Programming Languages and Paradigms
COMP 302

Instructor: Jacob Errington
School of Computer Science
McGill University

Lesson 4: compound types, constructors, and pattern matching

Last time...

- Tracing recursive functions.
- The call stack.
- Tail-recursive functions.
- Tail call optimization.
- Type synonyms
- Tuples, `fst`, `snd`
- Unpacking tuples with `let-in`.

Recap: type synonyms and tuples

- e.g. `type name = string` defines a type synonym called `name`
  - The types `name` and `string` are *interchangeable*.

Recap: type synonyms and tuples

- e.g. `type name = string` defines a type synonym called `name`
  - The types `name` and `string` are *interchangeable*.
- e.g. `name * height` is the type of tuples with a name on the left and a height on the right.
  - `let jake : name * height = ("jake", 182)`
  - The *type* uses the star `*` whereas the *expression* uses the comma `,`

Recap: type synonyms and tuples

- e.g. `type name = string` defines a type synonym called `name`
  - The types `name` and `string` are *interchangeable*.
- e.g. `name * height` is the type of tuples with a name on the left and a height on the right.
  - `let jake : name * height = ("jake", 182)`
  - The *type* uses the star `*` whereas the *expression* uses the comma `,`
- Access tuple components with functions `fst` and `snd`

Recap: type synonyms and tuples

- e.g. `type name = string` defines a type synonym called `name`
  - The types `name` and `string` are *interchangeable*.
- e.g. `name * height` is the type of tuples with a name on the left and a height on the right.
  - `let jake : name * height = ("jake", 182)`
  - The *type* uses the star `*` whereas the *expression* uses the comma `,`
- Access tuple components with functions `fst` and `snd`
- Or with *pattern matching*, using let-in:
  - `let is_tall (person : name * height) = snd person > 178`
  - `let is_tall p = failwith "exercise"`

This time...

- Disjoint unions aka “sum types” aka enumerations
- Constructors with fields
- Pattern matching
- Recursive types

Enumerated types and pattern matching on them

demo

Recap: enumerated types

- e.g. `type hand = Rock | Paper | Scissors` defines a new type by listing its values, namely `Rock`, `Paper`, and `Scissors`.
- These values are called constructors. They will be used to *construct* values of the type you're defining.
- Use pattern matching to direct control flow according to the value.

```
match e with
| Rock -> ...
| Paper -> ...
| Scissors -> ...
```

This is similar to a *switch statement* (in Java) or a *chain of if-else* (in python) to handle different cases.

Constructors with fields

demo

Recap: constructors with fields

- Constructors can have fields. A field holds some data together with the constructor, e.g.

```
type shape =
  | Circle of float
  | Square of float
  | Rect of float * float
  (* ^ separate fields with * as if a tuple *)
```

Recap: constructors with fields

- Constructors can have fields. A field holds some data together with the constructor, e.g.

```
type shape =
  | Circle of float
  | Square of float
  | Rect of float * float
  (* ^ separate fields with * as if a tuple *)
```

- Pattern matching syntax can extract the values of fields from the constructor.

```
let area (s : shape) : float = match s with
  | Circle r -> 3.14 *. r *. r
  | Square c -> c *. c
  | Rect (w, h) -> w *. h
```

Theory: pattern matching

`match e with p1 -> e1 | ... | pN -> eN`

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Syntax

- The scrutinee is an *expression*, usually a variable.

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Syntax

- The scrutinee is an *expression*, usually a variable.
- Each p_i is a *pattern*, i.e.

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Syntax

- The scrutinee is an *expression*, usually a variable.
- Each p_i is a *pattern*, i.e.
  - A variable such as `x` or a wildcard `_`.
    These variables are *bound* in the corresponding expression e_i.

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Syntax

- The scrutinee is an *expression*, usually a variable.
- Each p_i is a *pattern*, i.e.
  - A variable such as `x` or a wildcard `_`.
    These variables are *bound* in the corresponding expression e_i.
  - A tuple of other patterns: `(p1, ..., pN)`

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Syntax

- The scrutinee is an *expression*, usually a variable.
- Each p_i is a *pattern*, i.e.
  - A variable such as `x` or a wildcard `_`.
    These variables are *bound* in the corresponding expression e_i.
  - A tuple of other patterns: `(p1, ..., pN)`
  - A *constructor* applied to patterns: `Cnstr (p1, ..., pN)`

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Syntax

- The scrutinee is an *expression*, usually a variable.
- Each p_i is a *pattern*, i.e.
  - A variable such as `x` or a wildcard `_`.
    These variables are *bound* in the corresponding expression e_i.
  - A tuple of other patterns: `(p1, ..., pN)`
  - A *constructor* applied to patterns: `Cnstr (p1, ..., pN)`
- Each e_i is an expression to evaluate, contingent on p_i matching, together with variable bindings coming from p_i.

Theory: pattern matching

`match e with p1 -> e1 | ... | pN -> eN`

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Operational Semantics

1. Evaluate the scrutinee to a value `v`.

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Operational Semantics

1. Evaluate the scrutinee to a value `v`.
2. Line up `v` against each pattern p_i until we find one that matches.

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Operational Semantics

1. Evaluate the scrutinee to a value `v`.
2. Line up `v` against each pattern p_i until we find one that matches.
3. Patterns can contain variables; these become bound to the corresponding parts of `v`.

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Operational Semantics

1. Evaluate the scrutinee to a value `v`.
2. Line up `v` against each pattern p_i until we find one that matches.
3. Patterns can contain variables; these become bound to the corresponding parts of `v`.
4. The associated expression of the first matching branch is evaluated with any new bindings

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

- `e` is called the scrutinee
- Each `pi -> ei` is called a branch

Operational Semantics

1. Evaluate the scrutinee to a value `v`.
2. Line up `v` against each pattern p_i until we find one that matches.
3. Patterns can contain variables; these become bound to the corresponding parts of `v`.
4. The associated expression of the first matching branch is evaluated with any new bindings
5. That gives the result of the whole `match`-expression.

Theory: pattern matching

```
match e with p1 -> e1 | ... | pN -> eN
```

Static Semantics

- The type of each p_i must agree with the type of e
- The types of all the branch's bodies must agree with each other, i.e. all e_i have the same type.
- The type of the whole `match`-expression is the type of the e_i.

Recursive types

demo