Flat Domains and Recursive Equations in ACL2

John Cowles
Department of Computer Science
University of Wyoming
Laramie, Wyoming 82071
cowles@uwyo.edu

March 3, 2002

Abstract

Flat domains can be viewed as a "logic" of total functions in which every recursive equation has at least one function that satisfies it.

One formalization of flat domains in ACL2 is presented in some detail.

Flat domains are reviewed in enough detail to make this paper self contained for those who never knew or don't remember much about them.

1 Introduction

The following succinct, but accurate, description of ACL2 is taken from the introduction to the paper [9]:

> The ACL2 system [4, 5] consists of a programming language based on Common Lisp [10], a logic of total recursive functions[1], and a theorem prover. It contains a definitional principle essentially identical to that in Boyer and Moore's Nqthm [1] whereby function definitions are admissible only if a measure of the arguments can be shown to be decreasing in some well-founded sense, in every recursive call.

[1] That is, total computable functions such as those studied in computability theory [3].

ACL2's definitional principle ensures not only that an admissible definition defines an unique total function, but also that the definition can be used to actually compute the function. That is, at least in principle, any recursion terminates in the programming language implementation of the definition.

For example, consider the following proposed definitions, taken from [8], of `g` and `h`, shown in ACL2's Lisp-like syntax.

```
(defun g (n)
  (if (equal n 0)
      nil
      (cons nil
            (g (- n 1)))))
```

```
(defun h (n)
  (if (equal n 0)
      0
      (+ 1
         (h (- n 1)))))
```

These definitions are both inadmisible because termination is impossible to establish: Executing either `(g n)` or `(h n)`, in Lisp, for any negative integer `n`, eventually exhausts resources by attempting a nonterminating computation.

Although the defining recursive equations, given below, for `g` and `h` cannot be reliably used for computation, a question still remains: Are there any ACL2 functions that satisfy the defining recursive equations for `g` and `h`?

```
(equal (g n)
       (if (equal n 0)
           nil
           (cons nil
                 (g (- n 1)))))
```

```
(equal (h n)
       (if (equal n 0)
           0
           (+ 1
              (h (- n 1)))))
```

That is, can the defining equation for `g` or `h` be consistently added to ACL2's logic? Notice that the equations are syntactically similar, `g` conses `nil` where `h` adds `1`. As shown in [8], the defining equation for `h` can be safely added as an axiom, but adding the defining equation for `g` renders ACL2's logic inconsistent. Also shown in [8, 9], any tail recursive defining equation, such as the one shown below, can be consistently added to ACL2's logic.

```
(equal (h n)
       (if (equal n 0)
           nil
           (h (- n 1))))
```

The above examples show that ACL2's logic can sometimes be consistently extended by adding a recursive equation and sometimes not.

Functions defined on a so called *flat domain* can be viewed as a "logic" of *total* functions in which every recursive equation is consistent. That is,

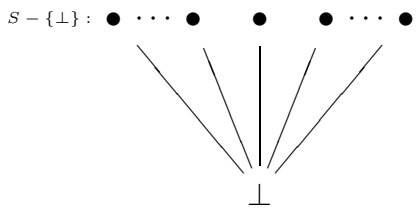

Figure 1: Graphical representation of a flat domain.

every recursive equation has at least one function that satisfies it. The next sections briefly review flat domains and their application to recursive equations. For the time being, Lisp's prefix notation is abandoned in favor of more conventional mathematical notation. Later sections apply the lessons learned from the study of flat domains to ACL2.

2 Flat Domains

A *flat domain*, as discussed in the fixpoint theory of program semantics [7, Chapter 5] and [3, Chapter 16], is a structure $< S, \sqsubseteq, \bot >$, where S is a set, $\bot \in S$, and the binary relation $\sqsubseteq$ is the partial order[2] defined by $x \sqsubseteq y \iff x = \bot \vee x = y$.

