Chapter 9

CLU and data abstraction

Contents

Dear Reader: This chapter is little changed from the related chapter in Sam Kamin's 1990 book Programming Languages: An Interpreter-Based Approach. *I have removed references to the Pascal code used in the original interpreter. This chapter will be revised in 2013. I am designing a new, statically typed version of μClu, as well as formal semantics, type rules, and a new interpreter. The type system of μClu will draw on and extend the polymorphic type system of Typed μScheme from Chapter 6. The new design will illustrate the following points:*

- *Bundling many operations into a single polymorphic "cluster" lightens the notational burden of fully explicit polymorphism and makes for a reasonably pleasant programming model. You'll be able to compare this restricted form of Typed μScheme with the different restricted form offered by μML's Hindley-Milner type system.*
- *Since in Clu, all you can do with a value of abstract type is use the operations of the value's cluster, we begin our type theory with the idea that a cluster could be represented as a record containing its exported operations. But to model Clu faithfully, this idea needs two additional developments:*
  - *Because clusters in Clu are* generative, *two clusters might have operations with exactly the same types and yet not be interchangeable. So the type of a cluster must include some notion of* identity.
  - *Unlike types we have seen so far, a cluster type may refer to itself. For example, a list cluster may contain a* `cdr` *operation which returns a list of the same type as the whole cluster. We therefore need a way to represent this self-reference without building infinite data structures. In this chapter we introduce the standard technique of writing $\mu\alpha.\tau$, in which μ is a binding construct a bit like λ or $\forall$. Only instead of being a function or a polymorphic type, $\mu\alpha.\tau$ stands for a solution to the recursion equation $\alpha = \tau$. For example, in the case of integer lists we might have the equation*

$$\alpha = \{\texttt{nil} : \alpha, \texttt{null?} : \alpha \to \texttt{bool}, \texttt{cons} : \texttt{int} \times \alpha \to \alpha, \ldots\}.$$

    As a representation of lists, any solution will do—that's the beauty of data abstraction. By inspecting the type system, it's easy to see that the clients of a cluster are be typechecked and compiled against the cluster's export list, and even if the representation is changed, the clients needn't be touched.

  The combination of a record of operations, an identity, a recursive type, and parametric polymorphism provides for a very expressive type system.
- *The new μClu will make it possible to present more interesting examples, which use Clu's innovative extension to parametric polymorphism: the* `where` *clause. The* `where` *clause provides yet another mechanism for providing comparison or equality functions, solving the problem described in Section 3.9.1 on page 103 without requiring* any *explicit argument-passing. We'll illustrate the principle with such examples as a polymorphic sort function and two different polymorphic priority queues.*

I hope hope to have completed the new chapter by the end of 2013.

Why does every language provide some form of procedure? Because calling a procedure frees the programmer from thinking about *how* a computation is performed; the procedure *hides the details of a computation.* For example, a procedure's local variables are private, so the caller of a procedure needn't worry about clashes between his own variables and the

	CLU	μClu	HW
Toplevel clusters	•	•	
Toplevel procedures	•	•	
Parametric polymorphism	•	•	
Checked where clauses in $\Lambda\tau.\cdots$	•	•	
Unchecked where clauses in $\tau[\tau]$	×	•	
Checked where clauses in $\tau[\tau]$	•		⋆
Nested, second-class procedures	•	×	
Non-nested, first-class procedures	×	•	
Multiple assignment	•	×	
Multiple results	•	•	
`signal`	•	•	
Try-catch	•		⋆
Iterators and for loops	•	×	
Export lists	•	×	
Export signatures	×	•	
Immutable records, variants	•	×	
Mutable records	•	•	
Mutable variants and `case`	•		•
Stratified syntax	•	×	
Own variables	•	×	
Multiple assignment	•	•	
`up` and `down`	•	•	
`cvt`	•		•
Implicit `letrec`	•	?	?
Dynamic types	•	×	×
Syntactic sugar for binary operators	•	•	
Syntactic sugar for record lvalues and rvalues	•		•
Syntactic sugar for array lvalues and rvalues	•		•

Notes:

- NR doesn't know how to typecheck recursive types

Figure 9.1: Design sketch of μClu

variables used in the procedure. If the implementation of procedure needs to be changed, most likely those changes will not will affect the rest of the program in unforeseen ways. In short, the interactions between the procedure and its clients—the procedures that call it—are primarily mediated by parameters and results; these interactions should be limited to what the procedure is *expected* to do, with no gratuitous side effects.[1] As has been understood since the first high-level programming languages were developed in the late 1950s, hiding implementation details from a procedure's clients provides inestimable benefits. The greatest advance in programming technique in the 1960s and 1970s was the realization that comparable benefits accrue from hiding the details of *the representation of data.*

In the late 1960s and the 1970s, many language designers were exploring ways to hide the representation of data. Ideas were tested and refined in Pascal, Simula 67, Alphard, Modula-2, and Clu, as well as many others. We have chosen Clu as a representative of these languages for several reasons:

- Clu offers an unusually clean, regular, orthogonal design. In particular, *all* types are treated as abstract types, and the types in the initial basis are on the same footing as the programmer's types.
- The designers of Clu developed not only a *programming language* but also a *methodology* for programming with abstraction (Liskov and Guttag 1986). One example of this methodology is the use of *exceptions* to signal the occurrence of nonsensical situations, such as asking for the first element of an empty list.
- The designers of many other languages have borrowed from Clu. Indeed, anyone today who designs a language involving abstraction, exceptions, iterators, or mutable abstract types would do well to study Clu.

An additional benefit of studying Clu is that it offers a form of parametric polymorphism that avoids both the horrendous syntactic clutter of Typed μScheme and the sometimes baffling error messages of ML type inference, while still providing lots of expressive power.

Clu is sometimes called an *object-based* language, the idea being that values are abstract "objects," with which the client code interacts at arm's length, through carefully defined abstract operations. The move for more data abstraction in programming languages can be seen as a first step towards today's desire for *object-oriented* languages. Indeed, we argue in Chapter 10 that "object-orientation = data abstraction + reuse." You will find facilities like Clu's in both C++ and Java, for example.

Clu doesn't quite fit with the other languages covered in this book. Scheme, ML, Smalltalk, and Prolog all lend themselves to knocking out small, interesting programs quickly. Clu programmers don't particular care about writing small programs quickly; they want to build large, rock-solid programs that will live a long time. Like the designers of ML, the designers of Clu created a powerful, expressive static type system that would catches many errors. But unlike the designers of ML, the designers of Clu wanted to break up programs into very small units and to minimize dependencies between units. This programming style, in which every type is sealed into a watertight compartment, shines brightest when applied to large programs—you can probably keep ten thousand lines of code in your

[1] The risk of gratuitous side effects explains why many teachers of programming frown on the use of global variables. Global variables have their place, but using them to control interactions between a procedure and its clients makes it much harder to understand how and when the procedure can be used safely. Global variables that are used to mediate interactions between procedures are one example of *shared mutable state*, which is almost always difficult to understand and use correctly. The powerful abstractions we have seen in μScheme and μML arose in part because the designers of Scheme and ML wanted to make it easy for programmers to avoid using mutable state entirely.

head, so if you're working with much less than a hundred thousand lines of code, it's hard to see the software-engineering benefits that Clu provides. We do our best to provide a μClu language that is small enough to understand, and to illustrate its use with suggestive programming examples, but if you want to develop a real feel for data abstraction, you will have to use your imagination more than in other chapters.

9.1 Data abstraction

What do we mean by "hiding the details of data representations?" We mean thinking more in terms of the *abstract* data that we would like to be manipulating in a program and less in terms of the *concrete* data that actually is being manipulated. Abstract languages encourage programmers to think about ideas, not about bits. Here are some examples:

Integers

Concrete view: Bit strings.

Abstract view: Mathematical integers.

This example may seem odd. After all, programmers are quite comfortable thinking of integers in the abstract; they are rarely concerned about their concrete representation as bit strings. It would be surprising, for example, to see this code in a C program:

```
if (n & 0x80000000)
  printf("n is negative\n");
else
  printf("n is nonnegative\n");
```

Programmers have learned to think abstractly about integers; they are much more likely to write `n < 0` than `n & 0x80000000`. And with good reason; the bit test baffles the non-expert, and it works correctly only on a computer that uses 32-bit two's-complement integers. The problem is that the bit test makes an unnecessary assumption about the representation of integers, whereas `n < 0` is guaranteed to work regardless of the representation.

Binary Tree

Concrete view: Record containing a `label` field, and two pointer fields to binary trees, `leftchild` and `rightchild`.

Abstract view: An object that can be queried for its label and for its left and right children.

Our concrete version of binary trees is just one possible representation among many. Yet, it would not be surprising to see a program containing this code:

```
r = t->rightchild;
printf("%s\n", r->label);
```

This code is guilty of the same crime as the sign-testing code: it makes unnecessary assumptions about the representation of binary trees. It would not survive a change of representation from records to parallel arrays or to strings, should such a change become desirable,[2] just as the sign-testing code would not survive a change to a 64-bit machine. By building in assumptions about data representations, we make our code more difficult to modify.

[2]Strings are a useful representation of many kinds of trees because by making pointers between nodes implicit, they reduce memory requirements dramatically. Knuth (1973) has many interesting examples of different representations of trees, as well as a discussion of their properties.

Priority queue

Abstract view: A collection of values in which a new value can be added efficiently and the smallest value (if any) can be found efficiently.

Concrete view: Array a in which element $a[i]$ is no larger than elements $a[2i]$ and $a[2i+1]$. No, wait! A binary tree in which the value at each node is no larger than the values at its left and right children.

Points on the plane

Concrete view: Record with two fields, giving the x and y coordinates.

Abstract view: A position on the plane, which can be determined in several ways: by its x and y coordinates, by its x coordinate, distance from the origin, and quadrant, by "polar coordinates," i.e., angle and distance, etc.

Again, code that assumes the stated concrete representation, using expressions like `p->xcoord`, is assuming too much; many representations of points are possible.

These examples suggest that programs should insulate themselves from the specific representation of data they use. Why could such insulation possibly be beneficial?

- It protects the client code from the effects of changes in representation. This protection is particularly important when the representation embodies a requirement, assumption, or design decision that is likely to change. For example, a new E-commerce venture may start out with a customer database suitable for at most a few hundred entries. If the venture is successful, however, it may need to support tens of thousands of customers, and the old data representations are not likely to remain appropriate. By hiding the representation from the client, the programmers can ensure a seamless move from one representation to another.

- It makes it easy to replace one representation with another, without breaking existing clients. For example, the most expensive operations in the μScheme interpreter from Chapter 3 are the operations on strings, names, and environments. Because the representations of names and environments are abstract, it is possible to replace the existing implementations with faster implementations (e.g., based on hash tables) while ensuring that existing clients continue to work.

- It protects internal "representation invariants" that can be essential to the efficiency, or even the correctness, of code that manipulates the representation. For example, the correctness of `strtoname` in Section 2.5.1 absolutely depends on having only one copy of any given name in the system, so that `strcmp(s, t) == 0` is always exactly equivalent to `strtoname(s) == strtoname(t)`. If the representation were exposed to clients, a careless or malicious client might break the internal invariants, introducing a bug into `strtoname`.

These benefits are magnified as a program evolves, especially when the program is modified by people who had nothing to do with the original code.

If you don't know the representation of a data type, what do *you know about it? You know the* operations *each data type provides, and you know* the effects of performing operations in sequence. *A simple and useful way to describe these effects is to use algebraic laws, as discussed in Section 3.4.2.*[3] *Let's revisit the examples above:*

[3]Not coincidentally, the method of *algebraic specification* was developed at the same time as CLU in the same MIT laboratory.

Integers *Integers have a rich algebraic structure which has been studied for centuries, known as a* ring. *For example, we know that* $x+y=y+x$, $x+(y\times z)=x\times y+x\times z$, $x+0=x$, $x-x=0$, *and many other useful laws. We also know that division by zero is not meaningful—and in Clu, any attempt to divide by zero is signalled by an* exception.

Binary Tree

Priority queue *At minimum, any implementation of a priority queue must provide a means of creating an empty queue, a means of inserting an element, and a means of removing a minimal element. If the representation is mutable, then the insert and "`deletemin`" operations can mutate queues in place; if the representation is immutable, a new queue will have to be returned along with any other results of interest. In Clu, the choice of whether a type is mutable is always left up to the programmer, but when we write specifications, we always use immutable versions of the operations. Here is one possible specification of a priority queue:*

```
deletemin(insert(empty, x)) = (x, empty)
deletemin(insert(q, y)) =
   (x, insert(q', y)) if deletemin(q) = (x, q') and x < y
   (y, q)             if deletemin(q) = (x, q') and y <= x
```

Because operations on immutable abstract types often produce multiple results, Clu allows a function to return any number of results.

