Syntheto: A Surface Language for APT and ACL2

Alessandro Coglio
Eric McCarthy
Stephen Westfold

Kestrel Institute

Daniel Balasubramanian
Abhishek Dubey
Gabor Karsai

VANDERBILT UNIVERSITY
INSTITUTE FOR SOFTWARE INTEGRATED SYSTEMS

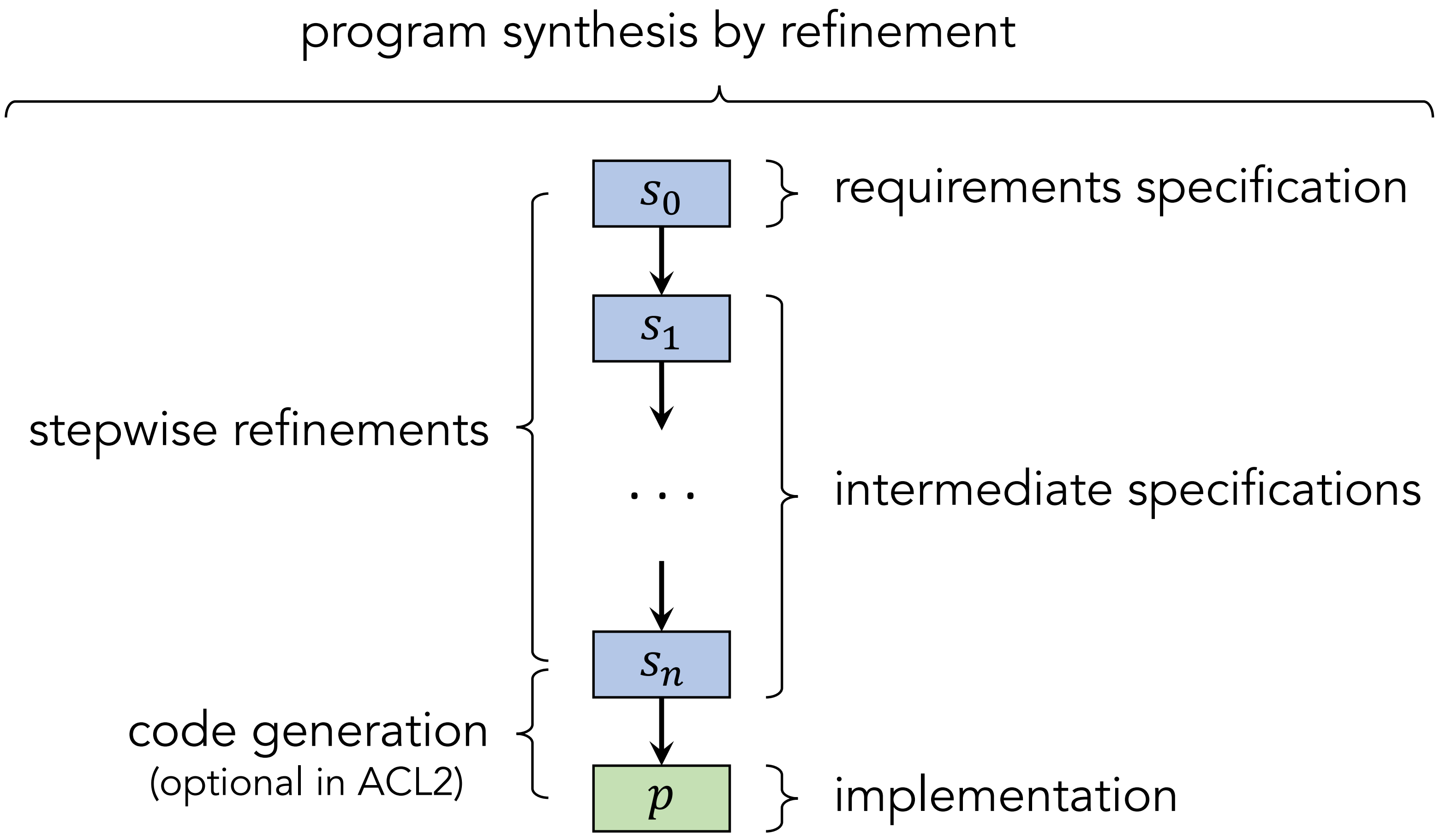
ACL2 WORKSHOP 2022

Background

APT (Automated Program Transformations) is a toolkit, built on ACL2, for formally verified program synthesis via transformational refinement.

