt$(t;n)$ stating that the binary tree t has *height* n.

$$\frac{}{\texttt{hgt}(\texttt{empty};\texttt{zero})} \tag{2.10a}$$

$$\frac{\texttt{hgt}(t_1;n_1) \quad \texttt{hgt}(t_2;n_2) \quad \texttt{max}(n_1;n_2;n)}{\texttt{hgt}(\texttt{node}(t_1;t_2);\texttt{succ}(n))} \tag{2.10b}$$

Prove that the judgment hgt defines a function from trees to natural numbers.

2.3. Given an inductive definition of *ordered variadic trees* whose nodes have a finite, but variable, number of children with a specified left-to-right ordering among them. Your solution should consist of a simultaneous definition of two judgments, t tree, stating that t is a variadic tree, and f forest, stating that f is a "forest" (finite sequence) of variadic trees.

2.4. Give an inductive definition of the height of a variadic tree of the kind defined in Exercise **2.3**. Your definition should make use of an auxiliary judgment defining the height of a forest of variadic trees and will be defined simultaneously with the height of a variadic tree. Show that the two judgments so defined each define a function.

2.5. Give an inductive definition of the *binary natural numbers*, which are either zero, twice a binary number, or one more than twice a binary number. The size of such a representation is logarithmic, rather than linear, in the natural number it represents.

2.6. Give an inductive definition of addition of binary natural numbers as defined in Exercise **2.5**. *Hint*: Proceed by analyzing both arguments to the addition, and make use of an auxiliary function to compute the successor of a binary number. *Hint*: Alternatively, define both the sum and the sum-plus-one of two binary numbers mutually recursively.

3.1.2 Admissibility

Admissibility, written $\Gamma \models_{\mathcal{R}} J$, 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 $\mathcal{R}$ when 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 $\mathcal{R}$. Given any derivations of $J_1, \ldots, J_n$ using the rules in $\mathcal{R}$, we may build a derivation of J using the rules in $\mathcal{R}$.

For example, the admissibility judgment

$$\mathtt{succ}(a)\ \mathsf{even} \models_{(2.8)} a\ \mathsf{odd} \tag{3.6}$$

is valid, because any derivation of $\mathtt{succ}(a)$ even from rules (2.2) must contain a subderivation of a odd from the same rules, which justifies the conclusion. This fact can be proved by induction on rules (2.8). That judgment (3.6) is valid may also be expressed by saying that the rule

$$\frac{\mathtt{succ}(a)\ \mathsf{even}}{a\ \mathsf{odd}} \tag{3.7}$$

is *admissible* relative to rules (2.8).

In contrast to derivability the admissibility judgment is *not* stable under extension to the rules. For example, if we enrich rules (2.8) with the axiom

$$\frac{}{\mathtt{succ(zero)}\ \mathsf{even}}\ , \tag{3.8}$$

then rule (3.6) is *inadmissible*, because there is no composition of rules deriving $\mathtt{zero}$ odd. 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, note that

$$\mathtt{succ(zero)}\ \mathsf{even} \not\vdash_{(2.8)} \mathtt{zero}\ \mathsf{odd},$$

because there is no derivation of the right-hand side when the left-hand side is added as an axiom to rules (2.8). Yet the corresponding admissibility judgment

$$\mathtt{succ(zero)}\ \mathsf{even} \models_{(2.8)} \mathtt{zero}\ \mathsf{odd}$$

is valid, because the hypothesis is false: there is no derivation of `succ(zero)` even from rules (2.8). Even so, the derivability

$$\texttt{succ(zero)}\ \texttt{even} \vdash_{(2.8)} \texttt{succ(succ(zero))}\ \texttt{odd}$$

is valid, because we may derive the right-hand side from the left-hand side by composing rules (2.8).

Evidence for admissibility can 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 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 $\mathcal{R}, r$ can be transformed into a derivation from $\mathcal{R}$ alone. Consequently, if we wish to prove a property of the judgments derivable from $\mathcal{R}, r$, when r is admissible with respect to $\mathcal{R}$, it suffices show that the property is closed under rules $\mathcal{R}$ alone, because its admissibility states that the consequences of rule r are implicit in those of rules $\mathcal{R}$.

3.2 Hypothetical Inductive Definitions

It is useful to enrich the concept of an inductive definition to allow rules with derivability judgments as premises and conclusions. Doing so lets us 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 set 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 when each J_i is a derivable consequence of Γ, augmented with the 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.

We require that all rules in a hypothetical inductive definition be *uniform* in the sense that they are applicable in *all* global contexts. Uniformity ensures that a rule can be presented in *implicit*, or *local form*,

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

in which the global context has been suppressed with the understanding that the rule applies for any choice of global hypotheses.

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 set of hypothetical rules $\mathcal{R}$ defines the strongest formal derivability judgment that is *structural* and *closed* under uniform 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. We write $\Gamma \vdash_{\mathcal{R}} J$ to mean 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 $\mathcal{P}(\Gamma \vdash J)$ when $\Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under the rules of $\mathcal{R}$ and under the structural rules.[1] Thus, for each rule of the form (3.9), whether structural or in $\mathcal{R}$, we must show that

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

But 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 Section 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 rules of a hypothetical inductive definition are uniform, the structural rules (3.11b) and (3.11c) are clearly admissible. Usually, rule (3.11a) must be postulated explicitly as a rule, rather than 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 set of objects. A *generic* judgment states that a judgment holds for any choice of objects replacing designated variables in the judgment. Another form of general judgment codifies the handling of symbolic parameters. A *parametric* judgment expresses generality over any choice of fresh renamings of designated symbols of a judgment. To keep track of the active variables and symbols in a derivation, we write $\Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J$ to say that J is derivable from Γ according to rules $\mathcal{R}$, with objects consisting of abt's over symbols $\mathcal{U}$ and variables $\mathcal{X}$.

