1 Double dispatch

Suppose we add string values and rational-number values to our expression language from lecture 18.
Further suppose we change the meaning of `Add` expressions to the following:

- If the arguments are ints or rationals, do the appropriate arithmetic.
- If either argument is a string, convert the other argument to a string (unless it already is one) and
  return the concatenation of the strings.

(For simplicity, the SML/Ruby/Java code associated with this summary does not change the meaning of
`Negate` or `Mult`; evaluation of these expressions raises a run-time error if either subexpression evaluates to
a value that is not an `Int`.)

The interesting change to the SML code is in the `Add` case of `eval`. We now have to consider 9 (i.e., 3 * 3)
subcases, one for each combination of values produced by evaluating the subexpressions. To make this
explicit and more like the object-oriented version considered below, we move these cases out into a helper
function `add_values` as follows:

```sml
fun eval e =
  case e of
  ... |
  Add(e1,e2) => add_values (eval e1, eval e2)
  ...

fun add_values (v1,v2) =
  case (v1,v2) of
  (Int i, Int j) => Int (i+j)
  | (Int i, String s) => String(Int.toString i ^ s)
  | (Int i, Rational(j,k)) => Rational(i*k+j,k)
  | (String s, Int i) => String(s ^ Int.toString i) (* not commutative *)
  | (String s1, String s2) => String(s1 ^ s2)
  | (String s, Rational(i,j)) => String(s ^ Int.toString i ^ "/" ^ Int.toString j)
  | (Rational _, Int _) => add_values(v2,v1) (* commutative: avoid duplication *)
  | (Rational(i,j), String s) => String(Int.toString i ^="/" ^ Int.toString j ^ s)
  | (Rational(a,b), Rational(c,d)) => Rational(a*d+b*c,b*d)
  | _ => raise BadResult "non-values passed to add_values"
```

Notice our functional decomposition generalizes in a straightforward way to binary (or in general, n-ary)
operations: we can use pattern-matching over tuples to enumerate all the cases.

Although nine cases might seem too many, that complexity is inherent to the problem. If many cases work
the same way, we can use wildcard patterns and/or helper functions to avoid redundancy. One common
source of redundancy is `commutativity`, i.e., the order of values not mattering. In the example above, there
is only one such case: adding a rational and an int is the same as adding an int and a rational. Notice how
we exploit this redundancy by having one case use the other with the call `add_values(v2,v1)`.

---

1Note that our focus is no longer on extensibility. We’re now just considering how to implement a bigger expression language,
not how to grow a smaller implementation into a bigger one.
We now turn to supporting the same addition in an object-oriented style, as in either our Java or Ruby code. The obvious first step is to have all the classes for values in our language, i.e., `Int`, `Rational`, and `String`, define an `add_values` method that takes one argument, the “other” value to be added. Then the `eval` method of the `Add` class would be as follows in Ruby:

```ruby
def eval
  e1.eval.add_values e2.eval
end
```

In Java, for type-checking purposes, we can create an abstract class `Value` (since non-value expression classes do not have an `add_values` method) and arrange things as follows:

```java
abstract class Exp {
  abstract Value eval();
  abstract String toString();
  abstract boolean hasZero();
}
abstract class Value extends Exp {
  abstract Value add_values(Value other);
}
class Add extends Exp {
  ...
  Value eval() {
    return e1.eval().add_values(e2.eval());
  }
}
```

By putting `add_values` methods in the `Int`, `String`, and `Rational` classes, we nicely divide our work into three pieces using dynamic dispatch depending on the class of the object that `e1.eval()` returns, i.e., the receiver of the `add_values` call in the `eval` method in `Add`. But then each of these three needs to handle three of the nine cases, based on the class of the second argument. One approach would be to, in these methods, abandon object-oriented style and use run-time tests of the classes to include the three cases. The Ruby code would look like this and the Java code would be similar using `instanceof`:

```ruby
class Int
  ...
  def add_values other
    if other.is_a? Int
      ...
    elsif other.is_a? Rational
      ...
    else
      ...
    end
  end
end
class Rational
  ...
  def add_values other
    if other.is_a? Int
      ...
    elsif other.is_a? Rational
      ...
    else
      ...
    end
  end
```
Although this approach works, it’s not very OOP. Rather, it is a mix of object-oriented decomposition (dynamic dispatch on the first argument) and functional decomposition (using \texttt{is\_a?} to figure out the cases in each method).

A “full” object-oriented solution would use just dynamic dispatch to choose among the nine cases, not \texttt{is\_a?} tests. Some languages have \textit{multimethods} to support this sort of operation. A multimethod allows multiple implementations (just like regular methods support implementations in different classes) and chooses which method is dispatched to using the classes of multiple arguments instead of just the receiver. If we had multimethods, we could define one called \texttt{add\_values}, provide nine implementations, and rely on dynamic dispatch to pick the right one.

