Flat Domains and Recursive Equations in ACL2

by

John Cowles
University of Wyoming

ACL2 is a logic of *total* functions.

- Some recursive equations have no satisfying ACL2 functions:

  No ACL2 function `g` satisfies this *recursive* equation

```
(equal (g x)
       (if (equal x 0)
           nil
           (cons nil (g (- x 1))))).
```

Theory of *flat domains* is a rival logic of *total* functions.

- Every recursive equation has at least one satisfying function.

Flat Domains

From the *fix-point theory* of program semantics.

A *flat domain* is a structure

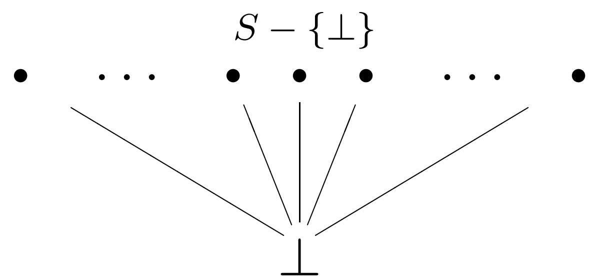
$$< S, \sqsubseteq, \perp >$$

, where

- S is a set,
- $\perp \in S$, and
- $\sqsubseteq$ is the partial order defined by

$$x \sqsubseteq y \Longleftrightarrow x = \perp \vee x = y.$$

Graphical representation of a flat domain:



- Graphical representation of the $\sqsubset$ relation defined by

$$x \sqsubset y \Longleftrightarrow x \sqsubseteq y \wedge x \neq y.$$

- The "flat part" is depicted by the vertices labeled with $S - \{\perp\}$.

Extend the partial order, $\sqsubseteq$, *componentwise* to

- tuples from $S \times S \times \cdots \times S$ by

$$< x_1, \ldots, x_n > \sqsubseteq < y_1, \ldots, y_n >$$
$$\iff \; x_1 \sqsubseteq y_1 \wedge \cdots \wedge x_n \sqsubseteq y_n$$

- functions $f, g : S \times \cdots \times S \to S$ by

$$f \sqsubseteq g \iff (\forall \vec{x} \in S^n)[f(\vec{x}) \sqsubseteq g(\vec{x})]$$

Flat Domains

Use *total functions* to model *partial functions*.

- Interpret

  $$f(\vec{x}) = \bot$$

  as meaning

  $f(\vec{x})$ is **undefined**.

- Interpret, for functions f and g,

  $$f \sqsubseteq g$$

  as meaning

  whenever $f(\vec{x})$ is defined,

  - $g(\vec{x})$ is also defined, and
  - $f(\vec{x}) = g(\vec{x})$.

Least Upper Bounds of Chains

Every chain of functions on S,

$$f_0 \sqsubseteq f_1 \sqsubseteq \cdots \sqsubseteq f_i \sqsubseteq \cdots,$$

has an unique *least upper bound*, $\sqcup f_i$.

- $\sqcup f_i$ is a function on S,
- for all j, $f_j \sqsubseteq \sqcup f_i$ and
- if f is any function such that for all i, $f_i \sqsubseteq f$, then $\sqcup f_i \sqsubseteq f$,
- define $\sqcup f_i(\vec{x})$ by cases:

  Case 1. $\forall i(f_i(\vec{x}) = \perp)$.
  Let $\sqcup f_i(\vec{x}) = \perp$.

  Case 2. $\exists j(f_j(\vec{x}) \neq \perp)$.
  Let $\sqcup f_i(\vec{x}) = f_j(\vec{x})$.

Flat Domains

Recursive Equations

Let F be a function variable and

let $\tau[F]$ be a term built by compositions involving F and other functions.

A *recursive equation* is of the form

$$F(\vec{x}) = \tau[F](\vec{x}).$$

A solution for such an equation is a function f such that for all $\vec{x}$,

$$f(\vec{x}) = \tau[f](\vec{x}).$$

Such a solution f is called a *fixed point* of the term $\tau[F](\vec{x})$.

Flat Domains

The Kleene Construction

A term $\tau[F]$ is *monotonic*:

- Whenever f and g are functions such that $f \sqsubseteq g$, then $\tau[f] \sqsubseteq \tau[g]$.

