Funcons-beta: Patterns

The PLanCompS Project

`Funcons-beta/Values/Abstraction/Patterns/Patterns.cbs`*

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.

*Suggestions for improvement: `plancomps@gmail.com`.
Issues: `https://github.com/plancomps/CBS-beta/issues`.

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

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
⇝ pattern(abstraction(map( )))

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

Funcon pattern-bind(I : identifiers) : ⇒ patterns
⇝ 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
⇝ 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(P_1 : values, P_2 : values) ⇝ pattern(abstraction(else(match(given, P_1), match(given, P_2))

$\mathsf{pattern\text{-}else}(P_1, P_2)$ matches all values matched by P_1 or by P_2. If a value matches P_1, that match gives the computed environment; if a value does not match P_1 but matches P_2, that match gives the computed environment; otherwise the match fails.

$$\textit{Funcon}\ \ \mathsf{pattern\text{-}unite}(\_ : \mathsf{values}, \_ : \mathsf{values}) : \Rightarrow \mathsf{patterns}$$

$$\textit{Rule}\ \ \mathsf{pattern\text{-}unite}(P_1 : \mathsf{values}, P_2 : \mathsf{values}) \rightsquigarrow \mathsf{pattern}(\mathsf{abstraction}(\mathsf{collateral}(\mathsf{match}(\mathsf{given}, P_1), \mathsf{match}(\mathsf{give}$$

$\mathsf{pattern\text{-}unite}(P_1, P_2)$ matches all values matched by both P_1 and P_2, then uniting the computed environments, which fails if the domains of the environments overlap.

Pattern matching

$$\textit{Funcon}\ \ \mathsf{match}(\_ : \mathsf{values}, \_ : \mathsf{values}) : \Rightarrow \mathsf{environments}$$

$\mathsf{match}(V, P)$ 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.

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

$$\textit{Rule}\ \ \frac{}{\mathsf{match}(\mathsf{datatype\text{-}value}(I_1 : \mathsf{identifiers}, {V_1}^* : \mathsf{values}^*), \mathsf{datatype\text{-}value}(I_2 : \mathsf{identifiers}, {V_2}^* : \mathsf{values}^*)) \rightsquigarrow \mathsf{s}}$$

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

$$\textit{Rule}\ \ \frac{\begin{array}{r}\mathsf{dom}(M_2) \neq \\ \mathsf{some\text{-}element}(\mathsf{dom}(\end{array}}{\mathsf{match}(M_1 : \mathsf{maps}(\_, \_), M_2 : \mathsf{maps}(\_, \_)) \rightsquigarrow \mathsf{if\text{-}true\text{-}else}(\mathsf{is\text{-}in\text{-}set}(K, \mathsf{dom}(M_1)), \mathsf{collateral}(\mathsf{match}(\mathsf{map\text{-}loo}}$$

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

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

$\mathsf{match\text{-}loosely}(V, P)$ 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 $\mathsf{match\text{-}loosely}(V : \mathsf{values}, \mathsf{pattern}(\mathsf{abstraction}(X))) \leadsto \mathsf{give}(V, X)$

Rule $\dfrac{}{\mathsf{match\text{-}loosely}(\mathsf{datatype\text{-}value}(I_1 : \mathsf{identifiers}, {V_1}^* : \mathsf{values}^*), \mathsf{datatype\text{-}value}(I_2 : \mathsf{identifiers}, {V_2}^* : \mathsf{values}}$

Rule $\dfrac{\mathsf{dom}(M_2) == \{\ \}}{\mathsf{match\text{-}loosely}(M_1 : \mathsf{maps}(\_,\_), M_2 : \mathsf{maps}(\_,\_)) \leadsto \mathsf{map}(\ )}$

Rule $\dfrac{\begin{array}{r}\mathsf{d}\\ \mathsf{some\text{-}ele}\end{array}}{\mathsf{match\text{-}loosely}(M_1 : \mathsf{maps}(\_,\_), M_2 : \mathsf{maps}(\_,\_)) \leadsto \mathsf{if\text{-}true\text{-}else}(\mathsf{is\text{-}in\text{-}set}(K, \mathsf{dom}(M_1)), \mathsf{collateral}(\mathsf{match\text{-}l}}$

Rule $\dfrac{P : \sim(\mathsf{datatype\text{-}values} \mid \mathsf{maps}(\_,\_))}{\mathsf{match\text{-}loosely}(DV : \mathsf{values}, P : \mathsf{values}) \leadsto \mathsf{if\text{-}true\text{-}else}(\mathsf{is\text{-}equal}(DV, P), \mathsf{map}(\ ), \mathsf{fail})}$

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

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

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

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

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

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

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

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

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