The GL Clause Processor

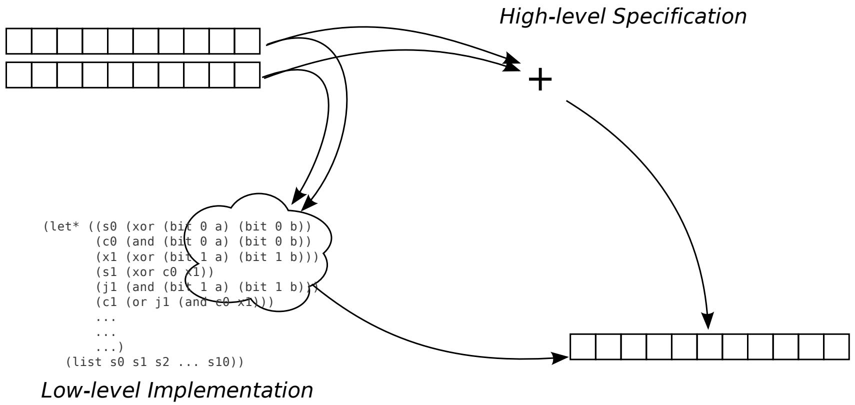
Sol Swords

September 23, 2009

Outline

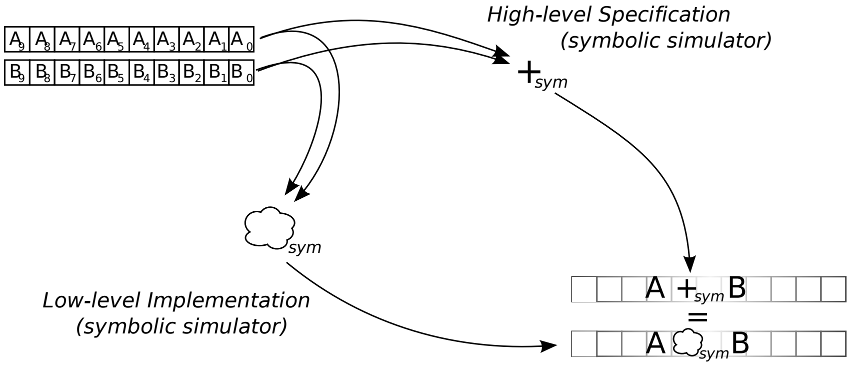
What is GL?

GL is a framework for proving difficult theorems by *symbolic simulation* using BDD-based Boolean reasoning.



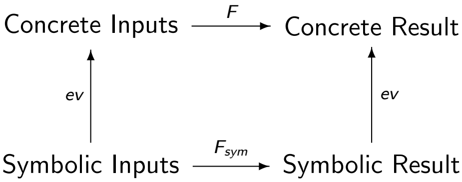
What is GL?

GL is a framework for proving difficult theorems by *symbolic simulation* using BDD-based Boolean reasoning.



Seen last time: Code transform

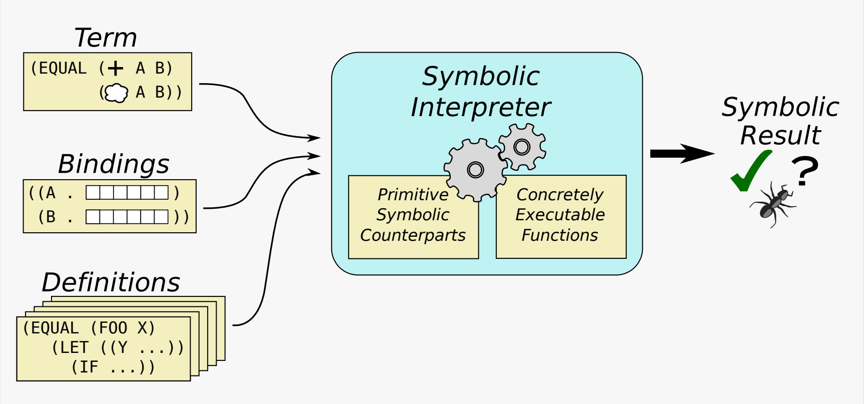
- Code transform creates *symbolic counterparts* for ACL2 functions
- Symbolic counterparts proven to correctly simulate their original functions



- Problem: Many proofs necessary, many new functions introduced, lots of theorem proving time, unreliable automation for proofs.

