Commentary on Practical Foundations for Programming Languages (Second Edition)

Robert Harper
Carnegie Mellon University

February 13, 2017

It may be useful to many readers for me to explain the many decisions made in the organization and presentation of ideas in PFPL, and to suggest some variations and extensions that I have considered during and after writing the book. The discussion necessarily reflects my own opinions.

1 Paradigms

The earliest attempts to systematize the study of programming languages made use of the concept of a *paradigm*, apparently from Thomas Kuhn's *The Structure of Scientific Revolutions*. The trouble is, with respect to programming languages at least, no one can say what is a paradigm, or when we know we have a new one. Nevertheless, it has become commonplace to classify languages into trendy-sounding categories such as "imperative", "functional", "declarative", "object-oriented", and "concurrent" programming. The convention is so pervasive that I am asked to justify why I do not adhere to it.

It's a matter of taxonomy versus genomics, as described in Stephen Gould's critique of cladistics (the classification of species by morphology) in his essay "What, if anything, is a zebra?" (Gould, 1983). According to Gould, there are three species of black-and-white striped horse-like animals in the world, two of which are genetically related to each other, and one of which is not (any more so than it is to any mammal). It seems that the mammalian genome encodes the propensity to develop stripes, which is expressed in disparate evolutionary contexts. From a genomic point of view, the clade of zebras can be said not to exist: there is no such thing as a zebra! It is more important to study the genome, and the evolutionary processes that influence it and are influenced by it, than it is to classify things based on morphology.

Paradigms are clades; types are genes.

2 Preliminaries

Part I, on syntax, judgments, and rules, is absolutely fundamental to the rest of the text, and is essential for a proper understanding of programming languages. Nevertheless, it is not necessary, or advisable, for the novice to master all of Part I before continuing to Part II. In fact, it might be preferable to skim Part I and begin in earnest with Part II so as to gain a practical understanding of the issues of binding and scope, the use of rules to define the statics and dynamics of a language, and the use of hypothetical and general judgments. Then one may review Part I in light of the issues that arise in a cursory study of even something so simple as a language of arithmetic and string expressions.

Having said that, the fundamental importance of the concepts in Part I cannot be overstressed. It is surprising that after decades of experience languages are still introduced that disregard or even flout basic concepts of binding and scope. A proper treatment of binding is fundamental to modularity, the sole means of controlling the complexity of large systems; it is dismissed lightly at the peril of the programmer. It is equally surprising that, in an era in which verification of program properties is finally understood as essential, languages are introduced without a proper definition. It remains common practice to pretend that "precise English" is a substitute for, or even preferable to, formal definition, despite repeated failures over decades to make the case. All that is needed for a precise definition is in Part I of the book, and exemplified in the remainder. Why avoid it?

3 Statics and Dynamics

Each language concept in PFPL is precisely specified by its *statics* and its *dynamics*, which are linked by a *safety* theorem stating that they cohere. The statics specifies which are the well-formed expressions of a language using context-sensitive formation constraints. The dynamics specifies how expressions are to be evaluated, usually by a transition system defining their step-by-step execution and when evaluation is complete. The safety theorem says that well-formed programs are either completely evaluated or are susceptible to transition, and that the result of such a transition is itself well-formed.

The statics of a language is usually an inductively defined *typing judgment* of the form $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$, stating that an expression e has the type τ uniformly in the variables $x_1, \ldots, x_n$ ranging over their specified types. The inductive definition takes the form of a collection of *typing rules* that jointly defined the strongest, or most restrictive, judgment closed under those rules. The minimality requirement gives rise to the important principle of *induction on typing*, an instance of the general concept of *rule induction* applied to the given typing rules. It is of paramount importance that the typing judgment obey the *structural properties* of a hypothetical general judgment as defined in Part I of the book. In particular typing must be closed under substitution, must assign the assumed type to each variable, and must be stable under adding spurious

variable assumptions. Experience shows that languages that fail to validate the variable and substitution properties are inherently suspect, because they violate the mathematical concept of a variable developed since antiquity.

The dynamics of a language is usually given as a *transition judgment* on execution states defining the atomic steps of execution together with a specification of what are the initial and terminal states. In many cases states are simply expressions, and transition is inductively defined by a collection of rules using Plotkin's method of *structural operational semantics*. In more sophisticated settings the state may contain additional information such as the contents of memory or the presence of concurrently executing processes, but the general pattern remains the same. An SOS specification of a transition system is the universal format used throughout PFPL. Besides being fully precise and perspicous, structural dynamics lends itself well to mechanization and verification of safety properties.

Other methods of specifying the dynamics are of course possible. An *abstract machine* is a transition system whose defining rules are premise-free (that is, are all axioms, and never proper inference rules). Usually it is straightforward to derive an abstract machine from a structural dynamics by introducing components of the state that account for the implicit derivation structure in the latter formulation. An *evaluation dynamics* consists of an inductive definition of the complete value, if any, of an expression in terms of the values of its constituent expressions and their substitution instances. An evaluation dynamics may be regarded as a characterization of multistep evaluation to a value. It can be convenient in some situations to think directly in terms of the outcome of evaluation, rather than in terms of the process of reaching it. But doing so precludes speaking about the *cost*, or *time complexity*, of a program, which may be defined as the number of steps required to reach a value. A *cost dynamics* defines not only the value of an expression, but also the cost of achieving that value. Costs need not be limited to step counts, as is illustrated in the chapter on parallelism.

