

# Funcons-beta: Patterns \*

The P<sub>L</sub>anCompS Project

Patterns.cbs | PLAIN | PRETTY

## OUTLINE

- Patterns
  - Simple patterns
  - Pattern matching

---

## Patterns

```
[ Datatype  patterns
  Funcon   pattern
  Funcon   pattern-any
  Funcon   pattern-bind
  Funcon   pattern-type
  Funcon   pattern-else
  Funcon   pattern-unite
  Funcon   match
  Funcon   match-loosely
  Funcon   case-match
  Funcon   case-match-loosely
  Funcon   case-variant-value ]
```

General patterns are simple patterns or structured patterns. Matching a pattern to a value either computes an environment or fails.

Simple patterns are constructed from abstractions whose bodies depend on a given value, and whose executions either compute environments or fail.

Structured patterns are composite values whose components may include simple patterns as well as other values.

Matching a structured value to a structured pattern is similar to assigning a structured value to a structured variable, with simple pattern components matching component values analogously to simple variable components assigned component values.

Note that patterns match only values, not (empty or proper) sequences.

*Meta-variables*  $T, T' <:$  values

---

\*Suggestions for improvement: [plancomps@gmail.com](mailto:plancomps@gmail.com).  
Reports of issues: <https://github.com/plancomps/CBS-beta/issues>.

## Simple patterns

*Datatype* `patterns` ::= `pattern`(`_` : `abstractions`(`values` ⇒ `environments`))

`patterns` is the type of simple patterns that can match values of a particular type.

`pattern`(`abstraction`(`X`)) constructs a pattern with dynamic bindings, and `pattern`(`closure`(`X`)) computes a pattern with static bindings. However, there is no difference between dynamic and static bindings when the pattern is matched in the same scope where it is constructed.

*Funcon* `pattern-any` : ⇒ `patterns`  
     $\rightsquigarrow$  `pattern`(`abstraction`(`map`()))

`pattern-any` matches any value, computing the empty environment.

*Funcon* `pattern-bind`(`I` : `identifiers`) : ⇒ `patterns`  
     $\rightsquigarrow$  `pattern`(`abstraction`(`bind-value`(`I`, `given`)))

`pattern-bind`(`I`) matches any value, computing the environment binding `I` to that value.

*Funcon* `pattern-type`(`T`) : ⇒ `patterns`  
     $\rightsquigarrow$  `pattern`(`abstraction`(`if-true-else`(`is-in-type`(`given`, `T`), `map`(), `fail`)))

`pattern-type`(`T`) matches any value of type `T`, computing the empty environment.

*Funcon* `pattern-else`(`_` : `values`, `_` : `values`) : ⇒ `patterns`  
*Rule* `pattern-else`(`P1` : `values`, `P2` : `values`)  $\rightsquigarrow$   
    `pattern`(`abstraction`(`else`(`match`(`given`, `P1`), `match`(`given`, `P2`))))

`pattern-else`(`P1`, `P2`) matches all values matched by `P1` or by `P2`. If a value matches `P1`, that match gives the computed environment; if a value does not match `P1` but matches `P2`, that match gives the computed environment; otherwise the match fails.

*Funcon* `pattern-unite`(`_` : `values`, `_` : `values`) : ⇒ `patterns`  
*Rule* `pattern-unite`(`P1` : `values`, `P2` : `values`)  $\rightsquigarrow$   
    `pattern`(`abstraction`(`collateral`(`match`(`given`, `P1`), `match`(`given`, `P2`))))

`pattern-unite`(`P1`, `P2`) matches all values matched by both `P1` and `P2`, then uniting the computed environments, which fails if the domains of the environments overlap.

## Pattern matching

*Funcon* `match`(`_` : `values`, `_` : `values`) : ⇒ `environments`

`match`(`V`, `P`) takes a (potentially structured) value `V` and a (potentially structured) pattern `P`. Provided that the structure and all components of `P` exactly match the structure and corresponding components of `V`, the environments computed by the simple pattern matches are united.

*Rule*  $\text{match}(V : \text{values}, \text{pattern}(\text{abstraction}(X))) \rightsquigarrow \text{give}(V, X)$

*Rule* 
$$\frac{l_2 \neq \text{"pattern"}}{\text{match}(\text{datatype-value}(l_1 : \text{identifiers}, V_1^* : \text{values}^*), \text{datatype-value}(l_2 : \text{identifiers}, V_2^* : \text{values}^*)) \rightsquigarrow \text{sequential}(\text{check-true}(\text{is-equal}(l_1, l_2)), \text{check-true}(\text{is-equal}(\text{length } V_1^*, \text{length } V_2^*)), \text{collateral}(\text{interleave-map}(\text{match}(\text{tuple-elements}(\text{given})), \text{tuple-zip}(\text{tuple}(V_1^*), \text{tuple}(V_2^*))))))}$$

