Commentary on Practical Foundations for Programming Languages (Second Edition)*

Robert Harper
Carnegie Mellon University

May 19, 2022

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 borrowed from Thomas Kuhn's *The Structure of Scientific Revolutions*. It is common to classify languages into trendy-sounding categories such as "imperative", "functional", "declarative", "object-oriented", "concurrent", "distributed", and "probabilistic" programming. These classifications are so widespread that I am often asked to justify why I do not adhere to them.

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 in-

stances. 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. It is possible to associate many different notions of cost to a program, such as the sequential or parallel time or space usage of a program. 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.

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 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.

Once partiality is introduced the lazy/eager distinction becomes vitally important. There are two issues to consider:

1. Are variables interpreted *by-name*[1] or *by-value*? That is, do variables range over *computations* or *values*?

2. Are value constructors for positive types *eager* or *lazy*? That is, do constructors compose values from values or from general computations?

The range of significance of a variable is determinative in several respects. First, because the dynamics of application of a λ-abstraction is defined by substitution, by-value variables demand evaluation of the argument before the call, the *call-by-value* interpretation. On the other hand, by-name variables are compatible with evaluating or not evaluating the argument before the call; the latter case is dubbed *call-by-name*. Similarly, because case analysis substitutes a constructor's argument for a variable in each branch, a by-value interpretation of variables requires a strict interpretation of constructors, whereas a by-name interpretation admits either formulation.

For these reasons when partiality is admitted, the statics should reflect the range of significance of variables (over values or over computations), as was done by Levy in his *call-by-push-value* formalism. The idea is to make a *modal* distinction between computations and values linked by a *modality* whose values are unevaluated computations. Variables range over values, and constructors form values from other values. This formulation scales well to account for parallelism by generalizing the unary modality to a family of n-ary modalities for each $n \geq 0$ that create a value from n unevaluated computations that are evaluated simultaneously by the elimination form.

[See also Harper (2019e).]

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 sys-

[1] It may have arisen from the mistaken idea that a suspended computation is a piece of program text.

tems 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 `case` e `{ }`, which evaluates e and branches on each of the zero possible outcomes. Regrettably, the standard notation used for a nullary case is `absurd`(e), which plainly suggests that it would induce some form of error during evaluation. But that is not so! 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 $\tau \rightharpoonup$ `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 $\tau \rightharpoonup$ `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, writing `void` for what is here called `unit`, and having no type corresponding to `void` type at all. Thus, in that notation, a function of type function of type $\tau \rightharpoonup$ `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 General Product and Sum Types

The notation for binary product and sum types is relatively standard. The nullary cases are somewhat less standard, with the most common discrepancy being the use of `void` for the unit type, and the consequent omission of the empty type itself. With the nullary and binary forms in hand, it is possible to encode n-ary forms by, say, right-association:

$$\tau_1 \times \cdots \times \tau_n \triangleq \tau_1 \times (\tau_2 \times (\ldots \times \tau_n))$$
$$\tau_1 + \cdots + \tau_n \triangleq \tau_1 + (\tau_2 + (\ldots + \tau_n))$$

with the $n = 0$ case being the nullary product or sum, and the $n = 1$ case being vacuous. Although helpful in principle, such encodings are unhelpful in practice—the nested associations, and positional notation, are unworkably onerous.

The alternative is to consider indexed n-ary forms as primitive, deriving the nullary forms as the $n = 0$ case, and choosing the indices $1, \ldots, n$ for the

positional notation. Following tradition, PFPL uses the notations $\langle \tau \rangle_{I \in i}$ and $[\tau]_{I \in i}$ for the indexed n-ary products and sums, respectively, with I being the finite set of indices i for the components. This notation is well-and-good by itself, but does not scale when considering indexed product *kinds* whose elements would be indexed n-tuples of constructors, clashing with the tuple-like notation for the product types themselves. The best solution is to generalize the infix product and sum notation to indexed n-ary forms

$$i_1 \hookrightarrow \tau_1 \times \cdots \times i_n \hookrightarrow \tau_n$$

and

$$i_1 \hookrightarrow \tau_1 + \cdots + i_n \hookrightarrow \tau_n.$$

These may be written more concisely as $\langle \tau_i \rangle_{i \in I}$ and $[\tau_i]_{i \in I}$, respectively. In both cases it is clear that a type is meant, and not a tuple of types of product kind.

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 form of pattern matching, albeit without of exhaustiveness or coverage checking), dynamically typed languages (see Section 15), 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.