This partial order is extended *componentwise* to tuples of elements of S, and then to functions on S, by

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

and for functions f and g, by

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

The "flat part" of flat domain is depicted by the row of vertices, labeled with $S - \{\bot\}$, in Figure 1. The figure shows a graphical representation of the $\sqsubset$ relation defined by $x \sqsubset y \iff x \sqsubseteq y \wedge x \neq y$.

2.1 Least Upper Bounds of Chains

Every chain of functions on S,

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

[2]Recall that a partial order is reflexive, antisymmetric, and transitive.

has an unique *least upper bound* denoted by $\sqcup f_i$. That is, $\sqcup f_i$ is a function that satisfies

- 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 $\sqcup f_i \sqsubseteq f$.

For each input $\vec{x}$, $\sqcup f_i$ is easily defined by cases: If for all i, $f_i(\vec{x}) = \perp$, then let $\sqcup f_i(\vec{x}) = \perp$. Otherwise, there is an j such that $f_j(\vec{x}) \neq \perp$. It must be that for all $k > j$, $f_k(\vec{x}) = f_j(\vec{x})$. So in this case, let $\sqcup f_i(\vec{x}) = f_j(\vec{x})$. Notice that in either case, for each $\vec{x}$, there exists an j such that for all $k \geq j$, $\sqcup f_i(\vec{x}) = f_k(\vec{x})$.

2.2 Monotonic Terms[3]

Let F be a variable ranging over the functions (of a fixed arity) on S and let $\tau[F]$ be a term built by compositions involving F and functions on S. For a function f (with the same number of inputs as F), $\tau[f]$ denotes the term obtained by replacing *every* occurrence of F in $\tau[F]$ with f. For each function f (with the appropriate number of inputs) on S, the term $\tau[f]$ also denotes a function on S.

Such a term $\tau[F]$ is *monotonic* just in case whenever f and g are functions such that $f \sqsubseteq g$, then $\tau[f] \sqsubseteq \tau[g]$.

If $\tau[F]$ is monotonic and

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

is a chain of functions, then clearly

$$\tau[f_0] \sqsubseteq \tau[f_1] \sqsubseteq \cdots \sqsubseteq \tau[f_i] \sqsubseteq \cdots$$

is also a chain. Furthermore, since, for all j, $f_j \sqsubseteq \sqcup f_i$, it must be that for all j, $\tau[f_j] \sqsubseteq \tau[\sqcup f_i]$. Thus $\tau[\sqcup f_i]$ is an upper bound of the chain

$$\tau[f_0] \sqsubseteq \tau[f_1] \sqsubseteq \cdots \sqsubseteq \tau[f_i] \sqsubseteq \cdots.$$

Since the least upper bound can be no larger than an upper bound,

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

Since the number of occurrences of F in $\tau[F]$ is finite, for every $\vec{x}$, it is always possible to find a j such that

$$\tau[\sqcup f_i](\vec{x}) = \tau[f_j](\vec{x}).$$

[3]These monotonic terms are a special case of what's known, in the literature of flat domains, as monotonic functionals.

This can be proved by structural induction on the term $\tau[F]$. Here is an example illustrating the proof.

Example. Suppose $\tau[F]$ is $F(F(x-1,y), F(y-1,x))$. Then, given x and y, there are j_1, j_2, and j_3 such that for all $k_1 \geq j_1$, for all $k_2 \geq j_2$, and for all $k_3 \geq j_3$

$$\begin{aligned} \sqcup f_i(x-1,y) &= f_{k_1}(x-1,y) \\ \sqcup f_i(y-1,x) &= f_{k_2}(y-1,x) \\ \sqcup f_i(\sqcup f_i(x-1,y), \sqcup f_i(y-1,x)) &= f_{k_3}(\sqcup f_i(x-1,y), \sqcup f_i(y-1,x)). \end{aligned}$$

Let $j = \max(j_1, j_2, j_3)$. Then

$$\sqcup f_i(\sqcup f_i(x-1,y), \sqcup f_i(y-1,x)) = f_j(f_j(x-1,y), f_j(y-1,x)).$$