APT transformations may be used to generate s_i from s_{i-1}, along with proofs of refinement.
APT transformation may require proving applicability conditions.
APT transformations include:

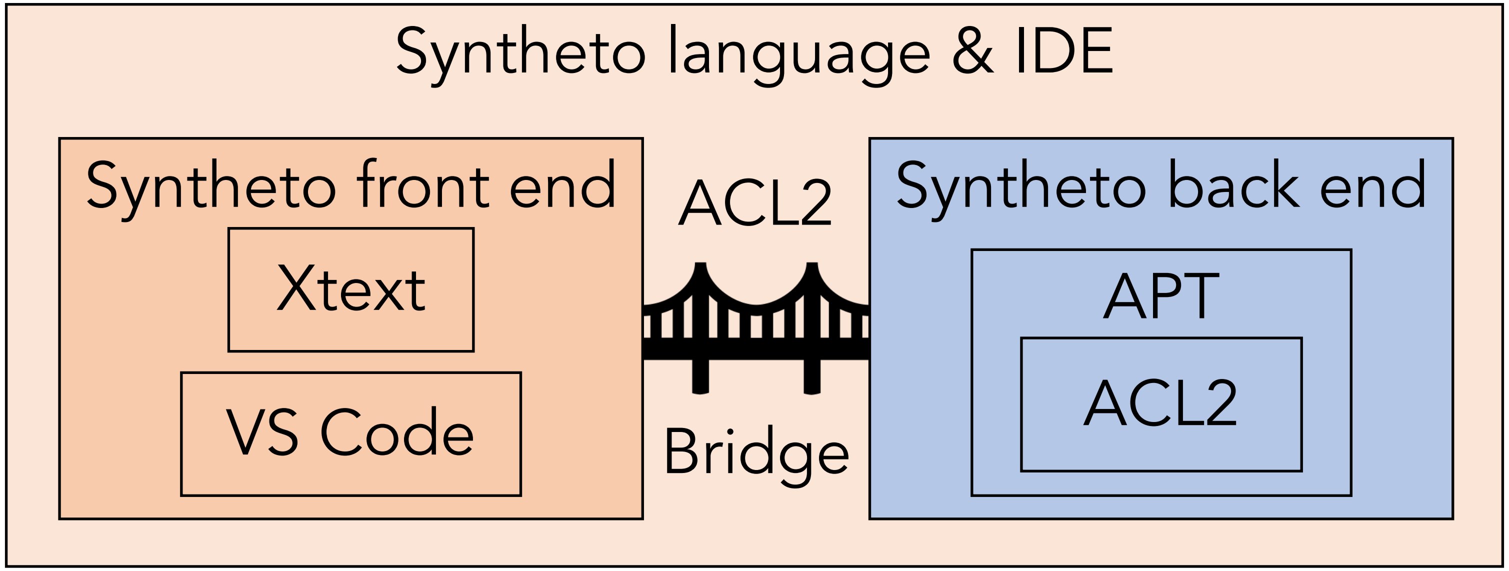
- Refine types isomorphically.
- Make functions tail-recursive.
- Simplify via rewrite rules.
- Incrementalize computations.
- And many others.



Users must have expertise in APT and ACL2.

Architecture

Syntheto uses ACL2 and APT, “hiding” them behind
(1) a strongly statically typed functional language and
(2) a notebook-style IDE based on VS Code.



