Overlapping pattern matches

Ben Sherman

November 3, 2016

0.1 Introduction and motivating examples

The purpose of this note is to explain how the notion of pattern matching in functional programming can be generalized to programming with continuous spaces, where the patterns should be allowed to overlap. This notion of pattern matching highlights the centrality of open covers, locality, and non-determinism for programming continuous functions.

The idea arose from assimilating two example programs. One is a toy example that I now like to call "Buridan's autonomous car." An autonomous car is approaching a yellow light and must decide whether to stop for the light or to proceed past the intersection. We imagine that the car's state is parameterized only by its distance from the light, a real number ($\mathbb{R}$), and that its output should be Boolean-valued ($\mathbb{B}$), indicating whether or not to brake. If the car is very far backwards, then the car should certainly brake, but if it is very far forwards, it should certainly proceed through the intersection, so the decision should not be a constant function. However, we notice that since the constant functions are the only possible continuous maps from $\mathbb{R}$ to $\mathbb{B}$ (in topological jargon, $\mathbb{R}$ is connected), we must do something else.

That "something else" is to allow non-determinism in the output decision. If it is both safe to stop as well as safe to go, it's alright to make either decision, even if one sometimes makes different decisions based on the *same exact* input value. There is space $\mathcal{P}^+_\Diamond(\mathbb{B})$ of non-empty subsets of $\mathbb{B}$ which should be the return value of the braking decision, rather than $\mathbb{B}$ itself. Then we can write the following function to specify the braking behavior:

$$\mathsf{brake?} : \mathbb{R} \to \mathcal{P}^+_\Diamond(\mathbb{B})$$

$$\mathsf{brake?}(s) \triangleq \mathsf{cases}(s) \begin{cases} \cdot > -1 & \Longrightarrow \quad \{\mathsf{false}\} \\ \cdot < 0 & \Longrightarrow \quad \{\mathsf{true}\}. \end{cases}$$

Note that while the two cases do indeed cover the entire space $\mathbb{R}$, they overlap. The computational interpretation is that the scrutinee of a cases expression is allowed to choose any case which it satisfies and then exhibit the behavior in that branch. Therefore, the result behavior is the union of the behaviors of all possible branches. If an input satisfies two cases, such as $-0.5 : \mathbb{R}$ in the above definition, then its "behavior" is the "maximum" of $\{\mathsf{true}\}$ and $\{\mathsf{false}\}$, which in this case is $\{\mathsf{true}, \mathsf{false}\}$. It is not always the case it's possible to take a union of behaviors of several programs, as it is here (in fact, this union exists for any two expressions of type $\mathcal{P}_\Diamond(A)$ for any A). In fact, as long as one can produce a maximum of two branches, when restricted to the intersection of their domains, then the cases expression will be well-defined.

The other example is the definition of multiplication of real numbers. It's quite different from the previous example: the pattern match has infinitely many cases (all overlapping each other), and

produces a deterministic function. I have defined the real numbers $\mathbb{R}$ using a generic construction of completion of metric spaces, taking the space $\mathbb{R}$ as the metric completion of the set $\mathbb{Q}$. The main construction that I have allows Lipschitz functions defined on metric "sets" (e.g., the set $\mathbb{Q}$ with its relevant metric) to be lifted to their metric completions. This is quite general, but already multiplication of real numbers is not Lipschitz, and therefore cannot be defined as the extension of multiplication of rational numbers. However, multiplication is *locally* Lipschitz; if one restricts one of the factors to have absolute value bounded by L, then L is a satisfactory Lipschitz constant for the multiplication. Therefore, it is possible to define multiplication with the overlapping pattern match

$$\begin{aligned} &\times : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ &x \times y \triangleq \mathsf{cases}(x) \left\{ [L : \mathbb{Q}^+] \quad x' \models -L < \cdot < L \quad \Longrightarrow \quad \mathsf{scale}_L(x', y) \right. \quad , \end{aligned}$$

where we have the family of functions $\mathsf{scale}_L : \{\mathbb{R} \mid -L < \cdot < L\} \times \mathbb{R} \to \mathbb{R}$ [1] for $L : \mathbb{Q}^+$ which are directly defined as the extension of Lipschitz functions on the rationals. In this definition, notice that we "bind" the pattern variable x' as a variable of type $\{\mathbb{R} \mid -L < \cdot < L\}$.

