Why do languages support structured data and abstraction mechanisms?

1 The Pragmatics Question

Although the *Core* language can express all functions that Church and Turing consider computable, you would have a hard time writing a program that allows people to manipulate geometric shapes in the two-dimensional plane. Consider the simplest shape of all: a Cartesian point. You remember that you need two numbers to describe the location of Cartesian points: the x coordinate and the y coordinate. But, all you have as a *Core* programmer are plain numbers. So now you wonder whether

> it is possible to represent the two coordinates of a Cartesian point with a single number.

While the answer is affirmative, it won't satisfy you. You will need a lot of code to combine the two numbers into one; you need even more code to extract the two coordinates from such a number; and you need a few other ways of manipulating such Cartesian coordinates.

Worse, when you look at a numeric constant in your program, you just don't know whether it represents this number or a coordinate. This observation has tremendous implications for code that must represent Cartesian coordinates in this manner. Imagine that your code treats a single number as the representation of a coordinate instead of a number representing a distance between two points. Or, your code may deal with a number that represents a distance as a coordinate. Since numbers come without any marker that classifies them as either plain numbers or representations of Cartesian coordinates, you cannot equip your code with appropriate checks. The best you can do is to create a rigorous and comprehensive test suite that ensures

that everything works properly. And if it doesn't, looking for a bug will be extremely painful.

This scenario reveals another problem with the *Core* language, namely, the complete lack of an abstraction facility. Clearly, the operations for turning two coordinates into a single number and for extract the x or y coordinates from a single number are going to show up all over a program. Even if we assume that they are short calculations—one-liners if we allow nested expressions—every reader of the program must assign a name to these (deeply) nested expressions: combine, number2x, number2y, and so on. In short, *Core* lacks the ability to name expressions and re-use them. More generally, the language lacks a construct for abstracting identical or similar pieces of code into a single form that can be referred to via a name.

In sum, any use of *Core* raises the following two related questions:

- how can developers represent structures information as data, and
- how can developers name—and abstract over— repeated pieces of code so that it becomes possible to reuse them easily.

1.1 Design Choices

This discussion ignores chars, Booleans, and other essentially atomic forms of data similar to numbers.

Historically, language creators gave distinct, unrelated answers to these two questions. To permit a direct form of representation of structured information as data inside of a programming language, they added arrays (of numbers). Arrays have come in many forms and shapes: one-dimensional, vector-like data; two-dimensional ones, resembling matrices; 0-based, 1-based, and programmer-defined based arrays; and many more flavors. In essence, though, an array is little more than a slice of memory from the underlying hardware, possibly equipped with some additional pieces, such as length information.

As for naming pieces of code and abstracting over differences, language creators introduced procedures and functions. The latter word was chosen in recognition of the relationship between language design and mathematical notation where functions—like those students get to know in pre-algebra courses—name expressions and abstract over the differences among similar expressions. Of course, due to the imperative nature of historical languages—quite similar to *Core*—they rarely represented mathematical functions.

Over time, the designers of programming languages realized that neither arrays nor plain functions met the needs of programmers concerning

the representation of complex information. Two different solutions to these questions emerged:

- The first one is the idea of a class, which combines several pieces of data via fields with methods. The latter enable programmers to define function-like facilities that process instances of this class.

- The second one combines first-class functions with structures, also known as records. Functions come with support such as pattern-matching for processing structures; if they are also first-class values, it is possible to store functions in structures.

At this moment in time, the first approach dominates the landscape of programming languages, though flavors of the second alternative are making a come back.

This chapter presents a model of the first approach, focusing on the most essential elements:

- Classes that have a fixed number of fields and methods.

- The construction of class instances must supply as many values as there are fields. The resulting values are so-called first-class objects, distinct from numbers and distinct from instances of other classes. They can be passed to methods as arguments, placed in variables, stored in fields, and so on.

- All fields and methods are public, meaning a piece of code with access to an instance of the class can read any field, modify the content of any field, and call any method.

This sparse model—lacking sub-classing and type-like features—is a good starting point to gain a first understanding of how the addition of classes to a language such as *Core* answers the two questions raised here. Additionally, even with all these simplifications, the model enables us to consider the rather dramatic increase in costs and benefits over *Core*.

1.2 Costs and Benefits

The addition of classes to a language such as *Core* comes with quite a number of pluses and minuses for the developer. As for the benefits, the are essentially the answers to the above questions, but let's spell them out:

- With classes, it becomes easy to formulate reasonably direct data representations for the information that a program is to process.
- Since class instances differ from numbers and each other, programming with classes also reduces the potential of conflating one piece of data for another. Coding becomes less error prone.
- Methods are a natural mechanism for naming repeated pieces of code.
- If they are parameterized—as they usual are—they also help programmers abstract over similar pieces of code. That is, methods abstract over pieces of code that differ in pieces that can be named and whose values can be passed into a method.
- The inclusion of methods in classes signals to future readers that these pieces of functionality belong to this specific form of data.

Unsurprisingly, the addition of a complex sub-language such as a class system also injects some complexities into programming that a purely numerical language doesn't suffer from. Most importantly, programmers must grok a complex model of computation, meaning syntax and semantics, before they can truly take advantage of the pragmatic benefits. Since a class-based language comes with a lot more syntax than say *Core*—meaning BNF productions and validity constraints—forming correct programs also becomes a more complex task. The semantics of a class-based language is also far more complicated than the one of *Core*:

- As a program instantiates a class, it allocates some space in the store for the values of the fields and the name of the class. Semantically, this space is a single piece of data. It can find its place in a variable or the field of another object; but it may also just be part of an expression that retrieves a field's value or calls a method.
- One consequence is that a programmer must become keenly aware of the distinction between lexical scope and dynamic extent. The life time of a variable corresponds to its *lexical scope*. By contrast, the life time of an object depends on dynamic properties of the program. As long as any variable or field can refer to an object located in the store, programmers consider the object as being alive; otherwise it is dead. Language researchers speak of the object's *dynamic extent*, i.e., the time between the birth of the object and its death. Semantically speaking, variables exist in environments, objects in stores.

Programmers must learn to consider dynamic extent, a far more complex notion than lexical scope, simply because it is a time slice of a program execution rather than a fixed region of text.

- Furthermore, because a program can point several variables and fields to the same object—that is, the same place in the store—changes to the object are visible from many different places in the program.

  Language researchers refer to this phenomenon as *aliasing*.

  Programmers must learn to reason about the aliasing of objects, especially whether they want the effect in some situation or not. And if they don't want it, they need ways to avoid it.

- In some way, a method call resembles the use of a function in pre-algebra, once the targeted method has been identified. It is this latter part of a method call that imposes work on the programmer. Since different classes may contain methods with the same name (yet different expression), the meaning of a method call is only clear when a programmer understands which object is the *target* of the call.

  Language researchers dub this concept *dynamic dispatch*, because the actual method is identified only during program execution when the "dispatch" takes place.

  A programmer must carefully think about code to ensure that code refers to the truly desired method. If a program comes with the classes Artist and Cowboy and both supply a method named draw, it is unlikely that these methods are meant to exhibit the same behavior.