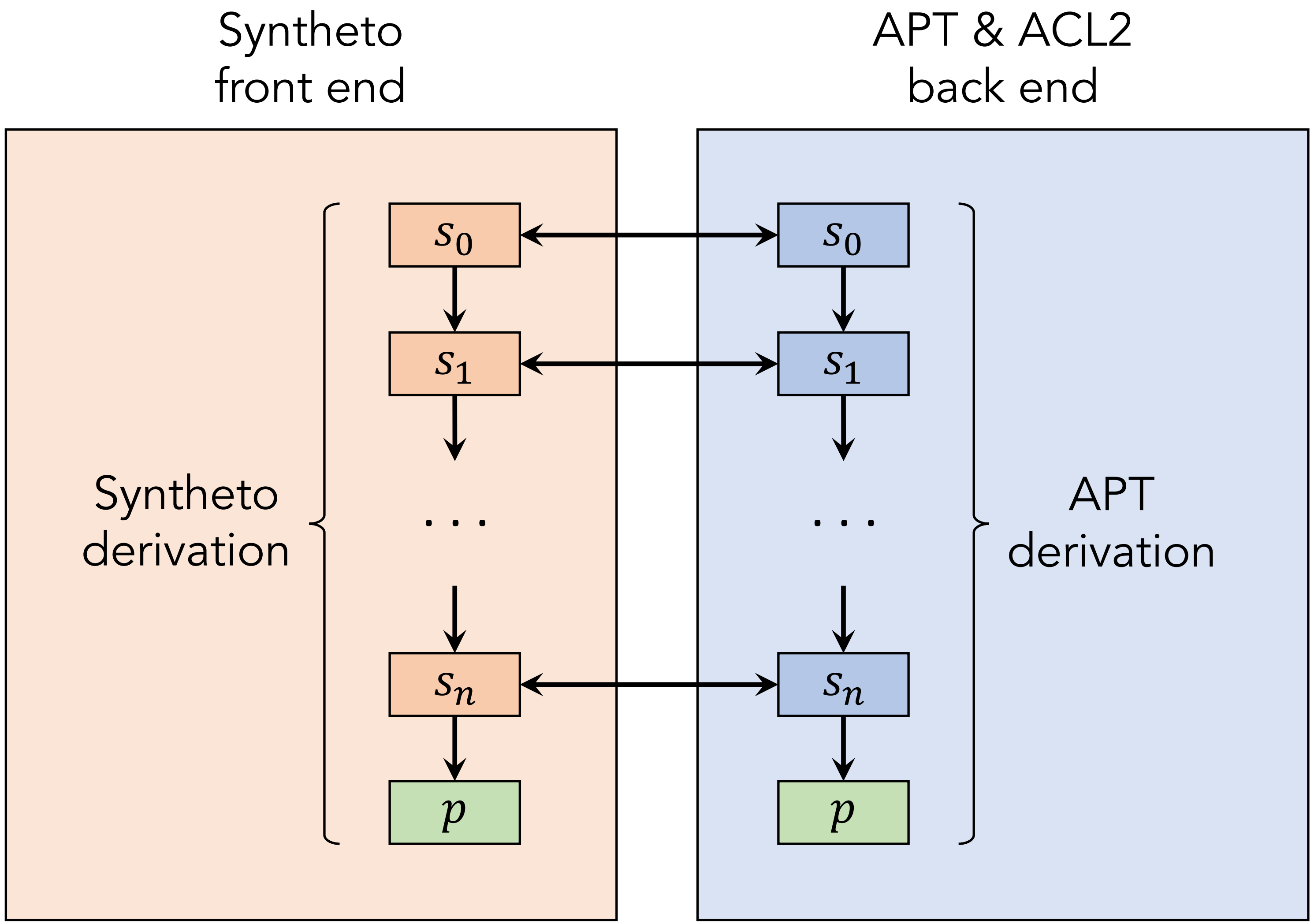
The purpose is to provide more familiarity and automation, making formal program synthesis more widely accessible.