Points on the plane

How can a programming language help and encourage programmers to maintain an abstract view of data? Ultimately, data must be represented concretely in integers, arrays, records, and so on. What can be done is to provide a syntactic "capsule" containing both the representation of a data type and all functions that need to know that representation. This capsule hides the representation from all outside programs. A program wishing to use this data type—a *client* of the data type—is given access only to the functions. For example, a Tree capsule defines functions *label*(*t*), *make-tree*(*l*, *t1*, *t2*), *leftchild*(*t*), and so on; a client of Tree can create and use trees via these operations, but can never discover how they are represented. Different languages can use different mechanisms to provide such a capsule; some are easier to use than others (Figure 9.2). In Clu, such a capsule is called a *cluster*, and it is very easy to use indeed.

9.2 The language

The idea of Clu is to allow users to define their own *data types*. A new data type is defined by writing a *cluster*, which defines how values of the type are represented and what functions are provided by that type. A value created by cluster operations is an *instance* of the cluster.

A cluster, then, consists of a datatype representation and a collection of functions operating on that representation. All the functions that need to know how the data is represented are included. For defining the representation itself, a programmer using full Clu has access to a large collection of built-in types and type constructors, plus language constructs for defining mutable and immutable sum and product types. To keep things simple, μClu provides only mutable product (record) types. Most important, the clients of a cluster may not

Language	*Mechanism*	*Explanation*
C	opaque structure	In C, a programmer can declare a structure type without giving its representation. A pointer to a value of such a type can only be manipulated abstractly, through functions that know the members of the structure. The "syntactic capsule" is the compilation unit, so it is typical to put the structure definition and all related functions into a single source file.
Scheme	function	In Scheme, S-expressions can always be probed with `null?`, `car`, `cdr`, etc.; the only truly opaque value is a function value. The only way to represent a truly abstract value is with a function; one gets the effect of operations on the function by passing it different arguments. The "syntactic capsule" is the body of the function. Abelson and Sussman (1985) contains several examples of this programming technique, including examples of using Scheme in object-oriented style.
ML	various	In ML, one can use the same techniques as in Scheme, but full ML also provides an `abstraction` keyword for defining abstract types. The Standard ML modules system also provides direct support for data abstraction; this support is called "opaque signature matching."
Smalltalk	instance variables	As we discuss in Chapter 10, Smalltalk values are defined in *classes*, and their representations are visible only inside the class in which they are defined (and its *subclasses*).
C++	`private`	C++ has the keyword `private`, which prevents clients from getting access to the detail's of a class's representation. The "syntactic capsule" is quite complex, as it may involve not only the body of the class, but also subclasses and "friends" classes.

Figure 9.2: Mechanisms for data abstraction found in different languages

use the operations of the representation; they may use only the functions exported by the cluster.

As an example, we define a cluster for representing points in the plane with integer coordinates. In μClu, the cluster starts by listing the operations that the cluster defines and makes available to the outside world, called the "exported" functions. These functions present an "abstract" picture of `Point` to the outside world (i.e. the clients). They need not make available each component of the representation (though in this case they do), because that kind of information is not inherent to point-hood, only to the specific representation chosen. Similarly, they need not allow each component to be altered individually (as in this case they don't), because such an operation can be done only with knowledge of the representation of values.

351 ⟨*point cluster using Cartesian representation* 351⟩≡

```
(cluster Point
  (export
    (new      (function (int int) Point))
    (abscissa (function (Point)   int))
    (ordinate (function (Point)   int))
    (reflect  (function (Point)   ()))
    (rotate   (function (Point)   ()))
    (nearer-origin  (function (Point Point) bool))
    (quadrant (function (Point) int)))
  (type rep (record (int x-coord) (int y-coord)))
  (private
    (sqrdist (function (Point) int)))
  (define new (x y) (up (make rep (x-coord x) (y-coord y))))
  (define abscissa (p) (rep$get-x-coord (down p)))
  (define ordinate (p) (rep$get-y-coord (down p)))
  (define reflect (p)
     (let (r (down p))
          (begin
             (set-x-coord! r (- 0 (get-x-coord r)))
             (set-y-coord! r (- 0 (get-y-coord r))))))
  (define rotate (p)
     (let ((r (down p)))
        (let ((temp (get-x-coord r)))
          (begin
             (set-x-coord! r (get-y-coord r))
             (set-y-coord! r (- 0 temp))))))
  (define nearer-origin (p1 p2) (< (sqrdist p1) (sqrdist p1)))
  (define sqrdist (p)
     (let ((r (down p)))
        (+ (* (get-x-coord r) (get-x-coord r))
           (* (get-y-coord r) (get-y-coord r)))))
)
```

We can't understand an abstract data type by looking at the representation; the whole point, after all, is that the representation is hidden. Instead, we need to understand the behavior of the exported functions, i.e., the *operations* on that type. As in Section 3.4.2, it is very helpful to classify the operations in three categories: *creators and producers*, which build new values of the abstract type, *observers*, which tell us something about a value of the abstract type, and *mutators*, which change a value of the abstract type. We use this classification to understand `Point`. Notice that we are not intending to explain the code itself; we are giving the *specification* of each function, i.e., the "abstract meaning" or the client's view.

The `Point` cluster in chunk 351 has only one creator:

`new`: Create a new `Point`, given its x and y coordinates.

The `Point` cluster exports several observers.

`abscissa` and `ordinate`: Return the x and y coordinates, respectively, of the argument.

`nearer-origin`: Test if the first argument is closer to the origin than the second.

`quadrant`: Return the quadrant number (also shown in Figure 9.3) of the point.

Finally, the `Point` cluster has two mutators:

`reflect` and `rotate`: Change the point to its reflection or rotation, respectively; these operations are illustrated in Figure 9.3.

The function `sqrdist` is used by `nearer-origin`, but it is not exported.

352 ⟨*initial basis stuff* 352⟩≡

```
(define sqr (x) (* x x))
(define abs (x) (if (< x 0) (- 0 x) x))
(define +1 (x) (+ x 1))
(define and (x y) (if x y x))
(define or (x y) (if x x y))
(define not (x) (if x 0 1))
(define <> (x y) (not (= x y)))
(define >= (x y) (or (> x y) (= x y)))
(define <= (x y) (or (< x y) (= x y)))
(define mod (m n) (- m (* n (/ m n))))
(define min (x y) (if (< x y) x y))
(define max (x y) (if (> x y) x y))
```

⟨*transcript* 353a⟩

```
-> (val p1 (Point$new 3 4))
-> (Point$rotate p1)
-> (Point$abscissa p1)
4
-> (Point$ordinate p1)
-3
-> (Point$reflect p1)
-> (Point$abscissa p1)
-4
-> (Point$ordinate p1)
3
-> (val p2 (Point$new 1 5))
-> (Point$nearer-origin p1 p2)
#t
```

```
-> (define enclosed-area (p1 p2)
     (abs (* (- (Point$abscissa p1) (Point$abscissa p2))
       (- (Point$ordinate p1) (Point$ordinate p2)))))
-> (enclosed-area p1 p2)
10
```

353a ⟨*transcript* 353a⟩≡ (352) 353b▷

```
-> (type intlist List[int])
-> (set x (intlist$cons 1 (intlist$cons 2 (intlist$nil)))) ; x is 1,2
-> (set y x) ; y is 1,2
-> (intlist$car x)
1
-> (intlist$car y)
1
-> (intlist$car (intlist$cdr x))
2
-> (intlist$rplaca y 3) ; y is 3,2, and so is x
-> (intlist$car x)
3
-> (intlist$car y)
3
-> (define length (l)
    (if (intlist$null? l) 0 (+1 (length (intlist$cdr l)))))
-> (length x)
2
-> (length y)
2
;
-> (define nth (n l)
     (if (= n 0) (intlist$car l) (nth (- n 1) (intlist$cdr l))))
-> (define changenth (n x l)
     (if (= n 0) (intlist$rplaca l x) (changenth (- n 1) x (intlist$cdr l)))))
```

ARRAYS SHOULD BE BUILT IN.

353b ⟨*transcript* 353a⟩+≡ (352) ◁353a 356▷

```
-> (type Array array[int])
-> (set A (Array$new 1 10))
-> (set i 0)
-> (while (< i 10) (begin (set i (+ i 1)) (Array$assign A i (* i i))))
-> (set i 0)
-> (while (< i 10) (begin (set i (+ i 1)) (print (Array$index A i))))
asdf
```

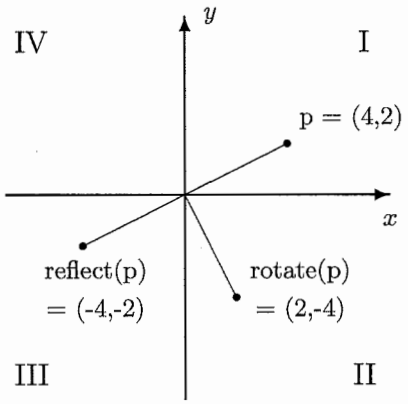

Figure 9.3: The operations of `Point`

We can make the specification more precise by giving algebraic laws, at least for creators, producers, and observers, as in Section 3.4.

```
(Point$abscissa (Point$new x y)) = x
(Point$ordinate (Point$new x y)) = y
```

Using algebraic laws to specify the behavior of mutators is trickier; we have to find notation to talk about the states of the point both *before* and *after* the mutation. The details are beyond the scope of this book.

The important point is that our discussion and specification do not mention of the representation of points. The operations of the cluster `Point` are operations that make sense for points in the plane, independent of any particular concrete representation. In the following session, the variables `p1` and `p2` contain instances of `Point`; though we know (from reading the code of `Point`) that these each contain two integers, we are not allowed to access those integers except as specifically permitted by the operations defined in the cluster. The function `enclosed-area` computes the area of a rectangle, given the locations of opposite corners:

```
-> (val p1 (Point$new 3 4))
-> (Point$rotate p1)
-> (Point$abscissa p1)
4
-> (Point$ordinate p1)
-3
-> (Point$reflect p1)
-> (Point$abscissa p1)
-4
-> (Point$ordinate p1)
3
-> (val p2 (Point$new 1 5))
-> (Point$nearer-origin p1 p2)
```

```
1

-> (define enclosed-area (p1 p2)
       (abs (* (- (Point$abscissa p1) (Point$abscissa p2))
          (- (Point$ordinate p1) (Point$ordinate p2)))))

-> (enclosed-area p1 p2)
10
```

The functions defined in cluster `Point` are always referenced, by its clients, via their *two-part names*, consisting of the cluster name and the function name, separated by "$".

Let's take a look at the cluster itself. As clients, we needed to know only the name of the cluster and the name and purpose of the functions it exports. We could think of the instances of the cluster as points in a perfectly abstract sense. Now we are placing ourselves in the role of the implementor of the cluster, and are going to look inside it.

As clients, we use the ideas of creators, producers, observers, and mutators as intellectual tools to understand an abstract type. As implementors, we will use three additional tools: the *representation* tells us how values of the abstract type are actually represented; the *representation invariant* tells us what representations we can expect to find at run time; and the *abstraction function* tells us the relationship between the concrete type and the idealized abstraction.

In ⟨*point cluster using Cartesian representation* 351⟩, the first thing after the export list is the line "`(type rep (record (int x-coord) (int y-coord)))`", which is followed by a sequence of function definitions. The "`rep`" line gives the representation; it says that points are to be represented by their x and y coordinates. The representation invariant is that these two points are integers. Although this fact is not stated explicitly in the program, it is important to the cluster's implementor, since it guarantees that operations like subtracting coordinates from 0 won't go wrong. Finally, the abstraction function is dead simple: the representation stands for the point in the plane whose coordinates are $(\mathtt{x-coord}, \mathtt{y-coord})$.

Again, this representation is of no interest to clients, but it is of central importance to the cluster's implementor. The operations are quite simple, once some basic facts about μClu are explained:

- `new` creates a new instance of `Point`, by applying the function `Point`. The latter is an *implicitly defined* function available within the cluster, which takes two arguments and creates a `Point`. In effect, it puts these two values into a package to which clients have access only via the cluster operations.

