COMPSCI 3MI3 - Principles of Programming Languages

Topic 11 - Data Structures

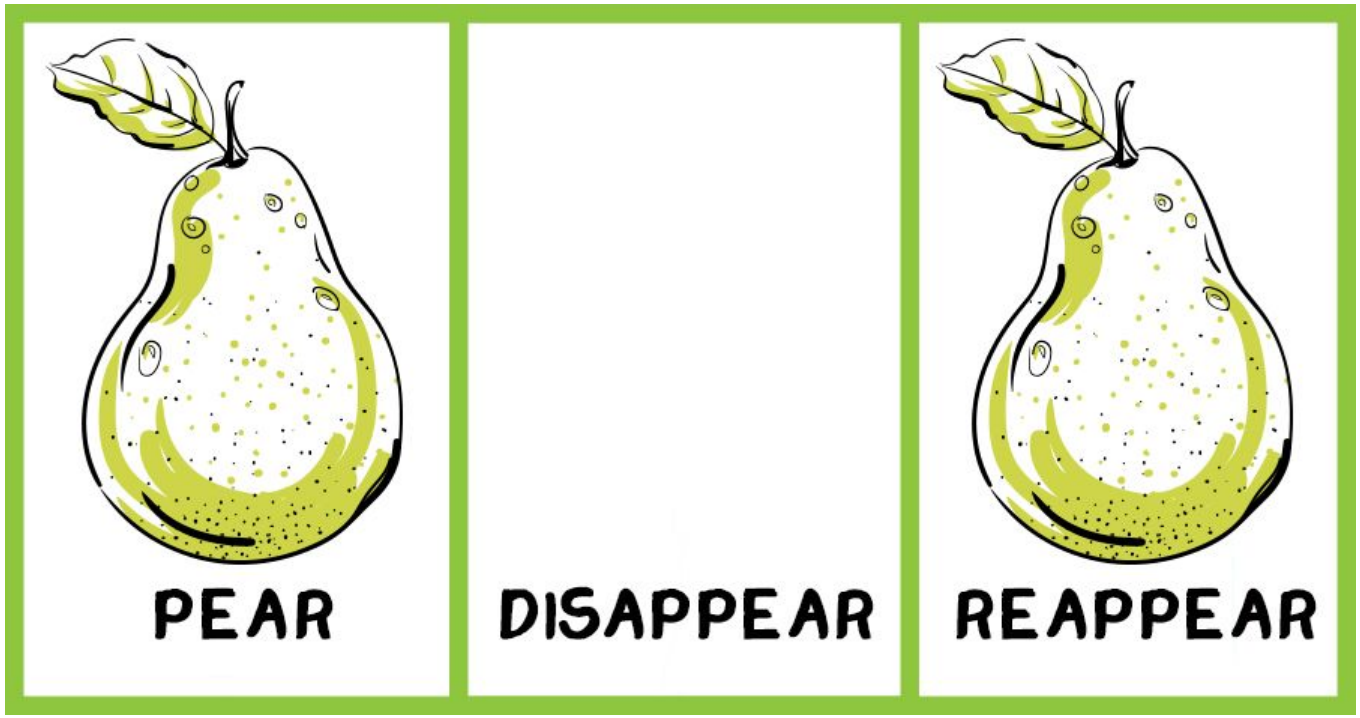
NCC Moore

McMaster University

Fall 2021

Adapted from "Types and Programming Languages" by Benjamin C. Pierce

Pairs



Adding Data Structures

Any good programmer knows that programming without data structures is like trying to bury the evidence with a soup spoon: slow and painful with a high probability of failure, and you should probably be asking yourself how you got in this position in the first place.

Over the course of this topic, we will introduce a variety of data structures to our calculus.

- Most of these will require additions to our internal calculus.
- We will therefore be adding syntax in the regular manner, and not through the use of derived forms.

The Most Basic Data Structure

We will begin by introducing **pairs**.