In sum, while a class system clearly benefits programmers in many work situations, it also raises the bar for thinking about code.

From the perspective of the language designer, adding classes to *Core* is a major undertaking. It means significant changes to the syntax, both grammar and validity constraints. It requires an extensive revision of the existing semantics plus the development of a semantics of classes: field access, field mutation, and method calls. All of these changes imply a good amount of implementation work: the parser, the validity checker, the compiler, and the run-time system. But, in the end, these additions are of tremendous help to working developers and therefore worth their careful design and implementation.

The rest of the chapter introduces *Class*, the syntax and semantics for a class-based extension of *Core*. With *Class* in hand, we can discuss the com-

plexities of class-based, object-oriented programming compared to rather simplistic model of computation from the preceding chapters.

Exercise 15. Imagine an extension of *Core* with structure-type definitions (records) and procedures. Which of the just presented benefits and costs of the class extension do not apply to this alternative?

Exercise 16. Consider an extension of the language proposed in exercise 15 so that functions are first-class values in addition to structures and numbers. In particular, assume that the fields of structures may contain functions. Which of the just presented benefits and costs of the class extension do not apply to this second alternative?

2 Project Language: *Class*

Adding the syntax of a class system to *Core* requires the addition of productions to the language's grammar and the introduction of appropriate validity rules. While the first is reasonably straightforward, the second one exposes a critical tension inherent to language design: which parts of a program to check *before* program execution and which ones to check *during* execution. Indeed, the tension goes back to the origins of computer science, concretely the theory of computation, which tells us that not all problems are solvable via algorithms.

2.1 *Class*: the Grammar

Figure 28 presents the grammar of *Class*, the class-based extension of *Core*. A *Class* program consists of a possibly empty sequence of class definitions followed by the program's *body*. In a language such as Java, the latter corresponds to the main method of the main class. The body is like a *Core* program that may also contain expressions and statements related to classes.

The keyword class marks a class definition, which consists of an embedded parenthesis-delimited sequence of the names of fields followed by a possibly empty sequence of method definitions. In turn, a method definition comes with a parameter list—a sequence of variable names surrounded by parentheses—and a method body; the latter is again a sequence of variable declarations, statements, and an expression. This expression computes the value that a method call returns.

Next, the Expression production enables object creation, access to a field, method calls, and object identification. Field modification, though, is a statement, because it has only an effect, not a result.

```
Program     ::= (Class* Declaration* Statement* Expression)

Class       ::= (class ClassName (FieldName*) Method*)

Method      ::= (method MethodName (Parameter*)
                  Declaration*
                  Statement*
                  Expression)

Declaration ::= (def Variable Expression)

Statement   ::= (Variable = Expression)
              | (if0 Expression Block Block)
              | (while0 Expression Block)

              | (Variable --> FieldName = Expression)

Block       ::= Statement
              | (block Declaration* Statement+)

Expression  ::= GoodNumber
              | Variable
              | (Variable + Variable)
              | (Variable / Variable)
              | (Variable == Variable)

              | (new ClassName (Variable*))
              | (Variable --> FieldName)
              | (Variable --> MethodName (Variable*))
              | (Variable isa ClassName)

The set of ClassNames is the same as the set of Variables.

The set of FieldNamess is the same as the set of Variables.

The set of MethodNames is the same as the set of Variables.

The set of Parameters is the same as the set of Variables.

The set of Variables consists of all Names, minus keywords.

The set of GoodNumbers comprises all inexact numbers
(doubles) between -1000.0 and +1000.0, inclusive.
```

Figure 28: The grammar of the *Class* language

Class names, field names, method names, and method parameters are drawn from the same set as variables. The BNF productions use different (meta)names for these sets just to explain their various roles in a program.

Also, these names indicate where it is possible to reference these names, a critical aspect of validity checking and semantics.

A second look at the grammar reveals that the word "this" isn't mentioned anywhere. But, as every student of computing knows, "this" seems to play an important role in method bodies. Specifically, the word "this" is a reference to the concrete object on which the method is called; it is often dubbed the *target* of a method call. Syntax design of a class-based language can choose to make "this" special or to treat it as an ordinary variable that, semantically, has meaning inside of a method body. The design of *Class* takes the second approach.

Exercise 17. Design an AST data representation for *Class*. Implement a parser for *Class* that maps an S-expression to an instance of AST. Re-use your solution from exercise 10 as much as possible.

2.2 *Class*: Scope

The scope of a class's name is the entire program. All expressions in a program may refer to any defined class.

Next, the parameters of a method are visible in its body, that is, the following sequence of declarations, statements, and return expression. Following tradition, the name this denotes the target object of a method call—by default and implicitly. To be precise, the method header introduces the name this, whose scope ranges from the end of the parameter list to the end of the method.

Parameters, including this, are just variables. Hence, a programmer may declare a variable this inside of a method or use the name of a parameter for a declared variable. Doing so will shadow the implicit declaration of this and the parameter of the same name, respectively.

By contrast, FieldNames and MethodNames do not have a scope. That is, expressions and statements may use any name as an "index"—i.e. use the -> notation—for a FieldName or a MethodName into an object. If the target object comes with an appropriately named field or method, the phrase has meaning; otherwise it doesn't.

Given these scoping rules, let's consider the two *Class* programs in figure 29. Both come with one class definition named C, and both create one instance of this class in the program's body. The program on the left uses C in the program's body, and it refers to this as the target object in the set method. In contrast, the program on the right uses C inside the class and

```
((class C (f)                          ((class C (f)
  (method set(p)                         (method make(p)
    (this -> f = p)                        (def this 66.0)
    42.0))                                 (new C(this))))

 (def one 1.0)                          (def one 1.0)
 (def obj (new C(one)))                 (def obj (new C (one)))
 (def fld (obj --> f))

 (obj --> f = (obj --> set(one)))       (obj = (obj --> make(one)))
 (one = (obj -> f))                     (one = (obj -> f))
 (one + fld))                           one)
```

Figure 29: Two simple *Class* programs

in the program's body. Furthermore, it declares a variable named this immediately below the method header of make, thus making it impossible to refer to the target object in the remainder of the method's body.

Exercise 18. Use your experience with object-oriented languages to explain the intuitive meaning of the two programs in figure 29.

2.3 *Class*: Validity, the Easy Part

Since the design of *Core* avoids "undefined variable" errors, the question arises whether similar properties can hold for the *Class* language. In other words, we are checking whether to impose validity constraints on well-formed *Class* programs with the goals of

- providing feedback to the programmer during code creation, and
- simplifying the semantics of the *Class* language.

As it turns out the design of the *Class* language comes with two kinds of validity constraints that programmers would find useful and that language designers could use to simplify the semantics. The first kind concerns the definitions of classes, the major new component of the Program production of the BNF grammar. The second kind concerns the additional expressions and statements that enable the creation and manipulation of instances. Surprisingly, the two kinds of validity concerns radically differ from each other, in nature and in eas-of-checking. While this section deals with the first, easy kind, the next one covers the difficult problems.

