Commentary on Practical Foundations for Programming Languages (Second Edition)

Robert Harper
Carnegie Mellon University

January 16, 2018

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.

Contents

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 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 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 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 my 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 I chose not to fudge the details.

5 Functions First, or Not

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. 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 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. On the other hand it is rather appealing to relate circular dependencies in a circuit to the fixed point theory of programs, and to show that this is what enables the passage from the analog to the digital domain. Without feedback, there is no state, without state, there is no computation.

6 The Empty and Unit Types

One 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 void type inevitably arises as the case of zero summands. By the type safety theorem there are no values of type `void`; it is quite literally void of elements, and may therefore be said to be empty. It follows that, in a total language, there can be no closed expressions of type `void`, and, in a partial language, that any such closed expression must diverge. Otherwise the expression would have to evaluate to a value fo `void`, and that it cannot do.

The natural elimination form for an n-ary sum is an n-ary case analysis, which provides one branch for each of the n summands. When $n = 0$ this would be a *nullary case analysis* of the form $\texttt{case}\, e\, \{\,\}$, which evaluates e and branches on each of the zero possible outcomes. Regrettably, the standard notation used for a nullary case is $\texttt{abort}(e)$, which plainly suggests that it would induce some form of error during evaluation. But that is not at all the case! The unhappy choice of notation confuses many readers, and it is best to avoid it in favor of the more plainly self-explanatory nullary case notation.

Because it is empty, an important use of the type `void` is to state that some expression must not return to the point at which it is evaluated. A particularly common use of `void` is in a type $\tau \rightharpoonup \texttt{void}$, which classifies functions that, once applied, *do not return* a value to the caller (by diverging, by raising an exception, or throwing to a continuation). That type is quite different from the type $\tau \rightharpoonup \texttt{unit}$, which, if it returns, returns the empty tuple (but might engender effects during execution). To be emphatic, *not returning a value* is different from *returning an uninteresting value*, yet, bizarrely, the industry standard is to confuse the two situations! Specifically, it is standard practice to write `void` for what is properly called `unit`, and having no proper `void` type at all! Thus,

in that notation, a function of type function of type $\tau \rightharpoonup \texttt{void}$ may well return to its caller, and there is no way to notate a function that definitely does not return. Words matter; reduction of expressiveness matters even more.

7 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 form of pattern matching, albeit without of exhaustiveness or coverage checking), dynamically typed languages (see Section 14), 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 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 modular program development. This leads to an emphasis on "just in time" compilation, another word for whole-program compilation, to defer code generation until the class hierarchy is known.

8 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 $\texttt{nat} \rightarrow \texttt{bool}$), 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.

9 Recursion in 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.

Separating self-reference from functions emphasizes its generality, and helps clarify a common misconception about recursive (self-referential) functions. No stack is required to implement self-reference. (Consider, for example, that a flip-flop is a recursive network of logic gates with no stack involved.) Stacks are used to manage control flow, regardless of whether function calls or recursion are involved; see Chapter 28 for a detailed account. The need for a stack to implement function calls themselves is a matter of *re-entrancy*, not recursion. Each application of a function instantiates its parameters afresh, and some means is required to ensure that these instances of the function are not confused when more than one application is active simultaneously. It is re-entrancy that gives rise to the need for additional storage, not recursion itself.

10 Parallelism

The treatment of fork-join parallelism[1] distinguishes two cases, the *static* and *dynamic*, according to when the degree of parallelism is determined. The static case is formulated using the definition construct $\texttt{par}\, e_1 = e_2 \,\texttt{and}\, x_1 = x_2 \,\texttt{in}\, e$, which substitutes the values of e_1 and e_2 for x_1 and x_2, respectively, within e. The binary form may easily be generalized to any statically determined degree

[1] Futures are another matter. They can be used to implement fork-join, and they also support pipelining, so they are more general. But they are also unstructured, impeding the formulation of a cost semantics.

of parallelism. Conversely, any static degree of parallelism may be realized by a cascade of binary splits. The dynamic case is formulated using a sequence generator with which is created a sequence of values whose length and content is not determined until run-time.