- `abscissa` and `ordinate` return the x and y coordinates of their argument, respectively, which also happen to be the two components of the point's representation. To get at these representation components, these functions apply the *selector* functions, which are also implicitly-defined and have the same names as the corresponding components. Thus, *within the cluster*, `x-coord` is defined to be a function that returns the `x-coord` component of any `Point`; outside the cluster, `x-coord` is undefined; and similarly for `y-coord`. Again, the idea is to deny clients access to the representation of points, so they are forced to think of points in the abstract.

- Another capability provided by implicitly-defined functions within the cluster, used by `reflect` and `rotate`, is modification of the representation of an instance. `set-x-coord` modifies the `x-coord` component of its first argument to be equal to its second argument. Note that this function *modifies its first argument*; it does not return a new instance of `Point`.

The value of hiding the representation from clients can best be illustrated by presenting an alternative representation, given in Figure 9.4. There, a point is represented by three values, the *absolute values* of its x and y coordinates, and its quadrant number. The representation invariant is that `x-mag` is nonnegative, `y-mag` is nonnegative, and `quad` is an integer from 1 to 4. The abstraction function is a bit difficult to state precisely; the representation stands for that unique point that is located an absolute distance of `x-mag` from the y axis, `y-mag` from the x axis, and that is located in the quadrant numbered `quad`, where the numbering is given in Figure 9.3.

Abstractly, the operations work exactly the same, and the session given above has identical results. If, however, clients had been allowed to use selectors, that fact could no longer be guaranteed, since the selectors of the two versions of `Point` are different. The situation is precisely analogous to the examples given in the introduction, where troublesome things happened when the user of a data type wrote representation-specific code. The whole idea of clusters is to discourage the writing of such code.

356 ⟨*transcript* 353a⟩+≡ (352) ◁353b 363b▷

```
-> (val p1 (Pointq$new 3 4))
-> (Pointq$rotate p1)
-> (Pointq$abscissa p1)
4
-> (Pointq$ordinate p1)
-3
-> (Pointq$reflect p1)
-> (Pointq$abscissa p1)
-4
-> (Pointq$ordinate p1)
3
-> (val p2 (Pointq$new 1 5))
-> (Pointq$compare p1 p2)
1
-> (define enclosed-area (p1 p2)
     (abs (* (- (Pointq$abscissa p1) (Pointq$abscissa p2))
       (- (Pointq$ordinate p1) (Pointq$ordinate p2)))))
-> (enclosed-area p1 p2)
10
```

357 ⟨*point cluster using magnitude-quadrant representation* 357⟩≡

```
(cluster Pointq
  (export
    (new      (function (int int) Point))
    (abscissa (function (Point)   int))
    (ordinate (function (Point)   int))
    (reflect  (function (Point)   ()))
    (rotate   (function (Point)   ()))
    (nearer-origin  (function (Point Point) bool))
    (quadrant (function (Point) int)))
  (rep (record (int x-mag) (int y-mag) (int quad)))
  (private
    (sqrdist (function (Point) int))
    (compute-quad (function (int int) int)))
  (define new (x y) (up (make rep (x-mag (abs x)) (y-mag (abs y))
                          (quad (compute-quad x y)))))
  (define abscissa (p)
    (if (> (get-quad (down p)) 2) (- 0 (get-x-mag (down p))) (get-x-mag (down p))))

  (define ordinate (p)
    (if (or (= (quad p) 2) (= (quad p) 3))
            (- 0 (get-y-mag (down p)))
            (get-y-mag (down p))))
  (define reflect (p)
     (set-quad! (down p) (+1 (mod (+1 (get-quad (down p))) 4))))
  (define rotate (p)
     (let ((temp (get-x-mag (down p)))
           (r    (down p)))
       (begin
         (set-x-mag! r (get-y-mag r))
         (set-y-mag! r temp)
         (set-quad! p (+1 (mod (get-quad r) 4))))))

  (define nearer-origin (p1 p2)
       (< (sqrdist p1) (sqrdist p2)))
  (define quadrant (p) (get-quad (down p)))
  ; compute-quad, sqrdist are not exported
  (define compute-quad (x y)
       (if (>= x 0)
           (if (>= y 0) 1 2)
           (if (< y 0) 3 4)))
  (define sqrdist (p)
         (+ (sqr (get-x-mag (down p))) (sqr (get-y-mag (down p)))))
)
```

Figure 9.4: Cluster Point

9.2.1 Syntax

The syntax of μClu draws elements from Typed Impcore and Typed μScheme, as well as adding unique elements of its own.

- Our standard imperative core
- Types, including function types, but no polymorphic types. (Polymorphism is restricted to clusters and to top-level procedures.)
- Let-bindings
- Cluster definitions
- Two-part names

def ::= `(cluster` *cluster-name* [`(`*type-formals*`)`] *exports types* {*impl*}`)`
| `(type` *type-name type-exp*`)`
| `(define` *function-name* [`(`*type-formals*`)`] `(`*formals*`)` [*returns*] [*signals*] *exp*`)`
| `(val` *variable-name exp*`)`
| *exp*

type-formals ::= { `'`*type-variable-name* | `(`*'type-variable-name* `where (`{*declaration*}`))` }
formals ::= {`(`*type-exp variable-name*`)`}
returns ::= `returns (`{*type-exp*}`)`
signals ::= `signals (`{*exception-name*}`)`
exports ::= `(export` {*declaration*}`)`
types ::= {`(type` *type-name type-exp*`)`}
declaration ::= `(`*name type-exp*`)`
| `(function` *function-name* (]{*type-exp*}`)` [*returns*] [*signals*]`)`
impl ::= `(val` *variable-name exp*`)`
| `(define` *function-name* `(`*formals*`)` [*returns*] [*signals*] *exp*`)`
exp ::= *value*
| *variable-name*
| *type-exp*`$`*function-name*
| `(@` *function-name* {*type-exp*}`)`
| `(set` *variable-name exp*`)`
| `(set (`{*variable-name*}`)` *exp*`)`
| `(if` *exp exp exp*`)`
| `(while` *exp exp*`)`
| `(begin` {*exp*}`)`
| `(make` *record-type* {`(`*field-name exp*`)`}`)`
| `(signal` *exception-name*`)`
| `(`*function* {*exp*}`)`
value ::= *integer*
function ::= *exp* | *sugared-function*
sugared-function ::= `+` | `-` | `*` | `/` | `=` | `<` | `>` | `and` | `or` | `not` | `print` | `set-`*name*`!` | `get-`*name*
type-exp ::= *type-name*
| `'`*type-variable-name*
| `(@` *cluster-name* {*type-exp*}`)`
| `(function (`{*type-exp*}`)` [*returns*] [*signals*]`)`
| `(record` {`(`*type-exp field-name*`)`}`)`

Names may not include the `[` or `]` characters, as these characters are reserved to provide syntactic sugar for arrays, as described in Exercises REFuclu.ex.record-sugar and REFuclu.ex.array-sugar.

9.2.2 Semantics

Functions defined inside a cluster have access to certain functions that cannot be used outside the cluster, namely:

- The names used in the representation (`x-coord` and `y-coord` in the first `Point` cluster, `x-mag`, `y-mag`, and `quad` in the second) can be used as functions which, when applied to an instance of the cluster, return the corresponding component of its representation. These functions are called *selectors*.
- For each selector, the name formed by prefixing its name by "`set-`" (`set-x-coord`, etc.) is a function of two arguments; when applied to an instance v_1 of the cluster and another value v_2, it sets the selected component of v_1 to v_2. These functions are *settors*.
- The name of the type (`Point`) is itself a function, called a *constructor*, which has as many arguments as the number of selectors. It forms a new instance of the type, having the given representation components.
- Each function defined in the cluster can be called by the other functions defined there, using its *one-part* name.

By contrast, access to the functions defined in a cluster from outside the cluster is strictly limited:

- Selectors, settors, and constructors cannot be used. This is to be expected, since these three types of functions cannot sensibly be used without knowledge about the type's representation.
- Functions defined in a cluster are referred to by two-part-names[4]. Furthermore, only those functions named in the cluster's export list can be used; others are private to the cluster.

9.2.3 Scope in CLU

The scope rules for variables are exactly as in Impcore. Variables are either formal parameters or global variables. This is equally true for functions defined within clusters as for functions defined at the top level.

For function names, the scope rules are an elaboration of those in Impcore, where all functions had global scope. A cluster defines a scope, making names available globally and also making some names visible within the cluster which are not visible outside it. In particular, the cluster exports the two-part names of the functions it defines. Internally, it defines both the one-part and two-part names of those functions, and also the constructor, selectors, and settors, as previously described.

9.2.4 Examples

μClu contains Impcore as a subset, so code from Chapter 2 still works. Our examples involve only what is unique to Clu, namely clusters.

359 ⟨*example cluster* `List` 359⟩≡

```
(cluster List [T]
```

[4]This is easy to forget, but doing so usually leads quickly to an "undefined function" error.

```
  (export
    (nil List[T])
    (null? (function (List[T]) bool))
    (cons  (function (T List[T]) List[T]))
    (get-car (function (List[T]) T       signals (Empty)))
    (get-cdr (function (List[T]) List[T] signals (Empty)))
    (set-car! (function (List[T] T) () signals (Empty)))
    (set-cdr! (function (List[T] List[T]) () signals (Empty))))
  (type cons (record (T car) (List[T] cdr)))
  (type rep (optional[cons]))
  (private
    (get-cons (List[T]) cons signals (Empty))
  )
  (val nil rep$none)
  (define null? (l) (not (rep$isSome (down l))))
  (define cons (x xs) (up (rep$some (make cons (car x) (cdr xs)))))
  (define get-car (l) (get-car (cons l)))
  (define get-cdr (l) (get-cdr (cons l)))
  (define set-car! (l x)  (set-car! (cons l) x))
  (define set-cdr! (l xs) (set-cdr! (cons l) xs))
)
```

Lists

The cluster `List` implements lists of values. The constructors are:

`nil`: The list of zero items.

`cons`: Take a list and a new element and returns a new, larger list. If `x` is the list $(3, 4, 5)$, then `(List$cons 2 x)` is $(2, 3, 4, 5)$. Note that `cons` is a constructor, not a mutator. This means that `x` is not changed, but rather that `cons` returns a new instance.

The observers are:

`car`: Return the first item in a list. Given `x` above, `(List$car x)` is 3. It is an unchecked run-time error to apply `car` to the empty list.

`cdr`: Return the list consisting of all items in the argument except the first. `(List$cdr x)` is $(4, 5)$. It is an unchecked run-time error to apply `cdr` to the empty list.

The `List` cluster also exports these mutators:

`set-car!`: "Replace the car," i.e., mutate the first argument, a list, by making its first element equal to the second argument. After calling `(List$set-car! x 8)`, `x` is $(8, 4, 5)$.

`set-cdr!`: "Replace the cdr," i.e., mutate the first argument, a list, by making its cdr equal to the second argument. If `x` is $(3, 4, 5)$, and `y` is $(7, 8, 9)$, then after calling `(List$set-cdr! x y)`, `x` is $(3, 7, 8, 9)$.

Here is a session using `List`:

```
-> (val x (List$cons 1 (List$cons 2 (List$nil)))) ; x is (1, 2)
-> (val y x) ; y is (1, 2)
-> (List$car x)
1

-> (List$car y)
1

-> (List$car (List$cdr x))
2

-> (List$set-car! y 3) ; y is (3, 2), and so is x
-> (List$car x)
3

-> (List$car y)
3

-> (define length (l)
    (if (List$null? l) 0 (+ 1 (length (List$cdr l)))))
-> (length x)
2
```

Another point to be made from this session is that the assignment `(val y x)` sets `y` and `x` to be the *exact same list*, so that subsequent changes to `y` via `set-car!` are at the same time changes to `x`. The technical name for this feature is *reference semantics*; the idea is

that assigning x to y makes both x and y references to the same object, so that if the object is mutated, both x and y see the mutation.

Having seen what List's are in the abstract, let us look at how they are represented. The representation has three components: type, a, and d. The representation invariant is that type is either 0 or 1, and furthermore, if type is 1, then d is an instance of List.[5] The abstraction function is given by the following rules:

- If type is 0, the concrete representation stands for the empty list.
- The type is 1, then the concrete representation stands for the list whose first element is a and whose remaining elements are given by d.

Note that if type-exp is 0, the values of a and d are irrelevant.

Arrays

A cluster that is frequently used, both for itself and to represent other types of data, is Array, implementing one-dimensional arrays with integer indices and arbitrary components. Unlike Array in full Clu, our arrays are not built-in and do not enjoy constant-time access to all components, but aside from that they provide the programmer with basically the same capability. To make the abstraction precise, an array is a *mapping* from integers to values, together with a lower and upper bound. The type comes with three operations.

