Recursion on Inductive Data Types

Deriving Recursion Patterns From Inductive Definitions
Lists as an Inductive Data Type

- \( \text{List} = \begin{cases} \text{nil} \\ \text{cons Element List} \end{cases} \)

Construction forms

- \( \text{nil} \)
- \( \text{(cons head tail)} \)

Elimination equations

- \( \text{(null nil)} \rightarrow t \)
- \( \text{(null (cons head tail))} \rightarrow \text{nil} \)
- \( \text{(first (cons head tail))} \rightarrow \text{head} \)
- \( \text{(rest (cons head tail))} \rightarrow \text{tail} \)
Induction and Recursion

- *Induction* and *recursion* are opposite perspectives of the same dual phenomenon.
- For each inductively defined data type there is an inferred pattern of recursion.

  » The (mathematical) theory underlying this approach is the *Martin-Löf Constructive Type Theory*. Its main applications lie in *automated programming, automated verification, type-system design*.

Induction and Recursion

• **Inductive definition (bottom-up)**
  
  – defining type **List**
  
    • **nil** is a **List**
    
    • if **head** is an **Element** and **tail** is a **List**, then (cons **head** **tail**) is a **List** (induction case)

• **Recursive pattern (top-down)**

  – define **process** **lst**\textsuperscript{List}

    • handle **lst** is **nil** (base case)
    
    • handle **lst** is (cons **head** **tail**)\textsuperscript{List} (recursive case)

      – [if necessary] call **process** on **tail**
Recursive Pattern for Lists

\[
\text{List} = \begin{cases} 
\text{nil} \\ 
\text{cons Element List} 
\end{cases}
\]

(\text{defun function-name (lst)}

\begin{align*}
\text{(cond} \\
& ((\text{null lst}) \ldots \text{base-case-processing}\ldots) \\
& \text{(otherwise} \ldots \text{induction-case-processing}\ldots \\
& \quad \text{if needed, call (function-name (rest lst))} \\
& \quad ) \\
& ) \\
& )
\end{align*}
Recursive Pattern for Lists (2)

(defun function-name (list)
  (if list
    (if needed, call (function-name (rest list))
    ...base-case-processing...
  )
  ...
)

List =
  | nil
  | cons Element List
(defun alt-sum (lst)
  (cond
   ((null lst) 0)
   (otherwise (+ (first lst) (alt-sum (rest lst))))
  ))
)
Copy

(defun alt-copy (lst)
  (if lst
      (cons (first lst) (alt-copy (rest lst)))
      ))
(defun alt-copy (lst)
  (if lst
      (cons (first lst) (alt-copy (rest lst)))
    ))

 Processing after recursion
Reverse

(defun alt-reverse (lst &optional acc)
  (if lst
      (alt-reverse (rest lst) (cons (first lst) acc))
      acc)
  )
  )
Reverse

(\textsf{defun} \textsf{alt-reverse} (\textit{lst} \textit{&optional} \textit{acc})
  (if \textit{lst}
    (\textsf{alt-reverse} (rest \textit{lst}) (cons (first \textit{lst}) \textit{acc}))
    \textit{acc}
  ))

\textit{lst}
\begin{tikzpicture}[node distance=1cm,auto]
  \node (a) {a};
  \node (b) [right of=a] {b};
  \node (c) [right of=b] {c};
  \node (d) [right of=c] {d};
  \node (e) [right of=d] {e};
  \draw[->] (a) -- (b);
  \draw[->] (b) -- (c);
  \draw[->] (c) -- (d);
  \draw[->] (d) -- (e);
\end{tikzpicture}

\textit{intermediary-list}
\begin{tikzpicture}[node distance=1cm,auto]
  \node (b) [right of=a] {b};
  \node (c) [right of=b] {c};
  \node (d) [right of=c] {d};
  \node (e) [right of=d] {e};
  \draw[->] (b) -- (c);
  \draw[->] (c) -- (d);
  \draw[->] (d) -- (e);
\end{tikzpicture}