Neither Ruby nor Java has multimethods. But there is a programming pattern for “coding up” multimethods. This pattern is called \textit{double dispatch} when the total number of arguments is 2 (the receiver plus 1 other), like with \texttt{add\_values}. With additional work, the technique extends to any number of arguments.

We will demonstrate double dispatch by using it to complete \texttt{add\_values} in object-oriented style. The relevant Ruby and Java code appears at the end. The idea is to have all our value classes implement three more methods:

- \texttt{addInt} takes an argument \texttt{other} and produces the result of addition where the left argument is an \texttt{Int other} and the right argument is \texttt{self / this}.
- \texttt{addString} takes an argument \texttt{other} and produces the result of addition where the left argument is a \texttt{Strng other} and the right argument is \texttt{self / this}.
- \texttt{addRational} takes an argument \texttt{other} and produces the result of addition where the left argument is a \texttt{Rational other} and the right argument is \texttt{self / this}.

So with three classes each implementing these three methods, we have the nine cases we need. Now we just need to dispatch to the correct one. Our \texttt{eval} method in \texttt{Add} already does the “first dispatch” to pick the \texttt{add\_values} method of its receiver. We then implement these methods to do the “second dispatch” by having \texttt{add\_values in Int call other.addInt self}, \texttt{add\_values in Strng call other.addString self}, and \texttt{add\_values in Rational call other.addRational self}.

\begin{verbatim}
class Int < Exp
  ...
def add_values v # first dispatch
  v.addInt self
end

def addInt v # second dispatch: other is Int
  Int.new(v.i + i)
end

def addString v # second dispatch: other is Strng (notice order flipped)
  Strng.new(v.s + i.to_s)
end

def addRational v # second dispatch: other is Rational
  Rational.new(v.i+v.j*i,v.j)
end
end

See the code associated with this lecture for the rest of the Ruby implementation.

2 Immutable objects

The code we’ve been looking at treats objects in a somewhat stylized way:

- Objects have only getters, not setters.
- No methods, other than initializers ever assign to instance variables.
- Anytime a “changed” object is needed, instead a new object is created.

This is an idea that comes out of functional programming, where values are immutable. In fact, objects coded in this style are sometimes called functional objects or value objects.

Suppose we’re programming a game with UFOs. The typical way of writing imperative objects for UFOs in Java might be as follows:

class UFO {
  private int x;
  private int y;
  UFO(int x, int y, ... ) {
    this.x = x; ....
  }
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) { this.x = x; }
  void setY(int y) { this.y = y; }
  ...
  void move() {
    x = x + deltaX;
    y = y + deltaY;
  }
}

Here’s the immutable (i.e., functional) equivalent:

class UFO {
private int x;
private int y;
UFO(int x, int y, ...) {
    this.x = x; ....
}
int getX() { return x; }
int getY() { return y; }
// no setters...
UFO move() {
    new UFO(x+deltaX, y+deltaY);
}

Java even provides some language support for this idiom. You can annotate fields as final, meaning the type system will reject any attempt to assign to them—except in constructors.

Making this scale to large-scale programs sometimes requires other programming patterns, too. But it’s worthwhile, because it reduces complexity hence bugs. In fact, object-oriented programming originated in part from a desire to eliminate assignment statements.

3 Subtyping

We previously studied static types for functional programs, in particular ML’s type system. ML uses its type system to prevent errors such as treating a number as a function. A key source of expressiveness in ML’s type system (not rejecting too many programs that do nothing wrong and programmers are likely to write) is parametric polymorphism, also known as generics.

So we should also study static types for object-oriented programs, such as those found in Java. If everything is an object (which is less true in Java than in Ruby), then the main thing we would want our type system to prevent is “method missing” errors, i.e., sending a message to an object that has no method for that message. If objects have fields accessible from outside the object (e.g., in Java), then we also want to prevent “field missing” errors. There are other possible errors as well, like calling a method with the wrong number of arguments.

Although languages like Java and C# have generics these days, subtype polymorphism, also known as subtyping, is the fundamental source of type-system expressiveness in object-oriented languages. ML does not have subtyping, by design. (Why would the designers leave it out? It would complicate type inference, for one thing.)

Our plan for the next couple lectures is to:

- study subtyping,
- compare subtyping and generics, determining which idioms are best supported by each, and
- combine subtyping and generics, showing that the result is even more useful than the sum of the two techniques.

4 Subtyping for Records and Functions

It would be natural to study subtyping using Java since it is a well-known object-oriented language with a type system that has subtyping. But Java has a number of fairly complicated features that could get in the way.

---

2“Though OOP came from many motivations, two were central. . . . [T]he small scale one was to find a more flexible version of assignment, and then to try to eliminate it altogether.” [Alan Kay, History of Smalltalk, 1993]
way of our understand the essence of subtyping.

