Construction Techniques

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
constantinos.constantinides@concordia.ca

January 18, 2022

What to construct?

- Unordered and ordered structures.
- Procedures (mathematical operations) and functions (software elements). Note: We will use the two terms interchangeably.
- Grammars.

Constructing functions that use lists

- In problem solving, the deployment of *recursion* implies that the solution to a problem depends on solutions to smaller instances of the same problem.
- Every recursive function consists of:
    - One or more *base cases*, and
    - One or more *recursive cases*.

From specification to implementation

- Each recursive case consists of:
- Splitting the data into smaller pieces (for example, with `head` and `tail`),
- Handling the pieces with calls to the current method (note that every possible chain of recursive calls must eventually reach a base case), and
- Combining the results into a single result.

Initial example: Obtaining a specification

- Suppose we need to define the function $f : \mathbb{N} \rightarrow \mathit{lists}(\mathbb{N})$ that accepts an integer argument and returns a list, such that

$$f(n) = \langle n, n-1, ..., 0 \rangle$$

Initial example: Constructing a computable function

- We transform the definition of $f(n)$ into a computable function using available operations on the underlying structure (list).
- We can use *cons* as follows:

$$\begin{aligned} f(n) &= \langle n, n-1, ..., 1, 0\rangle \\ &= cons(n, \langle n-1, ..., 1, 0\rangle) \\ &= cons(n, f(n-1)). \end{aligned}$$

- We can therefore define $f(n)$ recursively by

$$\begin{aligned} &f(0) = \langle 0\rangle. \\ &f(n) = cons(n, f(n-1)),\ \ for\ \ n > 0. \end{aligned}$$

Initial example: Unfolding the definition

- We can visually show how this works with a technique called "unfolding the definition" (or "tracing the algorithm").
- We can unfold this definition for $f(3)$ as follows:

$$
\begin{aligned}
f(3) &= cons(3, f(2)) \\
&= cons(3, cons(2, f(1))) \\
&= cons(3, cons(2, cons(1, f(0)))) \\
&= cons(3, cons(2, cons(1, \langle 0 \rangle))) \\
&= cons(3, cons(2, \langle 1, 0 \rangle)) \\
&= cons(3, \langle 2, 1, 0 \rangle) \\
&= \langle 3, 2, 1, 0 \rangle.
\end{aligned}
$$

Initial example: Transforming the algorithm into an implementation

- We can now build function `f` as follows:

Functional programming (Common Lisp)

```
(defun f (n)
  (if (= n 0)
    (cons 0 '())
    (cons n (f(- n 1)))))
```

Imperative programming (Pseudocode)

```
f(n) is
  if (n = 0) then display(0)
  else
     display(n)
     f(n - 1)
```

Initial example: Tracing the execution of the function

- We now can trace the execution of the Common Lisp function with sample input data, e.g.

```
f(2)  = (cons 2, f(1))
      = (cons 2, (cons 1, f(0)))
      = (cons 2, (cons 1, (cons 0, ())))
      = (cons 2, (cons 1, (0)))
      = (cons 2, (1 0))
      = (2 1 0).
```

Initial example: Executing the implementation

- We can execute the function as follows:

```
> (f 0)
(0)
> (f 3)
(3 2 1 0)
```

Putting everything together

- Obtain the specification.
- Construct a computable function.
- Unfold the definition (“trace the algorithm”).
- Transform the algorithm into implementation.
- Trace the execution of the implementation.
- Execute the implementation.

Second example: Obtaining a specification

- Consider function `dist` that accepts an atomic element n and a non-empty list Λ, and returns a list composed of ordered pairs, where in each pair the first coordinate is n and the second coordinate is each successive element of Λ.
- For example,

$$dist(a, \langle b, c, d\rangle) = \langle\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle\rangle$$

Second example: Constructing a computable function

- To detect recursion, we can re-write the equation by splitting up the list into its head and its tail:

$$\begin{aligned} dist(a, \langle b, c, d\rangle) &= \langle\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle\rangle \\ &= \langle\langle a, b\rangle, dist(a, \langle c, d\rangle)\rangle. \end{aligned}$$

  We can therefore provide the following computable function definition:

$$\begin{aligned} dist(x, \langle\rangle) &= \langle\rangle, \\ dist(x, \Lambda) &= cons(\langle x, head(\Lambda)\rangle, dist(x,\ tail(\Lambda))). \end{aligned}$$

Second example: Unfolding the definition

- We can now unfold the definition for $\mathit{dist}(w, \langle x, y \rangle)$ as follows:

$$\begin{aligned}
\mathit{dist}(w, \langle x, y \rangle) &= \mathit{cons}(\langle w, x \rangle, \mathit{dist}(w, \langle y \rangle)) \\
&= \mathit{cons}(\langle w, x \rangle, \mathit{cons}(\langle w, y \rangle, \mathit{dist}(w, \langle \rangle))) \\
&= \langle \langle w, x \rangle, \langle w, y \rangle \rangle.
\end{aligned}$$

Second example: Transforming the algorithm into an implementation

- The function is defined as follows:

```
(defun dist (n lst)
  (if (null lst)
      nil
      (cons (list n (car lst)) (dist n (cdr lst) ))))
```

Second example: Tracing the execution of the implementation

- We can trace the execution of function `dist` as follows:

```
(dist 'a '(b c d))
  = (cons (list 'a 'a) dist ('a '(b c)))
  = (cons '(a a) (cons ((list 'a 'b) dist ('a '(c)))))
  = (cons '(a a) (cons '(a b) (cons (list 'a 'c) '())))))
  = (cons '(a a) (cons '(a b) (cons '(a c) '())))
  = '((a a), (a b) (a c))
```

Function `consR`: Obtaining the specification

- Consider function $consR(\Lambda, el)$ which constructs a list by placing an element to the right of a list which is provided as the first argument.
- For example,

$$consR(\langle a, b, c \rangle, d) = \langle a, b, c, d \rangle.$$

Function `consR`: Constructing a computable function

- We can provide a recursive computable function definition as follows:

$$
\begin{aligned}
consR(\Lambda, el) = & \ if \ \Lambda \ = \ \langle\rangle \ then \ \langle el \rangle \\
& \ else \ \ cons(head(\Lambda), consR(tail(\Lambda), el)).
\end{aligned}
$$

Function `consR`: Unfolding the definition

- We can now unfold the definition for $consR(\langle a, b\rangle, c)$ as follows:

$$\begin{aligned}
consR(\langle a, b\rangle, c)) &= cons(a, (consR(\langle b\rangle, c))) \\
&= cons(a, (cons(b, consR(\langle\rangle, c)))) \\
&= cons(a, (cons(b, \langle c\rangle))) \\
&= \langle a, b, c\rangle.
\end{aligned}$$

Function consR: Transforming the algorithm into an implementation

- The function is defined as follows:

```
(defun consr (lst elt)
    (if (null lst) (list elt)
       (cons (car lst) (consr (cdr lst) elt))))
```

Guidelines for constructing functions

- Unless the function is trivial, we can break the logic into cases (with single or multiple selection statements).
- When handling lists, we would normally adopt a recursive solution. Treat the empty list as a base case.
- Normally we would operate on the head of a list (accessible with *head*) and recur on the tail of the list (accessible with *tail*).
- To skip the head of the list, simply recur on the tail of the list.
- To keep the head of the list as is, use *cons* to place it as the head of the newly constructed (returning) list (whose tail is determined by the recursive call).