Kleene's construction:

- When $\tau[F]$ is monotonic,

  $$F(\vec{x}) = \tau[F](\vec{x})$$

  always has a solution.

Flat Domains
The Kleene Construction

Kleene's construction:

- Use the term $\tau[F]$ to recursively define a chain of functions,

$$\begin{aligned} f_0(\vec{x}) &= \bot \\ f_{i+1}(\vec{x}) &= \tau[f_i](\vec{x}). \end{aligned}$$

- Since $\tau[F]$ is monotonic,

$$f_0 \sqsubseteq f_1 \sqsubseteq \cdots \sqsubseteq f_i \sqsubseteq \cdots$$

- Then,

$$\sqcup f_i = \tau[\sqcup f_i].$$

  That is, $\sqcup f_i$ is a solution for the recursive equation $F(\vec{x}) = \tau[F](\vec{x})$.

Turn ACL2 data into a flat domain

Impose a partial order, `$<=$`, on ACL2 data:

- specify a "least element", `($bottom$)`, *strictly* less than any other ACL2 datum

```
(defstub
    $bottom$ () => *)
```

- no other *distinct* data items are related:

```
(defun
   $<=$ (x y)
   (or (equal x ($bottom$))
       (equal x y)))
```

- `($bottom$)` plays the part of $\bot$ and `$<=$` plays the part of $\sqsubseteq$.

Chains of functions in ACL2

Formalize a chain of functions

$$f_0 \sqsubseteq f_1 \sqsubseteq \cdots \sqsubseteq f_i \sqsubseteq \cdots.$$

- Treat the index as an additional argument to the function, so $f_i(x)$ becomes `(f i x)` in ACL2.

- The `$<=$`-chain of functions is consistently axiomatized by

```
(implies (and (integerp i)
              (>= i 0))
         ($<=$ (f i x)
               (f (+ 1 i) x))).
```

Chains of functions in ACL2

Formalize the least upper bound, $\sqcup f_i$, of

$$f_0 \sqsubseteq f_1 \sqsubseteq \cdots \sqsubseteq f_i \sqsubseteq \cdots.$$

- Use `defchoose` to pick the appropriate "index" required in the definition of the least upper bound.

- ACL2 verifies this *formal* least upper bound is, in fact, the *least upper bound* of the chain.

Which ACL2 terms are monotonic?

Recall:

To ensure that Kleene's construction always produces

- a solution for the recursive equation

$$F(\vec{x}) = \tau[F](\vec{x}),$$

- the term $\tau[F]$ must be monotonic:

$$f \sqsubseteq g \Rightarrow \tau[f] \sqsubseteq \tau[g].$$

Which ACL2 terms are monotonic?

Tail Recursion. Let `test`, `base`, and `st` be arbitrary unary functions.

Consider a term $\tau[\mathrm{F}]$ of the form

```
(if (test x)
    (base x)
    (F (st x)))).
```

Such *tail recursive terms are always monotonic*.

- This means that tail recursive equations always have solutions.

- Another explanation for Pete & J's result that any tail recursive equation is satisfiable by some ACL2 function.

Such *tail recursive terms are always monotonic*:

Let `f` and `g` be functions such that `($<=$ (f x)(g x))`, [i.e., $\mathrm{f} \sqsubseteq \mathrm{g}$].

Case 1. `(test x)` is **not** `NIL`.

$\tau[\mathrm{f}](\mathrm{x}) = (\mathrm{base\ x}) = \tau[\mathrm{g}](\mathrm{x})$.

So $\tau[\mathrm{f}] \sqsubseteq \tau[\mathrm{g}]$.

Case 2. `(test x)` is `NIL`

Since $\forall y[(\mathrm{f}\ y) \sqsubseteq (\mathrm{g}\ y)]$,

$$\begin{aligned} \tau[\mathrm{f}](\mathrm{x}) &= (\mathrm{f\ (st\ x)}) \\ &\sqsubseteq (\mathrm{g\ (st\ x)}) \\ &= \tau[\mathrm{g}](\mathrm{x}). \end{aligned}$$

Thus $\tau[\mathrm{f}] \sqsubseteq \tau[\mathrm{g}]$.