A cost dynamics is both more and less than an evaluation dynamics. Reading the cost and value as outputs, it specifies both the value and cost of an expression, but reading the cost as input, it specifies only the values of expressions that are achievable with the specified cost. There is no inherent reason to prefer one reading to the other. For a specified e and n the cost dynamics may be defined as the n-fold head expansion of a value in terms of the transition dynamics. The transition dynamics may be recovered from the cost dynamics by defining e to transition to e' when, for any $n \geq 0$, if e' evaluates to v with cost n, then e evaluates to v with cost $n + 1$. Conversely, a cost dynamics may be seen as a specification of n-step transition to a value. Thus, a cost dynamics *is* the transition dynamics in disguise, and so properties of the latter may readily be transferred to the former. In particular, it is no surprise that one may characterize type safety properties using a cost dynamics, precisely because it contains the transition dynamics in terms of which safety is defined.

Transition systems are sometimes called "small step" dynamics, in contrast to evaluation relations, which are then called "big step" dynamics. There is no

accounting for taste, but in the author's opinion evaluation dynamics does not define a transition relation, but an evaluation relation, and is not comparable as a matter of size with a proper transition system. Moreover, in many cases the values related to expressions by an evaluation (or cost) dynamics are not themselves forms of expression. In such cases it is senseless to treat evaluation as a kind of transition; it is, quite plainly, just evaluation.

4 Partiality and Totality

A significant change in the second edition is that I have deferred discussion of partiality until the basics of type structure are established. The advantage of this approach is that it avoids distracting vacillations about partiality and totality in the midst of discussing fundamental issues of type structure. The disadvantage is that the deterministic dynamics is less well-motivated in the total case, and that many semantic properties of types do not carry over from the total to the partial case without signification complication (a prime example being the parametricity properties of type quantification.)

For most practical purposes it is a bit unnatural to emphasize totality. Instead, it might be preferable to start out with **PCF**, rather than **T**, and omit discussion of inductive and coinductive types in favor of recursive types. The chief difficulty is that the discussion of representation independence for abstract types is no longer valid; one must weaken the results to take account of partiality, which complicates matters considerably by forcing consideration of admissible predicates and fixed point induction. Perhaps this is a situation where "white lies" are appropriate for teaching, but for the the book I chose not to fudge the details.

5 Functions First

The illustrative expression language studied in Part II provides a starting point for the systematic study of programming languages using type systems and structural operational semantics as organizing tools. Systematic study begins in Part III with (total) functions, followed by **T**, a variation on Gödel's formalism for primitive recursive functions of higher type. This line of development allows me to get to interesting examples straightaway, but it is not the only possible route.

An alternative is to begin with products and sums, including the unit and empty types, as a starting point. The difficulty with this approach is that it is a rather austere starting point, and one that is not obviously motivated by the illustrative expression language, or any prior programming experience. The advantage is that it allows for the consideration of some very basic concepts in relative isolation. In particular, the booleans being definable as the sum of two copies of the unit type, one may consider binary decision diagrams as an application of sums and products. Bdd's are sufficient to allow modeling

of combinational logic circuits, for example, and even to generalize these to decision diagrams of arbitrary type. One can relate the formalism to typical graphical notations for bdd's, and discuss equivalence of bdd's, which is used for "optimization" purposes.

One might even consider, in the latter event, the early introduction of fixed points so as to admit representation of digital logic circuits, such as latches and flip-flops, but this would conflict with the general plan in the second edition to separate totality from partiality, raising questions that might better be avoided at an early stage.

6 The Empty (and Unit) Types

A primary motivation for including the empty type, `void`, in Chapter 11 is to ensure that, together with the binary sum type, $\tau_1 + \tau_2$, all finite sums are expressible. Were finite n-ary sums to be taken as primitive, then the empty type inevitably arises for the case of zero summands.

The notation for the elimination form for the type `void` is regrettable in that it leads to confusion for many students. The notation, $\texttt{abort}(e)$, which is relatively standard, suggests that this expression engenders some form run-time error, which it most definitely does not do. Rather, $\texttt{abort}(e)$ is a *nullary case analysis* with no cases, better written as $\texttt{case}\, e\, \{\ \}$, to reflect the interpretation of `void` as a nullary sum.

In a total language there are no closed terms of type `void`, for any such would necessarily evaluate to a value of this type, yet there are none. The primary use of `abort` is to express that a particular situation cannot arise, and hence can be regarded as complying with any required typing constraints. For example, the second branch of a binary case analysis on an expression of a type $\tau + \texttt{void}$ cannot ever be activated. But if the first branch is of a type τ, then so must be the second; this is achieved by using an `abort` (nullary case) of type τ. Perhaps `impossible`, or `contradiction`, would have been a better choice of terminology than `abort`.