To prove that the above definition is in fact valid, we must show that the cases suffice to cover $\mathbb{R}$, and that any two branches have a "maximum" on their region of overlap. The former should be clear, and we should be able to prove that branches always *agree* exactly on their region of overlap, so that maximum exists: it's the same as either branch. These proofs confirm the validity of the above definition and provide computational content so that the function can be run.

0.2 Patterns using sites of the gros topos

Let's get into the details, and in particular, we will generalize the notion of open covers to sites, which make for more convenient pattern matching.

First, some preliminary notions. For any two spaces X and Y, let $X \cong Y$ indicate that the spaces X and Y are homeomorphic, meaning that we have exhibited continuous maps $f : X \to Y$ and $g : Y \to X$ such that $g \circ f = \mathsf{id}_X$ as well as $f \circ g = \mathsf{id}_Y$. We call a map $f : A \to B$ is an *open embedding* if $f(A)$ is open and $A \cong f(A)$. The notation $f : A \hookrightarrow B$ indicates that the continuous map $f : A \to B$ is an open embedding.

The gros topos over **Top**, the category of topological spaces and their continuous maps, is a Grothendieck topos whose site is **Top** endowed with the coverage where a collection of open embeddings $(f_i : A_i \hookrightarrow B)_{i:I}$ is considered a cover if

$$\bigcup_{i:I} f_i(A_i) = B.$$

In the case where a cover $(f_i : A_i \hookrightarrow B)_{i:I}$ is *disjoint*, i.e., if $i \neq j$, then $f_i(A_i) \cap f_j(A_j)$ is empty, then we recover the ordinary notion of pattern matching. For instance, for two spaces $L, R : \mathsf{Space}$, we have the disjoint covering (often called a *partition*)

$$\{\mathsf{inl} : L \hookrightarrow L + R\,,\, \mathsf{inr} : R \hookrightarrow L + R\},$$

[1] For a space A and open $U : \mathcal{O}(A)$, the notation $\{A \mid U\}$ indicates the open subspace U of A.

and accordingly we define functions on coproducts via pattern matching, just as is possible in any functional programming language:

$$\mathsf{either}(f : L \to A)(g : R \to A)(x : L + R) : A \triangleq \mathsf{cases}(x) \begin{cases} \mathsf{inl}(\ell) & \Longrightarrow & f(\ell) \\ \mathsf{inr}(r) & \Longrightarrow & g(r) \end{cases}$$

We can imagine that this is "run" on points in the following way. First, a point $x : L + R$ is presented with the open cover

$$\top \vdash_{L+R} \mathsf{inl}(L) \vee \mathsf{inr}(R).$$

Suppose we have that x is actually on the L side, so that we learn $x \models \mathsf{inl}(L)$. Therefore, we now have a point $x' : \{L + R \mid \mathsf{inl}(L)\}$. Since we have proved that inl is an open embedding, there is a map $\mathsf{inl}^{-1} : \{L + R \mid \mathsf{inl}(L)\} \hookrightarrow L$, and so we can define $\ell \triangleq \mathsf{inl}^{-1}(x')$, and then follow the first case branch to compute $f(\ell)$ as the result.

Now, let's define the unique polymorphic map

$$\mathsf{distr}(x : A \times B + A \times C) : A \triangleq \mathsf{cases}(x) \begin{cases} \mathsf{inl}(a, b) & \Longrightarrow & a \\ \mathsf{inr}(a, c) & \Longrightarrow & a \end{cases}$$

Here, we have introduced the new notion that we can pattern match on products. The way that we "execute" this is, for instance, looking at the first branch, if we've already computed $\ell : A \times B$ as was previously described, we simply define $a \triangleq \mathsf{fst}(\ell)$ and $b \triangleq \mathsf{snd}(\ell)$. This sort of pattern matching seems to be specific for product spaces.