The concept of uniformity of a rule must be extended to require that rules be *closed under renaming and substitution* for variables and *closed under renaming* for parameters. More precisely, if $\mathcal{R}$ is a set of rules containing a free variable x of sort s, then it must also contain all possible substitution instances of abt's a of sort s for x, including those that contain other free variables. Similarly, if $\mathcal{R}$ contains rules with a parameter u, then it must contain all instances of that rule obtained by renaming u of a sort to any u' of the same sort. Uniformity rules out stating a rule for a variable, without also stating it for all instances of that variable. It also rules out stating a rule for a parameter without stating it for all possible renamings of that parameter.

Generic derivability judgment is defined by

$$\mathcal{Y} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \Gamma \vdash^{\mathcal{X}\,\mathcal{Y}}_{\mathcal{R}} J,$$

where $\mathcal{Y} \cap \mathcal{X} = \emptyset$. Evidence for generic derivability consists of a *generic derivation* ∇ involving the variables $\mathcal{X}\,\mathcal{Y}$. So long as the rules are uniform, the choice of $\mathcal{Y}$ does not matter, in a sense to be explained shortly.

For example, the generic derivation ∇,

$$\dfrac{\dfrac{\dfrac{}{x\ \mathsf{nat}}}{\mathsf{succ}(x)\ \mathsf{nat}}}{\mathsf{succ}(\mathsf{succ}(x))\ \mathsf{nat}},$$

is evidence for the judgment

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

provided $x \notin \mathcal{X}$. Any other choice of x would work just as well, as long as all rules are uniform.

The generic derivability judgment enjoys the following *structural properties* governing the behavior of variables, provided that $\mathcal{R}$ is uniform.

Proliferation If $\mathcal{Y} \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$, then $\mathcal{Y}, y \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$.
Renaming If $\mathcal{Y}, y \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$, then $\mathcal{Y}, y' \mid [y \leftrightarrow y']\Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [y \leftrightarrow y']J$ for any $y' \notin \mathcal{X}\,\mathcal{Y}$.
Substitution If $\mathcal{Y}, y \mid \Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} J$ and $a \in \mathcal{B}[\mathcal{X}\,\mathcal{Y}]$, then $\mathcal{Y} \mid [a/y]\Gamma \vdash^{\mathcal{X}}_{\mathcal{R}} [a/y]J$.

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. It is left implicit in the principle of substitution that the substituting abt is of the same sort as the substituted variable.

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

$$\mathcal{V} \parallel \mathcal{Y} \mid \Gamma \vdash^{\mathcal{U};\mathcal{X}}_{\mathcal{R}} J \quad \text{iff} \quad \mathcal{Y} \mid \Gamma \vdash^{\mathcal{U}\,\mathcal{V};\mathcal{X}}_{\mathcal{R}} J,$$

where $\mathcal{V} \cap \mathcal{U} = \emptyset$. Evidence for parametric derivability consists of a derivation ∇ involving the symbols $\mathcal{V}$. Uniformity of $\mathcal{R}$ ensures that any choice of parameter names is as good as any other; derivability is stable under renaming.

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

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

The variables $\mathcal{Y}$ are the *global variables* of the inference, and, for each $1 \leq i \leq n$, the variables $\mathcal{Y}_i$ 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 can be given in *implicit form*,

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

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

When specialized to a set of generic rules, the principle of rule induction states that to show $\mathcal{P}(\mathcal{Y} \mid \Gamma \vdash J)$ when $\mathcal{Y} \mid \Gamma \vdash_{\mathcal{R}} J$, it is enough to show that $\mathcal{P}$ is closed under the rules

3.4. Show that if $x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}} a\ \mathsf{comb}$, then there is a combinator a', written $[x]\,a$ and called *bracket abstraction*, such that

$$x \mid x\ \mathsf{comb} \vdash_{\mathcal{C}\cup\mathcal{E}} a'\,x \equiv a.$$

Consequently, by Exercise **3.2**, if $a''\ \mathsf{comb}$, then

$$([x]\,a)\,a'' \equiv [a''/x]a.$$

Hint: Inductively define the judgment

$$x \mid x\ \mathsf{comb} \vdash \mathsf{abs}_x\ a\ \mathsf{is}\ a',$$

where $x \mid x\ \mathsf{comb} \vdash a\ \mathsf{comb}$. Then argue that it defines a' as a binary function of x and a. The motivation for the conversion axioms governing `k` and `s` should become clear while developing the proof of the desired equivalence.

3.5. Prove that bracket abstraction, as defined in Exercise **3.4**, is *non-compositional* by exhibiting a and b such that $a\ \mathsf{comb}$ and

$$x\ y \mid x\ \mathsf{comb}\ y\ \mathsf{comb} \vdash_{\mathcal{C}} b\ \mathsf{comb}$$

such that $[a/y]([x]\,b) \neq [x]\,([a/y]b)$. *Hint*: Consider the case that b is y.

Suggest a modification to the definition of bracket abstraction that is *compositional* by showing under the same conditions given above that

$$[a/y]([x]\,b) = [x]\,([a/y]b).$$

3.6. Consider the set $\mathcal{B}[\mathcal{X}]$ of abt's generated by the operators `ap`, with arity (Exp, Exp)Exp, and λ, with arity (Exp.Exp)Exp, and possibly involving variables in $\mathcal{X}$, all of which are of sort Exp. Give an inductive definition of the judgment b closed, which specifies that b has no free occurrences of the variables in $\mathcal{X}$. *Hint*: it is essential to give an inductive definition of the hypothetical, general judgment

$$x_1, \ldots, x_n \mid x_1\ \mathsf{closed}, \ldots, x_n\ \mathsf{closed} \vdash b\ \mathsf{closed}$$

in order to account for the binding of a variable by the λ operator. The hypothesis that a variable is closed seems self-contradictory in that a variable obviously occurs free in itself. Explain why this is not the case by examining carefully the meaning of the hypothetical and general judgments.

Notes

1 Writing $\mathcal{P}(\Gamma \vdash J)$ is a mild abuse of notation in which the turnstile is used to separate the two arguments to $\mathcal{P}$ for the sake of readability.

2 The combinator $\mathtt{ap}(a_1;a_2)$ is written $a_1\ a_2$ for short, left-associatively when used in succession.

PART II

Statics and Dynamics

4 Statics

Most programming languages exhibit a *phase distinction* between the *static* and *dynamic* phases of processing. The static phase consists of parsing and type checking to ensure that the program is well-formed; the dynamic phase consists of execution of well-formed programs. A language is said to be *safe* exactly when well-formed programs are well-behaved when executed.