Let's start with the programs in figure 29. Both of them are clearly well-formed and valid programs. Now contrast this first program with the following class:

```
((class C (f) (method m(a) 42.0))
 (class C (f f)
   (method m(p p) 42.0)
   (method m(p) 21.0))
 (def one 1.0)
 (def instance (new C (one)))
 (def field    (instance --> f))
 (instance --> f = (instance --> m(one)))
 (one = (instance -> f))
 (one + field))
```

It defines two classes—with the same name. Worse, the second, boxed class exhibits several other issues that every major programming language flags:

- The symbol C is used twice to name a class.
- In the second well-formed class definition, f is the name of two fields.
- Similarly, both methods in the second class definition are named m.
- And finally, the first method definition has two parameters named p.

Clearly, two class definitions with the same name make no sense, because the natural scope for one class definition is the entire program, including the class definitions that precede it and those that follow.

Exercise 19. Explain in a similar manner why duplicate field names, duplicate method names, and duplicate parameter names make no sense.

Exercise 20. Explain why we admitted duplicate variable definitions in the same sequence of Declartions.

Turning these examples of meaningless program pieces into validity constraints is straightforward. The first one is about the global sequence of class definitions:

> *No two classes should have the same name.*

The second one is all about constraints about code within a single class:

> *No two fields should have the same name, no two methods should have the same name, and no two parameters of a method should have the same name.*

Since there are two distinct set of rules, a well-designed static checker should employ two passes:

- one for enforcing that all occurrences of ClassNames refer to defined classes; and
- another one for making sure that the various kinds of names within field and method definitions (ClassNames, FieldNames, MethodNames, and Parameters) are distinct.

Exercise 21. Design and implement a validity checker that enforces the validity rules for *Class*. The checker consumes error-free ASTs from exercise 17; if it finds errors it annotates the AST.

Once these validity constraints are successfully checked, we can think of the defined collection of ClassNames as a set. Indeed, it is just like the set of variables that are declared at the top of the program. And this suggests an analogous rule for the occurrences of ClassNames:

> *All occurrences of a ClassName in expressions and statements must refer the name of one of defined classes.*

Exercise 22. Turn this rule into a static checker,—The implementation follows the same pattern as checking that all variable references point to declared variables. Given a well-formed program, the set of classes is one direct component of its AST. This node contains the AST for all classes and, inside of those, their names. Hence, a static checker can enforce the rule by visiting all expressions and statements in the given program and checking that every occurrence of a ClassName is a member of the set of class names.

2.4 Undecidable Problems and Validity

Gödel, Church, and Turing discovered intrinsic limitations on logic and computing, respectively.

Before we continue our exploration of validity constraints for *Class*, we need to take a step back and consider the nature of computational problems. As it turns out, some problems are undecidable, and this idea is directly connected to the validity checking in language implementations.

Let's make this idea precise. A problem is a set S of data representations (of information) together with a subset $T \subseteq S$ of "positive" elements; all other elements are "negative.". Here information means programs in *Class*, and ASTs are data representations of these pieces of information.

Such a problem is *decidable* if there is a total function D from S to the Booleans such that for all $s \in S$

- if $D(s) = \mathit{true}$, then $s \in T$, and
- if $D(s) = \mathit{false}$, then $s \notin T$.

If there is no such total function D, the problem is *undecidable*.

Here is a first concrete example. Take the problem as the set S of all *Class* ASTs, and the set T as those ASTs for which an evaluation on suitably modified CESK machine terminates. People call this the "halting problem", and Turing proved that it is *undecidable*.

In the context of a class-based language, we get a second concrete example that is relevant to software developers. Let's again start from the set of all ASTs as the set S, but take as T all those field-access nodes in the AST that an evaluation o suitably modified CESK machine reaches. If so, we could also consider T as the set such the evaluation of (o –> f) always succeeds. The question is whether such problems are decidable.

At this point, you might wonder whether D could use the CESK machine to determine whether a specific property holds for some *Class* AST. In other words, why shouldn't D run the given program on this CESK machine and observe how field references are evaluated?

Given your experience with writing and running programs you know that program executions may not terminate. Using *Core*, a programmer may write (while0 0. (x = x)), which causes the (well-formed and valid) program to loop forever. In short, if D relied on the CESK machine, it wouldn't be a total function.

Stepping back, this brief excursion into computation theory relates validity checking to pragmatics. As a developer enters program text into an IDE, programming language tools such as the parser and validity checker are supposed to provide feedback. To be effective, these tools must always terminate and provide the IDE with an answer that can be rendered as feedback. Conversely, the problems that these tools address must be decidable. In particular, the developers want feedback *before*—and indeed *without*—running the program.

Note Just because a problem is undecidable does not mean that we cannot determine whether $s \in T$ for some specific s. The key is that we cannot do so for all s. Researchers who investigate IDE tools occasionally exploit this "loophole." They construct validity-checking tools that *attempt* to decide whether the code is valid and, if the answer isn't available within a certain amount of time, it is ignored. We ignore this ambitious idea here.

2.5 *Class*: Validity, the Undecidable Part

We can now return to the development of validity constraints for *Class* and consider some problems that look more ambitious than those of section 2.3 of this chapter. Based on your experience with class-based programming languages, you might wonder whether the following validity problems are decidable and thus worth adding to a model of *Class*:

- A (new C (x_1 ... x_n)) expression may fail because C does not have n fields.
- A field access (target → f) may fail because target is not an object or there is no field f.
- A field modification (target → f = rhs) may fail because target is not an object or there is no field f.
- A method call (target → m(...)) may fail because target is not an object or there is no method m.

While the first problem *is* decidable, the remaining three are undecidable.

Concretely, there are no functions that consumes a program's AST and returns true if, say, a field access expression cannot fail. Doing so would require inspecting the AST and to determine exactly which objects target may ever denote. And this kind of problem is exactly one of the undecidable ones that people investigated in the 1930s and found undecidable.

A subset of this problem is decidable, however. When the target is this and there is no parameter and no locally declared variable named this, a validity checker could determine whether the object comes with a field named f. Similarly if the target of a field mutation or a method call is this, a function could use the given AST to determine whether the named field or method is available.

Since these cases are rather special, and since the expressions in questions raise additional questions, we delay the treatment of such validity constraints. Technically speaking, Chapter VIII covers the addition of a type system to our language models, at which points validity checking works rather differently. Instead, we enrich the CESK semantics so that it comes with checks that determine whether a field access, a field mutation, or a method call is meaningful.

3 The CESK Machine for *Class*

A modified CESK machine serves to explain the semantics of classes. Since the collection of classes does not change over time, it plays the role of a constant for the transition function. The load function can set up this quasi-constant so that the transition function has access. As for the sets of control strings, environments, stores, and continuations, revising them is quite straightforward. In addition to the statements and expressions of *Core*, the ones for revised CESK machine must account for the additional statements and expressions in *Class*. Finally, a semantics for *Class* must assign meaning to instances of classes, which implies that the set of values now comprises both numbers and data representations of objects.

3.1 Representation of Objects

By comparison, numbers are easy to represent, which is why numeric expressions didn't deserve a discussion.