- *The constructor* new. (Array$new b s) creates an array that maps every integer to zero, whose lower bound is b, and whose upper bound is b + s.
- *The observer* index. (Array$index A i) returns the value to which array A maps integer i.
- *The mutator* assign. The effect of (Array$assign A i x) depends on the value of the index i. If the index falls within the bounds of the array, i.e., is as large as the lower bound but not as large as the upper bound, then the effect of (Array$assign A i x) is to mutate the mapping, so that in the new mapping, i is mapped to x, and the mappings of other integers are not affected. If the index falls outside the bounds, (Array$assign A i x) has no effect.

Looking at the implementation of the cluster, we see that arrays are represented by a record containing three values: base, size, and elts. The representation invariant is that base is an integer, size is a nonnegative integer, and elts is a list of length size. The size is therefore redundant, but having it cached in a record element makes some of the operations faster. The abstraction function is defined by the following rules:

- The lower bound of the array is base.
- The upper bound of the array is base + size.
- The mapping defined by the array is the function that maps the integer i to 0, when $i <$ base or $i \geq$ size, or to the nth element of list elts, where $n = i -$ size (and where list elements are numbered from 0).

[5] This kind of recursive representation, in which part of the representation of List may contain an instance of List, is perfectly legal in Clu.

363a ⟨*more Kamin stuff* 363a⟩≡ 364a▷

```
(cluster Pair
    ; Exports: fst, snd, mkPair
    (rep f s)
    (define fst (p) (f p))
    (define snd (p) (s p))
    (define mkPair (x y) (Pair x y))
)
```

Figure 9.5: Cluster `Pair`

Once we understand the representation invariant and abstraction function, it's easy to see how the implementations work. When `(new b s)` is called, it forms the new `Array` instance by creating a list of `s` zeros and packaging it together with `b` and `s`. Subscripting is a straight-forward process of cdr'ing down the list the correct number of times. `(assign A i x)` mutates the array `A` by mutating the list `elts`; note the use of `set-car!` in `changenth`.

ARRAYS SHOULD BE BUILT IN!

The operations `zerolist`, `nth`, and `changenth` are not really operations on arrays, but are instead operations on lists which are auxiliary to the array operations. `zerolist` is included as a non-exported function in the cluster, but `nth` and `changenth` are defined globally; we discuss this decision below.

Sparse Arrays

When an array is expected to be filled mostly with zero entries, the normal array representation may waste too much memory. Here we present a cluster for sparse arrays, in which an array is represented by a list of pairs.

- The representation still uses integers `base` and `size` and list `elts`.
- The representation invariant is different. `base` and `size` give the bounds of the array as before, but `elts` is now a list of (i_j, x_j) pairs such that no two i_j, $i_{j'}$ are equal, i.e., $j \neq j' \implies i_j \neq i_{j'}$. The length of `elts` need not bear any relation to `size`. Finally, every i_j appearing in `elts` falls within the bounds.
- The abstraction function for the bounds is the same, but the abstraction function for the mapping is different. The mapping is the function mapping i to 0, if there is no (i_j, x_j) pair in `elts` with $i_j = i$. If there is an (i_j, x_j) pair in `elts` with $i_j = i$, this pair is guaranteed to be unique (by the representation invariant), and the array maps i to x_j.

The cluster `Pair` is shown in Figure 9.5, and `SpArray` in Figure 9.6 The function `assoc` is the basic searching function; `(assoc i l)` returns either the empty list or a suffix $((i_j, x_j) \ldots (i_n, x_n))$ of the list `l`, such that `i` $= i_j$. Note that `SpArray` is mutable; in `assign`, `A` itself is changed either by consing on a new pair (i, x), or, if $i = i_j$ for some j, changing the j^{th} pair to (i, x).

Sparse arrays are a standard example of a change of data representations that should be transparent to the client. Any client of `Array` can be readily modified to be a client of `SpArray`, simply by changing function calls of the form "`Array$`$\cdots$" to "`SpArray$`$\cdots$".

363b ⟨*transcript* 353a⟩+≡ (352) ◁356 369b▷

```
-> (val A (SpArray$new 1 10))
```

364a ⟨*more Kamin stuff* 363a⟩+≡ ◁363a

```
(define assoc (i l)
       (if (intlist$null? l) l
            (if (= (Pair$fst (intlist$car l)) i)
                 l
                 (assoc i (intlist$cdr l)))))
```

364b ⟨*sparse-array cluster* 364b⟩≡

```
(cluster SpArray [T where (default T)]
  (export
    (new (function (int int) SpArray[T]))
    (at (function (SpArray[T] int) T))
    (put (function (SpArray[T] int T) ())) signals (Bounds))
  (type pair (record (int index) (T value)))
  (type rep (record (int base) (int size) (List[pair] elts)))
  (define new (b s) (up (make rep (base b) (size s) (elts List[pair]$nil))))
  (define at (A i)
     (let l ((assoc (get-elts (down A))))
       (if (List[pair]$null? l) T$default (get-value (List[pair]$car l)))))
  (define put (A i v)
     (let l ((assoc (get-elts (down A))))
       (if (List[pair]$null? l)
         (set-elts (down A) (List[pair]$cons (make pair (index i) (value v)) l))
         (set-value (List[pair]$car l) v))))
) ; bounds lost!!
```

364c ⟨*junk* 364c⟩≡

```
(define out-of-bounds (A i)
      (or (< i (base A)) (> i (- (+ (base A) (size A)) 1))))
```

Figure 9.6: Cluster SpArray

```
-> (val i 0)
-> (while (< i 10) (begin (set i (+ i 1)) (SpArray$assign A i (* i i))))
-> (set i 0)
-> (while (< i 10) (begin (set i (+ i 1)) (print (SpArray$index A i))))
```

Mutable types vs. immutable types

Clusters whose instances can be changed by cluster operations, i.e., clusters that export mutators, are called *mutable*. If a cluster is mutable, its operations use settors, but it is possible for a cluster to use settors *without* being mutable; an operation may create a new instance and then apply settors to it rather than to an argument. A later example, `Poly`, does exactly this. `SpArray`, `Array`, `List`, and both versions of `Point` are mutable; `Pair` is our only immutable cluster thus far. Note that mutability is an "abstract" property of clusters, one which must be appreciated by a client. Another way to say this is that it is a property of *data types*, not of the *clusters* that implement them.

It seems to help programmers to distinguish between mutable and immutable types. In ordinary programming languages, "small" values, such as integers, characters, and enumeration literals, are usually immutable. Structured values such as arrays and records are often mutable. Some programming languages give the programmer considerable control over mutability; for example, full Clu provides both mutable and immutable record types. Full ML provides mutable "arrays" and immutable "vectors."

The choice between mutable and immutable types, or equivalently, between constructors and mutators, is not always obvious. Programming with immutable types often requires extra copying. If one element of a mutable array changes, we can just mutate that single element in place. But if one element of an immutable vector changes, we have to build an entire new vector, and that may mean copying the old one. Immutable types have the great advantage that it is easier to understand their behavior; because they don't encapsulate mutable state, the same function applied to the same value always has the same effect. In other words, we don't have to reason about order of evaluation. Our study of Scheme and ML has shown that it is easy to write programs that don't use mutable types. [6] This tradeoff explains why small values are usually immutable, whereas large values are sometimes mutable; if a value is small (e.g., fits in one machine register), the cost of copying is the same as the cost of mutating, and we may as well get the advantages of immutability. One advantage of Clu is that the designers thought carefully about how to specify and implement mutable types, so if we are careful, we can program with them almost as easily as with immutable types.

Where should functions be defined?

Consider the function `nth`, used in cluster `Array`. There are three places in which it could plausibly be defined:

- In the `List` cluster, since it is a list operation. However, it does not need to know the representation used by `List`, so that it *can* be defined as a client. If we were to follow the principle of including in each cluster all the operations naturally associated with that type, even those which *could* be defined outside it, several bad things might occur in the long run:
  - Clusters would get unmanageably large.
  - If there were more than one cluster representing a type, these definitions would have to be repeated in each one, expanding the total volume of code and presenting a severe version control problem.
  - Most importantly, this policy implies making the cluster effectively public, to allow for new functions to be added. Having clusters be so public invites pro-

[6] Such programs may still use mutation, via `set`, but even that is often unnecessary.

grammers to write representation-dependent code, and thus mitigates the benefits of representation-independence.

According to these arguments, a cluster should be limited to those operations having a legitimate need to know the representation.

- At the top level. Succumbing to the arguments just given, the choice was made to define `nth` globally. However, this approach is not without disadvantages. Mainly, with `List`-oriented code spread all over, the programmer cannot easily determine whether a certain function has previously been written; re-use of code is hindered.
- As a non-exported function in `Array`. This approach would have the advantage of maintaining the independence and "territorial integrity" of `Array`. For `zerolist`, the decision was made that there is so little likelihood of outside interest that defining it within `Array` was justified. However, for a function of general interest like `nth`, it would end up being defined locally in many clusters, again leading to a problem of code bulk and version control.

Deciding where to define functions—like deciding what data types to use in a given program—is not an exact science. The programmer must weigh the value of a variety of benefits, for example: keep the cluster itself simple, including only truly representation-dependent code; keep all functions associated with a given data type together; maintain the independence of a cluster, by having it define what it needs. These benefits cannot all be realized at once, and the best tradeoff will depend on circumstances. In writing the code for this chapter, we tried to balance these principles in each particular case.

9.3 Implementation

The original implementation of μClu, written in Pascal, is omitted from this revision. The final revision will have a new implementation written in ML.

9.4 Example—Polynomials

This example is from Liskov and Guttag (1986, pg. 68). It is the cluster `Poly`, an immutable type whose elements are polynomials over a single variable, which we refer to as x. To simplify the implementation, we define a restricted abstraction: polynomials of degree at most 19.

367 ⟨*polynomial cluster* 367⟩≡

```
(cluster Poly
    ; Export: create, degree, coeff, zero?, add, minus, sub, mul, prnt
    (rep coeffs lo hi)
    (define create (c n)
        (begin
            (set A (Array$new 0 20))
            (Array$assign A n c)
            (Poly A n n)))
    (define degree (p) (hi p))
    (define coeff (p n)
        (if (or (< n (lo p)) (> n (hi p))) 0 (Array$index (coeffs p) n)))
    (define zero? (p) (= 0 (coeff p (lo p))))
    ⟨ring operations on polynomials 368a⟩
```

```
    (define prnt (p)
     (if (zero? p) (begin (print 0) (print 0))
       (begin
         (set expon (hi p))
         (while (>= expon (lo p))
           (if (= (coeff p expon) 0)
             (set expon (- expon 1))
             (begin (print (coeff p expon)) (print expon)
                    (set expon (- expon 1))))))))
  ; set-coeff, remove-zeros not exported
    (define set-coeff (p n c)
          (Array$assign (coeffs p) n c))
    (define remove-zeros (p) ; (lo p) is too low, and/or (hi p) too high
        (begin
            (while (and (= 0 (coeff p (lo p))) (<= (lo p) (hi p)))
                (set-lo p (+1 (lo p))))
            (if (> (lo p) (hi p)) ; p a zero polynomial
                (begin (set-lo p 0) (set-hi p 0))
                (while (= 0 (coeff p (hi p)))
                       (set-hi p (- (hi p) 1))))))
)
```

368a ⟨*ring operations on polynomials* 368a⟩≡ (367) 368b▷

```
(define add (p q)
    (begin
        (set result (create 0 0))
        (set-lo result (min (lo p) (lo q)))
        (set-hi result (max (hi p) (hi q)))
        (set i (lo result))
        (while (<= i (hi result))
            (begin
                (set-coeff result i (+ (coeff p i) (coeff q i)))
                (set i (+1 i))))
        (remove-zeros result)
        result))
```

368b ⟨*ring operations on polynomials* 368a⟩+≡ (367) ◁368a 368c▷

```
(define minus (p)
    (begin
        (set result (create 0 0))
        (set-lo result (lo p))
        (set-hi result (hi p))
        (set i (lo p))
        (while (<= i (hi p))
            (begin
                (set-coeff result i (- 0 (coeff p i)))
                (set i (+1 i))))
      result))
```

368c ⟨*ring operations on polynomials* 368a⟩+≡ (367) ◁368b 369a▷

```
(define sub (p q)
    (add p (minus q)))
```