The static phase is specified by a *statics* comprising a set of rules for deriving *typing judgments* stating that an expression is well-formed of a certain type. Types mediate the interaction between the constituent parts of a program by "predicting" some aspects of the execution behavior of the parts so that we may ensure they fit together properly at run-time. Type safety tells us that these predictions are correct; if not, the statics is considered to be improperly defined, and the language is deemed *unsafe* for execution.

In this chapter, we present the statics of a simple expression language, **E**, as an illustration of the method that we will employ throughout this book.

4.1 Syntax

When defining a language we shall be primarily concerned with its abstract syntax, specified by a collection of operators and their arities. The abstract syntax provides a systematic, unambiguous account of the hierarchical and binding structure of the language and is considered the official presentation of the language. However, for the sake of clarity, it is also useful to specify minimal concrete syntax conventions, without going through the trouble to set up a fully precise grammar for it.

We will accomplish both of these purposes with a *syntax chart*, whose meaning is best illustrated by example. The following chart summarizes the abstract and concrete syntax of **E**.

Typ	τ	::=	`num`	`num`	numbers
			`str`	`str`	strings
Exp	e	::=	x	x	variable
			`num`$[n]$	n	numeral
			`str`$[s]$	"s"	literal
			`plus`$(e_1; e_2)$	$e_1 + e_2$	addition
			`times`$(e_1; e_2)$	$e_1 * e_2$	multiplication
			`cat`$(e_1; e_2)$	e_1 ^ e_2	concatenation
			`len`(e)	$\lvert e \rvert$	length
			`let`$(e_1; x.e_2)$	`let` x `be` e_1 `in` e_2	definition

This chart defines two sorts, Typ, ranged over by τ, and Exp, ranged over by e. The chart defines a set of operators and their arities. For example, it specifies that the operator `let` has arity (Exp, Exp.Exp)Exp, which specifies that it has two arguments of sort Exp, and binds a variable of sort Exp in the second argument.

4.2 Type System

The role of a type system is to impose constraints on the formations of phrases that are sensitive to the context in which they occur. For example, whether the expression $\texttt{plus}(x;\texttt{num}[n])$ is sensible depends on whether the variable x is restricted to have type `num` in the surrounding context of the expression. This example is, in fact, illustrative of the general case, in that the *only* information required about the context of an expression is the type of the variables within whose scope the expression lies. Consequently, the statics of **E** consists of an inductive definition of generic hypothetical judgments of the form

$$\vec{x} \mid \Gamma \vdash e : \tau,$$

where $\vec{x}$ is a finite set of variables, and Γ is a *typing context* consisting of hypotheses of the form $x : \tau$, one for each $x \in \vec{x}$. We rely on typographical conventions to determine the set of variables, using the letters x and y to stand for them. We write $x \notin dom(\Gamma)$ to say that there is no assumption in Γ of the form $x : \tau$ for any type τ, in which case we say that the variable x is *fresh* for Γ.

The rules defining the statics of **E** are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{4.1a}$$

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{4.1b}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{4.1c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}} \tag{4.1d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1; e_2) : \texttt{num}} \tag{4.1e}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1; e_2) : \texttt{str}} \tag{4.1f}$$

$$\frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{len}(e) : \texttt{num}} \tag{4.1g}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) : \tau_2} \tag{4.1h}$$

In rule (4.1h), we tacitly assume that the variable x is not already declared in Γ. This condition may always be met by choosing a suitable representative of the α-equivalence class of the `let` expression.

the type `num` are addition and multiplication, and those for the type `str` are concatenation and length.

The importance of this classification will become clear once we have defined the dynamics of the language in Chapter 5. Then we will see that the elimination forms are *inverse* to the introduction forms in that they "take apart" what the introduction forms have "put together." The coherence of the statics and dynamics of a language expresses the concept of *type safety*, the subject of Chapter 6.

4.4 Notes

The concept of the static semantics of a programming language was historically slow to develop, perhaps because the earliest languages had relatively few features and only very weak type systems. The concept of a static semantics in the sense considered here was introduced in the definition of the Standard ML programming language (Milner et al., 1997), building on earlier work by Church and others on the typed λ-calculus (Barendregt, 1992). The concept of introduction and elimination, and the associated inversion principle, was introduced by Gentzen in his pioneering work on natural deduction (Gentzen, 1969). These principles were applied to the structure of programming languages by Martin-Löf (1984, 1980).

Exercises

4.1. It is sometimes useful to give the typing judgment $\Gamma \vdash e : \tau$ an "operational" reading that specifies more precisely the flow of information required to derive a typing judgment (or determine that it is not derivable). The *analytic* mode corresponds to the context, expression, and type being given, with the goal to determine whether the typing judgment is derivable. The *synthetic* mode corresponds to the context and expression being given, with the goal to find the unique type τ, if any, possessed by the expression in that context. These two readings can be made explicit as judgments of the form $e \downarrow \tau$, corresponding to the analytic mode, and $e \uparrow \tau$, corresponding to the synthetic mode.

Give a simultaneous inductive definition of these two judgments according to the following guidelines:

(a) Variables are introduced in synthetic form.

(b) If we can synthesize a unique type for an expression, then we can analyze it with respect to a given type by checking type equality.

(c) Definitions need care, because the type of the defined expression is not given, even when the type of the result is given.

There is room for variation; the point of the exercise is to explore the possibilities.

4.2. One way to limit the range of possibilities in the solution to Exercise **4.1** is to restrict and extend the syntax of the language so that every expression is either synthetic or analytic according to the following suggestions:

(a) Variables are analytic.

(b) Introduction forms are analytic, elimination forms are synthetic.

(c) An analytic expression can be made synthetic by introducing a *type cast* of the form $\texttt{cast}\{\tau\}(e)$ specifying that e must check against the specified type τ, which is synthesized for the whole expression.

(d) The defining expression of a definition must be synthetic, but the scope of the definition can be either synthetic or analytic.