- A pair is very simply defined as a collection of two terms:

$$\{t_1, t_2\} \tag{1}$$

- We will access the individual elements of a pair using **projection**.
  - The term $\{t_1, t_2\}.1$ evaluates to t_1.
  - The term $\{t_1, t_2\}.2$ evaluates to t_2.
- These two projections are the equivalents of the `fst` and `snd` functions in Haskell.

Pair Semantics

Congruence Rules

$$\frac{t \to t'}{t.1 \to t'.1} \quad \text{(E-Proj1)}$$

$$\frac{t \to t'}{t.2 \to t'.2} \quad \text{(E-Proj2)}$$

$$\frac{t_1 \to t_1'}{\{t_1, t_2\} \to \{t_1', t_2\}} \quad \text{(E-Pair1)}$$

$$\frac{t_2 \to t_2'}{\{v_1, t_2\} \to \{v_1, t_2'\}} \quad \text{(E-Pair2)}$$

By now, these congruence rules should be well familiar.

- Note that we have a separate congruence rule for evaluating the inside of either projection.
- Otherwise, execution is controlled the same way as with the congruence rules for function application or sequencing.

Pear Semantics Continued

In addition, we have the following elimination rules:

$$\{v_1, v_2\}.1 \rightarrow v_1 \qquad \text{(E-PairBeta1)}$$

$$\{v_1, v_2\}.2 \rightarrow v_2 \qquad \text{(E-PairBeta2)}$$

Again, these rules are quite straightforward.

- A pair by itself with no projection is considered a **value**.
- Aside from that, there really isn't that much to say about it... But can you think of how we can express the above as a derived form?

Ask Your Pair-ents!

It's a trick question!

- We already covered the derived forms of pairs and projection way back in topic 5!

$$\texttt{pair} = \lambda f : T_1.\lambda s : T_2.\lambda b : T_3.\ b\ f\ s \tag{2}$$

$$\texttt{fst} = \lambda p : T_1.\ p\ \texttt{tru} \tag{3}$$

$$\texttt{snd} = \lambda p : T_1.\ p\ \texttt{fls} \tag{4}$$

Ap-pair-ent Typings

How we go about typing pairs is slightly more interesting.

- We could make the decision to require the type of both arguments to be identical.
  - This would make a certain degree of sense, we might even be tempted to say that the type of the pair itself is the type of either component.
  - This sort of coercion would make our pairs something very different from the derived forms discussed above.
- A better way to think about it, however, is as so:
  - We have introduced a new *value*, $\{v_1, v_2\}$.
  - What sort of type could this new value be a canonical form of?

Mortal Pair-il

So, the question becomes, how do you represent the type of something that contains two types?

- We must **expand the type system!**

$$\langle T \rangle ::= \ldots \mid \langle T \rangle \times \langle T \rangle$$

This is known as the **product** or the **Cartesian Product** type constructor.

- If we consider the total number of possible combinations of the two types there are, it is the carnality of T squared.
- This informs the use of $\times$ as the type product operator.

Since we are adding this new type constructor, pairs **cannot** be a derived form, despite having clear and easy implementations in pure λ-Calculus

Typing Rules for Pairs

The following typing rules intuitively follow our evaluation semantics.

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2} \quad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.1 : T_1} \quad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.2 : T_2} \quad \text{(T-Proj2)}$$

Aside from the use of the new product type constructor, these typing rules are wholly unremarkable.

One thing to note is that we are maintaining our pattern of only having one typing rule per term of the calculus.

Tuples

Tip Top Tuples

Tuples are a straightforward generalization of pairs to data structures containing an arbitrary number of elements.

- Unfortunately, this means that we need to come up with a notation for uniformly describing systems of arbitrary arity.
- That is, we need a notation for a list of numbers.

The textbook uses the following notation for a tuple of n elements, which is more formal, but less readable.

$$t_i^{i \in 1..n} \tag{5}$$

For this course, we will opt for the less rigorous, but far more readable:

$$t_{1..n} \tag{6}$$

Syntax!