Although there is nothing wrong with this formulation, there is nothing especially right about it either. The parallel definition mechanism feels rather *ad hoc*, as does the sequence tabulation mechanism. It would be preferable if these, or similar, constructs could arise from general semantic considerations, rather than simply imposed from the outside, as it were, to achieve our ends. Moreover, the asymmetry between the static and dynamic cases seems suspect. Why is there not a symmetric treatment of both cases?

There is, but to see it requires taking a step back to think about the nature of parallelism. When formulating a parallel language the goal is not to somehow *impose* parallelism by some means not otherwise available, but rather to *harness* the parallelism that is already present. Functional languages (rather, those with an eager dynamics) give rise to parallelism without even trying. For example, to evaluate the sum $e_1 + e_2$, we need, in general, to have evaluated both e_1 and e_2, but nothing constraints how or when that may happen. All that is necessary is to express the dependency of the sum on the summands, and the parallelism will take care of itself. Thus, the essence of parallelism is not with introducing "simultaneity" or "concurrency" into a language (it is already there), but rather with expressing dependencies among computations. Oddly enough, *the essence of parallelism is sequencing*.

The best way to express the required sequencing is to draw a modal distinction between two forms of expression: *values*, which have already been evaluated, and *computations*, which have yet to be evaluated. Crucially, handling a value imposes no cost, but evaluating a computation takes work. The value/computation distinction is broadly similar to the expression/command distinction in Modernized Algol. Both have their roots in a formalism called *lax logic*, the core logic of dependency. The syntactic skeleton of a modally separated value and computation language looks like this:

$$\begin{array}{rcl} v & ::= & x \mid \overline{n} \mid \lambda\,(x:\tau)\,e \mid \langle v_1, v_2\rangle \mid \mid \ldots \\ e & ::= & \mathtt{ret}(v) \mid \mathtt{seq}(v; x.e) \mid v_1(v_2) \mid \ldots \end{array}$$

The statics defines the judgments $\Gamma \vdash v : \tau$ and $\Gamma \vdash e \div \tau$. The dynamics defines the judgment $e \Downarrow^c v$, where c is a cost graph.

The simplest computation, $\mathtt{ret}(v)$, simply returns the given value v, incurring unit cost for the service. One computation may be sequenced prior to another using the $\mathtt{comp}(\tau)$ type. Value of this type are suspended computations, $\mathtt{comp}(e)$, which are forced by the sequencing elimination form, $\mathtt{seq}(v; x.e)$. Thus,

$$\frac{}{\mathtt{comp}(e)\ \mathsf{val}} \qquad \frac{}{\mathtt{ret}(v) \Downarrow^{\mathbf{1}} v} \qquad \frac{e_1 \Downarrow^{c_1} v_1 \quad [v_1/x]e_2 \Downarrow^{c_2} v_2}{\mathtt{seq}(\mathtt{comp}(e_1); x.e_2) \Downarrow^{c_1 \oplus c_2} v_2}$$

In this setup one computation may depend on only one other, which ensures that the work and the span of a computation coincide, eliminating parallelism.

The solution is to generalize the sequencing construct to admit more than one dependent. Let us consider first the static case, in which one computation may depend on the values of some fixed number of other computations. This can be represented by generalizing the suspended computation type to a form of *lazy product type* whose values are tuples of unevaluated computations, and whose elimination form sequences the evaluation of the components of such a tuple before that of another computation. The sequencing is expressed using an *eager product type* whose values are tuples of values and whose elimination form is pattern matching. Ordinarily, the elimination forms for a lazy product type would be projections, so as to allow one component to be accessed without disturbing the other. Contrarily, we demand that all components of a lazy product be evaluated whenever it is used, and these evaluations are parallel because nothing precludes it being so.