369a ⟨*ring operations on polynomials* 368a⟩+≡ (367) ◁368c

```
(define mul (p q)
    (begin
         (set result (create 0 0))
         (if (> (+ (hi p) (hi q)) 19) result ; error!
             (if (or (zero? p) (zero? q)) result
                 (begin
                      (set-lo result (+ (lo p) (lo q)))
                      (set-hi result (+ (hi p) (hi q)))
                      (set p-hi (hi p))
                      (set q-hi (hi q))
                      (set q-lo (lo q))
                      (set i (lo p))
                      (while (<= i p-hi)
                           (begin
                                (set j q-lo)
                                (while (<= j q-hi)
                                          (begin
                                 (set-coeff result (+ i j)
                                       (+ (coeff result (+ i j))
                                          (* (coeff p i) (coeff q j))))
                                   (set j (+1 j))))
                              (set i (+1 i))))
                      result)))))
```

369b ⟨*transcript* 353a⟩+≡ (352) ◁363b 369c▷

```
(define diff (p)
    (begin
        (set n 1)
        (set pdx (Poly$create 0 0))
        (while (<= n (Poly$degree p))
            (begin
                (set pdx (Poly$add pdx
                       (Poly$create (* n (Poly$coeff p n)) (- n 1))))
                (set n (+1 n))))
        pdx))
```

369c ⟨*transcript* 353a⟩+≡ (352) ◁369b

```
(set p (Poly$create 5 2))
(set q (Poly$create 3 1))
(set r (Poly$add p q))
(Poly$prnt (diff r))
10
1
3
0
```

```
(define diff (p)
    (begin
       (set n 1)
       (set pdx (Poly$create 0 0))
       (while (<= n (Poly$degree p))
          (begin
             (set pdx (Poly$add pdx
                      (Poly$create (* n (Poly$coeff p n)) (- n 1))))
             (set n (+ 1 n))))
       pdx))
```

Figure 9.7: Function `diff`

Because polynomials are immutable, they export no mutators, only constructors and observers. The *primitive* constructor, used to create polynomials from scratch, is:

`create`: Given arguments c and n, return the polynomial cx^n.

The cluster also provides several *non-primitive* constructors. These functions are used to build new instances from existing instances.

`add`: Add two polynomials.

`minus`: Negate a polynomial.

`sub`: Subtract two polynomials.

`mul`: Multiply two polynomials.

Finally, the observers are:

`degree`: The highest power of x that has a non-zero coefficient.

`coeff`: The coefficient of a given power of x.

`zero?`: Test if a polynomial is zero.

`prnt`[7]: Print the non-zero coefficients and corresponding exponents. Polynomial $5x^2 + 7$ prints as:

```
5
2
7
0.
```

Here, then, is a use of `Poly`. The client function `diff`, given in Figure 9.7, does differentiation:

[7]Why "`prnt`" instead of "`print`"? Because `print` is called from within `prnt`, to print integers, and if the cluster had a function called `print`, then, by the scope rules, it would be the one invoked. In short, if a cluster function has the same name as a built-in (or any other globally-defined) function, it is impossible to call that built-in function from within the cluster.

```
-> (val p (Poly$create 5 2)) ; represents polynomial 5x^2
<userval>
-> (val q (Poly$create 3 1)) ; represents 3x
<userval>
-> (val r (Poly$add p q)) ; represents 5x^2 + 3x
<userval>
-> (Poly$prnt (diff r)) ; derivative of 5x^2 + 3x is 10x + 3
10
1
3
0
```

Polynomials are represented by three items of data: the 20–element array `coeffs`, giving the coefficients, and the integers `lo` and `hi`. The representation invariant is

- $0 \leq$ `lo` and `hi` < 20
- If $i <$ `lo`, then (`Array$index coeffs` i) is zero.
- If $i >$ `hi`, then (`Array$index coeffs` i) is zero.
- If there is a nonzero element in `coeffs` then `lo` $\leq$ `hi`, (`Array$index coeffs lo`) is nonzero, and (`Array$index coeffs hi`) is nonzero.

In other words, `lo` and `hi` give the exponents corresponding to the lowest and highest non-zero coefficients. The abstraction function is that the concrete representation stands for the polynomial $\sum_i$ `coeffs`$[i]x^i$. This abstraction function, together with the reprsentation invariant, implies that any concrete representation in which (`Array$index coeffs lo`) is zero stands for the zero polynomial.

The coding of the functions is generally straightforward. `mul` determines the coefficient of x^n by adding the products of all coefficients of x^i in `p` and x^j in `q` for which $i + j = n$. `add` adds the coefficients of x^i in `p` and `q`, for all i; when done, it calls `remove-zeros`, which adjusts `lo` higher and/or `hi` lower, if their coefficients are zero.

`Poly` is an immutable data type. The exported functions that return polynomials—`add`, `minus`, `sub`, and `mul`—return new instances of `Poly` and do not modify their arguments. Although the function `set-coeff` does mutate its argument, it is not exported and is applied only to new instances.

9.5 Clu as it really is

Clu is a language for building reliable programs. Data abstraction via clusters is the principal feature supporting this goal. It is not the only one, however. One of the most crucial is *compile-time type checking*, as in C, ML, and Java. Another is *exception handling*. Following a brief discussion of the real Clu syntax and some aspects of clusters which are absent from μClu, we present these other features of full Clu.

9.5.1 Syntax

Clu syntax is traditional, rather Pascal-like. References to cluster operations use two-part names just as in our version, but several data types we have defined are built-in and use special syntax; for example, arrays use Pascal syntax for indexing and assignment.

```
Poly = cluster is create, degree, coeff, iszero, add, minus, sub, mul, prnt

        rep = record[coeffs: array[int], lo, hi: int]

        create = proc (c, n: int) returns (Poly)
              A: array[int] := array[int]$create(0)
              A[n] := c
              return (up(rep${coeffs: A, lo: n, hi: n}))
        end create

        degree = proc (p: Poly) returns (int)
              return (down(p).hi)
        end degree

        coeff = proc (p: Poly, n: int) returns (int)
              r: rep := down(p)
              if (n < r.lo) | (n > r.hi)
              then return (0)
              else return (r.coeffs[n])
              end
        end coeff
        .
        .
        .
end Poly
```

Figure 9.8: Cluster *Poly* in real Clu

Here is our differentiation function (Figure 9.7) in full Clu:

```
diff = proc (p: Poly) returns (Poly)
      n: int := 1
      pdx: Poly := Poly$create(0,0)
      while (n <= Poly$degree(p))
          pdx := Poly$add(pdx, Poly$create(n*Poly$coeff(p, n), n-1))
          n := n+1
      end
      return (pdx)
end diff
```

9.5.2 Clusters in full Clu

Syntax

Figure 9.8 contains part of the code for the `Poly` cluster, as it would appear in full Clu. Some points may require explanation:

- A record with three components is used for the representation, introduced by the keyword **rep**. The notation for record types and record selection is close to that of Pascal.
- The *coeffs* component is declared to be of type *array[int]*. This is the type of arrays with integer entries. The index type of arrays is always *int*.

- In *create*:
  - Variables may be declared and intialized in the same statement.
  - The *create* operation on arrays has only one argument. It gives the lower index bound; the upper index bound is initially set to one less than this value (giving an empty array), but it can be increased dynamically.
  - The keyword **rep** is used in cluster operations as an abbreviation for the representation type. Record constants can be written by writing the type of the record (here, **rep** abbreviates it), a "$", then the value of each field, enclosed in curly braces.
  - The keyword **up** is used similarly to our constructor functions: it turns an element of the representation type into an instance of the cluster.
- In *degree*:
  - **down** is the inverse of **up**: it turns an instance of the cluster into an instance of the representation type. This permits record selection to be applied to p.

There is one other syntactic abbreviation that should be mentioned, because it occurs frequently. Note that **up** and **down** are used in very stereotyped ways: **down** is applied to instances of the cluster when they are passed in as arguments, and **up** is applied before instances are returned as results. To handle these situations more conveniently, the keyword **cvt** can appear in an argument list of a cluster function or as the return type; it causes **up** or **down** to be applied at the appropriate time. Thus, *create* and *degree* become:

```
create = proc (c, n: int) returns (cvt)
        ⋮
        return (rep${coeffs: A, lo: n, hi: n})
end create

degree = proc (p: cvt) returns (int)
        return (p.hi)
end degree
```

Parameterized clusters

Our version of *Poly* allows only polynomials with integer coefficients. There is no real reason for this, since addition and multiplication of polynomials is the same regardless of the type of coefficient; the only requirement is that it be possible to add, negate, and multiply coefficients, and test if they equal zero. In real Clu, *Poly* could be written to permit different types of coefficients, by using *type parameters*. Such clusters are called *generic*, or *polymorphic*[8]. Figure 9.9 gives part of the polymorphic version of *Poly*. The type parameter t is just like a type parameter in a `TYLAMBDA` in Typed μScheme (Chapter 6). The **where** clause can be modeled as an ordinary lambda abstraction over the values *add*, *mul*, *sub*, *zero*, and *iszero*.

Given the polymorphic cluster *Poly*, we can write:

[8]These are similar to the polymorphic type constructors of ML, like `'a list`.

```
Poly = cluster [t: type] is create, degree, coeff, iszero, add, minus, sub, mul, prnt
    where t has
        add, mul, sub: proctype (t,t) returns (t)
        zero: proctype () returns (t)
        iszero: proctype (t) returns (bool)

    rep = record[coeffs: array[t], lo, hi: int]
        ⋮
    add = proc (p, q: cvt) returns (cvt)
        result: rep := rep$create(array[t]$create(0),0,0)
        result.lo := min(p.lo, q.lo)
        result.hi := max(p.hi, q.hi)
        i: int := result.lo
        while (i <= result.hi)
            result.coeffs[i] := t$add(p.coeffs[i], q.coeffs[i])
            i := i+1
        end
        removezeros(result)
        return result
    end add
    ⋮
    end Poly
```

Figure 9.9: Cluster *Poly*, polymorphic version

```
intpoly: Poly[int]
realpoly: Poly[real]
polypoly: Poly[Poly[int]]
```

Each of these applications of *Poly* combines an explicit type application, such as is used in Typed μScheme, with an implicit ordinary application to the *add*, *mul*, and other functions of the argument cluster. A future edition of this book may present a version of μClu that supports static type checking with this sort of explicit polymorphism.

Iterators

Container abstractions are so called because they contain other values; lists, sets, arrays, and tables are common examples. Given a container C, one frequently wants to iterate over all of the things contained, i.e., to perform a computation of the form "forall $x \in C$ do ..." Functional languages like Scheme and ML provide `foldl`, `map`, and similar functions; as we show in Section 10.3.4, object-oriented languages provide special methods for iteration. The designers of Clu thought iteration was so important that they built special support into the language; full Clu clusters can define not only constructors, observers, and mutators, but also *iterators*. Iterators are not like the ordinary functions or procedures that implement constructors, observers, and mutators. Iterators can be used only in special **for** statements, which iterate over sequences of values.

We use *List* as an example. Here is an iterator that produces, one after another, of all the elements in a list; it would appear within the cluster, like a cluster operation. Assume lists are represented by records having a *car* and a *cdr* field (much as in our version); "~" is the logical "**not**" operator:

```
elements = iter (l: cvt) yields (int)
    while (~isnull(l))
        yield (l.car)
        l := l.cdr
    end
end elements
```

When *elements* is invoked, the **while** loop is executed as usual, but whenever the **yield** statement is encountered, execution of the loop is suspended and the argument of **yield** is "produced." The client calls this iterator within a **for** statement, which generalizes **for** statements of Fortran and C:

```
sum: int := 0
for i: int in List$elements (l) do
    sum := sum + i
end
```

The $i : int$ is a binding instance; the **for** statement binds i to the values produced by the iterator, one after another. For each such binding, it executes the body of the loop. On each iteration of the **for** loop, the iterator is resumed from just after the last-executed **yield**. When the iterator terminates (in this example, when *isnull*(l)), the **for** loop does likewise. Iterators can be simulated very easily in terms of nested functions, even in languages that only allow functions to be passed as arguments, not returned as results, They are nevertheless extremely convenient syntactic sugar.

9.5.3 Type checking

As we discuss in Chapter 7, there is a broad consensus among language designers and theoreticians that type information should be used to catch program errors *at compile time*. The design of a type system must balance users' needs for flexibility, efficiency, reliability, and freedom from cumbersome notation. The C programming language, for example, provides great flexibility and efficiency, as well as some reliability, with only a modicum of notation; types of variables must be declared, but the compiler does the rest. The reliability of C programs is compromised, however, because the type system permits the programmer to perform many unsafe conversions between pointer types. The ML type system, on the other hand, provides flexibilty and reliability with a minimum of notation, but at some cost in efficiency. As an extreme example, the type system of Typed μScheme provides even more flexibility, with the same efficiency as ML, but at great notational cost.