The expression (new C (x y z)) expresses the intention to create an instance of C whose three fields are initially associated with the values of the variable x, y, and z. This information—the class combined with the field values—needs a representation in our model, because this is how models assign meaning to expressions. Here are some possibilities::

1. The first five chapters present models that are completely mathematical. If we wanted to continue in this fashion, we would have to use a protocol for representing objects as bundles of locations in the store. For example, if a class has n fields, the representation of an instance could use n+1 slots in the store. One of them would contain the name of the class, and the remaining slots would contain the value of fields.

2. Another approach to extend the mathematical approach of the first five chapters is to use a vector-like concept from mathematics to represent an object—something like a Cartesian point—and to stick this representation into a single location. In practice, this idea means to pick some immutable data structure in the chosen implementation language and to stick instances of this data structure into the store. When the machine evaluates a field-access expression, the CESK transition function can retrieve the object from its location and index the designated field.

3. When it comes to modeling mutation, both purely mathematical approaches cause serious complications. Although possible in principle, we choose to use a third approach—the use of a *mutable* data

structure in the chosen implementation language. This choice comes with one clear downside: we can no longer interpret the model with our (quite simple) understanding of mathematics. One clear upside is that this approach is easy to implement.

We go with alternative 3, emphasizing ease of implementation over mathematical reasoning.

See "Definitional interpreters for higher-order programming languages" in ACM's Annual Conference Record (1974).

Note By tying the model to a mutable data structure in the implementation language, we choose the so-called *meta-language* approach. That is, our e model of semantics no longer just relies on a neutral understanding of mathematical concepts, such as sets and functions, but, at least to some extent, of a tru understanding of our chosen implementation language. As John Reynolds pointed out a long time ago, this may inject elements of the implementation language into the modeled language. With the CESK approach, it is fortunately possible to avoid most of the problems in a natural way, and the rest of the book makes sure that the only point of "contact" remains the data representation of objects.

Exercise 23. Design a data representation for instances of classes in your favorite programming language. Include the functionality (1) for accessing and modifying fields and (2) for retrieving the AST for a specific method, assuming a collection of classes is also given.

3.2 CESK States for *Class*

Let's recall what the content of the CESK machine's registers represents:

- Control strings direct the CESK machine. The register contains one of three kinds of elements: an expression to be evaluated; the *value* from an expression evaluation; an error token, causing the machine to stop; or the † token, indicating that the machine is searching for the next expression to evaluate.

  As pointed out, the set of values comprises both numbers and data representations of objects.

  The *Class* language adds several forms of expressions to *Core*: new, isa, target → f, and target → m(...). For each of these expressions, we need to equip the transition function with corresponding cases.

  Additionally, the search process may find a field-mutation statement, which places an expression into the C register. Once the value of such

an expression is found, the transition function can wrap up the execution of the statement via a use of meta-functionality.

- Environments associate variables with locations. While the CESK machine for *Core* allocates new locations only for variable declarations, the CESK machine for *Class* also allocates locations for the parameters of a method when it is called.
- Stores associate locations with *values*. Keep in mind that the set of values is more than just the set of numbers in this variant of the CESK machine.
- Continuations are potentially empty stacks of closures. Each such closure represents which part of the given program's instructions remain to be executed.

Figure 30 is an amendment to the notational conventions of figure 23. As you study the following subsections, consult these tables as needed.

name(s) and term(s)	standing in for
C	a name for a class
f	a name for a field
m	a name for a method
o	a program variable that is supposed to denote an object
x*	a potentially empty sequence of program variables
v	values, include numbers and objects
v*	a potentially empty sequence of values
obj	an object in the implementation language
thisL	a location for this
xl*	a potentially empty sequence of locations

Figure 30: Additional CESK Conventions for *Class* (also see figure 23)

As for the classification of the sets of states into initial states, intermediate states, and final states, the definitions remain the same. The set of initial states turns the variable declarations, statements, and return expression of a program into the first closure. Similarly, the final states are those that contain a value or the error token in the C register and an empty stack in the K register. All remaining states are intermediate states.

Exercise 24. Revise the data representation from exercise 13 for the states of the CESK machine for *Class*. Implement the load and unload functions, which turn a program AST into an initial state and unload a final state, respectively. Keep in mind that the load function must also extract the collection of class definitions from the given program AST and make it available to the transition function.

3.3 The CESK Transition Function: Adapting the Cases for *Core*

The definition of the CESK transition function from Chapter V almost works for *Core* statements and expressions within the *Class* model language. But, given that the set of values now comprises numbers and data representations of objects, the cases do not apply as-is. Consider the simple case of an addition expression the C register:

	Control	Environment	Store	Kontinuation
evaluate an addition				
before:	(y + z)	e	s	k
after:	+	e	s	k
where	yl = e[y]			
and	zl = e[z]			
and	yn = s[yl]			
and	zn = s[zl]			

Since locations in the store map to numbers, this definition is a straightforward three-step procedure: retrieve the location of the two variables (y, z) from the environment; retrieve the numbers associated with the respective locations (yl, zl); and stick the sum of the two numbers (+ yn zn) back into the C register. Now, however, the set of values comprises both numbers and objects, and as a result the addition may not always succeed.

Figure 31 presents an adaptation of the transition function. Like the CESK transitions for division in the preceding chapter, addition now requires two cases: one for success, when both variables refer to numbers in the current store, and one for failure, when at least one of them refers to an object. Beyond the cases for addition, figure 31 displays the case for a comparison expression. Like the success case for addition, it retrieves the values of the two variables and uses the =0? function to compare them.

All of the remaining rules of the CESK transition function from the preceding chapter work *mutatis mutandis*. For example, the cases for if state-

	Control	Environment	Store	Kontinuation
evaluate an addition (success)				
before:	(y + z)	e	s	k
subject to:	y and z are numbers			
after:	+	e	s	k
where	yl = e[y]			
and	zl = e[z]			
and	yn = s[yl]			
and	zn = s[zl]			

evaluate an addition (failure)				
before:	(y + z)	e	s	k
subject to:	y or z is not a number			
after:	error	[ ]	[ ]	[ ]

evaluate a comparison				
before:	(y == z)	e	s	k
after:	rr	e	s	k
where	yl = e[y]			
and	zl = e[z]			
and	yn = s[yl]			
and	zn = s[zl]			
and	rr = are yn and zn structurally equal?			

Figure 31: The CESK transition function for *Class*: addition, comparison

ments remain the same, though the interpretation of the phrase "n is 0.0" shifts to the following:

if n is a number (i.e., not an object) and n is equal to 0.0.

The same shift in meaning applies to the cases for while.

Exercise 25. Generalize the function =0?, originally specified in exercise 13, so that it works for the set of values in the context of *Class*. Two objects are equal if they have the same fields and the fields contain the same values.

Exercise 26. Adapt all cases of the *Core* transition function from the preceding chapter so that they apply to the CESK machine for *Class*.

3.4 Creating and Checking Instances

Figure 32 shows the transitions for creating instances of a class. While the validity checking ensures that C is a defined class, it does not guarantee that the new expression comes with the same number of values as there are fields in the class definition. If the two counts agree, the machine creates an instance of C and places it into the control-string register. Otherwise, the machine signals an error. See section 3.1 of this chapter on what the data representation of an instance combines.