Thus we have the following statics for lazy products, viewed as a generalization of delayed computation types:

$$\frac{\Gamma \vdash e_1 \div \tau_1 \quad \Gamma \vdash e_2 \div \tau_2}{\Gamma \vdash e_1 \mid\mid e_2 : \tau_1 \mathbin{\&} \tau_2}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \mathbin{\&} \tau_2 \quad \Gamma, x : \tau_1 \times \tau_2 \vdash e_2 \div \tau_2}{\Gamma \vdash \mathtt{parseq}(v_1; x.e_2) \div \tau_2}$$

Notice the interplay between the lazy and eager product types in the elimination rule! The first argument is required to be a (static) value, the second is an abstraction whose body is a general computation. The corresponding dynamics is given as follows:

$$\frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [\langle v_1, v_2 \rangle / x]e \Downarrow^{c} v}{\mathtt{parseq}(e_1 \mid\mid e_2; x.e) \Downarrow^{(c_1 \otimes c_2) \oplus c} v}$$

Thus, it is the elimination form for the parallel pair that induces the evaluation of the two components of the pair, expressing the dependency of its body on their values.

In the dynamic case, wherein the number of dependents is not determined until evaluation time, a similar method applies. The analogue of lazy tuples is a *sequence generator* which determines the length of the sequence and the value of each of its elements. The elimination form for generators creates an *eager sequence* whose size and elements are already determined. The statics mimics that for lazy tuples, albeit with sequences playing the role of eager tuples:

$$\frac{\Gamma \vdash v_1 : \mathtt{nat} \quad \Gamma, x : \mathtt{nat} \vdash e_2 \div \tau}{\Gamma \vdash \mid\mid\{v_1\}(x.e_2) : \tau\, \mathtt{pargen}}$$

$$\frac{\Gamma \vdash v_1 : \tau\, \mathtt{pargen} \quad \Gamma, x : \tau\, \mathtt{seq} \vdash e_2 \div \tau_2}{\Gamma \vdash \mathtt{genseq}(v_1; x.e_2) \div \tau_2}$$

The dynamics is also similar to that of lazy products:

$$\frac{[\overline{0}/y]e \Downarrow^{c_0} v_0 \quad \ldots \quad [\overline{n-1}/y]e \Downarrow^{c_{n-1}} v_{n-1} \quad [<v_0,\ldots,v_{n-1}>/x]e' \Downarrow^{c'} v'}{\texttt{genseq}(||\{\overline{n}\}(y.e); x.e') \Downarrow^{(c_0 \otimes \ldots \otimes c_{n-1}) \oplus c'} v'}$$

The generator is used to create a sequence of the specified length, which is then propagated to the dependent as a value of eager product type.

The modal distinction also resolves a technical problem with cost semantics, namely the need to distinguish between an already-computed value, and a computation that happens to be a value. Without the distinction there is an ambiguity about how to treat, for example, $\langle e_1, e_2 \rangle$, when both components turn out to already be values. Should one consider that the pair is already evaluated, because the components are values, or should one evaluate it in ignorance to derive that it is its own value? Either way, the cost is calculated incorrectly. For if you never evaluate it at all, then you never incur the cost for allocating it in the first place. But if you evaluate it every time you see it, then you repeatedly incur the cost of evaluation, which can be asymptotically significant when the values involved are not of a statically known size. The modal distinction resolves the ambiguity by distinguishing already-evaluated values from computations that happen to evaluate to themselves (rather, their correlates as values). Thus, the trivial computation, $\texttt{ret}(\langle v_1, v_2 \rangle)$, is distinguished from the sequential composition $\texttt{parseq}(\texttt{ret}(v_1) \mid\mid \texttt{ret}(v_2); x.\texttt{ret}(x))$.

11 Laziness and Eagerness

The distinction between laziness and eagerness is best explained in terms of the semantics of variables: do they range over all expressions of their type, or all values of their type? When all all expressions have a unique value, the distinction seldom matters. But for languages that admit divergent, or otherwise undefined, expressions, the distinction is important, and changes the character of the language significantly. Whether one is more "mathematical" than the other, as is often claimed, is a matter of debate, because the analogies to mathematics break down in the presence of partiality. Moreover, considerations of efficiency are particular to programming, but play no role in mathematics. Put another way, programs are not just proofs; they have a cost, and we care about it. One might even say that cost is what distinguishes mathematics from computation; we ought not expect two disparate subjects to coincide.