The type `void` may be used to state that an expression must not return to its site of evaluation (say, by continuing elsewhere). Thus, a function of type $\tau \rightharpoonup \texttt{void}$ must not return when applied. This is rather different from a function of type $\tau \rightharpoonup \texttt{unit}$, which may well return to the call site, perhaps having engendered effects along the way. Regrettably, the industry standard is to confuse `unit` with `void`, writing the latter for the former. Under that misinterpretation, a function with the result type `void` may well return. The idea of a function that has *no value* is confused with that of a function that has *no interesting value*. Indeed, there is no way to express the former, useful condition once `void` is used to stand for `unit`. The error is not just terminological, it subtracts from the expressive power of a language.

7 Laziness and Eagerness

In a total language the decision to evaluate constructors eagerly or lazily has no semantic significance, though it may have implications for efficiency of execution. Whether to evaluate applications by-name or by-value is a separate matter. Neither choice is uniformly more efficient; if an argument is not used, it is not evaluated under by-name, but if an argument is used more than once, it is evaluated once under by-value, but multiply by-name. By-need evaluation avoids the repetition, and therefore seems superficially superior. This impression is tempered by the observations that by-need evaluation is by no means optimal, and that laziness robs a language of expressive power compared to eagerness.

With regard to optimality, one need only consider *Gustav's Function*, which takes three booleans and is to yield `true` if any two of them do, regardless of whether the third diverges. A moment's reflection reveals that no deterministic evaluation strategy avoids divergence (or doing "extra work" even when all three arguments are divergent) because one cannot predict which of the two arguments would be best to evaluate first. This old example (dating back to the 1970's) neatly refutes any arguments about efficiency of by-need evaluation.

More importantly, uniform adoption of laziness is semantically destructive; it *subtracts* expressiveness, which is never a good thing. To be specific, lazy languages *must* lack any type of natural numbers, regardless of what else is added to the language to mitigate this egregious deficiency. This means, in particular, that mathematical induction is never valid as a reasoning principle about programs written in a lazy language! One can *never* validly argue that a function whose domain is the putative natural numbers has a property by considering its behavior on `zero` and `succ`; one must always consider, in addition, the possibility of divergence, which is a *value*, not an *effect*, in a lazy language. Similarly, there are no booleans in a lazy language, nor lists, nor trees, nor enumeration types, nor abstract syntax, to name a few common examples. For if all constructors are lazy, then one can form, for example, the infinite natural number consisting of an infinite stack of successors; but this is of course not a natural number. And even if one specifies that a constructor is eager, one nevertheless has the divergent value at *every* type, ruining the inductive character of any putative inductive types.

This restriction is pure loss. There is no way to recover inductive types in a lazy language. But, contrarily, it is straightforward to introduce laziness in the context of an eager language. In an eager language divergence is an effect, and one reasons about the values inhabiting a type. The natural numbers are indeed the natural numbers, and one has types of lists and trees, for example, that are absent in a lazy language. To obtain their lazy counterparts one need only introduce a type of memoized computations, as described in Chapter 36 of the text. It is pure gain to do so, because one has all the benefits of laziness without the drawbacks. What's not to like?

It is sometimes said that lazy languages are "more mathematical" than eager languages. This, too, is marketing nonsense. For example, whereas lazy

languages have products in the categorial sense, they lack sums; the situation is precisely the opposite with eager languages. Which is "more mathematical?" Neither, of course, they are simply different. (And, as just remarked, products can be defined in an eager language using suspensions, but sums are forever unavailable in lazy languages.)

More importantly, there is more to programming than just mathematics. Matters of complexity and efficiency are absent in mathematics, but are of the essence in programming. It is important to incorporate mathematical thinking into programming, but it is also important not to take this too far, to the point of farce, as is done with the marketing of lazy languages. In mathematics one does not distinguish a computation from a value, much less the process of determining the value of a complex expression, or the possibility of an expression diverging. These considerations are part of the essence of programming. Laziness does not avoid addressing these issues, it exacerbates the difficulty of managing them.

8 Static and Dynamic Classification

The importance of sum types for static classification of data cannot be overstated. They have long been ignored in seat-of-the-pants language designs, giving rise to absurdities such as "null pointers" in abstract languages (there are none), the notion of "dynamic dispatch" in "object-oriented" languages (it is a trivial instance of pattern matching, albeit without of exhaustiveness or coverage checking), so-called "dynamically typed" languages (see Section 12), and to *ad hoc* notions such as "enumeration types" (which are but sums of unit type).

Static classification naturally generalizes to dynamic classification, in which new classes may be generated at run-time. The same basic mechanisms extend smoothly from the static to the dynamic case, obviating the need for specialized mechanisms such as exception generation or communication channel allocation. Moreover, they provide a natural basis for ensuring confidentiality and integrity of data. Coupled with an independent type abstraction mechanism, one can create as many distinct dynamically classified types as one likes, simply by defining them all to be the one type of dynamically classified values and relying on abstraction to keep them distinct.