Which ACL2 terms are monotonic?

Primitive Recursion. Let `test`, `base`, and `st` be arbitrary unary functions.

Let `h` be a binary function.

Consider a term τ[F] of the form

```
(if (test x)
    (base x)
    (h x (F (st x)))))
```

Often such terms are **not** monotonic.

Such terms **are** monotonic
if `h` *always preserves* $\sqsubseteq$ in its second input:

$$y_1 \sqsubseteq y_2 \Rightarrow (\texttt{h x } y_1) \sqsubseteq (\texttt{h x } y_2)$$

Such primitive recursive terms **are** monotonic if `h` *always preserves* $\sqsubseteq$ in its second input:

Let `f` and `g` be functions such that
`($<=$ (f x)(g x))`, [i.e., $\mathtt{f} \sqsubseteq \mathtt{g}$].

Case 1. `(test x)` is **not** `NIL`.
$\tau[\mathtt{f}](\mathtt{x}) = (\mathtt{base\ x}) = \tau[\mathtt{g}](\mathtt{x})$.
So $\tau[\mathtt{f}] \sqsubseteq \tau[\mathtt{g}]$.

Case 2. `(test x)` is `NIL`
Since $\forall y[(\mathtt{f}\ y) \sqsubseteq (\mathtt{g}\ y)]$,

$$(\mathtt{f\ (st\ x)}) \sqsubseteq (\mathtt{g\ (st\ x)}).$$

Since `h` *always preserves* $\sqsubseteq$ in its second input,

$$\begin{aligned} \tau[\mathtt{f}](\mathtt{x}) &= (\mathtt{h\ x\ (f\ (st\ x))}) \\ &\sqsubseteq (\mathtt{h\ x\ (g\ (st\ x))}) \\ &= \tau[\mathtt{g}](\mathtt{x}). \end{aligned}$$

Thus $\tau[\mathtt{f}] \sqsubseteq \tau[\mathtt{g}]$.

Such primitive recursive terms **are** monotonic if `h` *always preserves* $\sqsubseteq$ in its second input:

$$\mathtt{y}_1 \sqsubseteq \mathtt{y}_2 \Rightarrow (\mathtt{h\ x\ y}_1) \sqsubseteq (\mathtt{h\ x\ y}_2)$$

From *Consistently Adding Primitive Recursive Definitions in ACL2*,

```
(equal (F x)
       (if (test x)
           (base x)
           (h x (F (st x))))).
```

A sufficient (but not necessary) condition on `h` for the existence of `F` is that `h` have a right fixed point.

That is, there is some `c` such that
`(h x c) = c`.

Restate in the terminology of flat domains:

A sufficient (but not necessary) condition on `h` for a primitive recursive term, $\tau[\mathrm{F}]$, to be monotonic is that `h` have a right fixed point.

Use: Such primitive recursive terms **are** monotonic
if $\mathtt{h}$ *always preserves* $\sqsubseteq$ in its second input:

$$\mathrm{y}_1 \sqsubseteq \mathrm{y}_2 \Rightarrow (\mathtt{h}\ \mathtt{x}\ \mathrm{y}_1) \sqsubseteq (\mathtt{h}\ \mathtt{x}\ \mathrm{y}_2)$$

To Prove: A sufficient (but not necessary) condition on $\mathtt{h}$ for a primitive recursive term, $\tau[\mathrm{F}]$, to be monotonic is that $\mathtt{h}$ have a right fixed point, $\mathtt{c}$.

Proof. Use the right fixed point $\mathtt{c}$ to build a flat domain:

- Use $\mathtt{c}$ for $\perp$ and
- $\sqsubseteq_{\mathsf{c}}$ for $\sqsubseteq$ where

$$x \sqsubseteq_{\mathsf{c}} y \Longleftrightarrow x = \mathsf{c} \vee x = y.$$

- Then

$$\mathrm{y}_1 \sqsubseteq_{\mathsf{c}} \mathrm{y}_2 \Rightarrow (\mathtt{h}\ \mathtt{x}\ \mathrm{y}_1) \sqsubseteq_{\mathsf{c}} (\mathtt{h}\ \mathtt{x}\ \mathrm{y}_2)$$

Which ACL2 terms are monotonic?