The distinction between laziness and eagerness is not wholly a matter of cost; it is also about expressiveness. In a lazy language *all* expressions—even the divergent ones—are considered to be *values* of their type. This makes it natural to consider general recursion (fixed points) at every type, because even those that diverge are treated as values of their type. Pairing is naturally lazy (neither component is evaluated until it is used), and functions are naturally "call-by-name", because even a divergent argument is regarded as a value. However, for sums the effect of including divergence is disastrous! There are

three booleans, true, false, and divergence, and so we cannot reason by cases in the ordinary sense. The "empty" type has one element, divergence, and sums have divergence as well as labelled, possibly divergent elements of their summands. The natural numbers, viewed as a recursive sum, not only includes divergence, but also an infinite number, an unending stack of successors. Lazy languages do not, and *cannot* have, a type of natural numbers; at the very least divergence is a value, invalidating proofs by mathematical induction. Similarly, such languages have no trees, no enumerations, no lists, no abstract syntax, nor any inductive types whatsoever. Somehow this does not seem to count against their supposedly "mathematical" nature, yet what could be less mathematical than to lack a type of natural numbers?

Rather than force divergence to always be a value, it makes more sense to introduce a *type constructor* whose values are either divergence, or an actual value of the given type. These are the suspension types considered in Chapter 36, which put control into the programmer's hands, rather than imposing it for doctrinal reasons. Doing so allows one to distinguish between the proper type `nat` of natural numbers and either form of the the lazy natural numbers, namely the minimally lazy natural numbers, `nat susp`, which contains the natural numbers plus divergence, and the ultra-lazy natural numbers whose arguments to successors are themselves suspended. Similarly, one can distinguish lists from streams, and consider intermediate variations in which, say, only the tails, or only the heads, of lists are suspended. Lazy languages have types *called* lists, but in no case are they lists!

12 Self-Reference and Suspensions

Suspension types are naturally self-referential, because of the indirection for memoization, so it makes the most sense to make them recursive. General recursion can be implemented using such suspensions, but it makes little sense to insist that recursion incur the overhead of memoization. Instead, one may consider other types whose elements are naturally self-referential, and arrange for that specifically. For example, one may consider that mutual recursion among functions is a primitive notion by introducing the *n-ary λ-abstraction*

$$\lambda^n\{\tau_1;\ldots;\tau_n\}(x, y_1.e_1;\ldots;x, y_n.e_n).$$

The n-ary λ defines n mutually recursive functions, each abstracted on them all collectively as the first argument, x, of each. (Typically x is spelled `this` or `self`, but it is a bad idea to attempt to evade α-conversion in this way.)

The n-ary λ evaluates to an n-tuple of ordinary λ-abstractions, as suggested by the following statics rule, which gives it a product-of-functions type:

$$\frac{\Gamma, x : \langle \tau_1 \rightharpoonup \tau_1', \ldots, \tau_n \rightharpoonup \tau_n' \rangle, y_i : \tau_i \vdash e_i : \tau_i' \quad (1 \leq i \leq n)}{\Gamma \vdash \lambda^n\{\tau_1;\ldots;\tau_n\}(x, y_1.e_1;\ldots;x, y_n.e_n) : \langle \tau_1 \rightharpoonup \tau_1', \ldots, \tau_n \rightharpoonup \tau_n' \rangle}$$

The dynamics implements the self-reference by unrolling:

$$\lambda^n\{\tau_1;\ldots;\tau_n\}(x,y_1.e_1;\ldots;x,y_n.e_n) \longmapsto \langle \lambda\,(y_1:\tau_1)\,e'_1,\ldots,\lambda\,(y_n:\tau_n)\,e'_n\rangle,$$

wherein each e'_i (for $1 \leq i \leq n$) is given by

$$[\lambda^n\{\tau_1;\ldots;\tau_n\}(x,y_1.e_1;\ldots;x,y_n.e_n)/x]e_i.$$

Admittedly, the unrolling dynamics "cheats" by substituting the n-ary abstraction for a variable, even though it is not a value. But it is tantamount to a value in that it evaluates to one (in one step). There is no harm in extending substitution to valuable expressions (those tantamount to a value); it is the general expressions, which may not terminate, that cause trouble.