The new way: Verified interpreter

- Interpreter carries out symbolic execution
  - Inputs (abstractly): term, symbolic bindings, set of definitions
  - Uses existing symbolic counterparts of some "primitives"
  - Can concretely execute a fixed set of functions



Verified Interpreter

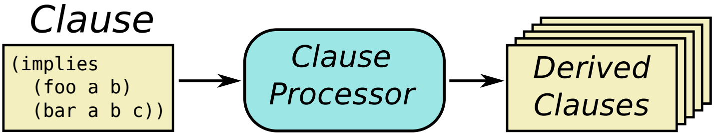
- Interpreter and primitive symbolic counterparts are verified; no need to generate and verify other symbolic counterparts.
  - Contrast with the "verifying compiler" approach.
- Performance: Sometimes slow to interpret through recursive definitions. Solution: each interpreter has
  - a fixed set of functions which it can directly execute on concrete values
  - a fixed set of symbolic counterparts which it can directly execute.

  May define new interpreters with different such sets of functions.
- Interpreter may be used in a *clause processor* to prove theorems.

What is a clause processor?

From ACL2 documentation: "A simplifier at the level of goals, where a goal is represented as a clause."

- User-defined function that takes one goal clause and produces a list of new clauses.
- Soundness contract: proving all of the new clauses suffices to prove the goal.
- May be verified (requires meta-level proof) or not (requires trust tag.)



Clause Processor Verification

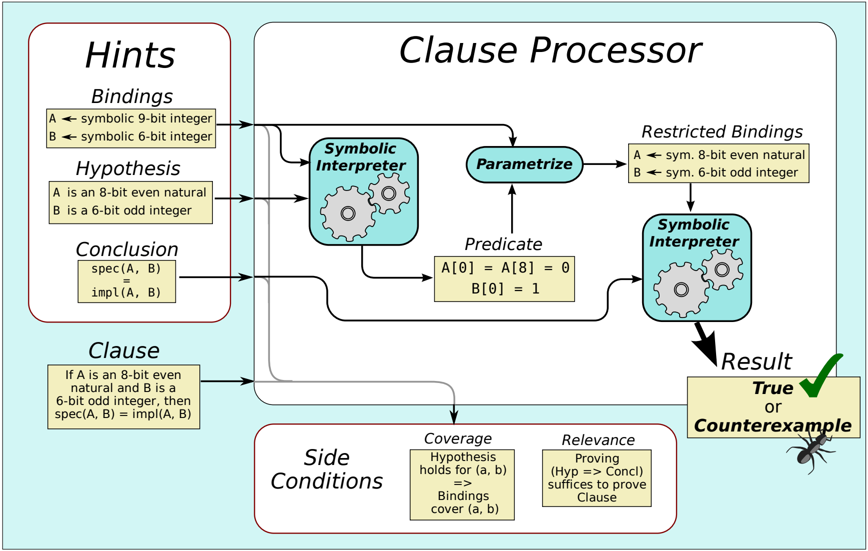
Prove correctness with respect to an *evaluator* function
`eval`(*Term*, *Alist*) → *Object* which gives a semantics to quoted terms. Example:

```
(eval '(if a (cons a 'b) 'foo) '((a . bar)))
   ⇒ (bar . b)
```

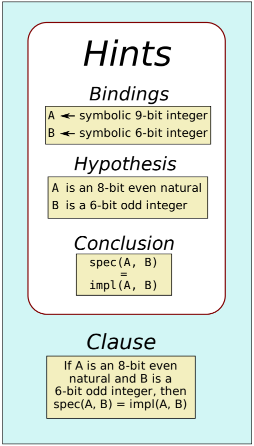
Clause processor correctness statement:

```
(implies (and ...    ;; well-formedness hyps
              (eval (conjoin-clauses
                      (clause-proc goal hints ...))
                     my-alist))
         (eval (disjoin goal) alist))
```

GL Clause Processor Flow

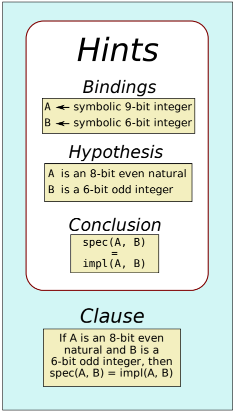


GL Clause Processor: Inputs

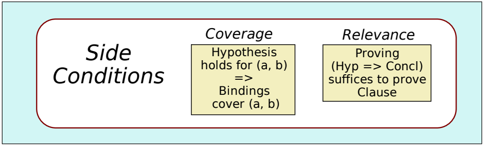


- Clause: the goal to be proved
- Hypothesis, conclusion, bindings: hints to the clause processor
- Bindings associate a symbolic object to each free variable in the clause
- Hypothesis gives "type"/"shape" constraints on variables
- Conclusion may further restrict variables (may itself be an `IMPLIES` term).