Reformulate your solution to Exercise **4.1** to take account of these guidelines.

Note

1 This point may seem so obvious that it is not worthy of mention, but, surprisingly, there are useful type systems that lack this property. Because they do not validate the structural principle of weakening, they are called *substructural* type systems.

5 Dynamics

The *dynamics* of a language describes how programs are executed. The most important way to define the dynamics of a language is by the method of *structural dynamics*, which defines a *transition system* that inductively specifies the step-by-step process of executing a program. Another method for presenting dynamics, called *contextual dynamics*, is a variation of structural dynamics in which the transition rules are specified in a slightly different way. An *equational dynamics* presents the dynamics of a language by a collection of rules defining when one program is *definitionally equivalent* to another.

5.1 Transition Systems

A *transition system* is specified by the following four forms of judgment:

1. s state, asserting that s is a *state* of the transition system.
2. s final, where s state, asserting that s is a *final* state.
3. s initial, where s state, asserting that s is an *initial* state.
4. $s \longmapsto s'$, where s state and s' state, asserting that state s may transition to state s'.

In practice, we always arrange things so that no transition is possible from a final state: if s final, then there is no s' state such that $s \longmapsto s'$. A state from which no transition is possible is *stuck*. Whereas all final states are, by convention, stuck, there may be stuck states in a transition system that are not final. A transition system is *deterministic* iff for every state s there exists at most one state s' such that $s \longmapsto s'$; otherwise, it is *non-deterministic*.

A *transition sequence* is a sequence of states $s_0, \ldots, s_n$ such that s_0 initial, and $s_i \longmapsto s_{i+1}$ for every $0 \leq i < n$. A transition sequence is *maximal* iff there is no s such that $s_n \longmapsto s$, and it is *complete* iff it is maximal and s_n final. Thus, every complete transition sequence is maximal, but maximal sequences are not necessarily complete. The judgment $s \downarrow$ means that there is a complete transition sequence starting from s, which is to say that there exists s' final such that $s \longmapsto^* s'$.

The *iteration* of transition judgment $s \longmapsto^* s'$ is inductively defined by the following rules:

$$\frac{}{s \longmapsto^* s} \tag{5.1a}$$

$$\frac{s \longmapsto s' \quad s' \longmapsto^* s''}{s \longmapsto^* s''} \tag{5.1b}$$

When applied to the definition of iterated transition, the principle of rule induction states that to show that $P(s, s')$ holds when $s \longmapsto^* s'$, it is enough to show these two properties of P:

1. $P(s, s)$.
2. if $s \longmapsto s'$ and $P(s', s'')$, then $P(s, s'')$.

The first requirement is to show that P is reflexive. The second is to show that P is *closed under head expansion*, or *closed under inverse evaluation*. Using this principle, it is easy to prove that $\longmapsto^*$ is reflexive and transitive.

The *n-times iterated* transition judgment $s \longmapsto^n s'$, where $n \geq 0$, is inductively defined by the following rules:

$$\frac{}{s \longmapsto^0 s} \tag{5.2a}$$

$$\frac{s \longmapsto s' \quad s' \longmapsto^n s''}{s \longmapsto^{n+1} s''} \tag{5.2b}$$

Theorem 5.1. *For all states s and s', $s \longmapsto^* s'$ iff $s \longmapsto^k s'$ for some $k \geq 0$.*

Proof From left to right, by induction on the definition of multi-step transition. From right to left, by mathematical induction on $k \geq 0$. □

5.2 Structural Dynamics

A *structural dynamics* for the language **E** is given by a transition system whose states are closed expressions. All states are initial. The final states are the *(closed) values*, which represent the completed computations. The judgment e val, which states that e is a value, is inductively defined by the following rules:

$$\frac{}{\texttt{num}[n]\ \mathsf{val}} \tag{5.3a}$$

$$\frac{}{\texttt{str}[s]\ \mathsf{val}} \tag{5.3b}$$

The transition judgment $e \longmapsto e'$ between states is inductively defined by the following rules:

$$\frac{n_1 + n_2 = n}{\texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \longmapsto \texttt{num}[n]} \tag{5.4a}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{plus}(e_1; e_2) \longmapsto \texttt{plus}(e_1'; e_2)} \tag{5.4b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\texttt{plus}(e_1; e_2) \longmapsto \texttt{plus}(e_1; e_2')} \tag{5.4c}$$

$$\frac{s_1 \hat{} s_2 = s \ \mathsf{str}}{\mathtt{cat}(\mathtt{str}[s_1];\mathtt{str}[s_2]) \longmapsto \mathtt{str}[s]} \tag{5.4d}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{cat}(e_1;e_2) \longmapsto \mathtt{cat}(e_1';e_2)} \tag{5.4e}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{cat}(e_1;e_2) \longmapsto \mathtt{cat}(e_1;e_2')} \tag{5.4f}$$

$$\left[\frac{e_1 \longmapsto e_1'}{\mathtt{let}(e_1;x.e_2) \longmapsto \mathtt{let}(e_1';x.e_2)}\right] \tag{5.4g}$$

$$\frac{[e_1 \ \mathsf{val}]}{\mathtt{let}(e_1;x.e_2) \longmapsto [e_1/x]e_2} \tag{5.4h}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (5.4a), (5.4d), and (5.4h) are *instruction transitions*, because they correspond to the primitive steps of evaluation. The remaining rules are *search transitions* that determine the order of execution of instructions.

The bracketed rule (5.4g) and bracketed premise on rule (5.4h) are included for a *by-value* interpretation of `let` and omitted for a *by-name* interpretation. The by-value interpretation evaluates an expression before binding it to the defined variable, whereas the by-name interpretation binds it in unevaluated form. The by-value interpretation saves work if the defined variable is used more than once but wastes work if it is not used at all. Conversely, the by-name interpretation saves work if the defined variable is not used and wastes work if it is used more than once.