Therefore for all $\vec{x}$, there is a j such that

$$\tau[\sqcup f_i](\vec{x}) = \tau[f_j](\vec{x}) \sqsubseteq \sqcup \tau[f_i](\vec{x});$$

so $\tau[\sqcup f_i] \sqsubseteq \sqcup \tau[f_i]$. Combined with $\sqcup \tau[f_i] \sqsubseteq \tau[\sqcup f_i]$, from above, it follows [7, Theorem 5-1, page 367] that for any monotonic term $\tau[F]$ and any chain of functions

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

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

3 Recursive Equations

Let F be a function variable and let $\tau[F]$ be a term. 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 said to be a *fixed point* of the term $\tau[F](\vec{x})$.

When $\tau[F]$ is monotonic, a construction due to Kleene [6, Proof of Theorem XXVI, pages 348–349] shows that the equation

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

always has a solution:

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

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

Since $\tau[F]$ is monotonic,

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

and

$$\tau[f_0] \sqsubseteq \tau[f_1] \sqsubseteq \cdots \sqsubseteq \tau[f_i] \sqsubseteq \cdots.$$

By definition $\tau[f_i] = f_{i+1}$, so $\sqcup\tau[f_i] = \sqcup f_i$. Then, from above,

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

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

Furthermore, the *least upper bound* $\sqcup f_i$ given by the Kleene construction is the *least fixed point* for the term $\tau[F](\vec{x})$ in the following sense. For any fixed point f of $\tau[F](\vec{x})$, $\sqcup f_i \sqsubseteq f$ This follows because any fixed point f must be an upper bound the chain f_i: Clearly $f_0 \sqsubseteq f$. If $f_i \sqsubseteq f$, since $\tau[F]$ is monotonic and f is fixed point, then $f_{i+1} = \tau[f_i] \sqsubseteq \tau[f] = f$.

3.1 Which Terms are Monotonic?

The answer to this question is explored with the help of the following examples.

Tail Recursion. Consider a term $\tau[F]$ of the form

$$\begin{aligned} &\mathsf{if}\ \mathrm{test}(x)\ \mathsf{then}\ \mathrm{base}(x) \\ &\qquad\qquad\ \mathsf{else}\ \ F(\mathrm{step}(x)). \end{aligned}$$

Let f and g be unary functions such that $f \sqsubseteq g$. Then, whenever $\mathrm{test}(x)$ holds, $\tau[f](x) = \mathrm{base}(x) = \tau[g](x)$; so in this case $\tau[f](x) \sqsubseteq \tau[g](x)$. When $\mathrm{test}(x)$ does not hold, since $\forall y[f(y) \sqsubseteq g(y)]$,

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

Thus $\tau[f] \sqsubseteq \tau[g]$. That is, *tail recursive terms are always monotonic.*

This means that tail recursive equations always have solutions. This observation gives another explanation for the result in [8, 9], that any tail recursive equation is satisfiable by some function.

Primitive[4] Recursion. Consider a $\tau[F]$ of the form

$$\begin{array}{ll} \textsf{if } \text{test}(x) \textsf{ then } & \text{base}(x) \\ \qquad\qquad \textsf{else } & h(x, F(\text{step}(x))). \end{array}$$

Let f and g be unary functions such that $f \sqsubseteq g$. Then, as above, whenever test(x) holds, $\tau[f](x) = \text{base}(x) = \tau[g](x)$.

Showing $\tau[F]$ is monotonic, when test(x) does not hold, reduces to showing

$$\tau[f](x) = h(x, f(\text{step}(x))) \sqsubseteq h(x, g(\text{step}(x))) = \tau[g](x).$$

As above, $f(\text{step}(x)) \sqsubseteq g(\text{step}(x))$. One way to ensure that

$$h(x, f(\text{step}(x))) \sqsubseteq h(x, g(\text{step}(x))$$

is to require that h always preserves the partial order $\sqsubseteq$ on its rightmost input:

For all x, y_1, and y_2, whenever $y_1 \sqsubseteq y_2$, then $h(x, y_1) \sqsubseteq h(x, y_2)$.