The distinction between dynamic and static classification sheds light on a major deficiency of so-called "object-oriented" languages. Because the totality of classes must be known statically, it follows that, for semantic reasons, that whole-program compilation is required, which completely defeats the concept of modularity in program development. To hide the defect, the error is marketed as "just-in-time compilation," echoing MBA-speak for certain business practices. It is not a question of efficiency of implementation, but of deficiency of expressiveness.

(See Section 14 for more on "object-oriented" programming.)

9 Inductive and Coinductive Types

The distinction between inductive and coinductive types is present only in the case of total languages; in languages with partiality they are subsumed by recursive types. For this reason it may be preferable to avoid the technical complexities of inductive and coinductive types, and simply work with recursive types in **FPC**. It is probably still worthwhile to discuss generic programming, which is of independent interest, and to discuss streams, which are characterized by their behavior, rather than their structure.

Streams raise an interesting question pertaining to the distinction between partial and total languages. It is not possible to write a `filter` function for streams that, given a binary predicate (a function of type `nat` $\rightarrow$ `bool`), produces the stream whose elements consist only of those elements for which the predicate evaluates to `true`. Intuitively, the code for `filter` does not know how long to wait for the next element to arrive; it can be arbitrarily far in the future of the stream, and in fact may not even exist in the case that the predicate is `false` for every remaining element! It is in the nature of a total language to build in the "proof" that it terminates on all inputs, but this is not possible for `filter`. Rather, to implement `filter` requires an *unbounded search* operation, analogous to a `while` loop in an imperative programming language or to a fixed point construction in a partial functional language.

10 PCF

Plotkin's language **PCF** is often called the *E. coli* of programming languages, the subject of countless studies of language concepts. The formulation given here is intended to make clear the connections with **T**, and the classical treatment of recursive (that is, computable) functions. The relation to **T** is important for explaining the significance of totality compared to partiality in a programming language.

I have chosen to formulate **PCF** in Plotkin's style, with a general fixed point operator, but, contrary to Plotkin, to admit both an eager and a lazy dynamics. This choice is not comfortable, because a general fixed point is a "lazy" concept in that the recursively defined variable ranges over computations, not merely values. I have waivered over this decision, but in the end can find no better way to expose the important ramifications of general recursion.

In particular, one can see immediately that if the successor is lazy, as it would be in a lazy language, then there is an "infinite natural number", which is to say that `nat` does not, and cannot be made to, represent the natural numbers! Nor are lists lists, nor are trees trees, nor is any inductive type definable in such a setting. This point about laziness and partiality seems often to be overlooked, when, contrarily, it ought to be emphasized.

An advantage of separating self-reference from functions is that it debunks the myth that recursive self-reference requires a "stack" for its implementation. (If you doubt this, consider that a flip-flop is self-referential, but of course in-

volves no "stack".) *En passant* doing so makes clear that a recursive function need not be re-entrant. A stack is required only if it is, and not otherwise. It is not a matter of "optimization", it is a matter of semantics.

11 Recursive Types

Recursive types (without any positivity requirement) only make sense for a language with partial functions, and, using them, one may implement self-reference with which to define them. Indeed, a single recursive type is sufficient to interpret the λ-calculus, the original universal model of computation. In the presence of partiality the distinction between eager and lazy evaluation is semantically significant, and the interplay between eager and lazy strategies is central to defining both inductive and coinductive types as recursive types.

Recursive types are the means by which the concept of a "data structure" can be given full expression. Too often students are taught that data structures are defined structurally, using "box and pointer" diagrams. But this approach excludes important cases involving laziness or functions, which are not depictable in those terms. For this reason alone it is useful to consider the general case, with the isomorphism between a recursive type and its unfolding playing the role of an "abstract pointer" that is not restricted to finitary representations of data in memory.

Recursive types are essential for many programming concepts, most obviously the definition of a self-referential value such as a function defined in terms of itself. Less obviously, recursive types are crucial for understanding the (misnamed) concept of dynamic typing (*q.v.*), for the representation of coroutines using continuations, and for a full treatment of dynamic dispatch (again, via self-reference). Although deceptively simple in the formalism, recursive types are a deep and fundamental concept that should be given full treatment in any account of programming languages.

12 Dynamic Typing

The whole of Part IX is devoted to so-called dynamic typing. The first point is that what is dynamic are not *types*, but rather *classes*, of data. The second, which is much the same, is that such languages are not *untyped*, but rather *uni-typed* (to use Dana Scott's neat turn of phrase), the one type in question being a recursive sum. Understanding this is essential to resolving the never-ending "debate" about typed *vs.* untyped languages.

The dynamic language **DPCF** hews close to the static language **PCF** to facilitate comparison. Needless to say, many variations are possible. It is a useful exercise to formulate a dynamic language, and to reformulate it in terms of recursive sums. For example, it is common to consider multi-argument (and even multi-result) functions. On careful analysis it becomes apparent that multiple arguments and results are but a concession to static typing in which one

admits that product types exist. Moreover, the typical syntax for multiple arguments is just a form of pattern matching for tuples. You may, if you wish, confine your attention to one type, but as the example of multiple arguments and results illustrates, you will not seriously want to do that.