A derivation sequence in a structural dynamics has a two-dimensional structure, with the number of steps in the sequence being its "width" and the derivation tree for each step being its "height." For example, consider the following evaluation sequence:

$$\begin{aligned}
&\mathtt{let}(\mathtt{plus}(\mathtt{num}[1];\mathtt{num}[2]);x.\mathtt{plus}(\mathtt{plus}(x;\mathtt{num}[3]);\mathtt{num}[4])) \\
&\quad\longmapsto\quad \mathtt{let}(\mathtt{num}[3];x.\mathtt{plus}(\mathtt{plus}(x;\mathtt{num}[3]);\mathtt{num}[4])) \\
&\quad\longmapsto\quad \mathtt{plus}(\mathtt{plus}(\mathtt{num}[3];\mathtt{num}[3]);\mathtt{num}[4]) \\
&\quad\longmapsto\quad \mathtt{plus}(\mathtt{num}[6];\mathtt{num}[4]) \\
&\quad\longmapsto\quad \mathtt{num}[10]
\end{aligned}$$

Each step in this sequence of transitions is justified by a derivation according to rules (5.4). For example, the third transition in the preceding example is justified by the following derivation:

$$\frac{\dfrac{}{\mathtt{plus}(\mathtt{num}[3];\mathtt{num}[3]) \longmapsto \mathtt{num}[6]}\ (5.4a)}{\mathtt{plus}(\mathtt{plus}(\mathtt{num}[3];\mathtt{num}[3]);\mathtt{num}[4]) \longmapsto \mathtt{plus}(\mathtt{num}[6];\mathtt{num}[4])}\ (5.4b)$$

The other steps are similarly justified by composing rules.

The principle of rule induction for the structural dynamics of **E** states that to show $\mathcal{P}(e \longmapsto e')$ when $e \longmapsto e'$, it is enough to show that $\mathcal{P}$ is closed under rules (5.4). For example, we may show by rule induction that the structural dynamics of **E** is *determinate*,


image
not
available

Proof From left to right, proceed by rule induction on rules (5.4). It is enough in each case to exhibit an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \to e_0'$. For example, for rule (5.4a), take $\mathcal{E} = \circ$, and note that $e \to e'$. For rule (5.4b), we have by induction that there exists an evaluation context $\mathcal{E}_1$ such that $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_0 \to e_0'$. Take $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1; e_2)$, and note that $e = \mathtt{plus}(\mathcal{E}_1; e_2)\{e_0\}$ and $e' = \mathtt{plus}(\mathcal{E}_1; e_2)\{e_0'\}$ with $e_0 \to e_0'$.

From right to left, note that if $e \longmapsto_{\mathsf{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \to e_0'$. We prove by induction on rules (5.7) that $e \longmapsto_{\mathsf{s}} e'$. For example, for rule (5.7a), e_0 is e, e_0' is e', and $e \to e'$. Hence, $e \longmapsto_{\mathsf{s}} e'$. For rule (5.7b), we have that $\mathcal{E} = \mathtt{plus}(\mathcal{E}_1; e_2)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_1 \longmapsto_{\mathsf{s}} e_1'$. Therefore, e is $\mathtt{plus}(e_1; e_2)$, e' is $\mathtt{plus}(e_1'; e_2)$, and therefore by rule (5.4b), $e \longmapsto_{\mathsf{s}} e'$. □

Because the two transition judgments coincide, contextual dynamics can be considered an alternative presentation of a structural dynamics. It has two advantages over structural dynamics, one relatively superficial, one rather less so. The superficial advantage stems from writing rule (5.8) in the simpler form

$$\frac{e_0 \to e_0'}{\mathcal{E}\{e_0\} \longmapsto \mathcal{E}\{e_0'\}}\ . \tag{5.9}$$

This formulation is superficially simpler in that it does not make explicit how an expression is decomposed into an evaluation context and a reducible expression. The deeper advantage of contextual dynamics is that all transitions are between complete programs. One need never consider a transition between expressions of any type other than the observable type, which simplifies certain arguments, such as the proof of Lemma 47.16.

5.4 Equational Dynamics

Another formulation of the dynamics of a language regards computation as a form of equational deduction, much in the style of elementary algebra. For example, in algebra, we may show that the polynomials $x^2 + 2x + 1$ and $(x + 1)^2$ are equivalent by a simple process of calculation and re-organization using the familiar laws of addition and multiplication. The same laws are enough to determine the value of any polynomial, given the values of its variables. So, for example, we may plug in 2 for x in the polynomial $x^2 + 2x + 1$ and calculate that $2^2 + 2 \times 2 + 1 = 9$, which is indeed $(2 + 1)^2$. We thus obtain a model of computation in which the value of a polynomial for a given value of its variable is determined by substitution and simplification.

Very similar ideas give rise to the concept of *definitional*, or *computational*, *equivalence* of expressions in **E**, which we write as $\mathcal{X} \mid \Gamma \vdash e \equiv e' : \tau$, where Γ consists of one assumption of the form $x : \tau$ for each $x \in \mathcal{X}$. We only consider definitional equality of well-typed expressions, so that when considering the judgment $\Gamma \vdash e \equiv e' : \tau$, we tacitly

assume that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$. Here, as usual, we omit explicit mention of the variables $\mathcal{X}$ when they can be determined from the forms of the assumptions Γ.

Definitional equality of expressions in **E** under the by-name interpretation of `let` is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash e \equiv e : \tau} \tag{5.10a}$$

$$\frac{\Gamma \vdash e' \equiv e : \tau}{\Gamma \vdash e \equiv e' : \tau} \tag{5.10b}$$

$$\frac{\Gamma \vdash e \equiv e' : \tau \quad \Gamma \vdash e' \equiv e'' : \tau}{\Gamma \vdash e \equiv e'' : \tau} \tag{5.10c}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \mathtt{num} \quad \Gamma \vdash e_2 \equiv e_2' : \mathtt{num}}{\Gamma \vdash \mathtt{plus}(e_1; e_2) \equiv \mathtt{plus}(e_1'; e_2') : \mathtt{num}} \tag{5.10d}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \mathtt{str} \quad \Gamma \vdash e_2 \equiv e_2' : \mathtt{str}}{\Gamma \vdash \mathtt{cat}(e_1; e_2) \equiv \mathtt{cat}(e_1'; e_2') : \mathtt{str}} \tag{5.10e}$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv \mathtt{let}(e_1'; x.e_2') : \tau_2} \tag{5.10f}$$

$$\frac{n_1 + n_2 \text{ is } n \text{ nat}}{\Gamma \vdash \mathtt{plus}(\mathtt{num}[n_1]; \mathtt{num}[n_2]) \equiv \mathtt{num}[n] : \mathtt{num}} \tag{5.10g}$$

$$\frac{s_1 \hat{} s_2 = s \text{ str}}{\Gamma \vdash \mathtt{cat}(\mathtt{str}[s_1]; \mathtt{str}[s_2]) \equiv \mathtt{str}[s] : \mathtt{str}} \tag{5.10h}$$

$$\frac{}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) \equiv [e_1/x]e_2 : \tau} \tag{5.10i}$$