Syntheto Usage

The user carries out derivations in Syntheto, in the notebook IDE.

The derivations are realized in APT/ACL2 behind the scenes.

There is a bidirectional translation between APT/ACL2 and Syntheto.



point_in_polygon_4.mnb

point_in_polygon_4.mnb

Users > westfold > Seafile > syntheto > examples > point_in_polygon_4.mnb > function path_vertices(edges:seq<edge>) returns (verti

+ Code + Markdown Run All Clear Outputs of All Cells ··· MIDAS NoteBook

```
function path_vertices(edges:seq<edge>) returns (vertices:seq<point>) ensures points2_p(vertices) {
  if (is_empty(edges)) {
    return empty;
  }
  else {
    let e:edge = first(edges);
    if (is_empty(rest(edges))) {
      return add(e.p1, add(e.p2, empty));
    }
    else {
      return add(e.p1, path_vertices(rest(edges)));
    }
  }
}
```

[ ] syntheto

```
// Inversion Theorems
theorem path_vertices_of_path
  forall(vertices:seq<point>)
    points2_p(vertices)
      ==> path_vertices(path(vertices)) == vertices

theorem path_of_path_vertices
  forall(edges:seq<edge>)
    path_p(edges)
      ==> path(path_vertices(edges)) == edges
```

[ ] syntheto

⊗ 0 ⚠ 0 Cell 4 of 10

Syntheto Language Features

- Strongly Statically Typed
  - Parameterized sequence, set, map and option types
  - Product and Sum types
  - Predicate Subtypes
  - Mutual Recursion
  - Primitive Types: integer, bool, char, string
- Functional but imperative-looking
- Functions: Executable and Non-executable (specifications)
- Theorems
- Transformations

Current Transformations

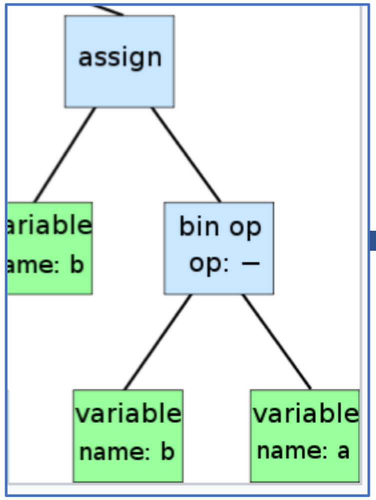
- **simplify**: Simplifies a function definition using enabled rewrite rules
- **finite_difference**: Adds a parameter to a function along with an invariant that the parameter is equal to a function of the existing parameters
- **tail_recursion**: Puts a function into tail-recursive form
- **isomorphism**: Replaces a parameter of one type by a parameter of an isomorphic type
- **rename_param**: Renames a parameter
- **drop_irrelevant_parameter**: Removes a parameter that is not needed
- **wrap_output**: Wraps a function call around the body of a function
- **restrict**: Adds a precondition on a function

Transfer Language

We want to send definitions and commands to ACL2 and to receive responses. How should we serialize the definitions?

On the ACL2 side, the Syntheto abstract syntax is defined primarily with FTY product types, sum types, list types, and some primitive types. The product types and sum types have handy "`make-`" macros that make values. *We use these S-expressions as the transfer language to transfer definitions in both directions.*