A unary function that always preserves $\sqsubseteq$ is said to be *monotonic*. So for all x, the unary function h_x defined by $h_x(y) = h(x, y)$ should be monotonic.

Thus, *a primitive recursive term is a monotonic term whenever* for each x, the unary function h_x *is a monotonic function.*

Which Functions are Monotonic?

By definition, an unary function f_1 is *monotonic* iff for all x, whenever $x_1 \sqsubseteq x_2$, then $f_1(x_1) \sqsubseteq f_1(x_2)$. It is straightforward to prove [7] that an unary function f_1 is *monotonic* iff either $\perp$ is a fixed point for f_1 (i.e. $f_1(\perp) = \perp$) or f_1 is constant (i.e., $\exists c_1 \forall x[f_1(x) = c_1]$).

[4]These terms are called primitive recursive because for a $\tau[F]$ with this particular form, the recursive equation $F(x) = \tau[F]$ is a generalization of the definitions by primitive recursion studied in computability theory [3]:

$$\begin{array}{rcl} F(x_1, \ldots, x_n, 0) & = & k(x_1, \ldots, x_n) \\ F(x_1, \ldots, x_n, t+1) & = & h(t, F(x_1, \ldots, x_n, t), x_1, \ldots, x_n) \end{array}$$

This observation gives another explanation for the result in [2], that can be paraphrased in the terminology of this paper: A sufficient (but not necessary) condition on h for a primitive recursive term, $\tau[F]$, to be monotonic is that h have a right fixed point, i.e., $\exists c \forall x [h(x, c) = c]$. To see this, use the right fixed point c to build the flat domain with c playing the part of $\perp$. Then for each x, the unary function h_x is a monotonic function, which, as noted above, is enough to guarantee that the primitive recursive term, $\tau[F]$, is a monotonic term.

Nested Recursion. Consider a $\tau[F]$ of the form

$$\begin{array}{l} \mathsf{if}\ \mathrm{test}(x)\ \mathsf{then}\ \ \mathrm{base}(x) \\ \qquad\qquad\quad \mathsf{else}\ \ F(F(\mathrm{step}(x))). \end{array}$$

Let f and g be unary functions such that $f \sqsubseteq g$. Then, as above, whenever $\mathrm{test}(x)$ holds, $\tau[f](x) = \mathrm{base}(x) = \tau[g](x)$.

Showing $\tau[F]$ is a monotonic term, when $\mathrm{test}(x)$ does not hold, reduces to showing

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

Since $f \sqsubseteq g$, both of the following hold,

$$\begin{array}{rcl} f(\mathrm{step}(x)) & \sqsubseteq & g(\mathrm{step}(x)) \\ f(g(\mathrm{step}(x))) & \sqsubseteq & g(g(\mathrm{step}(x))). \end{array}$$

One way to ensure that

$$f(f(\mathrm{step}(x))) \sqsubseteq g(g(\mathrm{step}(x))$$

is to require that f be a monotonic function: Since $f(\mathrm{step}(x)) \sqsubseteq g(\mathrm{step}(x))$, then $f(f(\mathrm{step}(x))) \sqsubseteq f(g(\mathrm{step}(x)))$. So

$$f(f(\mathrm{step}(x))) \sqsubseteq f(g(\mathrm{step}(x)) \sqsubseteq g(g(\mathrm{step}(x)).$$

Thus, one way *to ensure that $\tau[F]$ is a monotonic term, when the function variable F is nested more than one deep, is to restrict the variable F to range only over monotonic functions.*

Nested Recursion and Kleene's Construction

Equations (1) use $\tau[F]$ to recursively define a sequence of functions, $f_0,\ f_1, \ldots, f_i \ldots$. To ensure that the f_i form a $\sqsubseteq$-chain, $\tau[F]$ should be a monotonic term; and to ensure that $\tau[F]$ is a monotonic term, the function variable F should range only over monotonic functions. Hence, each of the f_i should be a monotonic function.