If you really wish to avoid this generalization, then you can instead regard the n-ary λ-abstraction to be a value of a special tuple-of-function type whose elimination form is project-and-apply to an argument. The dynamics of project-and-apply involves replacing the self-referential parameter by the tuple-of-lambda's, which is now deemed to be a value, and so the problem is evaded.

13 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.

14 Dynamic Typing

The whole of Part IX is devoted to 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 *un-i-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 fragment of your favorite dynamic language, then reformulate it as a static language with a privileged recursive sum type. For example, it is common to consider multi-argument (and even multi-result) functions. Under 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.

As the development in the book makes clear, the term "dynamic typing" is a misnomer—what is called a "type" in dynamic languages is, in fact, a static class. Dynamic type-checking is not *type* checking, it is rather the *class* checking that inheres in having types of classified values. The question remains, however, why restrict attention to a single type? Why willfully ignore the static structure that is unavoidably present in any dynamic language? Why pretend that having multiple arguments is not a concession to there being product types? Why insist that all values be classified? Why forego modular decomposition using types?

The most obvious reason is ignorance. Dynamic typing arose in Lisp long before there was any appreciation for the type structure of programming languages. The appeal of Lisp is powerful, especially if your only other experience is with conventional imperative programming languages. But Lisp's charms are not often separated from their historical context. Though once true, it is true no longer that Lisp is the sole opponent of all other programming languages. And so it remains unrecognized that Lisp, and its descendants, are statically typed languages after all, despite superficial appearances. Even after decades of experience, (recursive) sum types are unfamiliar, even unimagined, by many advocates of dynamic languages. Absent a proper understanding of recursive sums, dynamic typing seems appealing—precisely because it offers sums in the implicit form of run-time class checks. But dynamic typing robs the programmer of the all-important exhaustiveness check afforded by case analysis.

Another reason for preferring dynamic languages is their immediacy. It is infamous that every Lisp program does *something*, as long as the parentheses match. It is consequently very easy to write code that does something vaguely right, and to adopt a development strategy that amounts to debugging a blank

screen: get something going, then hack it into existence. Richly typed languages, on the other hand, impose the discipline that your code make minimal type sense, and demand that you consider the types of values in play. Opponents argue that the type system "gets in their way" but experience shows that this is hardly the case. Innovations such as polymorphic type inference make a mockery of the supposed inconveniences of a rich type discipline.

It seems to me that the most sensible argument for dynamic typing is the claim that behavioral typing subsumes structural typing. Behavioral typing, the argument goes, is more expressive, and can account for richer specifications than common structural type systems. The argument is not without merit. But it does not address the fundamental role of types, which is to express modular structure. Modularity is not a matter of behavior, it is a matter of protocol, a discipline imposed by the programmer on herself to ensure that programs decompose into neatly separable modules. Behavioral typing does not address such considerations, and so remains complementary to structural typing, rather than an alternative to it. In any case most dynamic languages in common use lack even a rudimentary behavioral type system, so the argument remains hypothetical.

15 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 that provide a modicum of tracking ability. Conditionals invariably attenuate what can be statically tracked, and are especially problematic when the subtype relation lacks meets, as it does in many popular languages.

16 Dynamic Dispatch

The discussion of dynamic dispatch is meant to address one of the more prominent aspects of object-oriented programming.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 cannot 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 seems to require structural subtyping, which may be taken as an indication of its dubious advisability. 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.

17 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 have 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

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

which only the active parts of the state are mentioned explicitly, the remaining being "framed in" using the usual rules of process calculus.

18 Sorts of Symbols

As discussed in Section 17 symbols in *PFPL* are used for open-ended indexing of families of operators. As new symbols are introduced, new instances of these families become available. For example, if symbols are being used as names of assignables, then the introduction of a new symbol, a with associated type τ, gives rise to the expression $\&[a]$, a reference to the assignable a, which is a value of the type $\tau\,\texttt{ref}$ of assignable references. The behavior of the dereferencing operations, $*e$ and $e_1 *= e_2$, is determined by the behavior of their underlying operations on assignables, $@a$ and $a := e$. Similar concepts, such as Lisp-like symbolic data, dynamic classification, and communication channels, are handled by similar means.