Again, this property could be checked but deferring this point to Chapter VIII is the pragmatic alternative.

	Control	Environment	Store	Kontinuation
evaluate a 'new' expression (success)				
before:	(new C x*)	e	s	k
subject to:	class C has the same number of fields as x* has elements			
after:	obj	e	s	k
where	v* = s[e[x*]], ...			
and	obj = create instance of C using v* for the values of the fields			
evaluate a 'new' expression (failure)				
before:	(new C x*)	e	s	k
subject to:	class C does not have the same number of fields as x* has elements			
after:	error	[ ]	[ ]	[ ]
evaluate an 'isa' expression				
before:	(o isa C)	e	s	k
after:	ans	e	s	k
where	obj = s[e[o]]			
and	ans = check whether obj is instance of C			

Figure 32: The CESK transition function for *Class*: instances

The third case in figure 32 concerns the isa expression. When such an expression shows up in the control-string register, the validity check of the syntax guarantees that C is to the name of a class defined in the source text of the given program. The machine must determine whether the value is an instance of C, and, if so, deliver an appropriate value.

Exercise 27. Design and implement a function that consumes the name of a class and a sequence of values to create an instance. Re-use the solution of exercise 23.

Exercise 28. Design and implement a function that consumes any value

and a class to determine whether (1) the value is an object and, if so, (2) it is an instance of the given class. As for the result of this function, see the discussion in exercise 13 and adapt as appropriate.

3.5 Field Access

In contrast to expressions involving class names, expressions involving field and method names need semantic checks that everything matches. For field-access expressions specifically, two situations may trigger faults if the machine doesn't recognize them first:

- The variable o may not refer to an object.
- While o may refer to an object, it may not have the specified field.

Since a field lookup can succeed only for objects with the desired field, the machine must signal an error in both situations.

	Control	Environment	Store	Kontinuation
evaluate a field reference (success)				
before:	(o → f)	e	s	k
subject to:	o is an object and has the desired field			
after:	ans	e	s	k
where	obj = s[e[o]]			
and	ans = get value of f from obj			
evaluate a field reference (failure)				
before:	(o → f)	e	s	k
subject to:	o may not be an object or does not have the desired field			
after:	error	[ ]	[ ]	[ ]

Figure 33: The CESK transition function for *Class*: field access

The two cases for the transition function are presented in figure 33. The first case specified a *before:* condition that includes "o is an object and has the desired field." Its *after:* state consists of the value in the field—whose retrieval depends on your choice of data representation—and the given environment, store, and continuation. As in other error cases, the second case in figure 33 places error into the C register and resets the remaining ones to initial values.

Exercise 29. Design and implement a function that consumes an object and the name of a field and that returns the value associated with this field in the given object. Keep in mind that due to the formulation of the transition function, it is guaranteed that the given object has the specified field. Re-use the solution of exercise 23.

3.6 Method Calls

In *Class*, method calls are expressions. Like a field-access expression, a method-call expression must satisfy certain conditions before the machine can perform it; see figure 34. In particular, the target (o) must refer to an object; the object must have the desired method; and the number of parameters of the method and the number of arguments must agree. If a CESK state meets these three conditions, it transitions to a state that has new content for each of its four registers:

- The environment register contains a table that associates the method's parameters and this with locations that the current store s does not associate with values.

- The store in the S register associates these newly allocated locations with the corresponding values. In particular, the location associated with this maps to the object itself.

- Finally, the machine pushes a new closure onto the continuation in the K register. This new closure combines the (abstract syntax tree) of the method's body with the old environment.

- The control-string register contains †, because the machine must find the next expression to evaluate. After all, the next expression to be evaluate is somewhere in the method's body (a sequence of definitions, statements, and a return expression).

It is for this case of the transition function that the name of the class is included with an object. Given the object that o denotes, the machine can retrieve the class definition by using the class name. The desired method is located in the AST of the class definition, meaning the machine can both check the conditions of the *before:* specification and extract the method's AST itself for the *after:* state.

Stop! Take a close look at figure 34. Why is it unnecessary to add a case concerning the return from a method call?

	Control		Environment	Store	Kontinuation
evaluate a method-call expression (success)					
before:	(o → m (x* ...))		e	s	k
subject to:	o is an object				
	... has the desired method				
	... the correct number of parameters				
after:	†		e1	s1	push[cl, k]
where	obj	=	s[e[o]]		
and	v*, ...	=	s[e[x*]], ...		
and	[para*, body]	=	the parameters & body of method m *in* the class of obj		
and	cl	=	*closure*: body *in* e		
and	thisL	=	a new (relative to s) location		
and	xl*, ...	=	as many new (relative to s) locations as elements in x*		
and	e1	=	[ ][para* = xl*], ... [this = thisL]		
and	s1	=	s[xl* = v*], ... [thisL = obj]		
evaluate a method-call expression (failure)					
before:	(o → m (x*))		e	s	k
subject to:	o may not be an object or				
	... does not have the desired method				
	... takes a different number of arguments				
after:	error		[ ]	[ ]	[ ]

Figure 34: The CESK transition function for *Class*: method call

Exercise 30. Design and implement a function that consumes an object, the collection of all defined classes, and the name of a method, and that returns the (AST of) the specified method. Keep in mind that due to the formulation of the transition function, it is guaranteed that the given object has the named method. Re-use the solution of exercise 23.

3.7 Field Mutation

Like a variable assignment, a field-mutation statement consists of a left-hand side and a right-hand side. For the statement to make sense, the left-hand side must refer to an object with a desired field, while the right-hand side is an expression. The expression's value is supposed to be the value of the object's field *after* the statement is evaluated. Consequently, a field-mutation statement comes with two places whose value must be determined: the object target position and the right-hand side.

	Control	Environment	Store	Kontinuation
search ends a field assignment				
before:	†	e	s	⟪ [ ], (o → f = rhs)::stmt*, r ⟫
after:	rhs	e	s	⟪ [ ], (o → f = rhs)::stmt*, r ⟫
execute a field assignment (success)				
before:	v	e	s	⟪ [ ], (o → f = rhs)::stmt*, r ⟫
subject to:	o is an object and has the desired field			
after:	†	e	s	⟪ [ ], stmt*, r ⟫
where	obj =	s[e[o]]		
and		set f in obj to v		
execute a field assignment (failure)				
before:	v	e	s	⟪ [ ], (o → f = rhs)::stmt*, r ⟫
subject to:	o may not be an object or does not have the desired field			
after:	error	[ ]	[ ]	[ ]

Figure 35: The CESK transition function for *Class*: field mutation

Figure 35 displays the three cases of the transition function that determine the meaning of a field-mutation statement:

1. The first case states that the machine is in search mode (†) and that the top-most closure of the continuation comprises a sequence of statements, starting with a field-mutation statement, and preceded by an empty sequence of definitions. The after: state's C register contains the right-hand side expression, meaning the machine is supposed to evaluate it next. Once C contains the value of the right-hand side expression, the machine distinguishes two cases.