Clearly f_0, defined by $f_0(x) = \bot$, is a monotonic function. Since f_{i+1} is defined by $f_{i+1}(x) = \tau[f_i](x)$, one way to ensure that all the f_i are monotonic functions is to require, of the term $\tau[F]$, that *whenever f is a monotonic function, then $\tau[f]$ is also a monotonic function.* Here are two ways to meet this requirement:

1. Monotonic functions are easily seen to be closed under composition. Thus, if $\tau[F]$ is a term built by compositions involving F and *monotonic functions*, then $\tau[F]$ is, indeed, a term such that whenever f is a monotonic function, then $\tau[f]$ is also a monotonic function. Thus, in this nested recursion example, test(x), base(x), and step(x) would have to be unary monotonic functions and the ternary if-then-else function would have to be replaced by the explicitly monotonic *sequential* if-then-else function, sq-if-then-else, that satisfies

$$\begin{array}{rcl} \textsf{sq-if}\ \text{true}\ \textsf{then}\ a\ \textsf{else}\ \bot & = & a \\ \textsf{sq-if}\ \text{false}\ \textsf{then}\ \bot\ \textsf{else}\ b & = & b \\ \textsf{sq-if}\ \ \bot\ \ \textsf{then}\ a\ \textsf{else}\ b & = & \bot \end{array}$$

2. A function is said to be *strict* if and only if the function returns $\bot$ whenever any of its inputs is $\bot$. Every strict function is monotonic. Notice that the function sq-if-then-else is not strict. If test(x) is strict and if-then-else is replaced by sq-if-then-else in this nested recursion example, then the resulting $\tau[F]$ always produces a monotonic function, whenever F is replaced by any unary function f. That is, for a term, $\tau'[F]$, of the form

$$\begin{array}{l} \textsf{sq-if}\ \text{test}(x)\ \textsf{then}\ \text{base}(x) \\ \qquad\qquad\qquad \textsf{else}\ \ F(F(\text{step}(x))), \end{array}$$

if test(x) is strict and $x \sqsubseteq y$, then for any unary function f, $\tau'[f](x) \sqsubseteq \tau'[f](y)$.

Most orthodox expositions [3, 7] of flat domains and monotonic terms simplify matters by always imposing the restrictions just mentioned above

in 1: *The term $\tau[F]$ must be composed entirely of F and monotonic functions, and F must vary only over monotonic functions.* Monotonic functions almost always treat $\perp$ in special ways that often cause mechanized proofs involving monotonic functions to "explode" with case-splits. Thus, monotonic functions should be avoided as much as possible. The previous three examples show that it is often possible to reduce the number of monotonic functions required in a recursive definition.

These examples suggest the following very rough and primitive heuristics for subterms, $\tau[F]$, of the form

$$\begin{array}{l} \mathsf{if}\ \text{test}(x)\ \mathsf{then}\ \text{t-term}(x) \\ \qquad\qquad\ \mathsf{else}\ \ \text{e-term}(x). \end{array}$$

- If F appears in t-term(x) or e-term(x), then, other function applications appearing in t-term(x) or e-term(x),
  1. need not be applications of monotonic functions, if they contain no applications of F;
  2. should be applications of monotonic functions, if they contain any application of F.
- 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), t-term(x), or e-term(x), then replace if by sq-if and ensure that test(x) is strict.

4 Formalizing Flat Domains in ACL2

Impose a partial order, `$<=$`, on ACL2 data by specifying a "least element", `($bottom$)`, that is *strictly* less than any other ACL2 datum and no other distinct data items are related:

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

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

ACL2 easily verifies that `$<=$` is a reflexive partial order on the ACL2 universe.