Syntheto AST in Java

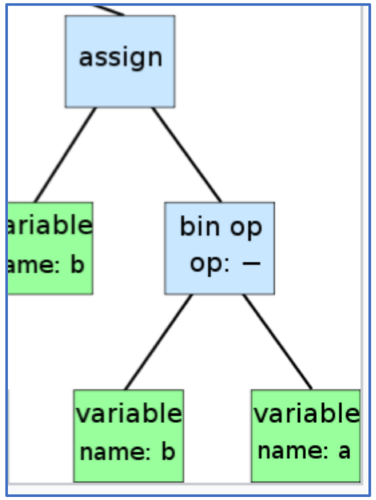


Serialize
methods
on AST classes

```
(SYNTHETO::MAKE-EXPRESSION-BINARY
 :OPERATOR (SYNTHETO::MAKE-BINARY-OP-GT)
 :LEFT-OPERAND
 (SYNTHETO::MAKE-EXPRESSION-VARIABLE
  :NAME (SYNTHETO::MAKE-IDENTIFIER :NAME "x"))
 :RIGHT-OPERAND
 (SYNTHETO::MAKE-EXPRESSION-LITERAL
  :GET (SYNTHETO::MAKE-LITERAL-INTEGER :VALUE 0)))
```

(read ..)
and
(make-event ..)

Syntheto AST in ACL2

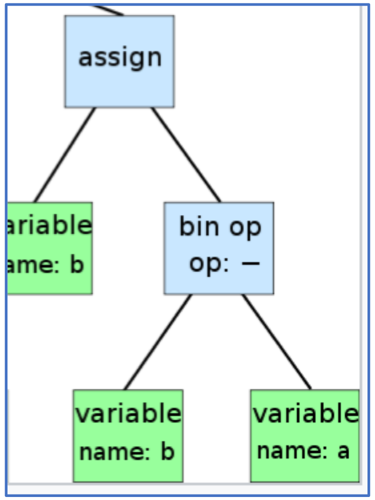


Transfer Language, return direction

Results of APT transformations are sent back to the front end.

To facilitate generating the transfer language from ACL2, for each AST node type we set up a ***make-myself*** macro such that `(make-myself x)` returns an S-expression that, when evaluated, makes `x`.

Syntheto AST in Java



Parse S-expression and use reflection to let each class build its instance

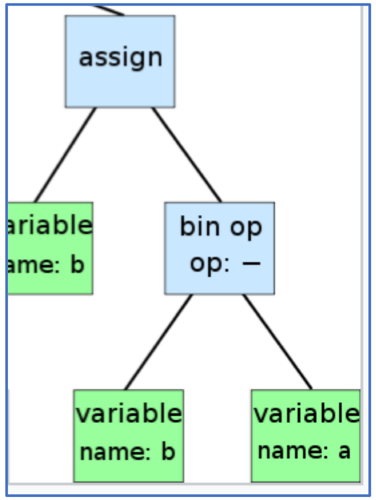
←

```
(SYNTHETO::MAKE-EXPRESSION-BINARY
 :OPERATOR (SYNTHETO::MAKE-BINARY-OP-GT)
 :LEFT-OPERAND
 (SYNTHETO::MAKE-EXPRESSION-VARIABLE
  :NAME (SYNTHETO::MAKE-IDENTIFIER :NAME "x"))
 :RIGHT-OPERAND
 (SYNTHETO::MAKE-EXPRESSION-LITERAL
  :GET (SYNTHETO::MAKE-LITERAL-INTEGER :VALUE 0)))
```

(make-myself ..)
and
(fmt ..)

←

Syntheto AST in ACL2



ACL2 Server

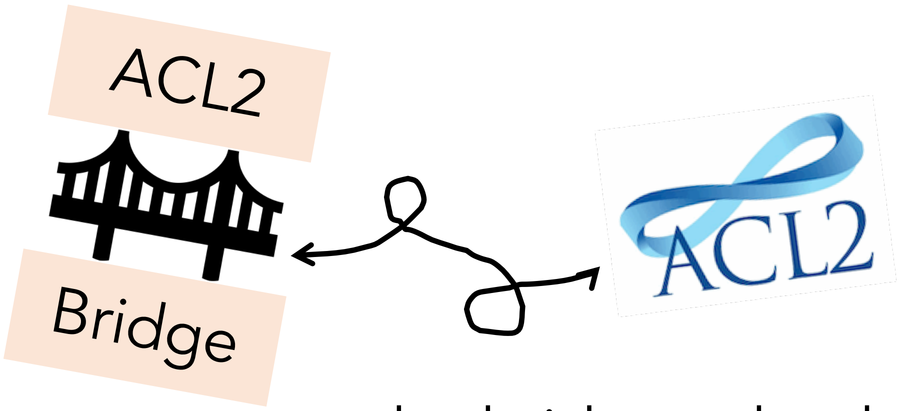
We want to send definitions and commands to ACL2 over a network connection, and to receive responses.