So at first we'll study subtyping using a made-up language called SML with Mutable Fields (MF, for short) that extends SML to have some new features related to records. This language exposes how subtyping should, and should not, work. Later, we'll apply what we've learned to the more complicated setting of a class-based, object-oriented language.

MF: a made-up language of records. To study the basic ideas behind subtyping, we will use records with mutable fields, as well as functions and other expressions. Our syntax will be a mix of ML and Java that keeps examples short and, hopefully, clear. For records, we will have expressions for making records, getting a field, and setting a field.

The evaluation rules for MF are as follows:

- In the expression \(\{f_1=e_1, f_2=e_2, \ldots, f_n=e_n\}\), each \(f_i\) is a field name and each \(e_i\) is an expression. The semantics is to evaluate each \(e_i\) to a value \(v_i\) and the result is the record value \(\{f_1=v_1, f_2=v_2, \ldots, f_n=v_n\}\). So a record value is just a collection of fields, where each field has a name and contents.
- For the expression \(e.f\), the semantics is to evaluate \(e\) to a value \(v\). If \(v\) is a record with an \(f\) field, then the result is the contents of the \(f\) field. Otherwise, an error occurs. The MF type system will prevent such errors.
- For the expression \(e_1.f = e_2\), the semantics it to evaluate \(e_1\) and \(e_2\) to values \(v_1\) and \(v_2\). If \(v_1\) is a record with an \(f\) field, then that field is updated to have \(v_2\) for its contents. Otherwise, an error occurs. The MF type system will prevent such errors. The result of evaluating \(e_1.f = e_2\) is \(v_2\), though usually our examples will ignore that “return value.”

Next, we define MF’s type system. It has a form of types for records, and typing rules for kind of expression. Record types are written \(\{f_1:t_1, f_2:t_2, \ldots, f_n:t_n\}\), just like in ML. For example, \(\{x:real, y:real\}\) is the type of records with two fields named \(x\) and \(y\) that hold contents of type \(\text{real}\). And

\[
\{\text{foo}: \{x : \text{real}, y : \text{real}\}, \text{bar} : \text{string}, \text{baz} : \text{string}\}
\]

is the type of records with three fields, where the \(\text{foo}\) field holds a (nested) record of type \(\{x:real, y:real\}\).

The type-checking rules for MF are as follows:

- If \(e_1\) has type \(t_1\), \(e_2\) has type \(t_2\), \ldots, \(e_n\) has type \(t_n\), then \(\{f_1=e_1, f_2=e_2, \ldots, f_n=e_n\}\) has type \(\{f_1:t_1, f_2:t_2, \ldots, f_n:t_n\}\).
- If \(e\) has a record type containing \(f : t\), then \(e.f\) has type \(t\). Otherwise, \(e.f\) does not type-check.
- If \(e_1\) has a record type containing \(f : t\) and \(e_2\) has type \(t\), then \(e_1.f = e_2\) has type \(t\). Otherwise, \(e_1.f = e_2\) does not type-check.

Assuming the “usual” ML typing rules for other expressions like variables, functions, arithmetic, and function calls, an example like this will type-check as we would expect:

```ml
fun distToOrigin (p:{x:real,y:real}) = Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```
In particular, the function `distToOrigin` has type `{x : real, y : real} -> real`, where we write function types with the same syntax as in ML.

This type system prevents run-time errors that would result from missing fields: no program that successfully type-checks would, when evaluated, try to look up a nonexistent field in a record.

**Subtyping in MF.** With our typing rules so far, this program would not type-check:

```
fun distToOrigin (p:{x:real,y:real}) = 
  Math.sqrt(p.x*p.x + p.y*p.y)
val c : {x:real,y:real,color:string} = {x=3.0, y=4.0, color="green"}
val five : real = distToOrigin(c)
```

In the call `distToOrigin(c)`, the type of the argument is `{x:real,y:real,color:string}`, but the type the function expects is `{x:real,y:real}`. That violates the usual SML typing rule that functions must be called with exactly the type of argument they expect. Nonetheless, the program above is safe: running it would not lead to accessing a field that does not exist.

So let’s make the type system more lenient: if an expression has record type `{f1:t1, ..., fn:tn}`, then let the expression also have a type where some of the fields are removed. For example, since expression `c` has type `{x:real,y:real,color:string}`, it can also have type `{x:real,y:real}`. Then the example will type-check! We say that `{x:real,y:real,color:string}` is a subtype of `{x:real,y:real}`. (Likewise, we could use `c` as an argument to a function of type `{color:string} -> int`, and `{x:real,y:real,color:string}` is a subtype of `{color:string}`.)

It might seem backwards that a subtype contains more fields than a supertype. Subtyping does not mean “subset on fields.” The right way to think about it is that there are “fewer” values of the subtype than of the supertype, because values of the subtype have more obligations—for example, having more fields.