2. The second case specifies that the *before:* state identifies o with an object in the store, specifically with an object that has the desired field. Its *after:* state description says that the machine is back in search mode, that the object in the store is modified, and that the evaluation of the field-mutation statement is complete.

3. The third case informs the language implementer that if o does not denote an object or if it denotes an object without the desired field, the language implementation is supposed to signal an error.

Note again how this expression is only evaluate when (1) the machine is in search mode (†) and (2) the statement is the first in a sequence of statements.

Exercise 31. Design and implement a procedure that consumes an object, the name of a field, and a value. It modifies the named field of the object, so that it is associated with the given value. Keep in mind that due to the formulation of the transition function, it is guaranteed that the given object has the specified field. Re-use the solution of exercise 23.

Exercise 32. The semantics of field-mutation statements evaluate the right-hand side of (o → field = rhs) *before* checking whether the left-hand side refers to an object. Hence, this arrangement leads to premature evaluations. Rewrite the instructions so that the right-hand is evaluated only if the entire instruction may succeed. Can a programmer observe the difference?

Exercise 33. Extend the transition from exercise 26 so that it includes the cases for object creation, instance checking, field access, method calls, and field mutation.

By combining the load, unload (see exercise 24) and transition functions with the runMachine function (see section 2 of Chapter III), you obtain a complete semantics for well-formed and valid *Class* programs.

4 Project Tasks

The exercises in the preceding two sections request a parser for *Class*, several *Class*-specific validity checkers, and an adaptation of the CESK machine to the new expressions and statements. It is now time to compose these pieces of functionality to a complete, executable model.

Exercise 34. Adapt the main function from figure 12 so that it can run the entire process of parsing *Class* program syntax, validating it, and determining its meaning.

The function should issue error strings in the following order:

- "syntax error" if the parser discovers any mistakes;
- "duplicate name error" if the *Class*-specific validity checker encounters two classes with the same name, a class with two identical field names, or a class with two identical method names;
- "undeclared name error" if the validity checker encounters an undeclared ClassName or variable.

- "runtime error" if the CESK transition function stops due to a final state with an error in its C register.

If the machine returns a number, main issues it as the final result.

4.1 Properly Evaluated Method (Tail) Calls

One of the puzzles in section 3 asks you to notice that method calls do not come with a return case in the transition function. Here is why this matters.

Language researchers say a method in *Class* is *tail-recursive* if its body comes with a return expression that calls this; indeed, this return expression is often called a *direct tail call*. The left column of figure 36 shows a concrete example.

```
((class While (count)
   (method w()
     (def d (this --> count))
     (def one -1.0)
     (def e (d + one))
     (if0 e
          (e = e)
          (this --> count = e))
     (other --> w ())))

 (def u 3.0)
 (def w (new While (u)))

 (w --> w ()))
```

```
((class While ()
   (method w(other)
     (other --> w (this))))

 (class Repeat ()
   (method w(other)
     (other --> w (this))))

 (def r (new Repeat ()))
 (def w (new While ()))

 (w --> w(r)))
```

Figure 36: Examples of direct and indirect tail-calling methods

Stop! How long will this program run?

Exercise 35. Evaluate this program with a direct tail call on your CESK implementation and continuously measure and print the size of the stack in K every time the transition function is called. What do you observe?

Indirect tail calls are return expressions that call a different method in either the same object or some other object. The code snippet on the right side of figure 36 illustrates indirect tail calls with two methods that are mutually recursive across class boundaries.

Stop! How long will this program run?

Exercise 36. Evaluate this program with indirect tail calls on your CESK implementation and continuously measure and print the size of the stack in K every time the transition function is called. What do you observe?

As these exercise demonstrate, the content of K is bounded. Specifically, it contains at most one frame, i.e. closure. If the semantics specification of a programming language satisfies this property and an implementation preserves it, the language is said to implement (method) calls properly. The concrete implication of this property concerns the consumption of space by a program; it does *not* come with any consequences concerning the consumption of time.

Note Old-fashioned language researchers speak of "tail call optimization," but this phrase is utter nonsense. What the careful formulation of the CESK machine shows is one natural way of specifying the behavior of (method) calls. It just so happens that "long time ago, in a galaxy far, far away" the first compiler writers happen to choose the other, space-consuming implementation of function calls. To defend their honor, they later dubbed the alternative implementation an "optimization."

5 Developers Must Learn to Think Hard

An executable semantics *can confirm a developer's reasoning, if the (immensely complex)* implementation *adheres to this (reasonably small) specification.*

With the addition of classes, reasoning about a program's properties becomes complex, as pointed out in the introduction of this chapter. fortunately, a model enables software developers to reason about program behavior; with an executable model, like the one for *Class*, they can also run quick experiments to see what happens, especially if they can annotate the source code.

This section presents examples that illustrate the new issues mentioned there: dynamic extent differs from lexical scope; object aliasing means a mutation in one place has a long-distance effect; and the meaning of field references and method calls are inherently dynamic properties.

5.1 Dynamic Extent vs Lexical Scope

When the CESK machine evaluates a variable declaration, it enters a specific lexical scope of the program text. It adds an association between the variable and a new location to the current environment to create a new one. It then places this environment back into the E register. As long as this new environment exists in the CESK machine—either in E or within a closure on

the stack in K, the variable is alive. By design, this life time ends when the CESK machine finishes the evaluation of the lexical scope associated with the variable declaration.

By contrast, when the CESK machine encounters a new expression in the program text, it creates an object and places it into the C register. What happens after that depends on the remaining instructions in k. For example,

```
(x = (new C ()))
```

This expression is ill-formed in our restrictive syntax but well-formed in most object-oriented programming languages.

would assign the new object to an existing variable, meaning the object would be accessible as long as the variable is alive. But,

```
((new C ()) --> m())
```

would just call a particular method and then the object would cease to exist.

Equipped with this simple explanation, take a look at this example:

```
((class Counter (count)
  (method getCount() (this --> count)))

 (def u 3.0)
 (def v 42.0)

 (block
   (def c (new Counter (u)))
   (v = c))

 (v --> getCount()))
```

The boxed region is the lexical scope for the program variable c. Its initial value is an instance of class Count. Otherwise the block contains a single assignment statement, which places the value of c into v, a program variable in the outer scope.

Thus, when the CESK machine encounters this nested block, it squirrels away the current content of E by pushing a closure onto the stack in K and then proceeds as follows:

1. The search inside the block places the new expression into Count and instantiates the class to yield a new instance.

2. Once this instance is in C, the CESK machine allocates a location cl that is not in the current store s; associates cl with the new instance in an extended store; and adds an association between c and cl to the current environment. Let's call this new environment ec.

3. Using ec, the machine finds and evaluates c. The result of s[e[l]] is the new instance.

4. Now that the instance is back in C, the machine completes the assignment statement. It looks up v's location in ec, yielding vl, and creates a store in which vl maps to the newly created instance of Count.

At this point, the block's instructions are exhausted and the CESK machine pops the content of K. The result is that ec is no longer reachable in any way, but vl still contains the instance of Count. Thus, calling getCount in the return expression succeeds and yields 3.0.