Rules (5.10a) through (5.10c) state that definitional equality is an *equivalence relation*. Rules (5.10d) through (5.10f) state that it is a *congruence relation*, which means that it is compatible with all expression-forming constructs in the language. Rules (5.10g) through (5.10i) specify the meanings of the primitive constructs of **E**. We say that rules (5.10) define the *strongest congruence* closed under rules (5.10g), (5.10h), and (5.10i).

Rules (5.10) suffice to calculate the value of an expression by a deduction similar to that used in high school algebra. For example, we may derive the equation

$$\mathtt{let}\, x\, \mathtt{be}\, 1 + 2\, \mathtt{in}\, x + 3 + 4 \equiv 10 : \mathtt{num}$$

by applying rules (5.10). Here, as in general, there may be many different ways to derive the same equation, but we need find only one derivation in order to carry out an evaluation.

Definitional equality is rather weak in that many equivalences that we might intuitively think are true are not derivable from rules (5.10). A prototypical example is the putative equivalence

$$x_1 : \mathtt{num}, x_2 : \mathtt{num} \vdash x_1 + x_2 \equiv x_2 + x_1 : \mathtt{num}, \tag{5.11}$$

which, intuitively, expresses the commutativity of addition. Although we shall not prove this here, this equivalence is *not* derivable from rules (5.10). And yet we *may* derive all of its closed instances,

$$n_1 + n_2 \equiv n_2 + n_1 : \mathtt{num}, \tag{5.12}$$

where n_1 nat and n_2 nat are particular numbers.

The "gap" between a general law, such as Equation (5.11), and all of its instances, given by Equation (5.12), may be filled by enriching the notion of equivalence to include a principle of proof by mathematical induction. Such a notion of equivalence is sometimes called *semantic equivalence*, because it expresses relationships that hold by virtue of the dynamics of the expressions involved. (Semantic equivalence is developed rigorously for a related language in Chapter 46.)

Theorem 5.5. *For the expression language* **E**, *the relation* $e \equiv e' : \tau$ *holds iff there exists* e_0 val *such that* $e \longmapsto^* e_0$ *and* $e' \longmapsto^* e_0$.

Proof The proof from right to left is direct, because every transition step is a valid equation. The converse follows from the following, more general, proposition, which is proved by induction on rules (5.10): if $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \equiv e' : \tau$, then when $e_1 : \tau_1, e_1' : \tau_1, \ldots, e_n : \tau_n, e_n' : \tau_n$, if for each $1 \leq i \leq n$ the expressions e_i and e_i' evaluate to a common value v_i, then there exists e_0 val such that

$$[e_1, \ldots, e_n/x_1, \ldots, x_n]e \longmapsto^* e_0$$

and

$$[e_1', \ldots, e_n'/x_1, \ldots, x_n]e' \longmapsto^* e_0.$$

□

5.5 Notes

The use of transition systems to specify the behavior of programs goes back to the early work of Church and Turing on computability. Turing's approach emphasized the concept of an abstract machine consisting of a finite program together with unbounded memory. Computation proceeds by changing the memory in accordance with the instructions in the program. Much early work on the operational semantics of programming languages, such as the SECD machine (Landin, 1965), emphasized machine models. Church's approach emphasized the language for expressing computations and defined execution in terms of the programs themselves, rather than in terms of auxiliary concepts such as memories or tapes. Plotkin's elegant formulation of structural operational semantics (Plotkin, 1981), which we use heavily throughout this book, was inspired by Church's and Landin's ideas (Plotkin, 2004). Contextual semantics, which was introduced by Felleisen and Hieb (1992), may be seen as an alternative formulation of structural semantics in which "search rules" are replaced by "context matching." Computation viewed as equational deduction goes back to the early work of Herbrand, Gödel, and Church.

Exercises

5.1. Prove that if $s \longmapsto^* s'$ and $s' \longmapsto^* s''$, then $s \longmapsto^* s''$.

5.2. Complete the proof of Theorem 5.1 along the lines suggested there.

5.3. Complete the proof of Theorem 5.5 along the lines suggested there.

6 Type Safety

Most programming languages are *safe* (or, *type safe*, or *strongly typed*). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, type safety for **E** states that it will never arise that a number is added to a string, or that two numbers are concatenated, neither of which is meaningful.

In general, type safety expresses the coherence between the statics and the dynamics. The statics may be seen as predicting that the value of an expression will have a certain form so that the dynamics of that expression is well-defined. Consequently, evaluation cannot "get stuck" in a state for which no transition is possible, corresponding in implementation terms to the absence of "illegal instruction" errors at execution time. Safety is proved by showing that each step of transition preserves typability and by showing that typable states are well-defined. Consequently, evaluation can never "go off into the weeds" and, hence, can never encounter an illegal instruction.

Type safety for the language **E** is stated precisely as follows:

Theorem 6.1 (Type Safety)**.**

1. *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$.
2. *If* $e : \tau$*, then either* e val*, or there exists* e' *such that* $e \longmapsto e'$.

The first part, called *preservation*, says that the steps of evaluation preserve typing; the second, called *progress*, ensures that well-typed expressions are either values or can be further evaluated. Safety is the conjunction of preservation and progress.

We say that an expression e is *stuck* iff it is not a value, yet there is no e' such that $e \longmapsto e'$. It follows from the safety theorem that a stuck state is necessarily ill-typed. Or, putting it the other way around, that well-typed states do not get stuck.