*Rule* 
$$\frac{\text{dom}(M_2) == \{ \}}{\text{match}(M_1 : \text{maps}(-, -), M_2 : \text{maps}(-, -)) \rightsquigarrow \text{if-true-else}(\text{is-equal}(\text{dom}(M_1), \{ \}), \text{map}(\text{ }, \text{fail}))}$$

*Rule* 
$$\frac{\text{dom}(M_2) \neq \{ \} \quad \text{some-element}(\text{dom}(M_2)) \rightsquigarrow K}{\text{match}(M_1 : \text{maps}(-, -), M_2 : \text{maps}(-, -)) \rightsquigarrow \text{if-true-else}(\text{is-in-set}(K, \text{dom}(M_1)), \text{collateral}(\text{match}(\text{map-lookup}(M_1, K), \text{map-lookup}(M_2, K)), \text{match}(\text{map-delete}(M_1, \{K\}), \text{map-delete}(M_2, \{K\}))), \text{fail})}$$

*Rule* 
$$\frac{P : \sim(\text{datatype-values} \mid \text{maps}(-, -))}{\text{match}(V : \text{values}, P : \text{values}) \rightsquigarrow \text{if-true-else}(\text{is-equal}(V, P), \text{map}(\text{ }, \text{fail}))}$$

*Funcon*  $\text{match-loosely}(\_ : \text{values}, \_ : \text{values}) : \Rightarrow \text{environments}$

$\text{match-loosely}(V, P)$  takes a (potentially structured) value  $V$  and a (potentially structured) pattern  $P$ . Provided that the structure and all components of  $P$  loosely match the structure and corresponding components of  $V$ , the environments computed by the simple pattern matches are united.

*Rule*  $\text{match-loosely}(V : \text{values}, \text{pattern}(\text{abstraction}(X))) \rightsquigarrow \text{give}(V, X)$

*Rule* 
$$\frac{l_2 \neq \text{"pattern"}}{\text{match-loosely}(\text{datatype-value}(l_1 : \text{identifiers}, V_1^* : \text{values}^*), \text{datatype-value}(l_2 : \text{identifiers}, V_2^* : \text{values}^*)) \rightsquigarrow \text{sequential}(\text{check-true}(\text{is-equal}(l_1, l_2)), \text{check-true}(\text{is-equal}(\text{length } V_1^*, \text{length } V_2^*)), \text{collateral}(\text{interleave-map}(\text{match-loosely}(\text{tuple-elements}(\text{given})), \text{tuple-zip}(\text{tuple}(V_1^*), \text{tuple}(V_2^*))))))}$$

*Rule* 
$$\frac{\text{dom}(M_2) == \{ \}}{\text{match-loosely}(M_1 : \text{maps}(-, -), M_2 : \text{maps}(-, -)) \rightsquigarrow \text{map}( )}$$
  

$$\text{dom}(M_2) \neq \{ \}$$

*Rule* 
$$\frac{\text{some-element}(\text{dom}(M_2)) \rightsquigarrow K}{\text{match-loosely}(M_1 : \text{maps}(-, -), M_2 : \text{maps}(-, -)) \rightsquigarrow \text{if-true-else}(\text{is-in-set}(K, \text{dom}(M_1)), \text{collateral}(\text{match-loosely}(\text{map-lookup}(M_1, K), \text{map-lookup}(M_2, K)), \text{match-loosely}(\text{map-delete}(M_1, \{K\}), \text{map-delete}(M_2, \{K\}))), \text{fail})}$$

*Rule* 
$$\frac{P : \sim(\text{datatype-values} \mid \text{maps}(-, -))}{\text{match-loosely}(DV : \text{values}, P : \text{values}) \rightsquigarrow \text{if-true-else}(\text{is-equal}(DV, P), \text{map}( ), \text{fail})}$$

*Funcon*  $\text{case-match}(\_ : \text{values}, \_ : \Rightarrow T') : \Rightarrow T'$

$\text{case-match}(P, X)$  matches  $P$  exactly to the given value. If the match succeeds, the computed bindings have scope  $X$ .

*Rule*  $\text{case-match}(P : \text{values}, X) \rightsquigarrow \text{scope}(\text{match}(\text{given}, P), X)$

*Funcon*  $\text{case-match-loosely}(\_ : \text{values}, \_ : \Rightarrow T') : \Rightarrow T'$

$\text{case-match}(P, X)$  matches  $P$  loosely to the given value. If the match succeeds, the computed bindings have scope  $X$ .

*Rule*  $\text{case-match-loosely}(P : \text{values}, X) \rightsquigarrow \text{scope}(\text{match-loosely}(\text{given}, P), X)$

*Funcon*  $\text{case-variant-value}(\_ : \text{identifiers}) : \Rightarrow \text{values}$

$\text{case-variant-value}(I)$  matches values of variant  $I$ , then giving the value contained in the variant.

*Rule*  $\text{case-variant-value}(I : \text{identifiers}) \rightsquigarrow \text{case-match}(\text{variant}(I, \text{pattern-any}), \text{variant-value}(\text{given}))$