The ACL2 Bridge did a lot of the work for us.

Some lessons:
* The ACL2 Bridge only works on Clozure Common Lisp (CCL).
* Protocol is simple and easy.
* JSON interface for returned S-expressions loses information, so we found it better to use S-expressions directly.
* When you send an S-expression, the ACL2 Bridge will read it in the listener thread in the ACL2 package, not in the package your main listener is in.

We found the ACL2 Bridge to be super-reliable code!

ACL2 Server



Once the definitions get across the bridge, what happens to them? There's no human looking at an ACL2 prompt.

Some of the issues:

Events must be serialized to the main thread, because memoization is not thread-safe.
Solution: The ACL Bridge has `TRY-IN-MAIN-THREAD` that shuttles forms from socket listeners to the main listener.

No defined API for submitting events and receiving machine-readable responses.
Solution: Matt Kaufmann came up with `NLD`, "Noninteractive `LD`", which doesn't expect to be executing in a REPL and which returns certain output messages as structured data rather than sending them to stdout.

Translation of Syntheto to ACL2

- Types → primitive and **fty** types
- Expressions → s-expressions
  - Use functions created by fty macros
  - Add typing and guard predicates
- Function definitions
  - Regular → defun and typing defthms
  - Quantified → defun-sk
- Specifications → defstub and defun-sk
- Theorems → defthm
- Transformations → one or more APT transformations

Example Translations

```
struct point
 { x: int,
   y: int }

function connected(e1:edge, e2:edge) returns (b:bool) {
  return e1.p2 == e2.p1;
}

// Given a list of points, return the list of edges
// that connect the points in sequence
function path(vertices:seq<point>) returns (p:seq<edge>)
  ensures path_p(p) {
  if (is_empty(vertices) || is_empty(rest(vertices))) {
    return empty;
  }
  else {
    let e: edge = edge(p1=first(vertices), p2=first(rest(vertices)));
    return add(e, path(rest(vertices)));
  }
}

theorem path_p_rest
 forall(edges:seq<edge>)
   !is_empty(edges) && path_p(edges)
     ==> path_p(rest(edges))
```