The Clu type system occupies an intermediate point in the design space. Like ML, Clu is safe; there are no loopholes in the type system. Clu is also very flexible, providing that flexibility using the same mechanism as ML: polymorphism via parameterized types. When compared with Typed μScheme, both ML and Clu provide simplified versions of parametric polymorphism, but the simplifications are different. Unlike ML but like Typed μScheme, Clu requires that programmers declare the types of variables. What simplifies Clu is that "cluster application" combines both type application (instantiation) and the application of higher-order functions. This simplification make the notational burden bearable. Moreover, because clusters and functions do not nest, it is possible for Clu to be implemented more or less as efficiently as typical implementations of C, as opposed to using the heap-allocated closures required by typical implementations of ML. In short, at the cost of some notation, Clu's type system provides convenience, flexibility, and efficiency, as well as the safety and reliability of compile-time type checking.

```
sort = proc [t: type] (a: array[t], lo, hi: int) returns (array[t])
       where t has lt: proctype (t,t) returns (bool)
       i: int := lo
       j: int := i+1
       while (i < hi)
             while (j <= hi)
                   if (t$lt(a[j], a[i]))
                   then
                         tmp: t := a[i]
                         a[i] := a[j]
                         a[j] := tmp
                   end
                   j := j+1
             end
             i := i+1
       end
end sort
```

Figure 9.10: Polymorphic procedure *sort*

We have mentioned that clusters can have type parameters. Procedures can have them also. As an example, we show a polymorphic sort routine in Figure 9.10.

9.5.4 Exception handling

The introduction of data abstraction creates a serious problem in the handling of "corner cases" or "error cases." For example, if the `cdr` operation is supposed to return a list, what can it do if it is applied to the empty list? In languages like C, C++, and Java, this is not a problem, because an implementation can always return the null pointer. But this strategy works only because C, C++, and Java expose the fact that the list is represented by a pointer. In Clu, where the representation of a cluster his hidden *completely*, the null-pointer trick is not possible. What is the poor programmer to do instead? It would be very annoying to add a special "error value" to every abstraction; not only would this clutter the specifications, but the clients would constantly have to check to see if the result of an operation was an error.[9] The designers of Clu thought carefully about how to handle exceptional cases with minimum inconvenience to the programmer, and they developed the idea of *signals*, now commonly called *exceptions* (Liskov and Snyder 1979). This was a major contribution of the language, because few language designers had confronted the issue. You have already seen the utility of exceptions, because our interpreters from Chapter 5 on make heavy use of them. Today, exceptions appear in most major languages, including Ada, C++, Java, ML, and Modula-3. In each of these languages, the exception mechanisms is closely based on the pioneering work of Clu. We describe a subset of Clu's exception-handling mechanism.

Three new statements are added to Clu to implement this feature. In what follows, *e-name* is an arbitrary identifer chosen (by the programmer) to represent some type of exception:

signal *e-name* This statement causes the current procedure to terminate abruptly; *e-name* determines what *exception handler* will be invoked. A procedure that wants to signal an exception must declare the exception name. Here is an example:

[9] Anyone who has ever forgotten to check the return code from a Unix system call should sympathize.

```
car = proc (l: cvt) returns (int) signals (nil_error)
      if (l.type = nil)
      then signal nil_error
      else return (l.carval)
      end
end car
```

statement$_1$ **except when** *e-name*: *statement*$_2$ **end** This compound statement defines the handler for exception *e-name* in the context of *statement*$_1$. *statement*$_1$ is executed; if, during its execution, *e-name* is signalled by a procedure that it has called, then *statement*$_2$ is immediately executed. An example is:

```
v := List$car(l)
        except when nil_error: v := 0 end
```

statement **resignal** *e-name* If *statement* may receive a signal *e-name* from a procedure that it calls, but it does not wish to handle the exception itself, it may signal the exception to its caller. This is equivalent to:

statement **except when** *e-name*: **signal** *e-name* **end**

Note that a statement always signals an exception to the calling routine. Thus, if the **signal** is executed in this statement:

··· `signal` *e-name* ··· `except when` *e-name*: *statement* `end`

statement is *not* executed. Rather, the procedure containing this entire statement is terminated, and *e-name* is signalled to its caller.

If a procedure fails to provide a handler for an exception, and if it receives a signal for that exception, the exception is automatically handled by signalling the special exception *failure*, which is implicitly declared globally in every Clu program.

9.6 Summary

In Liskov et al. (1977), the designers of Clu write:

> The motivation for the design of the Clu programming language was to provide programmers with a tool that would enhance their effectiveness in constructing programs of high quality—programs that are reliable and reasonably easy to understand, modify, and maintain.

Their principal method of providing such a tool to encapsulat data representations in *clusters*. Clusters enhance the readability of programs by allowing their division into logical units larger than procedures; they enhance their maintainability by restricting knowledge of data representations to those with a "need to know," thereby limiting the effects of code modifications; they also enhance their provability, by allowing the structuring of program proofs along data abstraction lines. In addition, Clu has strict type checking and an exception-handling feature.

9.6.1 Glossary

abstraction function A map from the instances of a cluster to the abstract values they represent. It should be part of the design of any cluster, even if not proved correct. To prove correctness, the abstraction function must be a homomorphism.

cluster A syntactic "capsule" in which all information concerning the representation of a data type is contained. By supporting the separation of the code that implements a data abstraction from the code that uses it, clusters promote well-structured code.

The idea of supporting data abstraction in this way appears in several languages, and different names are used. Although many details differ, the constructs serve basically the same purpose in each language. Some of the names are **class** (Simula 67,Smalltalk), **module** (Modula-2), **form** (Alphard), and **package** (Ada). Parnas (1972b) uses the term **information-hiding module**.

data abstraction An abstract data type, conceived of independent of any particular representation.

data-type specification A specification of the properties of a data abstraction independent of its representation. A completely formal method of proving cluster correctness would require such a specification.

Floyd-Hoare proof rules A set of rules embodying the programming knowledge pertaining to a particular language, and permitting proofs of programs in that language, like our proof rules for Impcore. Such proof systems have been developed for a variety of programming languages, mainly of the traditional, imperative type; see McGettrick (1982) for one example. They do not exist, however, for most of the languages discussed in this book.

homomorphism A map from one implementation of a cluster to another, which preserves the behavior of the operations defined on the cluster. Typically used as an abstraction function, where it maps a concrete representation to an ideal, abstract cluster. The homomorphism is the basis of a proof of correctness of the cluster.

inductive assertion An assertion about the relationships among the variables in a program. The basic method of program proving is to place inductive assertions around each statement in a program, and then prove that these assertions are necessarily true when control reaches that point in the program.

iterator A procedure-like construct in Clu that produces a *list* of values instead of a single value. A **for** statement is used to iterate over the elements of this list. The elements are produced not all at once, but on demand, as they are requested.

loop invariant An assertion P whose truth is preserved by the body of a loop. P is a loop invariant for the loop `(while e` S`)` if, whenever both P and `e` are true before executing S, P is still true afterwards.

representation invariant The condition that defines the instances of a cluster as a subset of the representing type. For example, instances of `SpArray` are represented by triples $\langle b, s, ((i_1, x_1) \ldots (i_n, x_n))\rangle$, satisfying the representation invariant that the i_j are distinct integers between b and $b+s-1$. Representation invariants are needed for proving the correctness of clusters.

9.6.2 Further Reading

In their excellent book on programming, Liskov and Guttag (1986) include an introduction to Clu and a concise reference manual. They also explain how Clu fits into the process of writing reliable software. Liskov et al. (1977) introduce the Clu approach to programming, and Liskov et al. (1981) provide a complete reference manual. Liskov and Snyder (1979) present Clu's exception mechanism.

The general topic of data abstraction as a useful concept for program development is discussed in classic papers by Wirth (1971), Parnas (1972b), and Dahl and Hoare (1972). Data abstraction is the centerpiece of several languages aside from Clu, including Simula 67 (Dahl and Hoare 1972; Birtwistle et al. 1973), Alphard (Wulf, London, and Shaw 1976), Modula-2 (Wirth 1982), and Ada. Chapter 2 of Abelson and Sussman 1985 contains a most interesting discussion about data abstraction in Scheme.

The basic reference for Ada is the Reference Manual (US DoD 1983). Barnes (1991) provides an excellent introduction to the language, and Cohen (1986) a very complete and detailed one; but these are just two among many introductory books on the language. Bishop (1986) covers general data abstraction concepts in an Ada context. All these references are for the 1983 version of the language; in 1995, Ada underwent a major revision and acquired object-oriented features (Ada 1995). The "Home of the Brave Ada Programmers" (`www.adahome.com`) contains many other references.

Program verification is among the most heavily researched topics in computer science over the past twenty years. Loeckx, Sieber, and Stansifer (1987) give a good overview. The introductory text by McGettrick (1982) gives a full set of rules for verification of Ada programs. Chapter 3 of Manna 1974 is an excellent introduction to verification of programs using integers and arrays. Hoare (1972) was the first to discuss the verification of data representations. This and related papers are reprinted in Hoare 1989. The subarea of *data-type specification* is surveyed by Liskov and Zilles (1975), and is also discussed by Liskov and Guttag (1986); there are many other papers on this topic (Guttag, Horwitz, and Musser 1978; Goguen, Thatcher, and Wagner 1979; Futatsugi et al. 1985; Parnas 1972a; Kamin 1983). Gries (1981), Reynolds (1981), and Jones (1980) have written fine books on *formal program development*, in which program-verification methods are applied to program construction.

9.7 Type theory of μClu

9.7.1 The representation of types

Like μML, μClu specializes the more general formulation of polymorphism found in Typed μScheme. As in μML, we stratify the type system so that a name bound in a type environment Γ has a potentially polymorphic type σ, whereas the type of a term e is always a monotype τ.

$$
\begin{array}{lll}
\sigma & ::= & \tau \mid \forall tyvar_1, \ldots, tyvar_n \,.\, \tau \\
\tau & ::= & \alpha \\
 & \mid & \langle \mathtt{id} = path, \mathtt{exports} = \mu c \,.\, \tau_{\mathtt{exports}} \rangle \\
 & \mid & \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\
 & \mid & \mathtt{function}\ (\vec{\tau}_a)\ \mathtt{returns}\ (\vec{\tau}_r)\ \mathtt{signals}\ (s_1, \ldots, s_n) \\
tyvar & ::= & \alpha\ [\mathtt{where}\ \tau_{\mathtt{has}}] \\
path & ::= & cluster\text{-}name \\
 & \mid & cluster\text{-}name[path\text{-}parameter]
\end{array}
$$

$$
\begin{array}{rcl}
\textit{path-parameter} & ::= & \textit{path} \\
& | & \alpha \\
& | & \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\
& | & \texttt{function}\ (\vec{\tau_a})\ \texttt{returns}\ (\vec{\tau_r})\ \texttt{signals}\ (s_1, \ldots, s_n)
\end{array}
$$

If we compare the σ and τ above with Typed μScheme's definition of τ on page 261, we notice these differences:

- A type constructor of kind $*$, which in Typed μScheme is only a name μ, in μClu has become a cluster type $\langle \texttt{id} = p, \texttt{exports} = \mu c\,.\,\tau_{\texttt{exports}} \rangle$. The path p plays the same role in μClu as μ does in Typed μScheme, but a cluster type has additional structure: a set of exported operations. Because the operations can mention the cluster, the type is recursive; the notation $\mu c\,.\,\tau_{\texttt{exports}}$ describes a standard representation of recursive types, which is explained in detail below.

- Both Typed μScheme and μClu have type variables, notated α.

- μClu cannot represent as many different kinds as Typed μScheme. In fact, every well-formed type τ has kind $*$, and the only higher kinds permitted are kinds of the form $*_1 \Rightarrow \cdots \Rightarrow *_n \Rightarrow *$. A type with this kind must be a *polymorphic cluster type* which expects n type arguments. If we have a polymorphic cluster named μ, and it takes n type parameters, we represent its type as $\forall \alpha_1, \ldots, \alpha_n . \langle \texttt{id} = p, \texttt{exports} = \mu c . \tau_{\texttt{exports}} \rangle$, where $p = \mu[\alpha_1, \ldots, \alpha_n]$. The type parameters α_i appear free in p, and they typically appear free in $\tau_{\texttt{exports}}$ as well.