The uniform treatment of operator indexing using symbols consolidates a number of otherwise disparate, but closely related, concepts, by separating the treatment of symbols *per se* from their application as assignables, classes, channels, and so forth. A shortcoming of their formulation in *PFPL*, though, is that I have only allowed for one "sort" of symbols to be active at a time. This immediately becomes problematic when we wish to have, say, assignables and dynamic exceptions, in play at the same time. Obviously not ever assignable should be construed as also being an exception name, nor every exception name as that of an assignable. What is required, in general, is to admit there to be several disjoint signatures in use at a time, each declaring a collection of active symbols and associating a type with each of them. Thus, in the case of assignables and exceptions, one would have a signature $\Sigma_{\texttt{asgn}}$ of assignables and a signature $\Sigma_{\texttt{excn}}$ of exception names, each governing its own realm independently of the other. Alternatively, one could introduce a notion of *species* for symbols, and associate a species, as well as a type, to each symbol in the signature. In the example the species would be `asgn` for assignables and `excn` for exception names.

To make this work the symbol allocation construct would have to be indexed by the sort of symbol being allocated so that it is added to the appropriate signature (or marked with the appropriate sort). This further suggests that symbol allocation should itself be broken out as a basic concept of syntax shared by all languages that make use of symbols. The issue of whether to consider scoped or scope-free allocation becomes one of the choice of structural congruence, the difference being the absence or presence, respectively, of Milner's scope extrusion principle. It is up to the dynamics of a specific language to determine when, if ever, the scope of symbol may be exited. At the least one must demand that the transition outcome be independent of the allocated symbol for the exit to be sensible. Mobility requirements in the statics ensure that the scope of a declaration may be exited whenever it ought to be in the sense of the concept under consideration.

19 Exceptions

My account of exceptions distinguishes the *control* mechanism from the *data* mechanism. In all languages that I know about these two aspects are not clearly separated, resulting in all manner of confusion. For example, most languages with exceptions have some form of "exception declaration" construct that introduces a mysterious thing called "an exception" that can be "thrown" or "raised" with an associated data value. The raised value can be "handled" by dispatching on the exception, recovering the associated value in the process. Some languages attempt to track the possibility of raising an exception in types, invariably with poor results. One reason for the failure of such methods is that it is not important to track what exceptions *can* be raised (which cannot be safely approximated), but rather what exceptions *cannot* be raised (which can be safely approximated). Worse, many programming methodology sources discourage, or even ban, the use of exceptions in programs, an egregious error based on not understanding the role of secrecy in an exception mechanism.

As regards the control aspect of exceptions, there is no difference between exceptions implemented as non-local control transfers and exceptions implemented using sums. Using sums the value of an exception is *either* a value of its intended type, *or* a value of the exception type that, presumably, provides some indication as to why a value of the intended type cannot be provided. The emphasis on the disjunctive meaning of sums is warranted. Many languages attempt to account for exceptional returns using absurdities such as the non-existent "null pointer" or similar devices whereby a certain range of returned values is to be regarded as having an entirely other meaning from the remaining values. Sums force the client to dispatch on whether the result is ordinary or exceptional, and there can be no ambiguity or confusion, nor can the distinction be elided. This is as it should be. Exceptions simply make this checking easier in cases where the "default" behavior is to propagate the exceptional value, rather than actually do anything with it, the handler being the locus of interpretation of the exceptional behavior. Admittedly most languages, astonishingly, lack sums, but this does not justify claiming that exceptions should also be ignored or repudiated!

The main criticism of exception is the fallacious belief that one cannot tell where the exception is handled, the non-local transfer being supposedly out of the programmer's control, or easily subverted accidentally or on purpose. Nonsense. Choosing the exception value correctly allows the programmer to establish an exception as a shared secret between the raiser and the handler that cannot be subverted. By declaring an exception that is known only to the raiser and handler, one can ensure that they, and only they, communicate with one another via the exception mechanism. The raiser is responsible to ensure the integrity of the exception value (that it satisfies some agreed-upon requirement) and the handler is responsible to ensure the confidentiality (only it can decode the exception value, whose integrity may be assumed). *Exceptions are all about shared secrets.*

Confidentiality, and integrity, are enforced using *dynamic classification*, which is defined in Chapter 33. Exceptions are dynamic classes that may mediate between the "raiser" and the "handler" of an exception. The introductory form of the class is provided only to those permitted to raise an exception of that class, and the eliminatory form only to those permitted to handle it. To all others the exception value is an indecipherable secret that passes through unscathed and uninterpreted. As this discussion makes clear, *exception classes are not fluids*, contrary to wide-spread belief. Although one may use fluid binding to implement the *control* aspect of exceptions, it has no bearing on their *data* aspect.

