CMPU 101 § 2 · Computer Science I

Designing Data Types

24 October 2022

Where are we?

We’ve seen numbers, strings, images, Booleans, tables, and lists.

These let us represent many kinds of real data quite naturally. But there are times when we’ll want something a bit different.

Defining structured data

Imagine that we’re doing a study on communication patterns among students.

While we don’t have the messages the students sent, we have the ***metadata*** for each message:

- sender
- recipient
- day of the week
- time (hour and minute)

This kind of metadata is quite important!

You may want to read

> John Bohannon, "Your call and text records are far more revealing than you think", *Science*, 2016

Imagine that we're doing a study on communication patterns between students.

While we don't have the messages the students sent, we have the metadata for each message:

sender

recipient

day of the week

time (hour and minute)

How should we store this data?

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*time ::* ...
`"4015551234"`	`"8025551234"`	`"Mon"`	...

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*time :: String*
`"4015551234"`	`"8025551234"`	`"Mon"`	`"4:55"`

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*time :: Number*
`"4015551234"`	`"8025551234"`	`"Mon"`	`295`

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*time :: List*
`"4015551234"`	`"8025551234"`	`"Mon"`	`[list: 4, 55]`

We could have a table, e.g.,

sender :: String	*recipient :: String*	*day :: String*	*hour :: Number*	*minute :: Number*
`"4015551234"`	`"8025551234"`	`"Mon"`	`4`	`55`

If we use multiple columns, we can access the components independently, by name, but if we use a single column, all of the “time” data is in one place.

To resolve this trade-off, we add structure: We can have a single data type that has named parts.

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

*The **name** of the data type*

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

*A **constructor** function that builds the data type*

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

*The **components** of the data*

After defining the data type,

```
data Time:
  | time(hours :: Number, mins :: Number)
end
```

we can call **time** to build **Time** values,

```
››› noon = time(12, 0)
››› half-past-three = time(3, 30)
```

and we can use dot notation to access the components:

```
››› noon.hours
12
››› half-past.mins
30
```

Our table could now be:

sender :: String	*recipient :: String*	*day :: String*	*time :: Time*
`"4015551234"`	`"8025551234"`	`"Mon"`	`time(4, 55)`

And we can write functions that use the hour and minute components, e.g.,

> **`message-before`** takes a row (representing a message) and a `Time` value and returns `true` if the message was sent before the specified time.

Defining conditional data

There are many applications where we need to represent times, and we can reuse our **`Time`** data definition.

For example, if we want to build a calendar, that's a collection of appointments, each of which has a

- Date
- Start time
- Duration
- Description

One possible design:

```
data Date:
  | date(year :: Number, month :: Number,
      day :: Number)
end

data Event:
  | event(date :: Date, time :: Time,
      duration :: Number, descr :: String)
end

calendar :: List<Event> = ...
```

Many calendar programs also offer a way to manage your to-do list.

Let’s say a to-do item has the following data:

- Task
- Deadline
- Urgency

We could have one list for calendar events and one for to-do items, but then we lose the benefit of having a single calendar with all our entries.

For many tasks (e.g., displaying entries sorted by date), we want both calendar events and to-do items.

Instead, we can define a *conditional data* type with multiple constructors:

```
data Event:
  | appt(date :: Date, time :: Time,
      duration :: Number, descr :: String)
  | todo(deadline :: Date, task :: String,
      urgency :: String)
end
```

Now a calendar can be a **`List<Event>`**, containing both types of events, e.g.,

```
calendar :: List<Event> =
  [list:
    appt(date(2022, 10, 23), time(13, 30),
      75, "CMPU 101"),
    todo(date(2022, 10, 24),
      "Use avocado", "high")]
```

But how do we work with a list where the items can have different parts?

Well, we've already seen the way to work with different varieties of data; it's **cases**!

For example, if we want to search our calendar for all events related to a term, we could write a function **`event-matches`**.

And we can use it to filter our calendar:

```
fun search-calendar(cal :: List<Event>,
    term :: String) -> List<Event>:
  doc: "Return just the calendar events that
contain the term"
  filter(
    lam(e): event-matches(e, term) end,
    cal)
end
```

Defining recursive data

A list is just a built-in kind of conditional data!

We used **`cases`** to tell apart its two possibilities – **`empty`** or **`link`**.

Now we can see how lists are defined:

```
data MyList:
  | my-empty
  | my-link(first :: Any, rest :: MyList)
end
```

Now we can see how lists are defined:

```
data MyList:
  | my-empty
  | my-link(first :: Any, rest :: MyList)
end
```

```
my-empty

my-link(1,
  my-link(2,
    my-link(3,
      my-empty)))
```

And just like we did for a List, we use this template to write a function that recursively processes the data:

```
fun my-list-fun(ml :: MyList) -> ...:
  doc: "Template for a fn that takes a MyList"
  cases (MyList) ml:
    | my-empty => ...
    | my-link(f, r) =>
      ... f ...
      ... my-list-fun(r) …
  end
where:
   my-list-fun(...) is ...
end
```

Every data definition has a corresponding template.

The more complex the data definition is – lots of variants, recursion, etc. – the more helpful it is to use the template!

Given a (recursive) data definition, you write a template by:

1 Creating a function header

2 Using cases to break the data input into its variants

3 In each case, listing each of the fields in the answer

4 Calling the function itself on any recursive fields

There's no need to define `MyList` when we already have `List`, but next class we'll see how the same idea of defining a recursive data type lets us create something new!

Class code:

`https://tinyurl.com/101-2022-10-23`

Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University

Doug Woos, Brown University