- Typed μScheme has a form for type application. *(The analogous form here is the path. Otherwise there's no application because where application is possible we always have the definition—need to tie into* `list` *example below ...)*

- Both Typed μScheme and μClu provide polymorphic types using the $\forall$ notation, but in μClu, it is possible to constrain a type parameter by means of a *where clause* (page PAGEREFuclu.where-clause). A type τ may be used to instantiate a type variable α `where` $\tau_{\texttt{has}}$ only if τ is a cluster type that exports every option listed in $\tau_{\texttt{has}}$.

- Instead of Typed μScheme's tuples, which have types of the form $\tau_1 \times \cdots \times \tau_n$, μClu provides records with named fields, which have types $\{\tau_1 : \}_1, \ldots, \tau_n : \}_n$. The names l_i are the names of the fields.

- Both Typed μScheme and μClu provide function types, but here it is μClu that is more expressive: just as a function may take any number of arguments, it may also return any number of results.[10] It may also signal an exception, and the set of exceptions signaled is part of a function's type.

Notes:

- Homework problem: make $\forall$ the only case for σ.

- The record type is as in ML.

[10] As seen on page PAGEREFuclu.term-multiple, μClu requires a special form of expression, `(multiple` $e_1, \ldots, e_n$`)`, which is used in the body of a function that returns multiple results. To call a function that returns multiple results, μClu also provides a `let-multiple` form. These are the introduction and elimination constructs for the result type vector $\vec{\tau_r}$ of a μClu function.

- The function type is a generalization of the function type used in Typed μScheme: a μClu function may return any number of results, and it may also raise an exception, which in Clu is called a *signal*. Unlike in ML, the signals a function may raise are part of its type, so a caller can guarantee to catch them.[11] (μClu provides facilities for raising signals but not catching them; see EXREFPAGEuclu.ex.catch.)
- A cluster type is distinguished from similar cluster types by its *identity*, which we represent as a *path*.
- To support programming with a cluster, we have to know what operations the cluster exports. These are the operations listed in the export list $\tau_{\texttt{exports}}$, which is always a record type. Within $\tau_{\texttt{exports}}$, the cluster itself is always referred to by the internal name c, which is an alpha-varying type variable.
- In the type of a polymorphic cluster, ∀-bound type variables may appear in both the identity (*path*) and $\tau_{\texttt{exports}}$ parts of the cluster. The cluster's internal name c appears only in $\tau_{\texttt{exports}}$.
- Full Clu has polymorphic functions within clusters; to support such functions we could change the form of a cluster type to be $\langle \texttt{id} = \mathit{path}, \texttt{exports} = \mu c\,.\,\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}\rangle$.

As an example, the polymorphic list cluster in ⟨*example cluster* `List` 359⟩ would have the type

$$\forall\alpha\,.\,\langle \texttt{id} = \texttt{list}[\alpha], \texttt{exports} = \mu l\,.\,\{\texttt{nil} : l, \texttt{null?} : l \to \texttt{bool}, \texttt{cons} : \alpha \times l \to \alpha, \ldots\}\rangle.$$

If we instantiate that cluster to the type `(@ list int)` the resulting cluster has type

$$\langle \texttt{id} = \texttt{list}[\texttt{int}], \texttt{exports} = \mu l\,.\,\{\texttt{nil} : l, \texttt{null?} : l \to \texttt{bool}, \texttt{cons} : \texttt{int} \times l \to \texttt{int}, \ldots\}\rangle.$$

No matter how many times we instantiate `(@ list int))`, the instantiated type's *identity* is always `list[int]`, so the multiple instances are interchangeable. In other words, just instantiating an existing cluster does not create a unique identity; only defining a new cluster creates a unique identity.

In fact, the type rule for instantiation is the same as in Typed μScheme; the only difference is that in addition to the polymorphic function *expression*, which we can do in Typed μScheme, we can also instantiate a polymorphic cluster *type*, which we cannot do in Typed μScheme. (See rule INSTX on page 382 and INSTT on page 382.)

9.7.2 Cluster types can be recursive

(In fact if they're not recursive, they're jolly well boring.)

The types of the exported operations may themselves mention the type of the cluster (for example `int$add` returns an `int`), which in turn lists the exported operations, which in turn mention the cluster... and it's turtles all the way down. To avoid an infinite representation, we have to invent

[11] In full Clu, it is in fact *required* that a caller catch the signals of every function called; the compiler enforces the requirement simply by examing the types of those functions. In most other languages with exceptions, a caller can choose not to handle the exception by instead to let the exception propagate to *its* caller, and so on up the stack. Because of this distinction between Clu's signals and exceptions in languages like ML, we use the original terminology to refer to signals.

9.7.3 Forms of typing judgments

A key feature of μClu is that it provides named *type abbreviations.*

In Typed μScheme, we use Δ as an environment mapping the name of a type constructor or type variable to its *kind.* In μClu we need a different kind of environment: one that maps the *name* of a type to its *definition.* We call this environment Θ. A name could be introduced into Θ if it is a type abbreviation or if it is a formal parameter to a polymorphic abstraction.

9.7.4 Type soundness

If is hoped that if $\Theta, \Gamma \vdash e : \tau$ and $\rho \vdash e \Downarrow v$, and if ρ is compatible with Θ and Γ, then v will be compatible with τ (EXREFPAGEuclu.ex.soundness).

9.7.5 Typing rules

Highlights

One of the most interesting new rules is the rule for selecting an exported operation from a cluster. The rule illustrates a typical way of handling recursive types: the use of a recursive type is tied to a particular kind of abstract syntax, and using that syntax causes the recursion to be "unrolled" by substituting the type for its name. Here's how it works when we select the component of a cluster:

$$\frac{\Theta \vdash t \downarrow \langle \texttt{id} = p, \texttt{exports} = \mu c\,.\,\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\rangle \qquad l_j = l}{\Theta, \Gamma \vdash \textsc{Component}(t, l) : \tau_j\{c \mapsto \langle \texttt{id} = p, \texttt{exports} = \mu c\,.\,\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\rangle\}} \quad (\textsc{Component})$$

Informally, the rule says we can extract a component only from a cluster type, and that when we find the type τ_j of the extracted component, we replace the internal name c of the cluster with the full cluster type.

(Special rules for extracting `get-`l and `set-`l from record types as if they were clusters.)

To use a polymorphic value, one chooses the types with which to instantiate the type variables. The notation $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ indicates the simultaneous substitution of τ_1 for α_1, τ_2 for α_2, and so on.

$$\frac{\Theta, \Gamma \vdash e : \forall\alpha_1, \ldots, \alpha_n.\tau \qquad \Theta \vdash t_i \downarrow \tau_i, 1 \le i \le n}{\Theta, \Gamma \vdash \textsc{inst\_exp}(e, \langle t_1, \ldots, t_n\rangle) : \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]} \quad (\textsc{InstX})$$

$$\frac{\Theta \vdash t \downarrow \forall\alpha_1, \ldots, \alpha_n.\tau \qquad \Theta \vdash t_i \downarrow \tau_i, 1 \le i \le n}{\Theta \vdash \textsc{inst\_ty}(t, \langle t_1, \ldots, t_n\rangle) \downarrow \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]} \quad (\textsc{InstT})$$

9.8 The interpreter

A type is either a cluster type, a record type, or a function type. A cluster type is uniquely identified by its path.

383a ⟨*types for μClu* 383a⟩≡ (390a) 383b▷

```
type 'a record = (string * 'a) list
type signal = name
datatype topty = FORALL of name list * ty
               | MONO   of ty
and         ty = RECORD    of ty record
               | FUNCTION  of ty list * ty list * signal list
               | CLUSTERTY of path * ty record (* identity and export list *)
               | TYVAR  of name
and       path = PATHCON of name * ty list (* named cluster *)
               | PATHVAR of name           (* cluster that is a type parameter *)
```

Instantiating a polymorphic type

Just as in Typed μScheme, only simpler, because FORALL can't be nested.

383b ⟨*types for μClu* 383a⟩+≡ (390a) ◁383a 383d▷

```
tysubst   : ty * ty env -> ty
subst     : ty          -> ty
pathsubst : path        -> path
```

```
exception BugInTypeChecking
fun tysubst (tau, varenv) =
  let fun subst (RECORD fields) = RECORD (map rsubst fields)
        | subst (FUNCTION (args, returns, signals)) =
                FUNCTION (map subst args, map subst returns, signals)
        | subst (CLUSTERTY (p, exports)) = CLUSTERTY (pathsubst p, map rsubst exports)
        | subst (TYVAR a) = (find(a, varenv)
                             handle NotFound _ => raise BugInTypeChecking)
      and rsubst (n, tau) = (n, subst tau)
      and pathsubst (PATHCON (c, taus)) = PATHCON (c, map subst taus)
        | pathsubst (PATHVAR a) =
            (case find(a, varenv)
               of tau as CLUSTERTY (p, _) => p (* N.B. where clause not checked! *)
                | _ => raise TypeError (String.concat
                          ["Type parameter '", a, " may only be instantiated by ",
                           "a cluster"]))
            handle NotFound _ => raise BugInTypeChecking
  in  subst tau
  end
```

383c ⟨*primitive type constructors for μClu* :: 383c⟩≡

```
("int",      intcluster) ::
("bool",     boolcluster) ::
("option",   optioncluster) ::
("array",    arraycluster) ::
```

383d ⟨*types for μClu* 383a⟩+≡ (390a) ◁383b 384▷

```
datatype tyex = TYPENAME     of name            (* type abbreviation or cluster name *)
              | TYVARX    of name             (* type variable *)
              | TYAPP     of name * tyex list  (* rename; considering TYAPPLY
 or CONAPPX; revisit uML an tuScheme XXX
 *)
              | FUNCTIONX of tyex list * tyex list * signal list
              | RECORDX   of tyex record
```

Elaboration of type expressions into types?

384 ⟨*types for μClu* 383a⟩+≡ (390a) ◁383d

```
elabTy   : topty env * tyex -> ty
monofind : name -> ty
```

```
fun elabTy (Delta, t) =
  let fun ty (TYPENAME n) = monofind n
        | ty (TYVARX a) = (monofind a; TYVAR a)
        | ty (TYAPP (n, ts)) =
            (case find (n, Delta)
               of FORALL (a's, tau) =>
                    (tysubst(tau, bindList(a's, map ty ts, emptyEnv))
                     handle BindListLength =>
                         raise TypeError (concat ["name ", n, " expected ",
                                                  Int.toString (length a's),
                                                  " type parameters; got ",
                                                  Int.toString (length ts)]))
                | MONO _ => raise TypeError (n ^ " is not polymorphic"))
        | ty (FUNCTIONX (args, returns, signals)) =
            FUNCTION (map ty args, map ty returns, signals)
        | ty (RECORDX fields) =
            RECORD (map (fn (n, t) => (n, ty t)) (withoutDuplicateNames fields))
      and withoutDuplicateNames [] = []
        | withoutDuplicateNames ((n, t) :: fs) =
            if List.exists (fn (n', _) => n = n') fs then
                raise TypeError ("field name " ^ n ^ " duplicated in record type")
            else
                (n, t) :: withoutDuplicateNames fs
      and monofind n = case find(n, Delta)
                         of MONO tau => tau
                          | FORALL _ => raise TypeError (String.concat
                                                ["Polymorphic cluster ", n, " must be ",
                                                 "instantiated before use"])
  in  ty t
  end
```

Substitution within types?

Type equality

As in Typed Impcore, many rules of μClu require that two types be the same. We can't simply use built-in equality to check, however, because when we compare two quantified types, the *names* of the type variables should be irrelevant. For example, the two types $\forall\alpha\,.\,\alpha$ `list` $\rightarrow$ `int` and $\forall\beta\,.\,\beta$ `list` $\rightarrow$ `int` should be considered to be equal. We implement the check by substituting one set of names for the other.

