Funcons-beta: Patterns *

The PLanCompS Project

`Patterns.cbs` | PLAIN | PRETTY

OUTLINE

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`.
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
⇝ 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)))))

pattern-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.

Funcon pattern-unite(_ : values, _ : values) : ⇒ patterns
Rule pattern-unite(P_1 : values, P_2 : values) ⇝
pattern(abstraction(collateral(match(given, P_1), match(given, P_2)))))

pattern-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

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 match(V : values, pattern(abstraction(X))) $\rightsquigarrow$ give(V, X)

Rule
$$\frac{I_2 \neq \text{“pattern”}}{\begin{array}{l}\text{match(}\\ \quad\text{datatype-value}(I_1 : \text{identifiers}, {V_1}^* : \text{values}^*),\\ \quad\text{datatype-value}(I_2 : \text{identifiers}, {V_2}^* : \text{values}^*)) \rightsquigarrow\\ \quad\text{sequential(}\\ \quad\quad\text{check-true}(\text{is-equal}(I_1, I_2)),\\ \quad\quad\text{check-true}(\text{is-equal}(\text{length } {V_1}^*, \text{length } {V_2}^*)),\\ \quad\quad\text{collateral(}\\ \quad\quad\quad\text{interleave-map(}\\ \quad\quad\quad\quad\text{match}(\text{tuple-elements}(\text{given})),\\ \quad\quad\quad\quad\text{tuple-zip}(\text{tuple}({V_1}^*), \text{tuple}({V_2}^*)))))\end{array}}$$

Rule
$$\frac{\text{dom}(M_2) == \{\ \}}{\begin{array}{l}\text{match}(M_1 : \text{maps}(\_, \_), M_2 : \text{maps}(\_, \_)) \rightsquigarrow\\ \quad\text{if-true-else}(\text{is-equal}(\text{dom}(M_1), \{\ \}), \text{map}(\ ), \text{fail})\end{array}}$$

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

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

Funcon match-loosely(_ : values, _ : values) : $\Rightarrow$ environments

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 match-loosely(V : values, pattern(abstraction(X))) $\rightsquigarrow$ give(V, X)

Rule
$$\frac{I_2 \neq \text{“pattern”}}{\begin{array}{l}\text{match-loosely(}\\ \quad\text{datatype-value}(I_1 : \text{identifiers}, {V_1}^* : \text{values}^*),\\ \quad\text{datatype-value}(I_2 : \text{identifiers}, {V_2}^* : \text{values}^*)) \rightsquigarrow\\ \quad\text{sequential(}\\ \quad\quad\text{check-true(is-equal}(I_1, I_2)),\\ \quad\quad\text{check-true(is-equal(length } {V_1}^*, \text{length } {V_2}^*)),\\ \quad\quad\text{collateral(}\\ \quad\quad\quad\text{interleave-map(}\\ \quad\quad\quad\quad\text{match-loosely(tuple-elements(given)),}\\ \quad\quad\quad\quad\text{tuple-zip(tuple}({V_1}^*), \text{tuple}({V_2}^*)))))\end{array}}$$

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

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

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

Funcon case-match(_ : values, _ : $\Rightarrow T'$) : $\Rightarrow T'$

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

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

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

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

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

Funcon case-variant-value(_ : identifiers) : $\Rightarrow$ values

case-variant-value(I) matches values of variant I, then giving the value contained in the variant.

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