Additionally, I'll note that sites (or Grothendieck pretopologies[2]?) should have the singleton cover which is the whole space ("reflexivity"), which corresponds to a pattern which may change to an isomorphic space but doesn't really break it apart, as well as a notion of composition of covers ("transitivity"), which corresponds to the potential to nest patterns. For instance, given the homeomorphism $\mathsf{add3} : \mathbb{Z} \hookrightarrow \mathbb{Z}$, we can get the inverse map using a pattern,

$$\begin{aligned} &\mathsf{subtract3} : \mathbb{Z} \to \mathbb{Z} \\ &\mathsf{subtract3}(x) \triangleq \mathsf{cases}(x) \left\{ \mathsf{add3}(y) \Longrightarrow y \right. \quad . \end{aligned}$$

Recall that this is not very impressive, since proving that $\mathsf{add3}$ is an open embedding requires providing such an inverse map in the first place.

In general, the covers will not be disjoint, so we need to deal with what happens when different cases overlap. For instance, consider the autonomous car which needs to decide whether to brake as it approaches a yellow light:

$$\begin{aligned} &\mathsf{brake?} : \mathbb{R} \to \mathcal{P}^{+}_{\Diamond}(\mathbb{B}) \\ &\mathsf{brake?}(s) \triangleq \mathsf{cases}(s) \begin{cases} x \models \cdot > -1 & \Longrightarrow & \{\mathsf{false}\} \\ y \models \cdot < 0 & \Longrightarrow & \{\mathsf{true}\} \end{cases} \end{aligned}$$

The syntax for this style of pattern match is up in the air. We note that we have the cover

$$\top \vdash_{\mathbb{R}} \cdot > -1 \vee \cdot < 0,$$

[2]See `https://ncatlab.org/nlab/show/Grothendieck+pretopology`.

and we can identify the open $\cdot > -1 : \mathcal{O}(\mathbb{R})$ with an open embedding $\{\mathbb{R} \mid \cdot > -1\} \hookrightarrow \mathbb{R}$. The pattern $x \models \cdot > -1$ is supposed to indicate that $x : \{\mathbb{R} \mid \cdot > -1\}$ is the variable that was "injected" into $\mathbb{R}$ by that open embedding. Though in this particular case, the x and y patterns in the case expression aren't even used.

We imagine the same process for computing the result of this expression, in some sense. We first present our input point s with the appropriate cover, and then follow that branch. But what if s satisfies several of the cases? Then it might follow any of the branches. So the behavior of the output should be the union of the behaviors in all possible branches. In this case, if we have $-1 < s < 0$, then s satisfies both cases. Here, it is clear that the right notion of the output point is the subspace $\{\mathsf{true}, \mathsf{false}\} : \mathcal{P}^+_\Diamond(\mathbb{B})$. The key property that makes this the "correct" answer is that, in terms of specialization order,

$$\{\mathsf{true}, \mathsf{false}\} = \max\left(\{\mathsf{true}\}, \{\mathsf{false}\}\right).$$

We note that the key fact is that it is possible to take *directed* suprema of points (as well as continuous maps), so we can specify the output result by saying its behavior is the union (i.e., supremum) of all possible behaviors, and we can create an implementation for this specification by ensuring that this supremum is the *directed* supremum of the behaviors of the each possible branch. We can ensure this by providing, for each pair of branches $f_i : A_i \hookrightarrow B$ and $f_j : A_j \hookrightarrow B$, an open embedding $f_{ij} : A_i \times_B A_j \hookrightarrow B$ from the pullback of f_i and f_j such that

$$f_{ij} = \max(f_i \circ e_i, f_j \circ e_j),$$

where $e_i : A_i \times_B A_j \hookrightarrow A_i$ and $e_j : A_i \times_B A_j \hookrightarrow A_j$ are the expected pullback maps. In the case where the open embeddings are simply open subspaces, $f_i : \{B \mid U_i\} \hookrightarrow B$ and $f_j : \{B \mid U_j\} \hookrightarrow B$, then the requirement reduces to producing a map $f_{ij} : \{B \mid U_i \wedge U_j\} \hookrightarrow B$ such that $f_{ij} = \max(f_i, f_j)$ in terms of specialization order (considering all these functions on their common domain $\{B \mid U_i \wedge U_j\}$). While this may seem like a special case, it's just as general: the difference is just up to the homeomorphism between a general space and the open subspace which is the image of the open embedding. That is, even given $f_i : A_i \hookrightarrow B$ and $f_j : A_j \hookrightarrow B$, it suffices to give a map