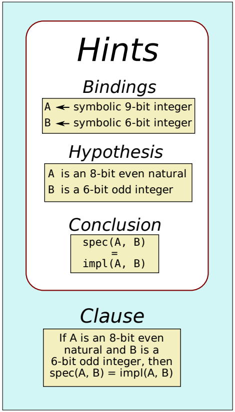
GL Clause Processor: Side Conditions



- Coverage:
  - Symbolic simulation (if successful) proves: *The conclusion holds of input vector x if x is a possible value of the symbolic inputs used in the simulation.*
  - To relate this to the hypothesis, must show: *If input vector x satisfies the hypothesis, then it is a possible value of the symbolic inputs.*



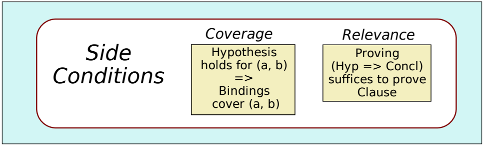
GL Clause Processor: Side Conditions



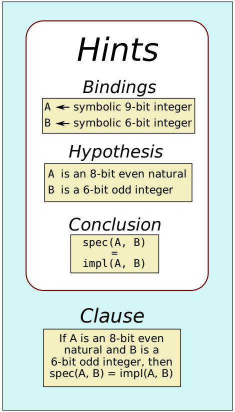
- Relevance:
  - Clause, hypothesis, conclusion are independent inputs to the clause processor
  - Symbolic simulation (with coverage) effectively proves

$$\textit{hypothesis} \Rightarrow \textit{conclusion}$$

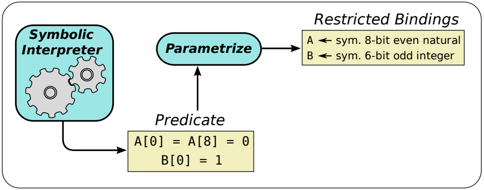
  - Therefore, prove that this implies the clause and we're done.
  - Typically trivial by construction.



GL Clause Processor: Parametrization

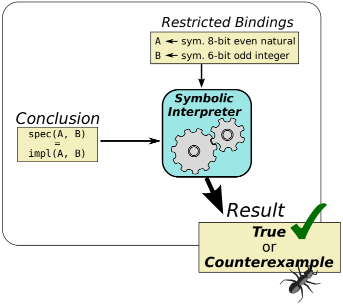


- Symbolic bindings may cover more than is accepted by the hypothesis - often better symbolic simulation performance is achievable if inputs cover less
- Symbolically simulating the hypothesis on the inputs yields a symbolic predicate
- Parametrization by that predicate yields new symbolic objects with coverage restricted to the space recognized by the hypothesis.

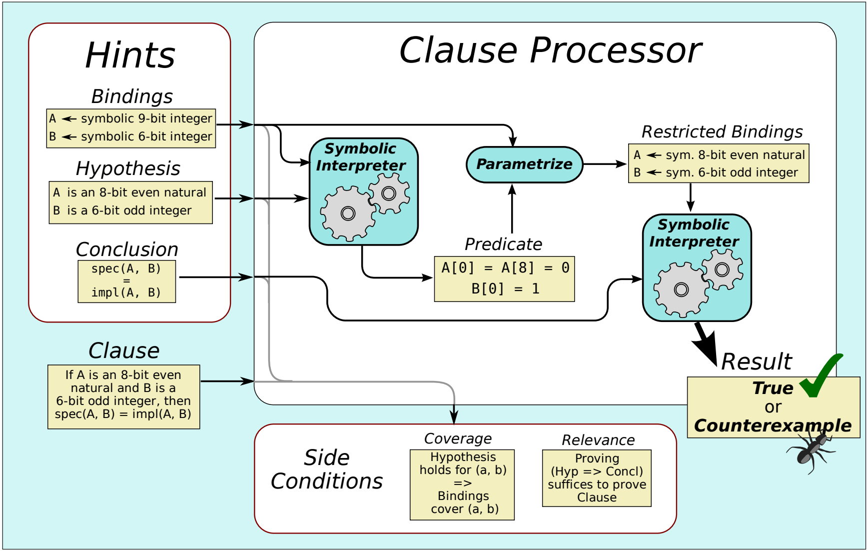


GL Clause Processor: Simulation

- Symbolically execute the conclusion to determine whether it holds on the space represented by the restricted bindings
- Result: often `T` or a set of counterexamples
- May fail or produce an ambiguous result (stack depth overrun, unimplemented primitive)