`Encapsulate`s are used to axiomatize various situations involving unary functions indexed by nonnegative integers: $f_0, f_1, \ldots, f_i, \ldots$. Indexed functions are formalized in ACL2 by treating the index as an additional argument to the function, so $f_i(x)$ becomes `(f i x)` in ACL2.

One of these situations is the case, of an `$<=$`-chain of functions, consistently axiomatized by

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

The least upper bound of the chain of functions is defined in ACL2 by using `defchoose` to pick an appropriate "index" required in the definition of the least upper bound. ACL2 verifies that this formal least upper bound is, in fact, not only an upper bound of the chain, but the *least* upper bound.

By adding the consistent axiom

```
(implies ($<=$ x y)
         ($<=$ (f i x)
               (f i y)))
```

to the preceding axiom, ACL2 verifies that the least upper bound of a `$<=$`-chain of *monotonic* functions is also a *monotonic* function. `Functional instantiation` shows that the least upper bound exists.

4.1 Formalizing the Kleene Construction

Recall that equations (1) use a monotonic term $\tau[F]$ to recursively define a chain of functions. The least upper bound of the chain is a solution to the the recursive equation $F(x) = \tau[F](x)$

A very ambitious formalization of Kleene's construction would *require the formalization of monotonic terms* $\tau[F]$ and the formal proof that whenever $\tau[F]$ is a monotonic term, then the construction produces a function f such that $f(x) = \tau[f](x)$. In fact, this ambitious formalization has yet to be undertaken.

A much less ambitious, but still useful, formalization abstracts away the monotonic term $\tau[F]$: Recall that the term $\tau[F]$ is used to define two chains,

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

$$g_0 \sqsubseteq g_1 \sqsubseteq \cdots \sqsubseteq g_i \sqsubseteq \cdots,$$

by

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

The argument showing that $\sqcup f_i = \tau[\sqcup f_i]$ is summarized as follows.

1. $\sqcup g_i = \sqcup f_i$, because $g_i = f_{i+1}$.

2. $g_i = \tau[f_i] \sqsubseteq \tau[\sqcup f_i]$, because $\tau[F]$ is a monotonic term. So $\tau[\sqcup f_i]$ is an *upper bound* of the g_i.

3. $\forall x \exists j (\tau[\sqcup f_i](x) = \tau[f_j](x) = g_j(x))$, because of the finite number of occurrences of F in $\tau[F]$. So there is a *Skolem function* h such that $\tau[\sqcup f_i](x) = \tau[f_{h(x)}](x) = g_{h(x)}(x)$.

4. Then $\sqcup g_i(x) \sqsubseteq \tau[\sqcup f_i](x) = g_{h(x)}(x) \sqsubseteq \sqcup g_i(x)$, by 2 and 3. Thus $\sqcup g_i = \tau[\sqcup f_i]$.

5. Then $\sqcup f_i = \sqcup g_i = \tau[\sqcup f_i]$, by 1 and 4.

The abstract version, leaving out the term $\tau[F]$, of this argument goes as follows.

- Use `encapsulate` to consistently axiomatize two `$<=$`-chains, `(f i x)` and `(g i x)`, related by `(equal (g i x)(f (+ i 1) x))`. Then ACL2 verifies that these two chains have the same least upper bound. This corresponds to 1 above.

- As in the previous item, use `encapsulate` to consistently axiomatize two `$<=$`-chains, `(f i x)` and `(g i x)`, related by `(equal (g i x) (f (+ i 1) x))`. This time two additional functions `(ub-g x)` and `(h x)` are consistently axiomatized so that

  - `(ub-g x)` is an upper bound of the `(g i x)`, i.e., `($<=$ (g i x) (ub-g x))`. So `ub-g` is playing the part of $\tau[\sqcup f_i]$ in 2 above.
  - `(equal (ub-g x)(g (h x) x))`. So `h` is the Skolem function h in 3 above.