6.1 Preservation

The preservation theorem for **E** defined in Chapters 4 and 5 is proved by rule induction on the transition system (rules (5.4)).

Theorem 6.2 (Preservation)**.** *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$.

it to be used in the denominator of a quotient. It is difficult to do this without ruling out too many programs as ill-formed. We cannot predict statically whether an expression will be non-zero when evaluated, so the second approach is most often used in practice.

The overall idea is to distinguish *checked* from *unchecked* errors. An unchecked error is one that is ruled out by the type system. No run-time checking is performed to ensure that such an error does not occur, because the type system rules out the possibility of it arising. For example, the dynamics need not check, when performing an addition, that its two arguments are, in fact, numbers, as opposed to strings, because the type system ensures that this is the case. On the other hand, the dynamics for quotient *must* check for a zero divisor, because the type system does not rule out the possibility.

One approach to modeling checked errors is to give an inductive definition of the judgment e err stating that the expression e incurs a checked run-time error, such as division by zero. Here are some representative rules that would be present in a full inductive definition of this judgment:

$$\frac{e_1 \text{ val}}{\texttt{div}(e_1;\texttt{num[0]}) \text{ err}} \tag{6.1a}$$

$$\frac{e_1 \text{ err}}{\texttt{div}(e_1;e_2) \text{ err}} \tag{6.1b}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ err}}{\texttt{div}(e_1;e_2) \text{ err}} \tag{6.1c}$$

Rule (6.1a) signals an error condition for division by zero. The other rules propagate this error upwards: if an evaluated sub-expression is a checked error, then so is the overall expression.

Once the error judgment is available, we may also consider an expression, `error`, which forcibly induces an error, with the following static and dynamic semantics:

$$\frac{}{\Gamma \vdash \texttt{error} : \tau} \tag{6.2a}$$

$$\frac{}{\texttt{error err}} \tag{6.2b}$$

The preservation theorem is not affected by checked errors. However, the statement (and proof) of progress is modified to account for checked errors.

Theorem 6.5 (Progress With Error)**.** *If $e : \tau$, then either e err, or e val, or there exists e' such that $e \longmapsto e'$.*

Proof The proof is by induction on typing, and proceeds similarly to the proof given earlier, except that there are now three cases to consider at each point in the proof. □

6.4 Notes

The concept of type safety was first formulated by Milner (1978), who invented the slogan "well-typed programs do not go wrong." Whereas Milner relied on an explicit notion of "going wrong" to express the concept of a type error, Wright and Felleisen (1994) observed that we can instead show that ill-defined states cannot arise in a well-typed program, giving rise to the slogan "well-typed programs do not get stuck." However, their formulation relied on an analysis showing that no stuck state is well-typed. The progress theorem, which relies on the characterization of canonical forms in the style of Martin-Löf (1980), eliminates this analysis.

Exercises

6.1. Complete the proof of Theorem 6.2 in full detail.

6.2. Complete the proof of Theorem 6.4 in full detail.

6.3. Give several cases of the proof of Theorem 6.5 to illustrate how checked errors are handled in type safety proofs.

7 Evaluation Dynamics

In Chapter 5, we defined evaluation of expressions in **E** using a structural dynamics. Structural dynamics is very useful for proving safety, but for some purposes, such as writing a user manual, another formulation, called *evaluation dynamics*, is preferable. An evaluation dynamics is a relation between a phrase and its value that is defined without detailing the step-by-step process of evaluation. A *cost dynamics* enriches an evaluation dynamics with a *cost measure* specifying the resource usage of evaluation. A prime example is time, measured as the number of transition steps required to evaluate an expression according to its structural dynamics.

7.1 Evaluation Dynamics

An *evaluation dynamics* consists of an inductive definition of the evaluation judgment $e \Downarrow v$ stating that the closed expression e evaluates to the value v. The evaluation dynamics of **E** is defined by the following rules:

$$\frac{}{\mathtt{num}[n] \Downarrow \mathtt{num}[n]} \tag{7.1a}$$

$$\frac{}{\mathtt{str}[s] \Downarrow \mathtt{str}[s]} \tag{7.1b}$$

$$\frac{e_1 \Downarrow \mathtt{num}[n_1] \quad e_2 \Downarrow \mathtt{num}[n_2] \quad n_1 + n_2 \,\mathsf{is}\, n \;\mathsf{nat}}{\mathtt{plus}(e_1; e_2) \Downarrow \mathtt{num}[n]} \tag{7.1c}$$

$$\frac{e_1 \Downarrow \mathtt{str}[s_1] \quad e_2 \Downarrow \mathtt{str}[s_2] \quad s_1 \hat{} s_2 = s \;\mathsf{str}}{\mathtt{cat}(e_1; e_2) \Downarrow \mathtt{str}[s]} \tag{7.1d}$$

$$\frac{e \Downarrow \mathtt{str}[s] \quad |s| = n \;\mathsf{nat}}{\mathtt{len}(e) \Downarrow \mathtt{num}[n]} \tag{7.1e}$$

$$\frac{[e_1/x]e_2 \Downarrow v_2}{\mathtt{let}(e_1; x.e_2) \Downarrow v_2} \tag{7.1f}$$

The value of a `let` expression is determined by substitution of the binding into the body. The rules are not syntax-directed, because the premise of rule (7.1f) is not a sub-expression of the expression in the conclusion of that rule.