GL Clause Processor Flow: Recap



Verifying GL Clause Processors

- First, verify the generic clause processor:
  - Crux: symbolic interpreter is faithful to an evaluator's interpretation of a given term (next slide)
  - Show that given the side conditions, if the interpreter's result is always true, then the clause is a theorem
- Automate the correctness proof of new clause processors by functional instantiation of the generic one
  - `DEF-GL-CLAUSE-PROCESSOR` macro provided; introduces and verifies a new GL clause processor.

Correctness of Interpreter

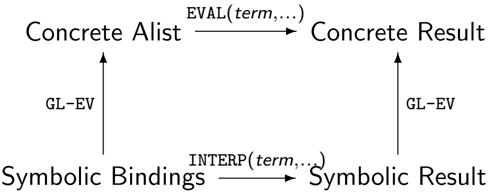
- *term*: what we're symbolically simulating
- *bindings*: association of symbolic objects to free variables of *term*
- *defs*: function definitional equations given to interpreter
- *env*: environment for symbolic object evaluation
- $\texttt{EVAL}(\mathrm{term}, \mathrm{alist}) \rightarrow \mathrm{obj}$: Evaluator for quoted ACL2 terms
- $\texttt{GL-EV}(\mathrm{sym\text{-}obj}, \mathrm{env}) \rightarrow \mathrm{obj}$: Evaluator for symbolic objects.
- $\texttt{INTERP}(\mathrm{term}, \mathrm{bindings}, \mathrm{defs}) \rightarrow \mathrm{sym\text{-}obj}$: Symbolic interpreter.

(Abstract) correctness statement:

$$\forall \mathit{term},\ \mathit{bindings},\ \mathit{defs},\ \mathit{env}\ .$$
$$(\forall \mathit{alist}\ .\ \texttt{EVAL}(\mathrm{conjoin}(\mathit{defs}), \mathit{alist}))$$
$$\Rightarrow \texttt{GL-EV}(\texttt{INTERP}(\mathit{term}, \mathit{bindings}, \mathit{defs}), \mathit{env})$$
$$= \texttt{EVAL}(\mathit{term}, \texttt{GL-EV}(\mathit{bindings}, \mathit{env}))$$

Correctness of Interpreter

- *term*: what we're symbolically simulating
- *bindings*: association of symbolic objects to free variables of *term*
- *defs*: function definitional equations given to interpreter
- *env*: environment for symbolic object evaluation
- EVAL(term, alist) → obj: Evaluator for quoted ACL2 terms
- GL-EV(sym-obj, env) → obj: Evaluator for symbolic objects.
- INTERP(term, bindings, defs) → sym-obj: Symbolic interpreter.



Assumed Definitions

- Definitions used by interpreter are not considered axiomatically true
- But we assume they are for the interpreter correctness statement
- Therefore, we are forced to emit them as output clauses from the clause processor.
- To automate their proofs, "label" each definition clause by adding a trivially true hypothesis and use computed hints to eliminate them
  - See "clause-processors/use-by-hint.lisp".

```
((not (use-these-hints
       '((:by (:definition len)))))
 (equal (len x)
        (if (consp x)
            (+ 1 (len (cdr x)))
          0)))
```

Instantiating derived clauses

```
(implies (eval (conjoin-clauses (clause-proc clause hints))
               some-alist)
         (eval (disjoin clause) original-alist))
```

- Problem: Certain derived clauses need to be instantiated with different alists or multiple times in the clause processor correctness proof
- Solution: May choose for `some-alist` any alist you want. Use a Skolem function:

  ```
  (defchoose falsifier (a) (x)
    (not (eval x a)))
  ```

  and choose:

  ```
  (falsifier (conjoin-clauses (clause-proc clause hints))).
  ```

- If *c* is a clause in the list `(clause-proc clause hints)`, then

  ```
  (eval (conjoin-clauses (clause-proc clause hints))
        (falsifier (conjoin-clauses (clause-proc clause hints))))
  ```

  implies for all a, `(eval c a)`.

Conclusions

- "Verified interpreter" rather than "verifying compiler" seems to be a win here.
  - Eliminates a lot of theorem proving
  - Little performance impact from interpretation (if you're careful)
- Challenging but surprisingly doable to verify complicated clause processors.
- Orchestration between clause processors and computed hints can be very powerful.