20 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 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 just 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 $\texttt{cmd}(\tau)$, 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

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

which is, of course, a command.

A subtext of the discussion of Modernized Algol is the remark that *Haskell is but a dialect of Algol*. The main concepts of the Haskell language were already present in Algol in 1960, and further elucidated by Reynolds in the 1970's. In particular so-called monadic separation of commands from expressions was present from the very beginning, as was the use of higher-order functions with a call-by-name evaluation order. The main advance of the Haskell language over Algol is the adoption of recursive sum types from ML, a powerful extension not contemplated in the original. Another significant advance is the elimination of assignables as forms of expression; the idea to make assignables seem sort of like mathematical variables was, in my opinion, an egregious error that is corrected in modern incarnations of the language.

21 Assignables, Variables, and Call-by-Reference

The original sin of high-level languages is the confusion of assignables with variables, allowing one to write formulas such as $a^2 + 2\,a + 1$ even when a is an assignable.[3] To make matters worse, most common languages don't have variables at all, but instead press assignables into service in their stead. It all seems fine, until one considers the algebraic laws governing the formulas involved. Even setting aside issues of machine arithmetic, it is questionable whether the equation $a^2 + 2\,a + 1 = (a+1)^2$ remains valid under the assignable-as-variable convention. For example, concurrency certainly threatens their equivalence, as would exceptions related to machine arithmetic. The strict separation of variables and assignables in Modernized Algol avoids these complications entirely by forcing access to assignables to be made explicit before any calculation can be done with their contents.

It is a good source of exercises to re-formulate a few standard language concepts using this separation. For example, most imperative languages lack any notion of variable, and must therefore regard the arguments to a procedure or function as assignables. It is a good exercise to formulate this convention in Modernized Algol: the argument becomes a variable, as it should be, and the procedure body is surrounded by a declaration of an assignable whose contents is initialized to that variable, with the choice of assignable name being that given as the procedure parameter. The next step is to consider the age-old notion of *call-by-reference*: given that the argument is an assignable, one may consider restricting calls to provide an assignable (rather than an expresion) of suitable type on which the body of the procedure acts directly. The assignable

[3] Indeed, the name "Fortran" for the venerable numerical computation language abbreviates the phrase "Formula Translator."

argument is *renamed* to be the call-site assignable for the duration of the call. From this point of view the standard concept of *call-by-reference* might better be termed *call-by-renaming*: it is more accurate and it even sounds similar!

To be more precise, consider a type of *call-by-renaming procedures*, written $\tau_1 \Rightarrow \tau_2$ whose values are procedures $\rho\{\tau\}(a.m)$, where a is an assignable symbol scoped within the command m. Absent call-by-renaming, such a procedure could be considered to be short-hand for the function

$$\lambda\,(x:\tau)\,\mathtt{cmd}(\mathtt{dcl}\,a := \tau\,\mathtt{in}\,m),$$

where x is a fresh variable, which allocates an assignable initialized to the argument for use within the body of the procedure. With call-by-renaming one must instead regard these procedures as a primitive notion, restrict calls to provide an assignable as argument, and define the action of a call as follows:

$$\mathtt{cbr}\{b\}(\rho\{\tau\}(a.m)) \longmapsto [a \leftrightarrow b]\,m,$$

where a is chose (by renaming of bound symbols) to be fresh. The assignable b must be within scope at the call site, and is provided to the body as argument. Notice that within the command m the assignable a is treated as such; it is *not* regarded as a reference, but rather accessed directly by $@\,a$ and $a := e$, as usual. The renaming on call ensures that these commands act on the passed assignable directly, without indirection.