Rule (7.1f) specifies a by-name interpretation of definitions. For a by-value interpretation, the following rule should be used instead:

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v_2}{\texttt{let}(e_1; x.e_2) \Downarrow v_2} \tag{7.2}$$

Because the evaluation judgment is inductively defined, we prove properties of it by rule induction. Specifically, to show that the property $\mathcal{P}(e \Downarrow v)$ holds, it is enough to show that $\mathcal{P}$ is closed under rules (7.1):

1. Show that $\mathcal{P}(\texttt{num}[n] \Downarrow \texttt{num}[n])$.
2. Show that $\mathcal{P}(\texttt{str}[s] \Downarrow \texttt{str}[s])$.
3. Show that $\mathcal{P}(\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n])$, if $\mathcal{P}(e_1 \Downarrow \texttt{num}[n_1])$, $\mathcal{P}(e_2 \Downarrow \texttt{num}[n_2])$, and $n_1 + n_2$ is n nat.
4. Show that $\mathcal{P}(\texttt{cat}(e_1; e_2) \Downarrow \texttt{str}[s])$, if $\mathcal{P}(e_1 \Downarrow \texttt{str}[s_1])$, $\mathcal{P}(e_2 \Downarrow \texttt{str}[s_2])$, and $s_1 \hat{} s_2 = s$ str.
5. Show that $\mathcal{P}(\texttt{let}(e_1; x.e_2) \Downarrow v_2)$, if $\mathcal{P}([e_1/x]e_2 \Downarrow v_2)$.

This induction principle is *not* the same as structural induction on e itself, because the evaluation rules are not syntax-directed.

Lemma 7.1. *If $e \Downarrow v$, then v val.*

Proof By induction on rules (7.1). All cases except rule (7.1f) are immediate. For the latter case, the result follows directly by an appeal to the inductive hypothesis for the premise of the evaluation rule. □

7.2 Relating Structural and Evaluation Dynamics

We have given two different forms of dynamics for **E**. It is natural to ask whether they are equivalent, but to do so first requires that we consider carefully what we mean by equivalence. The structural dynamics describes a step-by-step process of execution, whereas the evaluation dynamics suppresses the intermediate states, focusing attention on the initial and final states alone. This remark suggests that the right correspondence is between *complete* execution sequences in the structural dynamics and the evaluation judgment in the evaluation dynamics.

Theorem 7.2. *For all closed expressions e and values v, $e \longmapsto^* v$ iff $e \Downarrow v$.*

How might we prove such a theorem? We will consider each direction separately. We consider the easier case first.

Lemma 7.3. *If $e \Downarrow v$, then $e \longmapsto^* v$.*

Proof By induction on the definition of the evaluation judgment. For example, suppose that $\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n]$ by the rule for evaluating additions. By induction, we know that $e_1 \longmapsto^* \texttt{num}[n_1]$ and $e_2 \longmapsto^* \texttt{num}[n_2]$. We reason as follows:

$$\begin{array}{lll} \texttt{plus}(e_1; e_2) & \longmapsto^* & \texttt{plus}(\texttt{num}[n_1]; e_2) \\ & \longmapsto^* & \texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \\ & \longmapsto & \texttt{num}[n_1 + n_2] \end{array}$$

Therefore, $\texttt{plus}(e_1; e_2) \longmapsto^* \texttt{num}[n_1 + n_2]$, as required. The other cases are handled similarly. □

For the converse, recall from Chapter 5 the definitions of multi-step evaluation and complete evaluation. Because $v \Downarrow v$ when v val, it suffices to show that evaluation is closed under converse evaluation:[1]

Lemma 7.4. *If $e \longmapsto e'$ and $e' \Downarrow v$, then $e \Downarrow v$.*

Proof By induction on the definition of the transition judgment. For example, suppose that $\texttt{plus}(e_1; e_2) \longmapsto \texttt{plus}(e_1'; e_2)$, where $e_1 \longmapsto e_1'$. Suppose further that $\texttt{plus}(e_1'; e_2) \Downarrow v$, so that $e_1' \Downarrow \texttt{num}[n_1]$, and $e_2 \Downarrow \texttt{num}[n_2]$, and $n_1 + n_2$ is n nat, and v is $\texttt{num}[n]$. By induction $e_1 \Downarrow \texttt{num}[n_1]$, and hence $\texttt{plus}(e_1; e_2) \Downarrow \texttt{num}[n]$, as required. □

7.3 Type Safety, Revisited

Type safety is defined in Chapter 6 as preservation and progress (Theorem 6.1). These concepts are meaningful when applied to a dynamics given by a transition system, as we shall do throughout this book. But what if we had instead given the dynamics as an evaluation relation? How is type safety proved in that case?

The answer, unfortunately, is that we cannot. Although there is an analog of the preservation property for an evaluation dynamics, there is no clear analog of the progress property. Preservation may be stated as saying that if $e \Downarrow v$ and $e : \tau$, then $v : \tau$. It can be readily proved by induction on the evaluation rules. But what is the analog of progress? We might be tempted to phrase progress as saying that if $e : \tau$, then $e \Downarrow v$ for some v. Although this property is true for **E**, it demands much more than just progress—it requires that every expression evaluate to a value! If **E** were extended to admit operations that may result in an error (as discussed in Section 6.3), or to admit non-terminating expressions, then this property would fail, even though progress would remain valid.

One possible attitude towards this situation is to conclude that type safety cannot be properly discussed in the context of an evaluation dynamics, but only by reference to a structural dynamics. Another point of view is to instrument the dynamics with explicit checks for dynamic type errors, and to show that any expression with a dynamic type fault must be statically ill-typed. Re-stated in the contrapositive, this means that a statically well-typed program cannot incur a dynamic type error. A difficulty with this point of view

7.4. Augment the evaluation dynamics with checked errors, along the lines sketched in Chapter 5, using e err to say that e incurs a checked (or an unchecked) error. What remains unsatisfactory about the type safety proof? Can you think of a better alternative?

7.5. Consider generic hypothetical judgments of the form

$$x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$$

where v_1 val, $\ldots$, v_n val, and v val. The hypotheses, written Δ, are called the *environment* of the evaluation; they provide the values of the free variables in e. The hypothetical judgment $\Delta \vdash e \Downarrow v$ is called an *environmental evaluation dynamics*.

Give a hypothetical inductive definition of the environmental evaluation dynamics *without making any use of substitution*. In particular, you should include the rule

$$\frac{}{\Delta, x \Downarrow v \vdash x \Downarrow v}$$

defining the evaluation of a free variable.

Show that $x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ iff $[v_1, \ldots, v_n/x_1, \ldots, x_n]e \Downarrow v$ (using the by-value form of evaluation).

Note

1 Converse evaluation is also known as *head expansion*.