  Then ACL2 verifies `(equal (lub-f x)(ub-g x))`, where `lub-f` is the least upper bound of the `(f i x)`. This corresponds to the conclusion 5 above.

5 Utilizing the Abstract Kleene Construction in ACL2

Given a concrete recursive equation $F(x) = \tau[F](x)$ that we wish to study in ACL2, proceed as follows:

- Check if $\tau[F]$ is a monotonic term. ACL2 can aid in making this determination. If $\tau[F]$ is not a monotonic term, then use heuristics to replace a minimal number of functions in $\tau[F]$ with their monotonic or strict counterparts. This, of course, changes the original recursive equation from one that may not have a solution into a similar equation guaranteed to have a solution.

- Use $\tau[F]$ to define the chains `(f i x)` and `(g i x)`.

```
(defun
   f-chain (i x)
   (if (zp i)
       ($bottom$)
       τ[f-chain_(- i 1)] )
```

```
(defun
    g-chain (i x)
    (f-chain (+ 1 i) x))
```

- Define the least upper bound, `lub-f-chain` , of the chain, `f-chain`. Defchoose is used to choose the required "index", `lub-f-chain-i`.

```
(defun
    lub-f-chain (x)
    (f-chain (lub-f-chain-i x) x))
```

- Use $\tau[F]$ and `lub-f-chain` to define an upper bound, `ub-g-chain`, of `g-chain`.

```
(defun
   f (x)
   (lub-f-chain x))
```

```
(defun
   ub-g-chain (x)
   τ[f])
```

- Use the calls of F in $\tau[F]$ to help define the Skolem function called `h` above.

- ACL2 uses functional instantiaton to verify `(equal (lub-f-chain x) (ub-g-chain x))`.

- Finally, ACL2 uses the equality in the previous item to verify `(equal (f x)` τ`[f])`.

Examples

Tail Recursion. Construct an ACL2 function `f` satisfying the equation

```
(equal (f x)
       (if (test x)
           (base x)
           (f (st x)))).
```

Then following the above construction:

- τ[f], which is already a monotonic term, is

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

- Define:

```
(defun
    f-chain (i x)
    (if (zp i)
        ($bottom$)
        (if (test x)
            (base x)
            (f-chain (- i 1)(st x)))))
```

- Define `g-chain`, choose the "index" `lub-f-chain-i`, define `lub-f-chain` and `f` as outlined above.
- Define:

```
(defun
    ub-g-chain (x)
    (if (test x)
        (base x)
        (f (st x))))
```

- Define the Skolem function, called `h` above, and called `Skolem-f` here:

```
(defun
    Skolem-f (x)
    (lub-f-chain-i (st x)))
```

- Use functional instantiation to prove

```
(defthm
    lub-f-chain=ub-g-chain
    (equal (lub-f-chain x)(ub-g-chain x))
    ...)
```

- Finally, use the equality in the previous item to prove

```
(defthm
    generic-tail-recursive-f
    (equal (f x)
           (if (test x)
               (base x)
               (f (st x))))
    ...)
```

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

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

Since the two recursive calls of `Z` are contained inside the call to `*`, the heuristics suggest that `*` is the only monotonic function required for the construction to succeed. Unfortunately, `*` is *not* a monotonic function, a least with respect to the usual ACL2 version of $\perp$, `($bottom$)`. A strict (and hence monotonic) version of `*` would require for any `x`,

```
(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 `($bottom$)` and `$0<=$` in place of `$<=$`, where

```
(defun
    $0<=$ (x y)
    (or (equal x 0)
    (equal x y))).
```

Under these circumstances, the definition of `Z-chain` would be

```
(defun
    Z-chain (i x)
    (if (zp i)
        0
        (if (equal x 0)
            0
            (* (Z-chain (- i 1)(- x 1))
               (Z-chain (- i 1)(+ x 1))))))).
```

This example illustrates that any convenient ACL2 object can be used to play the role of `($bottom$)`. In this case, after `Z-chain` is defined, ACL2 reports:

```
We observe that the type of Z-CHAIN is described
by the theorem (EQUAL (Z-CHAIN I X) 0).
```

This means that during the systematic use of `Z-chain` to construct `Z`, ACL2 has essentially already deduced that `Z` is 0 on all inputs.

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))))))).
```

This example illustrates how to reduce recursive definitions of functions, with more than one input, to the unary case. The basic technique comes from the implementation of the macro `defpun` discussed in [8, 9].