We will add the term on the previous slide to our calculus.

$\langle t \rangle = \ldots$
$\quad | \quad \{\langle t \rangle_{1..n}\}$
$\quad | \quad \langle t \rangle.(1..n)$

- Further, we will define tuples containing only values to be values themselves:

$\langle v \rangle :: \ldots$
$\quad | \quad \{\langle v \rangle_{1..n}\}$

- Finally, we will also add the following type constructor:

$\langle T \rangle :: \ldots$
$\quad | \quad \{\langle T \rangle_{1..n}\}$

- The above notation lets us avoid messy and long tuple type notations, such as $T_1 \times T_2 \times T_3 \times \ldots \times T_n$

Evaluation Rules

$$\frac{j \in 1..n}{\{v_{1..n}\}.j \to v_j} \quad \text{(E-ProjTuple)}$$

$$\frac{t \to t'}{t.i \to t'.i} \quad \text{(E-Proj)}$$

$$\frac{t_j \to t'_j}{\{v_{1..(j-1)}, t_j, t_{(j+1)..n}\} \to \{v_{1..(j-1)}, t'_j, t_{(j+1)..n}\}} \quad \text{(E-Tuple)}$$

I'm Not Projecting, You Are!

- The projection elimination and congruence rules are straightforward adaptations of E-PairBeta1,2, and E-Proj1,2.
- The congruence rule E-Tuple actually describes a **set** of inference rules.
- True to form, we control the execution of the elements of the tuple by requiring all the elements preceding any particular value to be evaluated to values before the congruence rule can be applied.
- That is, to evaluate any element of a tuple, all elements to the left must be evaluated first.

Tuping Typles

$$\frac{\forall i \in 1..n \mid \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_{1..n}\} : \{T_{1..n}\}} \quad \text{(T-Tuple)}$$

$$\frac{j \in 1..n \qquad \Gamma \vdash t : \{T_{1..n}\}}{\Gamma \vdash t.j : T_j} \quad \text{(T-Proj)}$$

Again, the above inference rules should be relatively straightforward.

- T-Tuple states that in order for a tuple to be well typed, each of it's elements must be well typed.
- T-Proj states the inverse. If a tuple is well typed, an element projected from it will also be well typed.

This is Going On Your PERMANENT RECORD

- We got to tuples from pairs by generalizing their size.
- We can get to records from tuples by generalizing their means of access.

Whereas a tuple uses the natural numbers as a means of accessing it's members, we can generalize this to some element f of a set of **field names** $\mathcal{F}$.

- The textbook uses $l \in \mathcal{L}$ for "the set of label names."
- We will use $f \in \mathcal{F}$ because:
  - Field names and labels are synonymous in this context.
  - To avoid confusion with memory locations.

The step from tuples to records in our calculus is roughly the same as the step from tuples to dictionaries in Python.

Record Syntax

Syntax for records is highly similar to that of tuples.

$$\begin{aligned}\langle t \rangle ::= & \ \ldots \\ | & \ \{\langle f \rangle_{1..n} = \langle t \rangle_{1..n}\} \\ | & \ \langle t \rangle.\langle f \rangle\end{aligned}$$

Similarly to its cousins, a record containing only values is itself a value.

$$\begin{aligned}\langle v \rangle ::= & \ \ldots \\ | & \ \{\langle f \rangle_{1..n} = \langle v \rangle_{1..n}\}\end{aligned}$$

And field names will also be necessary to look up an element type in a record type, so we include the field name.

$$\begin{aligned}\langle T \rangle ::= & \ \ldots \\ | & \ \{\langle f \rangle_{1..n} : \langle T \rangle_{1..n}\}\end{aligned}$$

Evaluation Rules

All three of our data structures so far have used curly braces to emphasize that they are very straight forward extensions of each other.