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 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 about the distinction between partial and total languages. In the context of Chapter 15, it is not possible to write a `filter` function for streams that, given a binary predicate (a function of type `nat` $\to$ `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, or even not exist at all if the predicate is always false! But it is in the nature of a total language to build in the "proof" that it terminates on all inputs. To implement `filter` requires an *unbounded search* operation, using a fixed point construction that sacrifices guaranteed termination.

The definition of positive type operator given in Chapter 14 can be reformulated using hypothetical judgments by maintaining a context Δ of hypotheses of the form t_1 pos, $\ldots$, t_n pos as follows:

$$\frac{}{\Delta \vdash \texttt{unit pos}} \qquad \frac{}{\Delta \vdash \texttt{void pos}}$$

$$\frac{\Delta \vdash \tau_1 \text{ pos} \quad \Delta \vdash \tau_2 \text{ pos}}{\Delta \vdash \tau_1 \times \tau_2 \text{ pos}} \qquad \frac{\Delta \vdash \tau_1 \text{ pos} \quad \Delta \vdash \tau_2 \text{ pos}}{\Delta \vdash \tau_1 + \tau_2 \text{ pos}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ pos}}{\Delta \vdash \tau_1 \to \tau_2 \text{ pos}}$$

The reflexivity and weakening properties of the hypothetical judgment ensure that $\Delta, t \text{ pos} \vdash t \text{ pos}$. But even though $\Delta \vdash \tau \text{ pos}$ implies that $\Delta \vdash$

τ type, the entailment $\Delta, t\ \mathtt{pos} \vdash t\ \mathtt{type}$ is *not* valid. And this is a good thing, because this ensures that the domain of a function type be independent of the positive type variables in scope!

For a given positive type operator $t\,.\,\tau\ \mathtt{pos}$, define $T(\sigma)$ to be the substitution instance $[\sigma/t]\tau$. The inductive and coinductive types may be represented in **F** as follows:

$$\begin{aligned}
\mu(\,t\,.\,\tau\,) &\triangleq \forall(\,r\,.\,(\,T(r) \to r\,) \to r\,)\\
\mathtt{fold}\{t\,.\,\tau\}(\,e\,) &\triangleq \Lambda(\,r\,)\,\lambda\,(\,a : T(r) \to r\,)\,a(\,m(\,e\,)\,),\ \text{where}\\
m(\,e\,) &\triangleq \mathtt{map}\{t\,.\,\tau\}(\,x\,.\,\mathtt{rec}\{t\,.\,\tau\,;t\}(\,x\,.\,a(\,x\,)\,;x\,)\,)(\,e\,)\\
\mathtt{rec}\{t\,.\,\tau\,;\rho\}(\,x\,.\,e'\,;e\,) &\triangleq e[\,\rho\,](\ \lambda\,(\,x : T(\rho)\,)\,e'\,)
\end{aligned}$$

$$\begin{aligned}
\nu(\,t\,.\,\tau\,) &\triangleq \exists(\,s.(\,s \to T(s)\,) \times s\,)\\
\mathtt{unfold}\{t\,.\,\tau\}(\,e\,) &\triangleq\ \mathtt{open}\,e\,\mathtt{as}\,s\,\mathtt{with}\,m\,\mathtt{in}\,g(\,m\,),\ \text{where}\\
g(\,m\,) &\triangleq \mathtt{map}\{t\,.\,\tau\}(\,x\,.\,\mathtt{rec}\{t\,.\,\tau\,;t\}(\,x\,.\,a(\,x\,)\,;x\,)\,)(\,m\,)\\
\mathtt{gen}\{t\,.\,\tau\,;\sigma\}(\,x\,.\,e'\,;e\,) &\triangleq\ \mathtt{pack}\,\sigma\,\mathtt{with}\,\langle\lambda\,(\,x : \sigma\,)\,e', e\rangle\,\mathtt{as}\,\nu(\,t\,.\,\tau\,)
\end{aligned}$$

10 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 refute a widespread misconception about recursive 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.

11 Parallelism

There is an ambiguity in the cost semantics for parallelism regarding the distinction between an expression that is yet to be evaluated, but which happens to be a value, and an expression that has already been evaluated. In a by-value language variables stand for already-evaluated expressions, and hence, when encountered during evaluation, should not incurr additional cost. On the other hand certain expressions, such as $\langle e_1, e_2 \rangle$, where e_1 and e_2 are values, should incur at least the cost of allocation. Never charging for value pairs is wrong, and charging for them each time they are encountered is also wrong (because substitution replicates their occurrence.) The solution is to introduce a modal distinction between computations and values in the statics, and to impose costs only on computations.