Since the recursive calls to `f` are nested in the innermost call to `if`, the heuristics suggest it should be possible define an ACL2 function `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 an ACL2 macro implementation of sq-if, the monotonic sequential version of `if`, and `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 designated above by `(*)`. However, ACL2 can verify the following, showing that any such `f` can fail to satisfy `(*)` 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)))))))).
```

Here is the unary version of the desired equation, `(**)`.

```
(equal f1 (x)
       (let ((x1 (car x))
             (x2 (cadr x)))
            (if (equal x1 0)
                (lt-st-+ x2 1)
                (sq-if (lt-st-equal x2 0)
                       (f1 (list (- x1 1) 1))
                       (f1 (list (- x1 1)
                                 (f1 (list x1 (- x2 1))))
        )))))).
```

The heuristics suggest the second `list` should be monotonic. It is replaced with a right-strict version of (binary) `list`, `RT-ST-LIST` satisfying `(equal (RT-ST-LIST x 'undef$) 'undef$)`.

Here is the final definition for the chain used to construct an unary function `f1`:

```
(defun
    f1-chain (i x)
    (if (zp i)
        'undef$
        (if (eq x 'undef$) ;;Ensure f1 is strict and
            'undef$        ;;thus a monotonic function.
            (let ((x1 (car x))
                  (x2 (cadr x)))
                 (if (equal x1 0)
                     (lt-st-+ x2 1)
                     (sq-if (lt-st-equal x2 0)
                            (f1-chain (- i 1)
                                      (list (- x1 1)
                                            1))
                            (f1-chain
                             (- i 1)
                             (rt-st-list (- x1 1)
                                         (f1-chain
                                          (- i 1)
                                          (list x1
                                                (- x2 1))
                                          )))))))).
```

The unary function `f1` is used to define a function `f` satisfying the equation designated above by `(**)`:

```
(defun
    f (x1 x2)
    (f1 (list x1 x2))).
```

6 Conclusion

The theory of flat domains and its application to recursive equations have been reviewed. The ideas of a monotonic term and a monotonic function are the keys to ensuring that recursive equations have a solution.

Wholesale use of monotonic functions is undesirable from the practical point of view of mechanically implementing and verifying the constructions suggested by the theory of flat domains. Preliminary heuristics have been suggested to ensure that a term is monotonic while minimizing the number of applications of monotonic functions in the term.

One formalization of flat domains in ACL2 has been presented in some detail. Examples illustrate both success and limitations with this methodology for dealing with recursively defined functions in ACL2.

References

[1] R. Boyer and J Moore. *A Computation Logic Handbook.* Academic Press, second edition, 1997.

[2] J. Cowles, Consistently Adding Primitive Recursive Definitions in ACL2, ACL2 Workshop 2002.

[3] M. Davis, R.Sigal, and E. Weyuker. *Computability, Complexity, and Languages.* Academic Press, second edition, 1994.

[4] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, 2000.

[5] M. Kaufmann, P. Manolios, and J Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press, 2000.

[6] S. Kleene. *Introduction to Metamathematics*, Van Nostrand, 1952.

[7] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[8] P. Manolios and J S. Moore. Partial Functions in ACL2. In M. Kaufmann and J S. Moore, Editors, *ACL2 Workshop 2000 Proceedings*, October 30–31, 2000, University of Texas at Austin.

[9] P. Manolios and J S. Moore. Partial Functions in ACL2. January 2001, submitted for publication.

[10] G. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 1990.