S. Hayashi and N. Nakano, *PX: A Computational Logic* (MIT Press, London, 1988), Price £24.75 (hardback), ISBN 0-262-08174-1.

PX is a logic for reasoning about programs. It is a type-free constructive logic based on Feferman's theory T_0. It formalizes a simple programming language similar to pure Lisp, and implements the idea that proofs are programs in the sense that it allows the mechanical extraction of programs from proofs of their specifications. The theory is designed to agree with classical logic whenever this does not interfere with program extraction. A proof checker for PX has been written. It incorporates a macro facility that allows the use of derived inference rules to make proofs somewhat more legible. The system can extract Lisp programs from proofs, includes a facility to pretty print (using TEX) proofs, and allows the use of other theorem provers to prove hypotheses whose proofs are unneeded by the extractor.

PX can reason about programs written in a simple functional programming language. The language is an applicative order language similar in spirit to pure Lisp. The variables of the language are of two kinds, total variables which always stand for values, and partial variables which need not. Partial variables serve some of the purposes of metavariables since whenever an expression containing a partial variable has a property so does any expression resulting from substitution for that variable. Hence, it is possible to state and prove certain general theorems about expressions. Among the possible values for expressions are classes. These are the names of sets of values. The logic is a predicate logic with quantification over classes. Four primitive formulas express properties of an expresion: that it has a value, that it is equal to another term, that its value is a class, and that its value is a member of the set named by the class value of another expression. The logic also includes a formula constructor that, given a list of expressions and formulas, produces a formula logically equivalent to the first formula on the list that is paired with an expression with a non-nil value. Formulas including this constructor are particularly useful in specifying inductive classes. A syntactically recognizable subset of the formulas, the rank-zero formulas, is defined. They never have interesting computational content, so classical logic may be used in their proofs. In particular, double negation elimination, limited to rank-zero formulas, is a rule of the system. All four of the primitive formulas noted above are of rank zero.

Classes are at the heart of PX. With the exception of a handful of primitive classes whose existence is asserted by axiom, the existence of a given class is derived using two principles. Of the two, conditional induction generation (CIG) is the more important. For each formula meeting certain syntactic conditions it asserts the existence of a class naming a set defined by that formula. More specifically, if A is such a formula, and X and a variables, there is a class $\mu X\{a \mid A\}$ which names the least fixed point of the function mapping X to $\{a \mid A\}$. Other variables may appear free in A, so class constructors are also specifiable using CIG. Associated with this principle are three axioms. The first asserts that the class exists. The second characterizes membership for the class: an object is a member of $\mu X\{a \mid A\}$ exactly when it

satisfies the formula that results when $\mu X\{a|A\}$ is substituted for X in A. The third is an induction principle that allows facts about members of the class to be proved by structural induction. The conditions on A are present for two reasons. First, they ensure that class membership remains free of computational content. Second, they ensure that the desired fixed point really exists. Using CIG, it is possible to define most of the types that occur in programming languages: function spaces, products, lists, and so on. The other principle asserts the existence of dependent sums of classes.

The principal purpose of PX is the extraction of programs from constructive proofs of their specifications. Specifications are sequents stating that whenever the input belongs to certain a class, an output value exists and has the right properties. When such a sequent has been proved, it is possible to extract a program that meets the specification. The structure of the program reflects the structure of the proof. In particular, inductive proofs generate recursive programs. In PX, one often proves such a sequent in two steps. First, prove a sequent in which the class to which inputs must belong has been changed. Second, show that the new input class contains the second. The formula to be proven in the second stage is of rank zero, so it does not contribute to the program to be extracted and may be proved classically. Additionally, this means that the program may be extracted once the first part of the proof has been finished. Of course, until it has been proven there is no guarantee that the function behaves correctly on all elements of the intended domain, only that it behaves correctly on members of the new domain. Usually, the new domain is chosen so that the induction used to prove the theorem has the right structure; careful choice of domain allows the extraction of efficient programs.

This book is not a general overview of programming logics, nor is it a user manual for the PX system. Instead it is a detailed, theoretical discussion of PX, together with some examples, and a brief description of the implementation. Three of the six chapters are devoted to presenting the theory of the system. They formally define the theory upon which PX is based, give its intended semantics, and define the notion of realizability which is the theoretical basis for extraction. The others are devoted to describing the implementation of PX; providing some simple examples of program development, and showing that enough programs can be derived (PX is extensionally complete for the partial recursive functions); and showing how certain type-disciplines, namely the use of dependent sums for modules, and type polymorphism may be encoded in PX. There are a few places where the formalism is insufficiently motivated, particularly early in the book, which leaves the reader to absorb complicated definitions without knowing exactly what is being defined. But this problem is not severe enough to render the book impenetrable.

R. L. CONSTABLE
Department of Computer Science
Carnell University
Pittsburgh, PA, USA