[See (Harper, 2018a,b).]

12 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 expressions have a unique value, the distinction seldom matters. But for languages that admit divergent, or otherwise undefined, expressions, the distinction is important, and affects the character of the language significantly. Whether one is more "mathematical" than the other, as is often claimed, is a matter of debate, because analogies to mathematics break down in the presence of partiality. Moreover, considerations of efficiency are fundamental for computation, but play no role in mathematics. Put another way, programs are not just proofs; they have a cost, and it matters. One might even say that cost is what distinguishes mathematics from computation; it is not to be expected that the two disparate subjects 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, even when it diverges. Pairing is lazy (neither component is evaluated until it is used), and functions are "call-by-name", because even a divergent argument is a value. Neither of these is particularly problematic, but for sums the inclusion of divergence is disastrous. For example, there are not two, but *three* booleans, true, false, and divergence; even worse, they are not distinguishable by case analysis because it is impossible to detect divergence. Absurdly, the "empty" type has one element, namely divergence, and the "unit" type has two elements. Worse, besides divergence, sums also include injections of divergent values from the summands. It is possible to introduce "strict sums" that rule out injected divergence, but divergence itself is still present. Similarly, the so-called natural numbers, viewed as a recursive sum type, not only include divergence, but also an unavoidable infinite stack of successors. One may again consider a strict version of natural numbers, which rules out infinity, but in any case includes divergence. Consequently, reasoning by mathematical induction is *never valid* for a lazy language. What could be *less* mathematical than to lack a type of natural numbers?

The problem is the very idea of a lazy language, which imposes a ruinous semantics on types. An eager semantics supports the expected semantics of types (for example, the natural numbers are the natural numbers) and, moreover, admits definition of the lazy forms of these types using suspension types (Chapter 36). A value of a suspension type is either a value of the underlying type, or a computation of such a value that may diverge. There being no representation of the eager types in a lazy language, it follows that, *eager languages are strictly more expressive than lazy languages.*

[See Harper (2019e).]

13 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:

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

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

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

The foregoing 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 (ones that have a value); it is the *non-valuable* expressions, which may not terminate, that cause trouble.

If you really wish to avoid this generalization, then you may instead regard the n-ary λ-abstraction to be a value of a tuple-of-functions type whose elimination form projects a function from the tuple and applies it to an argument. By deeming a tuple-of-λ's as a value the minor cheat in the dynamics of unrolling is evaded. A specialized tuple-of-functions type is sometimes called an *object type*, whose self-referential components are then called *methods*.

14 Recursive Types

Recursive types (with no positivity restriction) only make sense for a language with partiality, because they allow the definition of self-referential

expressions that diverge when evaluated. 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 these strategies is central to the representation of 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 always defined structurally, and may be depicted using "box and pointer" diagrams. But this excludes important structures involving laziness or functions, which are not representable in such a simplistic manner. 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 more general than a memory address.

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.

As a technical matter, the statics for the `fold` operation of recursive types requires that $\texttt{rec}\,t\,\texttt{is}\,\tau\ \mathsf{type}$, but for the premise to be properly formulated demands that $[\texttt{rec}\,t\,\texttt{is}\,\tau/t]\tau\ \mathsf{type}$. But if the former holds, then, by induction on the formation rules for types, it follows that $t\ \mathsf{type} \vdash \tau\ \mathsf{type}$, from which the formation of the unrolling follows by substitution. Alternatively the formation rule for recursive type may be formulated using the hypothetical judgment

$$\texttt{rec}\,t\,\texttt{is}\,\tau\ \mathsf{type} \vdash [\texttt{rec}\,t\,\texttt{is}\,\tau/t]\tau\ \mathsf{type},$$

from which the unconditional formation of the unrolling follows. The bound variable t in $\texttt{rec}\,t\,\texttt{is}\,\tau$ is not so much a variable, but a "back pointer" to the recursive type itself. The alternative formulation captures this interpretation more accurately than does requiring that $t\,.\,\tau$ be a well-formed type operator.

15 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 *uni-typed* (using 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.

From this perspective a dynamic language is a static language that confines its attention to this one distinguished type. Yet in practice few dynamic languages adhere to this dogma. For example, such languages invariably admit multiple arguments to functions, and often admit multiple results as well. But this is none other than a concession to static typing! The domains and ranges of such functions are finite products, or perhaps lists, of the distinguished type, rather than single values of that type. So, multiple types are essential, but why stop with just these?