- The following evaluation rules are straightforward adaptations of those for tuples.

$$\frac{j \in 1..n}{\{f_{1..n} = v_{1..n}\}.f_j \rightarrow v_j} \quad \text{(E-ProjRcd)}$$

$$\frac{t \rightarrow t_1'}{t_1.f \rightarrow t_1'.f} \quad \text{(E-Proj)}$$

$$\frac{i \in 1..(j-1) \qquad k \in (j+1)..n \qquad t_j \rightarrow t_j'}{\{f_i = v_i, f_j = t_j, f_k = t_k\} \rightarrow \{f_i = v_i, f_j = t_j', f_k = t_k\}} \quad \text{(E-Rcd)}$$

Record Scratch

The evaluation rule E-ProjRcd relies on a slightly informal convention, which we should make explicit.

- If $\{f_{1..n} = v_{1..n}\}$ is a record and f_j is the label of its j^{th} field, then $\{\text{f\_1..n} = \text{v\_1..n.f\_j }\}$ evaluates in one step to the j^{th} value, v_j.
- Both this rule, and E-ProjTuple could be reformulated to make this behaviour more explicit.
- However, this would come at great cost to readability, so we will keep the rule in the form listed.

Ordering Records On The Internet!

In some sense, the definition provided above fails to fully abstract tuple-style numeric element access into record-style field access.

- Consider the congruence rule E-Rcd.
  - Following the python example of dictionaries, it would be reasonable to expect something like E-Rcd to be impossible.
  - The numeric sequencing of field names is relied upon to guarantee the determinacy of the evaluation of a record.
  - If the set of field names was simply a pure set, no such ordering would be implied.
  - The way we've defined them, we are closer to python's **ordered dictionary**, than it's default dictionary.
- Whether the language assumes unordered records are primitive, or uses primitive ordered records and evaluation rules which allow reordering, is a design decision that may have performance implications.

Accidentally Hit Record!

The typing rules for records are about what you would expect.

$$\frac{\forall i \in 1..n \mid \Gamma \vdash t_i : T_i}{\Gamma \vdash \{f_{1..n} = t_{1..n}\} : \{f_{1..n} : T_{1..n}\}} \quad \text{(T-Rcd)}$$

$$\frac{j \in 1..n \qquad \Gamma \vdash t : \{f_{1..n} : T_{1..n}\}}{\Gamma \vdash t.f_j : T_j} \quad \text{(T-Proj)}$$

- So, for example, the following record which indicates the degree to which science fiction franchises have been ruined by Hollywood since the financial crash of 2008:
  - `{startrek=95.6, starwars=99.9, foundationseries=false}`
- Would have type
  - `{Float, Float, Bool}`

Game Set Match!

Let's take a brief diversion from our study of data structures, and examine something we can do with them: **pattern matching!**

- In Haskell, pattern matching is ubiquitous, and the delight of many a new Haskell programmer.
- In Haskell, complex data types can be broken down into labelled subcomponents via pattern matching. For example:

```
snd :: (a, b) -> b
snd (x, y) = y
```

- With our current toolkit, we can create the rudiments of a pattern matching system.

I'm Detecting a Pattern...

Our pattern matching operation will be a generalization of **let bindings**.

$\langle t \rangle ::= \ldots$
$\quad | \quad \text{let } \langle p \rangle = \langle t \rangle \text{ in } \langle t \rangle$

In order to accomplish this, we need to open up a new syntactic category for patterns.

$\langle p \rangle ::= \langle x \rangle$
$\quad | \quad \{ \langle f \rangle_{1..n} = \langle p \rangle_{1..n} \}$

We'll be pattern matching over records/tuples/pairs, so we require a pattern which matches the data structure we are trying to match.

- Hence, a pattern may be either a lone variable, or a record containing only variable names or other patterns.

We Have A Match Captain!