13 Subtyping

Structural subtyping is, according to Pfenning (2008), based on the principle that a value should have the same types as its η-expansion. For example, the subtyping principle for function types may be justified by considering any expression e of type $\tau_1' \to \tau_2$. Under the assumptions that $\tau_1 <: \tau_1'$ $\tau_2 <: \tau_2'$, then e should also have the type $\tau_1 \to \tau_2'$, because its expansion, $\lambda\,(x_1 : \tau_1)\, e(x_1)$, does under those assumptions. Notice that if e were a λ-abstraction, the expansion would not be needed to obtain the weaker typing, but when e is a variable, for example, the expansion has more types than the variable as declared, so subtyping adds some flexibility. Similarly, one may justify the n-ary product subtyping principle on similar grounds. Given an n-tuple e, we may form an m-tuple from it consisting of the first $m \leq n$ projections from e, in order, to form a narrower view of e.

The treatment of numeric types stretches the meaning of structural subtyping to its limits. The idea is that an integer n may be "expanded" to the rational $n \div 1$, and a rational may be "expanded" to a real (say, a Cauchy sequence) representing the same rational. Whether these count as η-expansions is questionable; it is more accurate to say that they are justified by canonical choices of inclusions that are implicit in the interpretation of subtyping. Whereas one may consider a dynamics in which values are identified with their η-expansions (so that a narrow tuple may, in fact, be wider than its type would indicate), it is less clear that this would be sensible for the numeric types—unless all numbers were to be represented as reals under the hood!

But if that were the case, the inclusions would more naturally be interpreted as behavioral, rather than structural, subtypes. More precisely, the subtyping principles would better be read as isolating certain reals as rationals and certain rationals as integers according to a *semantic*, rather than *syntactic*, criterion. As such it is not always possible to determine whether a given real number is rational or integral by purely mechanical means; it is, rather, a matter of proof. In practice one gives syntactic conditions that suffice for practical situations, which is exactly what I have done in Chapter 25 in developing a syntactic behavioral type system to track the class of a value in **DPCF**. In truth one cannot determine the class of an arbitrary computation, but in practice one can often get by with some relatively simple syntactic conditions.

14 Dynamic Dispatch

The discussion of dynamic dispatch is meant to address one of the more prominent aspects of "object-oriented" programming.[1] A great deal of emphasis is placed on the idea of an "object" as a collection of "methods" that act on shared "instance" data. Many a claim hinges on this supposedly distinguishing characteristic of "object-oriented programming." For example, it is common to set up a false comparison to "abstract types", which is said to conflict with "object-oriented" programming. Yet, as is shown in Chapter 26 dynamic dispatch is a particular use of data abstraction, to which it thereby can hardly be opposed.

The main point of Chapter 26 is to make clear that the "method-oriented" and "class-oriented" organizations of code are isomorphic, and hence interchangeable in all contexts. *There is no distinction between the two organizations; they can be interchanged at will.* The central idea, a version of which also appears in Abselson and Sussman's *Structure and Interpretation of Computer Programs*, is what I call the *dispatch matrix*, which determines the behavior of each method on each class of instance. The dispatch matrix is inherently symmetric in that it favors neither the rows nor the columns. One may, if one wishes, take a row-oriented or a column-oriented, view of the dispatch matrix, according to taste. One may also interchange one for the other at will, without loss or damage. There is nothing to the choice; it reflects the fundamental duality between sums and products, a duality that is equally present in matrix algebra itself.

The distinction between the row- and column-oriented organizations would appear to arise when considering *inheritance*, whereby one can specify the behavior of some new methods as a modification or extension of some existing behaviors. The asymmetries between the two organizations can be easily resolved by making the dispatch matrix, which defines the behaviors of objects, as the unit of inheritance (Chapter 27). Inheritance requires structural subtyping, which may be seen as an indication of its dubious advisability as a language construct. Nevertheless, it is useful to study even the bad ideas in language design so as to identify them and avoid making the same mistakes in the future.

15 Modalities and Monads in Algol

As an homage to the master, the name "Modernized Algol" (Chapter 34) is chosen to rhyme with "Idealized Algol," Reynolds's reformulation (Reynolds, 1981) of the original, itself often described as "a considerable improvement on most of its successors." The main shortcoming of Algol, from a modern perspective, is that its expression language was rather impoverished in most

[1] The scare quotes are necessitated by the (intentional) vagueness and fluidity of the topic. Its one stable characteristic is that its proponents insist that object-oriented programming is never is what it, in fact, is. Being too profound for mortals, it's a topic we'll have to defer to the afterlife; certainly hell is object-oriented, but what may be the situation in heaven is not so clear. Meanwhile, some remarks about the actual world may be helpful.

respects, perhaps to achieve the *a priori* goal of being stack-implementable, a valid concern in 1960. On the other hand, its main longcoming, so to say, is that it made a *modal distinction* between expressions and commands, which was later to be much celebrated under the rubric of "monads." Indeed, Reynolds's formulation of Algol features the `comm` type, which in fact forms a monad of unit type.