$$f_{ij} : \{B \mid f_i(A_i) \wedge f_j(A_j)\} \hookrightarrow B,$$

since

$$\{B \mid f_i(A_i) \wedge f_j(A_j)\} \cong A_i \times_B A_j.$$

For the inhabited $\Diamond$-powerspace $\mathcal{P}^+_\Diamond(A)$ of any space A, this maximum *always* exists: it's just the familiar join operation, which takes two non-deterministic operations and executes either one non-deterministically. Its general specification is

$$\cdot \vee \cdot : \mathcal{P}^+_\Diamond(A) \times \mathcal{P}^+_\Diamond(A) \to \mathcal{P}^+_\Diamond(A)$$
$$x \vee y \models \Diamond P \triangleq x \models \Diamond P \vee y \models \Diamond P,$$

and this definition makes it clear why this produces the maximum in terms of specialization order.

0.3 General specification of overlapping patterns

So in general, a general pattern match on a term $x : A$ to produce a value in B, might look like

$$\mathsf{cases}(x) \left\{ [i : I] \quad f_i(x_i) \quad \Longrightarrow e_i(x_i) \quad , \right.$$

where i ranges over the index set I, and $f_i : A_i \hookrightarrow B$ is an open embedding, and $e_i : A_i \to B$ is a general expression, and $x_i : A_i$ is a pattern. The cases expression above entirely determines the *specification* of the program, assuming it is valid, but we still need to prove two conditions to show that it is valid, which are also then used in the *implementation* of the program. That means that these proofs will affect the *computational* behavior of the program, meaning that different proofs might result in different responses to open covers when points are evaluated. The two conditions are

Covering

$$\top \vdash_B \bigvee_{i:I} f_i(A_i)$$

Gluing For each $i, j : I$, there is a map

$$f_{ij} : A_i \times_B A_j \to B$$

such that

$$f_{ij} = \max(f_i \circ e_i, f_j \circ e_j),$$

where $e_i : A_i \times_B A_j \hookrightarrow A_i$ and $e_j : A_i \times_B A_j \hookrightarrow A_j$ are the expected pullback maps.

Sometimes, we can automatically/generically discharge either of these two goals. For instance, if the output space B is such that any two (generalized) points have a maximum in terms of specialization order, then the *gluing* condition is automatically fulfilled. For instance, this holds for any lower powerspace $\mathcal{P}_\Diamond(A)$, including the supported subspaces $\mathcal{P}^+_\Diamond(A)$.

0.4 Catch-all cases

If the output space B has a bottom point $\bot_B : B$, then in some sense we can leave out the *covering* condition as well. We note that for any cases expression

$$\mathsf{cases}(x) \begin{cases} [i : I] \quad f_i(x_i) & \Longrightarrow e_i(x_i) \end{cases}$$

which is already valid where the output space B has a bottom point, we can add a catch-all case

$$\mathsf{cases}(x) \begin{cases} [i : I] \quad f_i(x_i) & \Longrightarrow e_i(x_i) \\ \_\_ & \Longrightarrow \bot_B \end{cases}$$

which has the same meaning (where the __ wildcard indicates the subspace $\top$ of the input space A which is in fact the entire space). The catch-all case already suffices to cover the whole space, of course.

Since adding the catch-all case doesn't change the meaning, it makes sense to keep track of which spaces B have bottom points, and when a cases expression is written whose output space B has a bottom point, the catch-all case can be automatically added.

For any space A there is a construction to produce the "lifted" space $A_\bot$ which adds in a bottom point, and in fact, I'm pretty sure that any space B which has a bottom point satisfies $B \cong B_\bot$, which means that *every* space with a bottom point looks like $X_\bot$ for some X. For each lifted space $A_\bot$, there is importantly the open embedding $\mathsf{strict} : A \hookrightarrow A_\bot$.