Stop! Why does it yield 3.0?

In this example, ec denotes the scope of the boxed statement; to be even more precise, it denotes the lexical scope of the declaration of c. As the process description explains, the environment comes into existence and disappears. By contrast, the store keeps getting extended and never disappears. Hence, it is possible to assign the instance of Count to a variable outside the scope of the nested block, a scope whose meaning is just e. Put differently, the instance is alive beyond the lexical scope defined by the box, because it is in the store.

Stop! When does the life of an instance end?

Note The dynamic extent of an instance is determined in an implicit manner. Most generally speaking, if at any point during an evaluation of a program it is possible to replace an instance in the store with a number, say 42.0, and the rest of the program evaluation does not change, the instance is *dead*. The earliest such point in a program's evaluation determines the life span of the instance; the time between the creation of the instance and this point is its dynamic extent.

One consequence of a finite dynamic extent is that the location associated with an instance could be reused for other values. Recognizing this property and getting the location ready for reuse is called *garbage collection*. The CESK machine, as presented, comes without a garbage collection mechanism. Hence, all instances remain in the store until the end of the program evaluation. If we limited the store to a finite size, evaluations would occasionally exhaust all available locations and fail due to this limit. Mathematical models of languages, including executable models, ignore such limits, but a programmer must keep them in mind.

5.2 Object Aliasing

In the preceding subsection, two variables briefly referred to the same object, though one of them immediately went out of scope. When two variable simultaneously refer to the same object, programming language people speak of *object aliasing*, and programmers must be keenly aware of aliased objects. After all, modifying the object via one variable reference means a reference via the second variable is going to notice the change.

Consider the following program:

```
((class Counter (count)
  (method getCount()
    (this --> count))
  (method upCount()
    (def crt (this --> count))
    (def one 1.0)
    (this --> count = (crt + one))
    0.0))

 (def u 3.0)
 (def c (new Counter (u)))
 (def d c)

 (u = (d --> upCount()))

 (c --> getCount()))
```

Its three variable declarations create a single object—an instance of Counter—and give it two names: c and d. The following statement is really just a method call, because the value of left-hand side variable is not used again. Key is that the right-hand side uses d to initiate the method call, not c. A quick look at the definition of Counter shows that this method call increases the count value of the object. But, when the return expression observes the value of count via c, it notices this change from 3.0 to 4.0.

A programmer with an understanding of the CESK semantics can think through the relationship among the variables, their locations, and the various objects. In this particular case, the environment for the program scope consists of three variables associated with three locations. The call to upCount pushes this environment onto the continuation, together with the remainder of the assignment statement and the return expression. Once the CESK machine has determined the effects and result of the method call, this environment is restored and the assignment to u is executed. At this point, the machine has nothing left to do but evaluate the return expression of the program: (c –> getCount()).

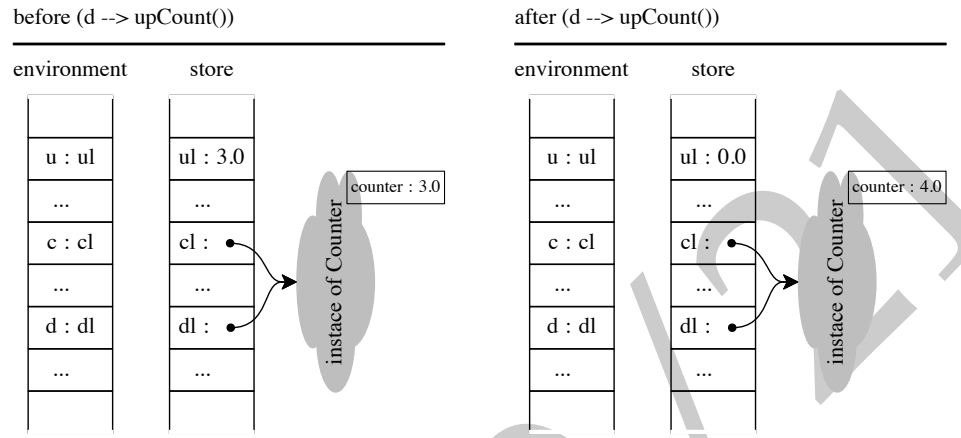

Figure 37: A diagram of object aliasing

Figure 37 shows how a programmer can imagine this state transition graphically. It consists of two diagrams: one of the *before:* state and one of the *after:* state—meaning before and after the evaluation of

```
(d --> upCount())
```

The dots in the tables aren't necessary for the example. They indicate that environments for larger programs contain more entries.

Each diagram comes with two tables: the environment and the store. The environment associates variables with locations, and it doesn't change. The store associates locations with values, which changes as needed. For example, the location ul maps to 3.0 in the *before:* state and to 0.0 in the *after:* state. Stop! Why is this the case?

Each state diagram contains two arrows that point to a cloud: one from cl, the location for c, and one from dl, the location for d. The cloud represents the only instance of Counter that the sample program creates. This object's single field is displayed as a box near the top-right. In the *before:* diagram, this box contains counter : 3.0; in the *after:* state, it is counter : 4.0. Since both the locations for c and d point to the same object, the change is visible via references to either variable in the program. Hence, (d –> getCount()) is 4.0.

Exercise 37. Turn this example of a *Class* program with object aliasing into a unit test for main from exercise 34.

Exercise 38. Construct an example like the one from this subsection in your favorite programming language and observe the effects of aliasing.

class definition

```
(class C (f)
  (method eq (other)
    (def old (this --> f))
    (def myf old)
    (def urf (other --> f))
    (def res 1.0)

    (if0 (old isa C)
      (this --> f = 42.0)
      (this --> f = (old + res)))

    (urf = (other --> f))
    (myf = (this --> f))

    (res = (urf == myf))
    (this --> f = old)

    res))
```

program body

```
(def one 1.0)
(def c (new C (one)))
(def d c)
(def e (new C (c)))

(one = (c --> eq(d)))
(d =  (c --> eq(e)))

(one + d)
```

Figure 38: Intensional (also known as) pointer equality

Exercise 39. Two objects are *extensionally* equal if they are instances of the same class and each of the corresponding fields contains extensionally equal values.

Two objects are *intensionally* equal if mutating the field of one affects the same change in the other. Programmers dub this form of equality "pointer equality" due to diagrams such as those in figure 37.

In the presence of parallelism, this eq *method won't function properly, which is why language implementation provide intensional equality as a basic, fast instruction.*

Stop! Develop a class with a single method, eq. This method consumes another instance of the same class and determines whether the two are intensionally equal.

The left-hand side of figure 38 shows a solution. Its right-hand side presents an example of a use of this method. Explain.

5.3 Call-by-value Prevents Variable Aliasing

What all this tells us, is that the CESK semantics determines the values of the arguments of a method call and hands them to the method. Language researchers dub this transfer of a value as a parameter-passing mechanism. This particular one is an idea that students actually get to know in American pre-algebra courses: if $f(x) = \ldots\ x\ \ldots$ is a function definition, then a use of f, say $f(40+2)$, is evaluated by substituting 42 for x into the expression of the right-hand side of the definition of f.