Let's take a look at the evaluation rules for pattern matching.

$$\texttt{let}\ \ p = v\ \ \texttt{in}\ \ t \to \mathit{match}(p, v)t \qquad \text{(E-LetV)}$$

$$\frac{t \to t'}{\texttt{let}\ \ p = t\ \ \texttt{in}\ \ t \to \texttt{let}\ \ p = t'\ \ \texttt{in}\ \ t} \qquad \text{(E-Let)}$$

The congruence rule is nothing special, but what's with the mysterious *match* function?

- In Haskell:
  - Inputs get applied to the pattern, and components of the input are identified with various variable names.
  - Then, occurrences of those variable names in the return expression are **substituted for** the input components.
  - Whatever this *match* function is, we know we want it to **create substitutions**.

Match 3 is Lazy Game Design

In this course, whenever we need to define the behaviour of a function or relation, we always specify it as "the smallest function or relation satisfying a set of inference rules."

- We shall define the behaviour of *match* using the same mechanism.

$$match(x, v) = [x \mapsto v] \qquad \text{(M-Var)}$$

$$\frac{\forall i \in 1..n \mid match(p_{1..n}, v_{i..n}) = \sigma_{1..n}}{match(\{f_{1..n} = p_{1..n}\}, \{f_{1..n} = v_{1..n}\}) = \sigma_1 \circ \sigma_2 \circ ... \circ \sigma_n} \qquad \text{(M-Rcd)}$$

Where $\circ$ is function composition.

Match My Moves!

Let's go through a few examples.

$$\texttt{let } \{x, y, z\} = \{2, 4, 6\} \texttt{ in } x \cdot z - y$$

$$\xrightarrow{\texttt{E-LetV}} match(\{x, y, z\}, \{2, 4, 6\})(x \cdot z - y)$$

$$\xrightarrow{\texttt{M-Rcd}} match(x, 2) \circ match(y, 4) \circ match(z, 6)(x \cdot z - y)$$

$$\xrightarrow{\texttt{M-Var}} \xrightarrow{\texttt{M-Var}} \xrightarrow{\texttt{M-Var}} [x \mapsto 2][y \mapsto 4][z \mapsto 6](xz - y)$$

$$\xrightarrow{\texttt{Subst}} \xrightarrow{\texttt{Subst}} \xrightarrow{\texttt{Subst}} 2 \cdot 6 - 4$$

$$\xrightarrow{\texttt{Arith}} \xrightarrow{\texttt{Arith}} 8$$

$$\not\rightarrow$$

Mix and Match

$$\texttt{let } \{x, y\} = \{2, \{4, 6\}\} \texttt{ in } ((\lambda t.\lambda f.f)\, x\, y)$$

$$\xrightarrow{\texttt{E-LetV}} match(\{x, y\}, \{2, \{4, 6\}\})((\lambda t.\lambda f.f)\, x\, y)$$

$$\xrightarrow{\texttt{M-Rcd}} match(x, 2) \circ match(y, \{4, 6\})((\lambda t.\lambda f.f)\, x\, y)$$

$$\xrightarrow{\texttt{M-Var}} \xrightarrow{\texttt{M-Var}} [x \mapsto 2][y \mapsto \{4, 6\}]((\lambda t.\lambda f.f)\, x\, y)$$

$$\xrightarrow{\texttt{Subst}} \xrightarrow{\texttt{Subst}} (\lambda t.\lambda f.f)\, 2\, \{4, 6\}$$

$$\xrightarrow{\texttt{E-AppAbs}} (\lambda f.f)\, \{4, 6\}$$

$$\xrightarrow{\texttt{E-AppAbs}} \{4, 6\}$$

$$\not\twoheadrightarrow$$

Typing and soundness for pattern matching will be the subject of next week's tutorial.

Listing Our Complaints.

The final data structure we'll cover is the **list**.

- Unlike tuples, which are heterogeneously typed, lists are homogeneously typed.
- The list semantics we are about to discuss are precisely those used by Haskell.