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 imperative, especially object-oriented, 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 appearances and vigorous protestation by advocates. 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 expression 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 sort-of 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.

16 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, $\lambda\,(\,x_1 : \tau_1\,)\,e(\,x_1\,)$, would, given 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, the m-tuple consisting of the first $m \leq n$ projections from e provides a narrower view of it.

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 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 is done in Chapter 25 with 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 joins, as it does in many popular languages.

17 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 misleading 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 Abelson 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 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. In particular, the isomorphism

$$\prod_{c \in C} \prod_{d \in D} (\tau^c \to \rho_d) \cong \left(\sum_{c \in C} \tau^c \right) \to \left(\prod_{d \in D} \rho_d \right)$$

states that a dispatch matrix determines and is determined by a mapping from the "row space" of instances of the classes to the "column space" of behaviors of the methods, much as in linear algebra.

[See also Harper (2019c).]

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 from use in 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 new assignables may be allocated 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.

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. 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 "local" 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.

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

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 that seems to arise from not appreciating 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 "null pointer," a distinguished value whose meaning is different from that of every other value of its type. Instead of reserving a certain value to have a certain meaning, a sum type expands the range of values to include an additional "special" value that is unambiguously different from an "ordinary" value. Using sums forces the client to dispatch on whether the result is ordinary or exceptional, as it should do in such a situation. 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 lack sums, but this does not justify claiming that exceptions should be ignored or repudiated!

A common criticism of exceptions is that one cannot tell where an exception is handled, the non-local transfer being said to be out of the programmer's control, or easily subverted, accidentally or on purpose. Nonsense. Dynamic classification allows one to ensure that each exception is a

"shared secret" between a raiser and a handler that cannot be subverted. 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). In short *exceptions are shared secrets.*

[See Harper (2019d).]

20 Modalities and Monads in Algol

In 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 long-coming, 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.

Implicit in the discussion of Modernized Algol is the observation that *the Haskell language is 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.

[See also Harper (2019e) and Harper (2018c).]

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 arising from machine arithmetic. The strict separation of variables and assignables in Modernized Algol avoids these complications 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 instructive to formulate this

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

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. Another exercise 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 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*.

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

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

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:

$$(\ \rho\,(\, a : \tau \,)\, m\,)(\, b \,) \longmapsto a \leftrightarrow bm,$$

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 18 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 τ `opt`, maintaining the invariant that the underlying value is always of the form `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 reference 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 the concept of a mutable object is self-standing and generalizes that of an assignable reference.

23 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

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

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.

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.* With that in hand, all that is left of process calculus is non-deterministic composition, and the transmission of messages on the medium.

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

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. 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 be *capable of taking a step*, or be *capable of undertaking an action* in coordination with another process. Theorem 40.3 summarizes the desired safety property using labeled transitions.

[See Harper (2019b) and Harper (2019a).]

24 Type Classes

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 signature from instances of the signatures on which it depends. The automatic instantiation mechanism composes such functors to obtain instances of a signature based on circumstantial evidence. The techniques used rely on such suppositions as there being at most one linear ordering for a given type, which is patently false and a common source of trouble. Dreyer et al. (2007) separates the declaration of an instance from its

use in a particular situation, which avoids unnecessary suppositions and supports the smooth integration of the module mechanisms described in the text.

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. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

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

Robert Harper. Types and parallelism. Supplement to Harper (2016), Fall 2018a. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/par.pdf`.

Robert Harper. Relating transition and cost semantics for parallel pcf. Supplement to Harper (2016), Fall 2018b. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/parsos.pdf`.

Robert Harper. Stacks by-name and by-value. Supplement to Harper (2016), Fall 2018c. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/stacks.pdf`.

Robert Harper. Automata and concurrency. Supplement to Harper (2016), Fall 2019a. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/automata.pdf`.

Robert Harper. Concurrent Algol, Modernized. Supplement to Harper (2016), Fall 2019b. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/mca.pdf`.

Robert Harper. Dynamic dispatch as an abstract type. Supplement to Harper (2016), Fall 2019c. URL https://www.cs.cmu.edu/~rwh/pfpl/supplements/ddadt.pdf.

Robert Harper. Exceptions: Control and data. Supplement to Harper (2016), Fall 2019d. URL https://www.cs.cmu.edu/~rwh/pfpl/supplements/exceptions.pdf.

Robert Harper. PCF-By-Value. Supplement to Harper (2016), Fall 2019e. URL https://www.cs.cmu.edu/~rwh/pfpl/supplements/pcfv.pdf.

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.