Nested Recursion. Let `test`, `base`, and `st` be arbitrary unary functions.

Consider a term τ[F] of the form

```
(if (test x)
    (base x)
    (F (F (st x))))
```

Often such terms are **not** monotonic.

Such terms **are** monotonic
if `F` *always preserves* $\sqsubseteq$:

$$y_1 \sqsubseteq y_2 \Rightarrow (\mathtt{F}\ y_1) \sqsubseteq (\mathtt{F}\ y_2)$$

That is, **restrict** the variable `F` to range only over functions that *always preserve* $\sqsubseteq$.

Nested Recursion and Kleene's Construction

Recall Kleene's construction:

- Use the term $\tau[\mathtt{F}]$ to recursively define a chain of functions,

$$\begin{aligned} f_0(\mathtt{x}) &= \bot \\ f_{i+1}(\mathtt{x}) &= \tau[f_i](\mathtt{x}). \end{aligned}$$

- Since $\tau[\mathtt{F}]$ is *monotonic*,

$$f_0 \sqsubseteq f_1 \sqsubseteq \cdots \sqsubseteq f_i \sqsubseteq \cdots$$

- To ensure $\tau[\mathtt{F}]$ is *monotonic*, the function variable `F` should range only over functions that *always preserve* $\sqsubseteq$.

- That is, each f_i should *always preserve* $\sqsubseteq$.

Nested Recursion and Kleene's Construction

To ensure that each f_i *always preserves* $\sqsubseteq$:

- Clearly, f_0, defined by $f_0(\mathtt{x}) = \bot$, *always preserves* $\sqsubseteq$.

- **Require**: *Whenever* f *always preserves* $\sqsubseteq$, *then* $\tau[f]$ *is also a function that always preserves* $\sqsubseteq$.

Nested Recursion and Kleene's Construction

Requirement. *Whenever f always preserves $\sqsubseteq$, then $\tau[f]$ is also a function that always preserves $\sqsubseteq$.*

Orthodox Solution. Functions, that always preserve $\sqsubseteq$, are closed under composition.

- **Restrict** $\tau[\mathtt{F}]$ to compositions involving `F` and functions that always preserve $\sqsubseteq$.

- So `test`, `base`, `st`, and `if` should all be functions that always preserve $\sqsubseteq$

```
(if (test x)
    (base x)
    (F (F (st x))))
```

- **Problem.** ACL2's `if` does **not** preserve $\sqsubseteq$.

Nested Recursion and Kleene's Construction

Problem. ACL2's `if` does **not** preserve $\sqsubseteq$.

- Assume $\perp \neq$ NIL.
- Then $\perp \sqsubset$ NIL, but
- $(\texttt{if}\ \perp\ 0\ 1) = 0 \not\sqsubseteq 1 = (\texttt{if NIL 0 1})$

Solution. Replace ACL2's `if` with a *sequential* version, `sq-if`, that always preserves $\sqsubseteq$.

$$
\begin{array}{lcl}
(\texttt{sq-if}\ \ \perp\ \ \texttt{b c}) & = & \perp \\
(\texttt{sq-if NIL b c}) & = & \texttt{c} \\
(\texttt{sq-if}\ \ \texttt{a}\ \ \texttt{b c}) & = & \texttt{b}\ \ \text{if a} \neq \perp \wedge \text{a} \neq \texttt{NIL}
\end{array}
$$

Nested Recursion and Kleene's Construction

Requirement. *Whenever* f *always preserves* $\sqsubseteq$*, then* $\tau[f]$ *is also a function that always preserves* $\sqsubseteq$.

Non-Orthodox Solution. Replace ACL2's `if` with the sequential version, `sq-if`, and

Make sure `test` is **strict**.

- A function is *strict* iff the function returns $\perp$ whenever any of its inputs is $\perp$.
- Every strict function always preserves $\sqsubseteq$.
- The function `sq-if` is **not** strict.