A list is a finite collection of elements, which are uniformly typed.
We will discuss syntax, typing and semantics for the following:

Term	Haskell Syntax
empty list	`[]`
list constructor	`x:xs`
test for empty list	`null xs`
head of a list	`head xs`
tail of a list	`tail xs`

Landing on Luxury Syntax

Like λ abstractions, each of our list terms will require **type annotation**.

$$
\begin{array}{lll}
\langle t \rangle & :: & \ldots \\
& | & \mathsf{nil}[\langle T \rangle] \\
& | & \mathsf{cons}[\langle T \rangle]\ \langle t \rangle\ \langle t \rangle \\
& | & \mathsf{isnil}[\langle T \rangle]\ \langle t \rangle \\
& | & \mathsf{head}[\langle T \rangle]\ \langle t \rangle \\
& | & \mathsf{tail}[\langle T \rangle]\ \langle t \rangle
\end{array}
$$

Both empty lists and lists containing only values will be values themselves.

$$
\begin{array}{lll}
\langle v \rangle & ::= & \ldots \\
& | & \mathsf{nil}[\langle T \rangle] \\
& | & \mathsf{cons}[\langle T \rangle]\ \langle v \rangle\ \langle v \rangle
\end{array}
$$

Evaluating Cons

- In the same way that `true` has no evaluation rules, `nil` has no evaluation rules.
- Since `cons` doesn't really do anything except exist, we only have two congruence rules for it:

$$\frac{t_1 \to t_1'}{cons[T]\ t_1\ t_2 \to cons[T]\ t_1'\ t_2} \quad \text{(E-Cons1)}$$

$$\frac{t_2 \to t_2'}{cons[T]\ v_1\ t_2 \to cons[T]\ v_1\ t_2'} \quad \text{(E-Cons2)}$$

Evaluating `isnil`

The evaluation rules for `isnil` are a bit more interesting, and should remind us strongly of our old friend isZero.

$$isnil[S](nil[T]) \rightarrow true \qquad \text{(E-IsNilNil)}$$

$$isnil[S](cons[T]\, v_1\, v_2) \rightarrow false \qquad \text{(E-IsNilCons)}$$

$$\frac{t_1 \rightarrow t_1'}{isnil[T]\, t_1 \rightarrow isnil[T]\, t_1'} \qquad \text{(E-IsNil)}$$

Evaluating Head and Tail

Evaluation Rules for `head` and `tail` should give us no surprises. You just pop either the first or second component of cons respectively.

$$head[S](cons[T]\,v_1\,v_2) \rightarrow v_1 \qquad \text{(E-HeadCons)}$$

$$tail[S](cons[T]\,v_1\,v_2) \rightarrow v_2 \qquad \text{(E-TailCons)}$$

$$\frac{t \rightarrow t'}{head[T]\,t \rightarrow head[T]\,t'} \qquad \text{(E-Head)}$$

$$\frac{t \rightarrow t'}{tail[T]\,t \rightarrow tail[T]\,t'} \qquad \text{(E-Tail)}$$

Typing Lists

Before we begin our typing rules for lists, we must update our set of types.

- Just as with references, it makes sense to be able to differentiate a list of Booleans from a single Boolean at the level of the type system.
- Therefore, we expand our types as follows:

$$\begin{array}{rcl} \langle T \rangle & ::= & \ldots \\ & | & \text{List } \langle T \rangle \end{array}$$

Typing Rules

$$\Gamma \vdash nil[T] : List\,T \qquad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : List\,T}{\Gamma \vdash cons[T]\ t_1\ t_2 : List\,T} \qquad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash t : List\,T}{\Gamma \vdash isnil[T]t : Bool} \qquad \text{(T-IsNil)}$$

$$\frac{\Gamma \vdash t : List\,T}{\Gamma \vdash head[T]t : T} \qquad \text{(T-Head)}$$

$$\frac{\Gamma \vdash t : List\,T}{\Gamma \vdash tail[T]t : List\,T} \qquad \text{(T-Tail)}$$

Only the usual subtleties apply.

- The main utility provided here is the prevention of elements of incompatible types to a list.

Last Slide Comic