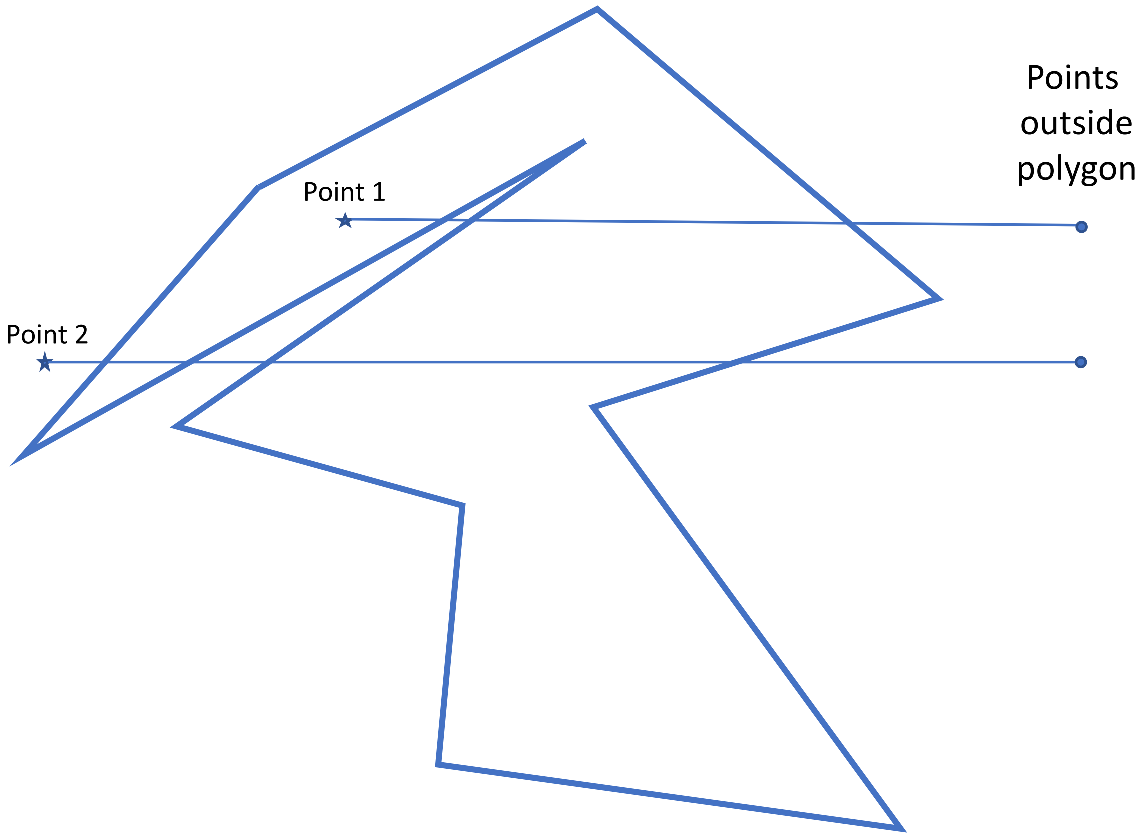
```
(fty::defprod point
  ((x int) (y int))
  :tag :point)

(define connected (e1 e2)
  :returns (b booleanp)
  (and (edge-p e1) (edge-p e2)
       (equal (edge->p2 e1) (edge->p1 e2))))

(define path ((vertices sequence[point]-p))
  :measure (len vertices)
  :guard (sequence[point]-p vertices)
  :prepwork ((local (include-book "kestrel/lists-light/len" :dir :system)))
  :returns (p sequence[edge]-p :hyp :guard)
  (if (or (not (mbt (and (sequence[point]-p vertices))))
          (endp vertices) (endp (cdr vertices)))
      nil
    (let ((e (MAKE-edge :p1 (car vertices)
                        :p2 (car (cdr vertices)))))
      (cons e (path (cdr vertices)))))
  ///
  (defret path-ENSURES :hyp :guard
    (path_p p)
    :hints (("Goal" :in-theory (enable path_p)))))

(defthm path_p_rest
  (implies (path_p edges)   ; remove-hyps removed 2 hyps
           (path_p (cdr edges)))
  :hints (("Goal" :in-theory (enable path_p))))
```

Back Translation of Transformed Functions

- Infer types of variables
  - Directly from guards
  - Simple inference on body
- Strip typing and guard predicates from function body
  - Can result in significantly simplified expression
- Exploit invertible naming scheme
- Currently supported APT transformations do not introduce functions that cannot be back-translated

Example Problem: Point in Polygon

A point is in a polygon if there are an odd number of edge crossings to a point outside the polygon.