385 ⟨*type checking for* μ*Clu* 385⟩≡ 388a▷

```
datatype ty = RECORD    of ty record
            | FUNCTION  of ty list * ty list * signal list
            | CLUSTERTY of path
and path    = PATH      of name * ty list
```

```
eqType  : tyex      * tyex      -> bool
eqTypes : tyex list * tyex list -> bool
```

```
fun eqType (RECORD r, RECORD r') = eqRecords (r, r')
  | eqType (FUNCTION (args, ress, signals), FUNCTION (args', ress', signals')) =
      eqTypes (args, args') andalso eqTypes (ress, ress') andalso
      eqSets (signals, signals')
  | eqType (CLUSTERTY (p, taus), CLUSTERTY (p', taus')) =
      eqPath (p, p') andalso eqTypes (taus, taus')
  | eqType _ = false
and eqTypes (t::taus, t'::taus') = eqType(t, t') andalso eqTypes(taus, taus')
  | eqTypes ([], []) = true
  | eqTypes _ = false
and eqRecords (r, r') =
      let fun matches n = eqType(find(n, r), find(n, r'))
      in  length r = length r' andalso
          forall (fn (n, tau) => eqType(tau, find(n, r'))) r
          handle NotFound _ => false
      end
and eqSets (signals, signals') =
    length signals = length signals' andalso
    forall (fn s => member(s, signals')) signals
```

Standard clusters of μClu

We now have all the type machinery we need to implement μClu. We continue by presenting the abstract syntax, values, and type rules of μClu.

9.8.1 Abstract syntax, values, and evaluation of μClu

As with the concrete syntax, there are only minor differences between the abstract syntax of μScheme and the abstract syntax of μClu. We add two new expressions, TYLAMBDA and TYAPPLY, which introduce and eliminate polymorphic types. We drop LETREC. We also require that the parameters of functions have explicit types.

386 ⟨*abstract syntax and values for μClu* 386⟩≡ (390a) 387a▷

```
datatype exp = LITERAL  of value
             | VAR      of name
             | COMPONENT of tyex * name
             | SET      of name list * exp
             | IFX      of exp * exp * exp
             | WHILEX   of exp * exp
             | BEGIN    of exp list
             | LETX     of let_kind * (name * exp) list * exp
             | APPLY    of exp * exp list
             | MAKE     of tyex * exp record
             | SIGNAL   of name
             | SUGAR    of sugar
             | TYAPPLY  of name * tyex list
                   (* revisit XXX -- make consistent  with tuScheme *)
and let_kind = LET | LETSTAR
and    sugar = PRIM of name
             | GET_SUGAR  of name
             | SET_SUGAR  of name

and    value = BOOL of bool
             | NUM  of int
             | SYM  of name
             | RVAL of value record
             | VVAL of name * value
             | FVAL of value list -> value list


exception RuntimeError of string
```

387a ⟨*abstract syntax and values for μClu* 386⟩+≡ (390a) ◁386

```
datatype toplevel = EXP     of exp
                  | DECLARE of declaration list
                  | DEFN    of definition
                  | CLUSTERDEF of { name : name
                                  , tyargs : tyformal list
                                  , exports : declaration list
                                  , abbrevs : (name * tyex) list
                                  , private : declaration list
                                  , impl    : definition list
                                  }
                  | USE     of name
and definition    = DEFINE of name * userfun
                  | VAL    of name * exp
and tyformal      = TYFORMAL of name * constraint list
withtype declaration = name * tyex
     and constraint  = name * tyex
     and userfun     = { tyformals : tyformal list option
                       , formals   : (name * tyex) list
                       , returns   : tyex list
                       , signals   : name list
                       , body      : exp
                       }
```

9.8.2 Type rules for μClu

- More complex type environment, to support dot notation
- Type abbreviations

Type rules for expressions

Type rules for top-level items

Type checking

387b ⟨*type checking for μClu* [[**prototype**]] 387b⟩≡

```
exception LeftAsExercise
fun topty _ = raise LeftAsExercise
```

388a ⟨*type checking for μClu* 385⟩+≡ ◁385

```
fun addCluster ({name, tyargs, exports, abbrevs, private, impl}, (delta, gamma, rho)) =
  let fun addDecl Delta ((name, tyex), gamma) = bind (name, asType Delta tyex, gamma)
      fun addAbbrev ((name, tyex), Delta) = bind (name, asType Delta tyex, Delta)
       (* horrid recursive knot here *)
      val thistype = CLUSTERTY (PATHCON (name, map TYVAR tyargs),
                                map (fn (n, t) => (n, asType Delta' t)) exports)
      val Delta'  = foldr (fn (a, Delta) => bind (a, TYVAR a, Delta)) Delta tyargs
      val gamma'  = foldr (addDecl Delta') gamma exports
      val Delta'' = foldl addAbbrev Delta' abbrevs
      val rep = find ("rep", Delta'') handle _ =>
                raise TypeError ("No 'rep' type defined in cluster " ^ name)
      val gamma'' = foldr (addDecl Delta'') gamma' private

                | CLUSTERDEF of { name : name
                                , tyargs : tyformal list
                                , exports : declaration list
                                , abbrevs : (name * tyex) list
                                , private : declaration list
                                , impl    : definition list
                                }


\subsection{The rest of an interpreter for {\uclu}}


\subsubsection{Evaluation}
```

388b ⟨*evaluation* [[clu]] 388b⟩≡ (390a) 388c▷

```
⟨definition of separate (generated automatically)⟩
exception NotYetImplemented
fun eval(e, rho) =
  let fun ev _ = raise NotYetImplemented
  in  ev e
  end
```

Top-level processing

388c ⟨*evaluation* [[clu]] 388b⟩+≡ (390a) ◁388b 389b▷

```
topeval : toplevel * value ref env * (string->unit) -> value ref env
fun topCheckEval (t, envs as (gamma, rho), echo) =
  case t
    of USE filename => use readCheckEvalPrint ucluSyntax filename envs
     | CLUSTERDEF c =>
       let val envs = addCluster (c, envs)
           val _    = echo ("cluster " ^ #name c)
       in  envs
       end
     | DECLARE decls => foldl addDecl envs decls
     | EXP e         => topCheckEval (DEFN (VAL ("it", e)), envs, echo)
     | DEFN (DEFINE (name, args, body)) => raise NotYetImplemented
     | DEFN (VAL (name, e)) => raise NotYetImplemented
```

The implementation of use is parameterized by readEvalPrint so we can share it with other interpreters. Function use creates a top-level reader that does not prompt, but it uses writeln to be sure that responses are printed.

389a ⟨*implementation of* use 389a⟩≡ (390a)

```
fun use readCheckEvalPrint filename rho =
      let val fd = TextIO.openIn filename
          val reader = defreader (false, filereader (filename, fd), cluSyntax)
          fun writeln s = app print [s, "\n"]
          fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
      in  readCheckEvalPrint (reader, writeln, errorln) rho
          before TextIO.closeIn fd
      end
```

The definition of defreader is relegated to chunk in Appendix E.

The read-eval-print loop

389b ⟨*evaluation* [[clu]] 388b⟩+≡ (390a) ◁388c

```
and readCheckEvalPrint (reader, echo, errmsg) rho =
  let fun loop rho =
        let fun continue msg = (errmsg msg; loop rho)
            fun finish () = rho
        in  loop (topeval (reader(), rho, echo))
            handle EOF => finish()
                  ⟨more read-eval-print handlers (generated automatically)⟩
        end
  in  loop rho
  end
```

Initializing the interpreter

389c ⟨*initialization* [[clu]] 389c⟩≡ (390a) 389d▷

389d ⟨*initialization* [[clu]] 389c⟩+≡ (390a) ◁389c

```
fun runInterpreter noisy =
  let val rho = []
      fun writeln s = app print [s, "\n"]
      fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
      val reader = defreader (noisy, filereader ("standard input", TextIO.stdIn), cluSyntax)
  in  ignore (readEvalPrint (reader, writeln, errorln) rho) handle EOF => ()
  end
```

```
runInterpreter : bool -> unit
```

Putting all the pieces together

We stitch together the parts of the implementation in this order:

390a ⟨*uclu.sml* 390a⟩≡
```
⟨environments (generated automatically)⟩
exception LeftAsExercise
exception TypeError of string
⟨definition of separate (generated automatically)⟩
⟨types for μClu 383a⟩
⟨lexical analysis [[uclu]] (generated automatically)⟩
⟨abstract syntax and values for μClu 386⟩
⟨parsing for μClu (generated automatically)⟩
⟨readers (generated automatically)⟩
⟨implementation of use 389a⟩
⟨evaluation [[clu]] 388b⟩
⟨initialization [[clu]] 389c⟩
⟨command line (generated automatically)⟩
```

390b ⟨*additions to the μClu initial basis* 390b⟩≡

9.9 Exercises

Learning about the language

1. The data type `BinTree` has as its values binary trees with labelled nodes, and as operations:

   `make-node`: Create a child-less tree, given its label.

   `make-tree`: Create a new tree, given its label and two subtrees.

   `children?`: Test if a tree has children.

   `label`: Return the label of a tree.

   `leftchild`: Return the left child of a tree.

   `rightchild`: Return the right child of a tree.

   Implement `BinTree` using two different representations:

   (a) A recursive representation, such as we gave for `List`.

   (b) Represent a tree by a single list, which has either one element (for child-less trees) or three elements (the label and the two subtrees).

2. Implement a `Queue` cluster, with operations: `empty-queue`, `empty?`, `enqueue`, `front`, `without-front`. Use it to define a function to traverse a `BinTree` in level order.

3. A mutable version of `BinTree` might have operations `change-label`, `add-leftchild`, and `add-rightchild`. `add-leftchild` and `add-rightchild` have as arguments *labels* (not trees), so they add only a single node, replacing whatever subtree might have existed before. Make mutable versions of the two `BinTree` clusters previously given.

   Then define a new mutable representation of `BinTree`, based upon the tree representation used in the heap-sort algorithm. Namely, a tree is represented by an array `A`, indexed from 1, the label of the root being at `A[1]` and the children of node i at `A[`$2i$`]` and `A[`$2i+1$`]`. (A special value needs to be reserved to indicate that a node is not present.)

4. Implement both immutable and mutable versions of cluster `Tree`, containing trees of arbitrary degree. You must choose appropriate operations in each case. Note that the `make-tree` approach to constructing trees does not work here, because every operation must have a fixed number of arguments. Program level-order traversal for these new clusters.

5. Implement a cluster `RatNum` of rational numbers; you should include the major arithmetic operations (addition, subtraction, multiplication, division), as well as comparisons (`<<` and `==`). Rational numbers can be represented simply by pairs of integers, representing numerator and denominator; it is a good idea to keep these integers in *reduced* form, meaning that the numerator and denominator should have no common factors. Implement a square root function based upon the Newton-Raphson method (page 460).

6. Implement a cluster `Matrix` that represents two-dimensional matrices; then implement `SpMatrix`, using a sparse representation of your own choosing. As a client, write function `rotate`, which performs a 90-degree clockwise *in-place* rotation of a matrix.

7. Using either of the `Point` clusters, define a cluster `Window`, which contains rectangles with integer vertices, with operations:

   `new`: Given any two opposing corners of a window, create the window.

   `lower-left`: Return the lower left corner of a window. (Note that this need not have been the first argument to `new`.)

   `upper-right`: Return the upper right corner of a window.

   `intersect?`: Check if two windows overlap.

   `intersection`: Return the window that is the intersection of two intersecting windows.

   Then make this a mutable data type, adding operations:

   `horizontal-move`: Move the first argument, a window, horizontally either to the right or the left, according as the second argument is positive or negative.

   `vertical-move`: analogously.

   `grow`: Given a window and two integers, grow or shrink in the horizontal and vertical directions based upon the second and third arguments, respectively, growing for positive integers and shrinking for negative ones. In all cases, the lower left corner of the window does not move.

Learning about program verification

8. Verify the body of the exponentiation loop (page PAGEREFuclu.expon).

9. We give two versions of the factorial function, the first purely iterative and the second recursive:

```
(define fac (x)
    (begin
       (set result 1)
       (while (<> x 0)
          (begin
             (set result (* result x))
             (set x (- x 1))))))

(define recfac (x)
    (if (= x 0) 1 (* x (recfac (- x 1)))))
```

   Prove: `x` > 0 {`(fac x)`} `result` $=$ `x`!. Then prove the corresponding result for `recfac`, after transforming it as per section REFuclu.funcver.

10. Transform the `listsum` function to iterative form and prove it using the iteration rule. Comment on the relation between the iteration rule and the recursive function rule.

11. Transform the exponentiation function from page PAGEREFuclu.expon to recursive form, and prove it using the recursive function rule.

12. Prove the correctness of the second `Point` cluster. In this case, the first `Point` cluster can serve as the "abstract" cluster. As a first step, you may assume that all the operations in the second `Point` cluster are correct, but you should then prove each of them.

13. Prove the correctness of the `Poly` cluster. You will need to define a cluster of abstract polynomials. As a first step, you may assume that all `Poly` operations are correct, but you should then prove each of them.