22 Assignable References and Mutable Objects

In Section 17 the concept of a *reference* is completely separable from that of an assignable. A reference is a means of turning a symbol—of whatever sort—into a value of reference type. It is a level of indirection in the sense that the operations acting on references simply extract the underlying symbol and perform the corresponding operation for that symbol. The available operations on references depend on the sort of the symbol. In the case of assignables the operations include `getref` and `setref`, which code for the `get` and `set` operations associated with the underlying assignable. One may also consider the references admit a *equality test* that determines whether or not they refer to the same underlying symbol. Such a test is definable by simply changing the type of the stored value from τ to $\tau\,\mathtt{opt}$, maintaining the invariant that the underlying value is always of the form $\mathtt{just}(-)$, except during an equality test. To check equality of references, simply save the contents of one reference, then set it to `null`, and check whether the contents of the other is also `null`. If so, they are the same reference, otherwise they are not; be careful to restore the changed value before returning!

Assignable references generalize naturally to *mutable objects*. Think of a mutable object as a collection of assignables (its *instance data*) declared for use within a tuple of functions (its *methods*) acting on those assignables.[4] A refer-

[4]For this one needs free assignables, of course, otherwise the tuple cannot be returned from the scope of the declaration.

ence is a single assignable, say `contents`, that is accessible only to the `getref`, `setref`, and `eq` methods just described. More generally, a mutable object can have any number of private assignables governed by any number of methods that act on them. Thus, *assignable references are a special case of mutable objects*. One may say informally that "mutable objects are references" or speak of "references to mutable objects", but in fact the concept of a mutable object is self-standing and generalizes that of an assignable reference. There is no need to speak of "object references" or "references to objects", because objects are themselves (generalized) references.

23 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.

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. 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.

24 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 in using labeled transition systems to express both the steps of computation as silent actions,[5] and the willingness to assert or query a signal as polarized transitions labeled by that action. Contrary to Milner, instead of having "names" and "co-names", I have two polarities of actions indexed by signal names.

Another difference from Milner is 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

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

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. (I suppose it would be possible to admit spontaneous events, but I have not found a need for it within the context of the book.)

One may then discuss the declaration of new signals, but no interesting issues arise until message passing is introduced. So the next order of business is to generalize signals to *channels*, which carry data, and which break the symmetry between assertion and query of signals. Now channels have a sending and a receiving side, with the sender providing the data and the receiver obtaining it. Channels may be synchronous or asynchronous, according to whether the sender is blocked until a receiver receives the message. The asynchronous form is more general, at least in the presence of channel references, for one can implement a receipt notification protocol corresponding to what is provided by synchronous send.

In the π-calculus the only messages that can be sent along a channel are other channels (or finite sequences of them in the so-called polyadic case). Instead, I choose to examine the question of the types of messages from the outset, arriving at the π-calculus convention as a special case involving recursive and reference types. When channels are declared, the type of their data values must be specified, and remains fixed for the lifetime of the channel. There is no loss of generality in insisting on homogeneous channels, the heterogeneous case being handled using sums, which consolidate messages of disparate types into a single type with multiple classes of values. To mimic the π-calculus, I consider a type of *channel references*, which are analogous to *assignable references* as they arise in **MA**. The process calculus notion of *scope extrusion* captures the distinction between scoped and (scope-)free channels found in **MA**.

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 (**CA**), an imperative language with a modal distinction between expressions and commands. Because assignables are definable using processes, the commands are entirely concerned with dynamic classification and synchronization. Two formulations of synchronization, *non-selective* and *selective*, are considered. 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.

The formulation and proof of type safety for the concurrent language **CA** in, to my knowledge, novel. The type system is structural; it makes no attempt to ensure the absence of deadlock, nor should it.[6] 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.

25 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.

[6] 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*.

26 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 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.

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. What, if anything, is a programming paradigm? Cammbridge University Press Author's Blog, April 2017. URL `http://www.cambridgeblog.org/2017/05/what-if-anything-is-a-programming-paradigm/`.

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.

Shriram Krishnamurthy. Teaching programming languages in a post-linnaean age. In *ACM SIGPLAN Workshop on Undergraduate Programming Language Curricula*, 2008. URL `https://cs.brown.edu/~{}sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/`.

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.

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.