In Reynolds's case the restriction to `unit`-typed commands was not problematic because there expressions are allowed to depend on the contents of the store. In private communication Reynolds himself made clear to me that, for him, this is *the* essential feature of the Algol language and of Hoare-style logic for it, whereas I, by contrast, regard this as a mistake—one that becomes especially acute in the presence of concurrency, for then multiple occurrences of the same "variable" (that is, assignable used an expression) raises questions of how often and when is the memory accessed during evaluation. The answer matters greatly, even if memory accesses are guaranteed to be atomic. Thus, **MA** differs from Reynolds's **IA** in that assignables are not forms of expression, which means that expression evaluation is independent of the contents of memory (but not of its domain!).

MA stresses another point, namely that the separation of commands from expressions is that of a *modality*, and not merely that of a *monad* (see Pfenning and Davies (2001) for more on this topic.) The modality gives rise to a connective that has the structure of a monad, but this observation is posterior to the modal distinction on which it is constructed. Without it, there are only evaluatable expressions with types of the form $\mathtt{cmd}(\tau)$, and nothing ever executes. Somewhere there has to be a command that executes an encapsulated command without itself being encapsulated, and it is there that the underlying modality is exposed. This shows up in the Haskell language as the device of "automatically" running an expression whose type happens to be that of the "IO monad." This is achieved by using the derived form

$$\mathtt{do}\, e \triangleq \mathtt{bnd}\, x \leftarrow e\,;\mathtt{ret}\, x,$$

which is, of course, a command.

16 Process Calculus

Chapter 39 is devoted to the development of process calculus in the style of Milner (1999). The advantage of discussing process calculus in isolation is that one can develop some of the central ideas free of commitment to their realization in a programming language. The disadvantage is that to do so requires a number of *ad hoc* devices, such as name generation, that really have nothing to do with concurrency *per se*, but are merely a technical device for limiting the visibility of communication among a network of processes. But this is exactly what is provided by dynamic classification (as described in Chapter 33). Once that is recognized, process calculus degenerates into non-deterministic composition, which is in any case the essence of concurrency.

From this point of view, the entire enterprise of process calculus is absorbed into the larger problem of concurrent programming languages in which one has available type structure denied to a simple-minded process calculus. Remove the channel labeling machinery, and there is nothing left to study. Moreover, dynamic classification is of independent interest, and has no particular association with concurrency. Other than to make this point, there is really very little justification for studying process calculus any more. Still, given its influence, it is worth dissecting it and exposing the more fundamental structure wrapped into it.

This viewpoint is adopted in Chapter 40 on Concurrent Algol, which begins with broadcast communication of dynamically classified values, and refines this into selective communication restricted to messages of a specific class. There is no explicit reliance on process calculus; readers may wish to skim or omit Chapter 39, and proceed directly to concurrency in the context of Modernized Algol. Among many further directions to consider, It would be interesting to extend selective communication to admit a more general form of pattern matching than just the simple specification of a channel on which to communicate. The channel specifies a class of messages to consider, and one may wish to consider more general specifications that depend on the content of the message, as well as its channel.

For the benefit of those familiar with the π-calculus and related formalisms, some justification of the formulation given here may be helpful.

1. Sums/choice. Rather than consider choice between two *processes*, I instead consider choice of *events* in roughly the sense of Reppy's Concurrent ML. One reason is that, in the absence of distributivity, choices among processes form clusters that may as well be broken out as a separate concept. Another is that choice among events is causal, whereas choice among processes is not. It seems more realistic, at least from a programming language viewpoint, to consider causal choice. For example, it should be possible to *compute* an event dynamically, and then synchronize on it, rather than require it to be determined statically.

2. Channels and references. Channel names are symbols whose purpose is to index open-ended families of operators. As such a channel is not a value, but it may index a family of values, called *references*, throughout the text. Thus, one does not pass a channel *per se*, but rather a reference to a channel, which is a value. This raises the important distinction, present elsewhere in PFPL, between the static and dynamic form of an operation, in this case send and receive. The static form has a manifest channel, the dynamic form determines the channel during execution. These are different, and should be recognized as such.

3. Types. It seems pointless to insist on a type-free message-passing discipline, particularly in the context of a practical programming language. On the contrary, it is enlightening to consider what is an appropriate type for messages even in a stripped-down formalism such as the π-calculus.

Unsurprisingly, the universality of the π-calculus amounts to the implicit use of a recursive type used both positively and negatively.

17 Type Classes

The concept of a *type class* is often associated with the Haskell language, but the concept was introduced by David MacQueen in his design of modules for ML (see Milner et al. (1997) for the history.) As is explained in Chapter 44, a type class is a descriptive signature, one that merely specifies the availability of certain types and operations in a structure (module), whereas an abstraction is prescriptive in that it specifies exactly what is to be visible in a structure.

The term "type class" in the Haskell language refers not only to the descriptive form of signature described in Chapter 44, but also to a means of automatically deriving an instance of it based on a global context of declarations. Instance declarations are simply functor declarations that build instances of a target class from instances of the classes on which it depends. The automatic instantiation mechanism composes such functors to obtain an instance of a type class based solely on the underlying type of the class. Consequently, a type can instantiate a class in at most one way, so that, for example, the naturally numbers can be linearly ordered in exactly one way in order to suit the derivation mechanism. Dreyer et al. (2007) separates the declaration of an instance from its use in a particular derivation scenario, allowing for a smooth integration of modules and type classes.