3 crossings so point 1 is inside

4 crossings so point 2 is outside

Main Function

```
/* number of times edge0 crosses  edges */
function crossings_count_aux
                  (edge0: edge, edges: seq<edge>)
  assumes path_p(edges)
  returns (n: int) ensures n >= 0 {
  if (is_empty(edges)) {
    return 0;
  }
  else {if (edges_intersect(edge0, first(edges))) {
    return 1 + crossings_count_aux(edge0, rest(edges));
  }
  else {
    return crossings_count_aux(edge0, rest(edges));
  }}
}
```

Transformation Sequence

```
function crossings_count_aux_1 =
  transform crossings_count_aux
    by tail_recursion {new_parameter_name = count}

function crossings_count_aux_2 =
  transform crossings_count_aux_1
    by restrict {predicate = natp(count)}

function crossings_count_aux_3 =
  transform crossings_count_aux_2
    by isomorphism {parameter = edges,
                    new_parameter_name = vertices,
                    old_type = path_p,
                    new_type = points2_p,
                    old_to_new = path_vertices,
                    new_to_old = path,
                    simplify = true}

function crossings_count_aux_4 =
  transform crossings_count_aux_3
    by wrap_output {wrap_function = odd}

function crossings_count_aux_5 =
  transform crossings_count_aux_4
    by finite_difference {expression = odd(count),
                          new_parameter_name = count_odd,
                          simplify = true}

function crossings_count_aux_6 =
  transform crossings_count_aux_5
    by drop_irrelevant_param {parameter = count}
```

Main Function Transformation

```
/* number of times edge0 crosses  edges */
function crossings_count_aux
                (edge0: edge, edges: seq<edge>)
  assumes path_p(edges)
  returns (n: int) ensures n >= 0 {
  if (is_empty(edges)) {
    return 0;
  }
  else {if (edges_intersect(edge0, first(edges))) {
    return 1 + crossings_count_aux(edge0, rest(edges));
  }
  else {
   return crossings_count_aux(edge0, rest(edges));
  }}
}
```

```
function crossings_count_aux_5
                (edge0:edge,vertices:seq<point>,count_odd:bool)
  assumes (points2_p(vertices) && path_p(path(vertices)))
  returns (b:bool) {
  if (is_empty(vertices) || is_empty(rest(vertices))) {
     return count_odd;
     }
  else {
     return crossings_count_aux_2
                (edge0,rest(vertices),
                 (edge_points_intersect
                    (edge0.p1,edge0.p2,first(vertices),first(rest(vertices)))
                  ? !count_odd : count_odd));
     }
  }
```

Final ACL2 Function

```
(defun crossings_count_aux_6 (edge0 vertices count_odd)
 (declare (xargs :ruler-extenders :all
                 :guard (and (points2_p vertices)
                             (edge-P edge0))
                 :measure (len (path vertices))))
 (and (mbt (points2_p vertices))
      (if (or (not (mbt (edge-P edge0)))
              (not (consp vertices))
              (not (consp (cdr vertices))))
          count_odd
        (crossings_count_aux_6
          edge0
          (rest1 vertices)
          (if (edge_points_intersect (edge->p1$INLINE edge0)
                                     (edge->p2$INLINE edge0)
                                     (car vertices)
                                     (car (cdr vertices)))
              (not count_odd)
            count_odd)))))
```

```
function crossings_count_aux_5
            (edge0:edge,vertices:seq<point>,count_odd:bool)
  assumes (points2_p(vertices) && path_p(path(vertices)))
  returns (b:bool) {
  if (is_empty(vertices) || is_empty(rest(vertices))) {
    return count_odd;
    }
  else {
    return crossings_count_aux_2
              (edge0,rest(vertices),
               (edge_points_intersect
                 (edge0.p1,edge0.p2,first(vertices),first(rest(vertices)))
                ? !count_odd : count_odd));
    }
  }
```

Future Work

- Language Enhancement
  - User type parameterization
  - Imperative-looking constructs such as loops
  - Support for more APT transformations
- Prover Interaction
  - Hints for prover
  - Feedback for failed proofs in Syntheto terms
- Improved IDE capabilities
- Syntheto Execution
  - Ability to interactively run ACL2 code with results in Syntheto syntax
  - Generation of Java with ATJ or C code with ATC