\textit{acc} when calling \textsf{\textit{alt-reverse} \textit{intermediary-list}}

\textit{acc} when calling \textsf{\textit{alt-reverse} \textit{lst}}

Processing before recursion \textit{(tail-recursion)}
Nested Lists as an Inductive Data Type

- **NestedList** =
  - nil
  - atom (other than nil)
  - cons NestedList NestedList

<table>
<thead>
<tr>
<th>Construction forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
</tr>
<tr>
<td>(cons left right)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Elimination equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(null nil) → t</td>
</tr>
<tr>
<td>(null (cons left right)) → nil</td>
</tr>
<tr>
<td>(consp nil) → nil</td>
</tr>
<tr>
<td>(consp atom) → nil</td>
</tr>
<tr>
<td>(consp (cons left right)) → t</td>
</tr>
<tr>
<td>(first (cons left right)) → left</td>
</tr>
<tr>
<td>(rest (cons left right)) → right</td>
</tr>
</tbody>
</table>
Recursive Pattern for Nested Lists

```
(defun function-name (nested-list)
  (cond
    ((null nested-list) ...nil-base-case-processing...)
    ((consp nested-list) ...induction-case-processing...
      if needed, call (function-name (first nested-list))
      (function-name (rest nested-list))
    )
    (otherwise ...atom-base-case-processing...)
  )
)
```
Recursive Pattern for Nested Lists (2)

```
(defun function-name (nested-list)
  (if (consp nested-list)
      ;; induction-case-processing...
      ;; if needed, call (function-name (first nested-list))
      ;; (function-name (rest nested-list))
      ;; base-cases-processing...(nil, atom)
      )
)
```

\[\text{NestedList} =
\begin{align*}
\mid & \text{nil} \mid \text{atom} \\
\mid & \text{cons NestedList NestedList}
\end{align*}\]
Count Elements of Nested Lists

(defun alt-count-nodes (nested-list)
  (cond ((null nested-list) 0)
        ((consp nested-list)
         (+ (alt-count-nodes (first nested-list))
            (alt-count-nodes (rest nested-list)))
        (otherwise 1)))

NestedList =
  | nil | atom
  | cons NestedList NestedList
NestedList =
         | nil | atom
         | cons NestedList NestedList

Copy Nested List

(defun alt-copy-nested-list (nested-list)
  (if (consp nested-list)
      (cons (alt-copy-nested-list (first nested-list))
            (alt-copy-nested-list (rest nested-list))
        nested-list)
  )
)
Trees as an Inductive Data Type

- \textbf{Tree} = \\
  \begin{itemize}
  \item \text{nil}
  \item \text{list Tree Element Tree}
\end{itemize}

\textbf{Type definition}

\textbf{Construction forms}

\textbf{Elimination equations}

- nil
- (list \text{left elm right})

  - (null nil) → t
  - (null (list \text{left elm right})) → nil
  - (first (list \text{left elm right})) → left
  - (second (list \text{left elm right})) → elm
  - (third (list \text{left elm right})) → right
Recursive Pattern for Trees

(defun function-name (tree)
  (cond
    ((null tree) ...base-case-processing...)
    (otherwise ...induction-case-processing...
      if needed, call (function-name (first tree))
                    (function-name (third tree)))
  ))
)
Tree =

| nil            |
| list Tree Element Tree |

Recursive Pattern for Trees (2)

(defun function-name (tree)
  (if tree
      ...
      \arrow \text{induction-case-processing}...
      \downarrow \text{if needed, call (function-name (first tree))}
      \downarrow \text{(function-name (third tree))}
      ...base-case-processing...
      )
  )
)
Tree =
  | nil
  | list Tree Element Tree

Count Elements of Tree

(defun alt-count-nodes (tree)
  (if tree
    (+ (alt-count-nodes (first tree)) 1
        (alt-count-nodes (third tree)))
  0))
Copy Tree

(defun alt-copy-tree (tree)
  (if tree
    (list (alt-copy-tree (first tree))
          (second tree)
          (alt-copy-tree (third tree)))
    )
  )
)