18 Symbols and References

One of the innovations of PFPL is the consolidation of a number of seemingly disparate notions by breaking out the concept of symbols for their applications to the semantics of various language constructs. Symbols are used in their own right as atoms whose identity can be compared to a given atom. They are used as fluid-bound identifiers and as assignables by a finite mapping associating values to identifiers. They are used as dynamically generated classifiers to enforce confidentiality and integrity of data in a program. And they are used as channels for synchronized message-passing in concurent programs. The commonality is simply this: symbols support *open-ended indexing* of families of related operators. For example, there is one `get` and `set` operator for each assignable, and we may allocate new assignables at will, implicitly giving rise to a new pair of get and set operators. The situation is similar for each of the language concepts just mentioned.

Symbols are central to the uniform treatment of references throughout the book. Rather than being tied to mutation, the concept of a reference appears in each of the applications of symbols mentioned above. In each case the primitive operations of a construct are indexed by statically known (explicitly given) symbols. For example, in the case of mutation, there are `get` and `set` operations

for each assignable, and in the case of dynamic classification, the introductory form is indexed by the class name, and the elimination form is similarly indexed by a particular class. The purpose of references is to extend these operations to situations in which the relevant symbol is not statically apparent, but is instead computed at run-time. So, there is a `getref` operation that takes as argument a reference to an assignable; once the referent is known, the `getref` steps to the `get` for that assignable. Classes, channels, and so forth are handed similarly. *References having nothing to do with mutation.* Rather, references are a means of deferring the index of an operation until execution time. Thus, symbols are not values, but references to them are.

The dynamics of constructs that involve symbols involves an explicit association of types to the active symbols whose interpretation is determined by the construct itself.[2] To achieve this requires that the dynamics maintain the signature of active symbols, which in turn requires that some type information be made explicit in the language. This can lead to some technical complications, but overall it is preferable to omitting it, because the information cannot otherwise be recovered if it were omitted.

Another reason to maintain the signature in the dynamics is that doing so facilitates the identification of states with processes in process calculus. For example, in the case of mutation, the signature specifies the active assignables, which are themselves modeled as processes executing concurrently with the main program, responding to `get` and `set` requests on their channel. Besides being conceptually appealing, using process notation facilitates a *substructural* formulation of the dynamics in which only the active parts of the state are mentioned explicitly, the remaining being "framed in" using the usual rules of process calculus.

19 Type Inference

I am often asked why I do not discuss type inference. The short answer is that type inference is a matter of implementation, not semantics, and does not belong in a book about the genetics of programming languages. PFPL is intended as but the first part of a Knuthian effort to give a precise formulation to both the semantics and the implementation of programming languages. The second, *Principled Implementation of Programming Languages*, or *PIPL*, concerns the systematic transformation of usable source languages into directly implementable target languages using type-directed and type-preserving transformations throughout.

Early drafts of PFPL included material on specifying concrete syntax and transforming concrete into abstract syntax, the formalism of abstract binding trees being the outermost classification of phrases according to type structure. These drafts also included chapters on type inference, pattern matching and compilation, and on garbage collection, all of which have been suppressed

[2] For example, in the case of mutation, the type associated to an assignable is the type of its associated value.

from the published editions of PFPL. Instead, these will become part of the planned PIPL, where they properly belong.

As to the question of type inference, it is important to distinguish the associated concept of a type scheme from the more fundamental concept of a polymorphic type. The main idea of type inference is to define a non-deterministic *elaboration* of a source language in which types may be omitted to a target language in which they may not. The elaboration process "guesses" the missing information in such a way that it results in a well-typed program (Milner et al., 1997; Harper and Stone, 2000). Each such elaboration is said to be an *instance* of the original, partially-typed program. To avoid repetitious elaboration of definitions on each use, it is typical to characterize the entire class of instances using a *type scheme*, which in simple cases resembles a polymorphic type in that it has type variables representing unconstrained guesses. The type scheme is usually derived by a form of unification, which finds the most general, *principal*, solution to a system of constraints that gives rise to the same class of instances. The principal type scheme is used to characterize instances of a defined identifier.

In simple cases type schemes can resemble polymorphic types with prefix type quantification, but this coincidence is deceiving. In truth a type scheme is an elaboration-time mechanism that has little or no semantic consequences. Put another way, even if the target language is as polymorphic as one may like, it is nevertheless necessary to employ a separate notion of type scheme to manage type inference during elaboration. The two concepts are entirely separable, and one does not replace the other. Thus, in addition to general considerations of separation of concerns, it would be misleading to discuss type inference and type schemes in PFPL. It would only invite confusion with polymorphism, which deserves careful consideration in its own right.

20 Types and Processes

The treatment of concurrency is divided into two parts: Chapter 39 introduces abstract process calculus, and Chapter 40 incorporates concurrency into Modernized Algol. The central theme is that *concurrency is entirely a matter of indeterminacy*. Although this is a commonly accepted idea, the usual treatments of concurrency involve quite a lot more than just that. Why is that?

