EQUATIONS: function definitions by dependent pattern-matching and recursion

Matthieu Sozeau, $\pi.r^2$, Inria Paris & IRIF

Functional Programming Lecture
October 7th 2019
Aarhus University
Aarhus, Danemark

Typical example

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

- An equational presentation rather than a computational one. You declare the equations the function should satisfy rather than the way it is computed using a cascade of `match..with`.
- Patterns = **well-typed** refinements of the arguments
- We can refine the **entire** context at once
  $\Rightarrow$ crucial for *dependent* pattern-matching.
- **First-match** semantics + **inaccessible** patterns ensure an operational reading of the clauses

Outline

Pattern-matching and unification

Idea: reasoning up-to the theory of equality and constructors

Example: to eliminate t : vector A m, we unify with:

1. vector A O for vnil
2. vector A (S n) for vcons

Unification $t \equiv u \rightsquigarrow Q$ can result in:

- $Q = \texttt{Fail}$
- $Q = \texttt{Success}\ \sigma$ (with a substitution σ);
- $Q = \texttt{Stuck}\ t$ if t is outside the theory (e.g. a constant)

Two successes in this example for $[m := 0]$ and $[m := \mathsf{S}\ n]$ respectively.

Unification rules

SOLUTION

$$\frac{x \notin \mathcal{FV}(t)}{x \equiv t \rightsquigarrow \texttt{Success}\ \sigma[x := t]}$$

OCCUR-CHECK

$$\frac{C \text{ constructor context}}{x \equiv C[x] \rightsquigarrow \texttt{Fail}}$$

DISCRIMINATION

$$\frac{}{\mathsf{C}\ \_ \equiv \mathsf{D}\ \_ \rightsquigarrow \texttt{Fail}}$$

INJECTIVITY

$$\frac{t_1 \ldots t_n \equiv u_1 \ldots u_n \rightsquigarrow Q}{\mathsf{C}\ t_1 \ldots t_n \equiv \mathsf{C}\ u_1 \ldots u_n \rightsquigarrow Q}$$

PATTERNS

$$\frac{p_1 \equiv q_1 \rightsquigarrow \texttt{Success}\ \sigma \quad (p_2 \ldots p_n)\sigma \equiv (q_2 \ldots q_n)\sigma \rightsquigarrow Q}{p_1 \ldots p_n \equiv q_1 \ldots q_n \rightsquigarrow Q \cup \sigma}$$

DELETION

$$\frac{}{t \equiv t \rightsquigarrow \texttt{Success}\ []}$$

STUCK

$$\frac{\text{Otherwise}}{t \equiv u \rightsquigarrow \texttt{Stuck}\ u}$$

Unification examples

- $\mathsf{O} \equiv \mathsf{S}\ n \rightsquigarrow \texttt{Fail}$
- $\mathsf{S}\ m \equiv \mathsf{S}\ (\mathsf{S}\ n) \rightsquigarrow \texttt{Success}\ [m := \mathsf{S}\ n]$
- $\mathsf{O} \equiv m + \mathsf{O} \rightsquigarrow \texttt{Stuck}\ (m + \mathsf{O})$

`Stuck` cases indicate a variable to eliminate, to refine the pattern-matching problem (here variable m).

Pattern-matching compilation uses unification to:

- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable

Pattern-matching compilation

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

cover($m\ n : \mathsf{nat} \vdash m\ n : (m\ n : \mathsf{nat})$)

Pattern-matching compilation

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

cover$(m\ n : \mathsf{nat} \vdash m\ n) \rightarrow \mathsf{O}\ \mathsf{O} \equiv m\ n \rightsquigarrow \mathtt{Stuck}\ m$

Pattern-matching compilation

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

Split($m\ n$: nat $\vdash$ $m\ n$, m, [ ])

Pattern-matching compilation

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.

Split(m n : nat ⊢ n m, m, [
  cover(n : nat    ⊢ O n)
  cover(m' n : nat ⊢ (S m') n)])
```

Pattern-matching compilation

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.

Split(m n : nat ⊢ m n, m, [
  Split(n : nat ⊢ O n, n, [
    Compute(⊢ O O ⇒ true),
    Compute(n' : nat ⊢ O (S n') ⇒ false)]),
  cover(m' n : nat ⊢ (S m') n)])
```

Pattern-matching compilation

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.

Split(m n : nat ⊢ m n, m, [
  Split(n : nat ⊢ O n, n, [
    Compute(⊢ O O ⇒ true),
    Compute(n' : nat ⊢ O (S n') ⇒ false)]),
  Split(m' n : nat ⊢ (S m') n, n, [
    Compute(m' : nat ⊢ (S m') O ⇒ false),
    Compute(m' n' : nat ⊢ (S m') (S n') ⇒ equal m' n')])])
```

Outline

1. Dependent Pattern-Matching 101
   - Pattern-Matching and Unification
   - Covering

2. Tutorial
   - In Coq
   - What Are Inaccessible Patterns, you ask?

Dependent pattern-matching

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).

Equations tail A n (v : vector A (S n)) : vector A n :=
  tail A n (@cons ?(n) _ v) := v.
```

Each variable must appear only once, except in inaccessible patterns.

cover($A\ n\ v$: vector A (S n)) $\vdash A\ n\ v$)

Dependent pattern-matching

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).

Equations tail A n (v : vector A (S n)) : vector A n :=
  tail A n (@cons ?(n) _ v) := v.
```

Each variable must appear only once, except in inaccessible patterns.

```
Split(A n (v : vector A (S n)) ⊢ A n v, v, [
  Fail; // O ≠ S n
  cover(A n' a (v' : vector A n') ⊢ A n' (@cons ?(n') a v'))])
```

Dependent pattern-matching

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).

Equations tail A n (v : vector A (S n)) : vector A n :=
  tail A n (@cons ?(n) _ v) := v.
```

Each variable must appear only once, except in inaccessible patterns.

```
Split(A n (v : vector A (S n)) ⊢ A n v, v, [
 Fail; // S n ≠ O
 Compute(A n' a (v' : vector A n') ⊢ A n' (@cons ?(n') a v')
    ⇒ v')])
```

Refinement across objects

```
Equations nth {A n} (v : vector A n) (f : fin n) : A :=
  nth (@cons _ x _) (fz _) := x;
  nth (@cons ?(n) _ v) (fs n f) := nth v f.
```