Although it seems straightforward to use this same idea for method and function calls in programming languages, the designers of languages have experimented with a rather large variety of ideas over the past six decades: call-by-value, call-by-reference, call-by-copy-in-copy-out, and several more. The reason for this experimentation is that a straightforward adaptation of the mathematical concept faces two problems:

The designers of Algol and Haskell also experimented with when *to determine the value of an argument. We ignore this dimension.*

- The first one is due to the presence of assignment statements in most programming languages, for both plain variables or the fields of objects. Assignments do not exist in mathematics. But, since methods are added so that programmers can abstract over repeated pieces of code, the question arises whether methods should also abstract over code with assignments—to variables or fields of objects.

- The second one concerns the notion of what is passed, that is, the question of what programmers should consider an argument *value*. As you may have noticed, all of our semantic models—from the CS machine to the CESK machine—explicitly define what the set of values is. In the context of *Class*, the set of values comprises numbers and objects—indeed, mutable objects from the CESK implementation language. Hence, abstracting over field assignments in *Class* code is straightforward as the preceding sub-section showed.

Given this background, old timers of the software profession like(d) to confront job interviewees with the following puzzle:

> Write a function in your favorite programming language that swaps the values of two given variables. Alternatively, explain why writing this function is impossible.

What they are getting at, is the natural follow-up question to the preceding sub-section, namely, whether it is possible to alias variables just like objects. If so, a method call would be able change the value of a variable at the call site, and writing a "swap" method would be straightforward.

A look at the CESK machine of *Class* answers this question. Every method parameter gets associated with a brand new location, and each of these locations maps to the corresponding value in the argument list, independently of what kind of value is passed in. This transfer of values from one place in a program to another via a method call is dubbed *call-by-value*.

> *If a programming language uses call by value only, variable aliasing and its effects are impossible.*

```
((class Counter ()
   (method upCount(x)
     (x = 42.0)
     0.0))

 (def c (new Counter ()))
 (def y 1.0)
 (def uselessResult
   (c --> upCount(y)))

 y)
```

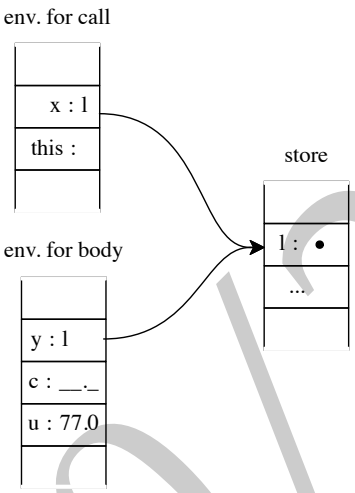

Figure 39: What variable aliasing would look like, if it existed in *Class*

In Class, all arguments to method calls are always variables (to simplify the semantics). Briefly imagine a relaxation of this constraint.

Now contrast call-by-value with an alternative that re-uses a location when the parameter is a variable. This alternative is referred to as call-by-reference. Figure 39 illustrates this idea with an example. The code on the left-hand side passes y (boxed) to upCount. In a call-by-reference world, the semantics would look up the location associated with y and point the parameter, x, to the same location. The right-hand side of the figure explains this situation with a diagram; technically, the diagram explains the environment and store right after the CESK machine begins its search for an expression in the body of upCount. It is easy to see that an assignment statement to x inside of upCount would change the value of y, too.

> *A programming language with some form of call-by-reference argument passing enables variable aliasing.*

Just like with object aliasing, a programmer may intend the aliasing of variables across method boundaries—to abstract over repeated variable assignment patterns—or may accidentally cause variable aliasing. In the second case, the programmer may observe strange behaviors of the code. To assist client programmers, the creator of a method that uses call by reference should advertise any possible effects via a purpose-and-effect statement (method documentation), because a by-reference indicator alone does

not suffice. Such a statement should help programmers with understanding aliasing effects.

Note Over time, programming language creators have recognized that the downsides of call-by-reference argument passing outweigh the upsides. As a result many modern languages use call-by-value only.

The C family of languages, including C#, remains an exception, offering several by-reference parameter-passing mechanisms. In addition, C# supports forms of in and out parameters, inherited from the Ada language.

Old timers used to C-style coding may also make statements such as "objects are passed by reference" and "variables are passed by value." Such statements simply show a lack of understanding of semantic models. Objects in a *Class*-like language *are* semantic values and are mutable; no confusion of terminology is needed to explain these effects. Worse, variables don't even exist when a program is executed, so they can't be "passed."

5.4 Dynamic Dispatch

Method calls and field access/mutation pose the problem of *dynamic dispatch*. Consider a schematic method call expression: (o –> m(a, ...)). The method name m is just that, a name. Without any knowledge of the target object o, a programmer cannot know which method code is going to be evaluated in reaction to this expression. Since it is undecidable to determine which object o denotes from program text, the actual object becomes known only as we determine the program's semantics—that is, when the program runs.

```
(class Cowboy ()                (def a (new Artist ()))
   (method draw()               (def c (new Cowboy ()))
     1.0))                      (def x 0.0)

 (class Artist ()               (if0 [          ]
   (method draw()                    (x = a)
     666.0))                         (x = c))
                                (x --> draw())
```

Figure 40: An example of dynamic dispatch

Figure 40 displays an old and well-known illustration of dynamic dispatch for methods. The class definitions of the program—on the left side—introduces Cowboy and Artist, both of which come with a method named

draw. The names are to suggest that "drawing" for each is considered a rather different action. Here the methods return 1.0 and 666.0, respectively.

The program's body—on the right side of the figure—creates an instance of each class, named c and a, respectively. Next a conditional statement selectively assigns one of these objects to x. Finally, the return value is determined by calling draw on x.

Key to this code snippet is the empty box in the test position of the conditional. In *Class*, a programmer could create a random number generator, and the conditional could check whether a generated number is greater or equal to some limit. In an actual class-based language, say Java, a programmer could use an expression to check whether it is Wednesday or whether the number of inches of rainfall today exceeded some given amount. In short, it is easy to see how the content of x is not determined until the conditional is evaluated, even for simplistic example like the one in figure 40.

Like the object aliasing aspect of parameter-passing, dynamic dispatch is desirable and potentially dangerous. It is desirable when a form of information needs a data representation that requires several classes, all of which would implement a method with the same name. If you have chosen a class-based language for the projects of this book, you know that your data representation of ASTs is an example of this kind. By contrast, dynamic dispatch poses a problem when the data flow and control flow of some code is opaque. In that case, the code may call a method with the correct name and unexpected behavior.

Note While a developer can warn clients about by-reference parameters in the documentation, it is impossible to do so for dynamic dispatch. Only disciplined program development can help a programmer.

Exercise 40. Develop a *Class* program that illustrates dynamic dispatch and use it to formulate a unit test for main from exercise 34.—You may wish to extend *Class* with a (heads-or-tail) expression that defers to a random-number generator from your implementation language. **Hint** Consider testing the for "tail."

Exercise 41. Develop a *Class* program that illustrates dynamic dispatch for field access. Use it to formulate a unit test for main from exercise 34. Re-use the relevant parts of the solution to the preceding exercises.