Nested Recursion and Kleene's Construction

Non-Orthodox Solution. When `test` is strict, the term

```
(sq-if (test x)
       (base x)
       (F (F (st x))))
```

always produces a strict function, whenever `F` is replaced by any unary function `f`.

Every strict function always preserves $\sqsubseteq$.

Primitive heuristics for ensuring terms are monotonic

For subterms, $\tau[\mathrm{F}]$, of the form

```
(if (test x)
    (then x)
    (else x))
```

- If `F` appears in `(test x)`, then replace `if` by `sq-if`.

- If `F` is nested more than one deep in any of `(test x)`, `(then x)`, or `(else x)`, then replace `if` by `sq-if` and ensure that `(test x)` is strict.

Primitive heuristics for ensuring terms are monotonic

- If `F` appears in `(then x)` or `(else x)` then, other function applications appearing in `(then x)` or `(else x)`,

  1. need not be applications of functions that always preserve $\sqsubseteq$, if they contain no applications of `F`;

  2. should be applications of functions that always preserve $\sqsubseteq$, if they contain any application of `F`.

  Example. `(h (F (st x)))`
  `st` need not preserve $\sqsubseteq$
  `h` should preserve $\sqsubseteq$

Zero Function. Construct an ACL2 function `Z` satisfying the equation

```
(equal (Z x)
       (if (equal x 0)
           0
           (* (Z (- x 1))(Z (+ x 1))))).
```

- The two recursive calls of `Z` are contained inside the call to `*`.
- The heuristics suggest that `*` is the only function required to preserve $\sqsubseteq$.
- Unfortunately, `*` does not preserve $\sqsubseteq$ with respect to the usual ACL2 version of $\perp$, `($bottom$)`.

- A strict version of * would require

```
(equal (* ($bottom$) x) ($bottom$))
(equal (* x ($bottom$)) ($bottom$)).
```

  Fortunately, the above two equations do hold if `($bottom$)` is replaced by `0`,

```
(equal (* 0 x) 0)
(equal (* x 0) 0).
```

  Therefore, the entire construction can be carried out using `0` in place of `($bottom$)`.

This example illustrates that any convenient ACL2 object can be used to play the role of `($bottom$)`.

Ackermann's Function. Construct an ACL2 function `f` satisfying

```
(equal (f x1 x2)
       (if (equal x1 0)
           (+ x2 1)
           (if (equal x2 0)
               (f (- x1 1) 1)
               (f (- x1 1)
                  (f x1
                     (- x2 1))))))).
```

The heuristics suggest it should be possible to find `f` that satisfies:

```
(equal (f x1 x2)
       (if (equal x1 0)
           (+ x2 1)
           (SQ-IF (LT-ST-EQUAL x2 0)
                  (f (- x1 1) 1)
                  (f (- x1 1)
                     (f x1
                        (- x2 1))))))).
```

- Here `SQ-IF` is the monotonic sequential version of `if`,

- `LT-ST-EQUAL` is a left-strict version of `equal` satisfying

```
(equal (LT-ST-EQUAL 'undef$ y)
       'undef$).
```

- Here `'undef$` is used in place of `($bottom$)`.

The heuristics are too primitive. No such ACL2 function was proved to exist. But, experimentation shows it is possible to define an ACL2 function `f` satisfying

```
(equal (f x1 x2)
       (if (equal x1 0)
           (LT-ST-+ x2 1)
           (sq-if (lt-st-equal x2 0)
                  (f (- x1 1) 1)
                  (f (- x1 1)
                     (f x1
                        (- x2 1)))))).
```

- Here `LT-ST-+` is a left-strict version of (binary) + satisfying

  ```
  (equal (LT-ST-+ 'undef$ y) 'undef$).
  ```

Of course any function `f` satisfying this last equation may not satisfy the original equation. However, ACL2 can verify the following, showing that any such `f` can fail to satisfy the original equation only when the second input is `'undef$`:

```
(implies (not (equal x2 'undef$))
         (equal (f x1 x2)
                (if (equal x1 0)
                    (+ x2 1)
                    (if (equal x2 0)
                        (f (- x1 1) 1)
                        (f (- x1 1)
                           (f x1
                              (- x2 1)))
))))).
```