We can map a function over lifted spaces with the overlapping pattern

$$\mathsf{map}_\perp(f : A \to B)(x : A_\perp) : B_\perp \triangleq \mathsf{cases}(x) \left\{ \mathsf{strict}(z) \quad \Longrightarrow \quad \mathsf{strict}(f(z)), \right.$$

which I think of as Haskell-ish, as it forces its argument to compute the result, but if the input is $\perp$, then so is the result.

We have that the Sierpínski space Σ (the space of "truth values") is homeomorphic to $*_\perp$ (where $*$ is the unit type), and in fact, it is convenient to take this as the definition of the Sierpínski space, where we have

$$\begin{aligned} \mathsf{true} : \Sigma &\triangleq \mathsf{strict}(\mathsf{tt}) \\ \mathsf{false} : \Sigma &\triangleq \perp. \end{aligned}$$

Furthermore, we also notice that the Sierpínski space automatically satisfies the gluing condition, as we also have $\Sigma \cong \mathcal{P}_\Diamond(*)$ (where $\mathsf{false} : \Sigma$ maps to $\{\} : \mathcal{P}_\Diamond(*)$ and $\mathsf{true} : \Sigma$ maps to $\{\mathsf{tt}\} : \mathcal{P}_\Diamond(*)$). Therefore, whenever writing any overlapping pattern match which outputs to Σ, no additional proofs are necessary! Simply writing the specification with the overlapping pattern is sufficient.

This yields some cute ways of writing the logical "and" and "or" operations for Σ:

$$\begin{aligned} \wedge &: \Sigma \times \Sigma \to \Sigma \\ \wedge(p) &\triangleq \mathsf{cases}(p) \left\{ \mathsf{strict}, \mathsf{strict} \quad \Longrightarrow \quad \mathsf{true} \right. \\ \vee &: \Sigma \times \Sigma \to \Sigma \\ \vee(p) &\triangleq \mathsf{cases}(p) \begin{cases} \mathsf{strict}, \_ & \Longrightarrow \quad \mathsf{true} \\ \_, \mathsf{strict} & \Longrightarrow \quad \mathsf{true} \end{cases} \end{aligned}$$

The first definition perhaps looks similar to the Haskell definition

```
and :: () -> () -> ()
and () () = ()
```

which looks trivial, but has an important computational interpretation of forcing both of its arguments whenever the return value is forced. However, the definition of $\vee$, which performs the "or" operation, has no analog in Haskell. The "similar" Haskell definition

```
or :: () -> () -> ()
or () _ = ()
or _ () = ()
```

will in fact behave computationally as if the second pattern were missing (i.e., it forces the first argument but not the second). This operation is often known as the "parallel-OR" operation, which Martín Escardó discusses in [Esc04]. It has this name because of the interpretation of the space Σ representing semi-decision procedures: if we have two semi-decision procedures, we can create a procedure which halts if and only if either of the component procedures halts by interleaving them in a parallel fashion. Such a general notion of interleaving is unavailable in most programming languages, as far as I'm aware.

Note that, viewing Σ as $*_\bot$, the "and" operation readily generalizes to

$$\mathsf{pair}_\bot : A_\bot \times B_\bot \to (A \times B)_\bot$$
$$\mathsf{pair}_\bot(p) \triangleq \mathsf{cases}(p) \left\{ \mathsf{strict}(x), \mathsf{strict}(y) \quad \Longrightarrow \quad \mathsf{strict}(x, y) \right.$$

but it isn't possible to generalize "or" to something that would look like $\mathsf{either}_\bot : A_\bot \times A_\bot \to A_\bot$. The fact that the "or" operation works for the Sierpínski space is because $*$ has only one element, so one cannot tell if they observed the result due to either the left argument or the right argument, while for general A, this may not be the case. This is why the Haskell `or` definition is problematic. If we instead view Σ as $\mathcal{P}_\Diamond(*)$, then the "and" operation generalizes to intersection, whereas the "or" operation generalizes to union. It doesn't seem that writing intersection and union on $\mathcal{P}_\Diamond(A)$ can be phrased as overlapping pattern matches, however (unless A is discrete).

References

[Esc04] Martín Escardó. Synthetic topology: of data types and classical spaces. *Electronic Notes in Theoretical Computer Science*, 87:21–156, 2004.