In Chapter 39 I develop Milner-style process calculus, starting with a simple synchronization calculus based on *signals*. Two processes that, respectively and simultaneously, assert and query a signal can synchronize on their complementary actions to take a silent action, a true computation step. The formulation follows Milner is using labeled transition systems to express both the steps of computation as silent actions,[3] and the willingness to assert or query a signal as polarized transitions labeled by that action. This approach is taken directly from Milner, except that instead of having "names" and "co-names", I

[3] Milner called these τ-transitions; I call them ϵ-transitions, as they are called in automata theory.

instead have two forms of action indexed by the signal name.

The formulation differs from Milner's in that I draw a distinction between processes and events. An event is a sum of atomic events, which are continuations conditioned on an action (assert or query a signal). A process is a parallel composition of atomic processes, the most important of which is synchronization on an event. Making the separation avoids condundrums in Milner's formalism, such as intermixing of choice and parallel composition, that need not be considered in a programming language context. In effect all choices are causal in that they are mediated by the possibility of taking an action.

One may then discuss the declaration of new signals, but no interesting issues arise until such time as message passing is available. So the next order of business is to generalize signals to *channels*, which carry data. Doing so introduces the fundamental asymmetry between sending and receiving, whereas contrarily assertion and querying are fully symmetric (which is which is a matter of convention). Sender continuations[4] are given no data on invocation, whereas receiver continuations are given the received datum.

In a departure from conventional accounts of process calculus, such as the π-calculus, I immediately address the question of the *type* of data that can be passed along a channel. When channels are declared, the type of their data values must be declared, and is fixed for the lifetime of the channel. This is no restriction, because sums are always present in any civilized language, allowing one to pass values of various types along the same channel using mechanisms already in place for other reasons. An interesting case is the possibility of there being a type of *channel references*, which are directly analogous to *assignable references* as they arise in Modernized Algol. By passing channel references one may change the communication topology of a collection of processes. This raises the concept of *scope extrusion*, the distinction between scoped and (scope-)free channels.

The final step in the development of process calculus is the realization that *channels are nothing but dynamic classes* in the sense of Chapter 33. There is only one shared communication medium (the "ether" if you will) on which one communicates dynamically classified values. The classifiers identify the "channel" on which the message is sent, and its associated data constitutes the data passed on that "channel." By controlling which processes have access to the constructor and destructor for a class one can enforce the integrity and confidentiality, respectively, of a message. *None of this has anything to do with concurrency.* Indeed, the entire matter is but one application of dynamic classification. With that in hand, all that is left of process calculus is parallel composition, which is indeterminacy of evaluation, and the transmission of messages on the medium.

Chapter 40 is devoted to the integration of these ideas into a programming language, called Concurrent Algol, an imperative language with a modal distinction between expressions and commands. Because assignables are definable using processes, the commands are entirely concerned with dynamic

[4]Present only in the synchronous calculus; there are none in the asynchronous calculus.

classification and synchronization. Communication is asychronous, because the synchronous case is easily definable using a callback channel; it requires no mechanism, such as a message queue, to mediate its behavior. I consider two formulations of synchronization, *non-selective* and *selective*. Non-selective communication resembles the behavior of an ethernet transceiver: packets are drawn from the ether, and are passed to the receiving process to dispatch on their channels and payloads. A typical, but by no means forced, pattern of communication is to handle packets that one recognizes, and to re-emit those that one does not, exactly in the manner of the aforementioned transceiver. Selective communication integrates dynamic class matching so that the only packets received are those with a known channel, and correspondingly known type of payload. Busy-waiting is thereby avoided.

An important innovation is the formulation and proof of type safety for Concurrent Algol. The type system is structural; it makes no attempt to ensure the absence of deadlock, nor should it.[5] Consequently, one must formulate the progress property carefully to allow for the possibility of a process that is willing to communicate, but is unable to do so for lack of a matching partner process. This is very neatly stated using labeled transitions: *a well-typed process that is not the terminal process is always capable of undertaking an action*. Spelled out, a well-typed, non-terminal process may either *take a step* of computation (including a synchronization), or be *capable of undertaking* an action (either a send or a receive). Theorem 40.3, which is paradigmatic for any concurrent programming language, concisely summarizes the desired safety property using labeled transitions.

References

Derek Dreyer, Robert Harper, and Manuel M. T. Chakravarty. Modular type classes. In *Proc 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, January 2007.

S.J. Gould. *Hen's Teeth and Horse's Toes*. Norton, 1983. ISBN 9780393311037. URL `https://books.google.ca/books?id=EPh9jD0XwR4C`.

Robert Harper and Christopher A. Stone. A type-theoretic interpretation of standard ML. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 341–388. The MIT Press, 2000. ISBN 978-0-262-16188-6.

Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[5] This would be a good use for type refinements, which are discussed in Chapter 25. Deadlock is a matter of program *behavior*, not program *structure*.

Frank Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. In C.Benzmüller, C.Brown, J.Siekmann, and R.Statman, editors, *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*, Studies in Logic 17, pages 303–338. College Publications, 2008.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

John C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.