

A Formalization of the Yul Language and Some Verified Yul Code Transformations

Alessandro Coglio Eric McCarthy

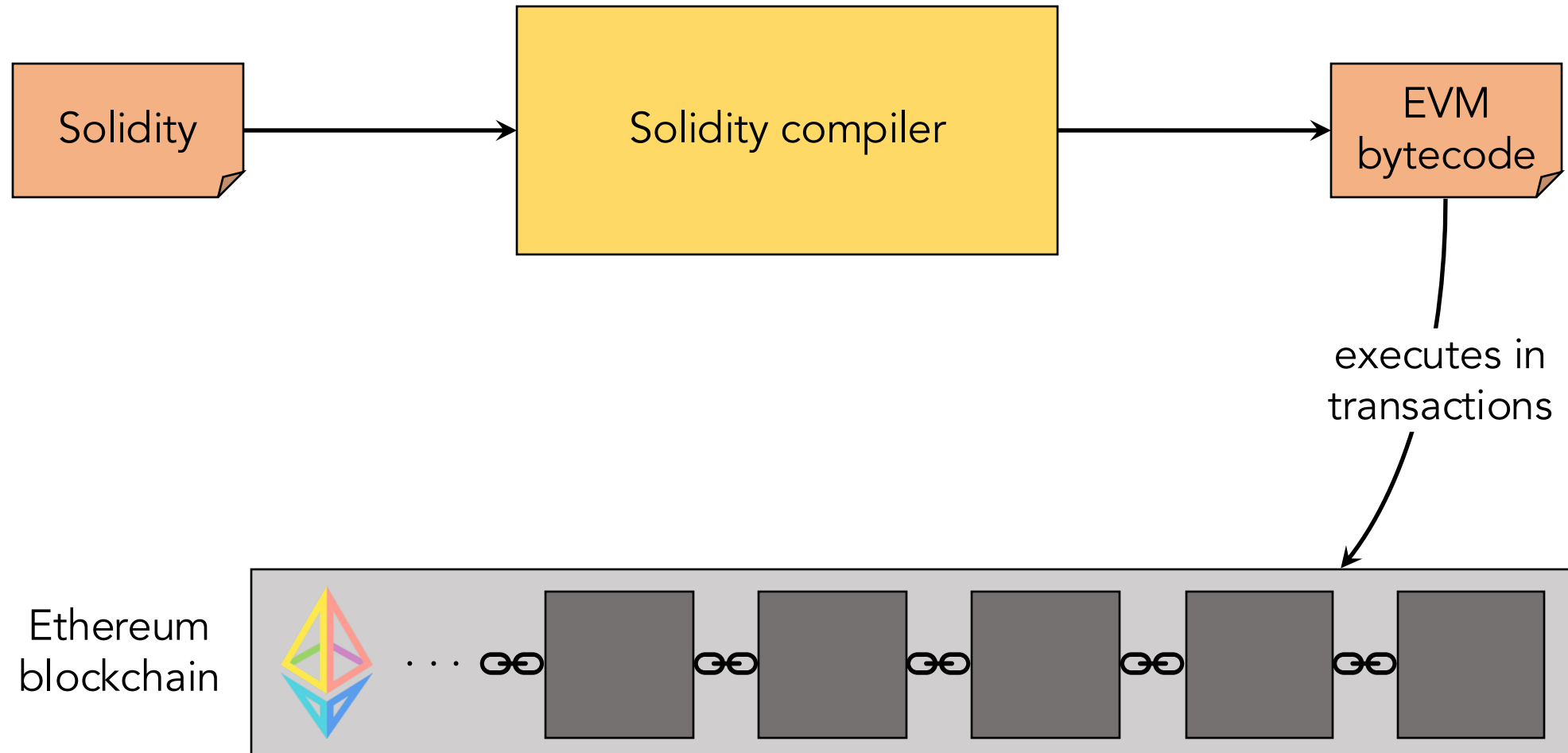


**Kestrel
Institute**

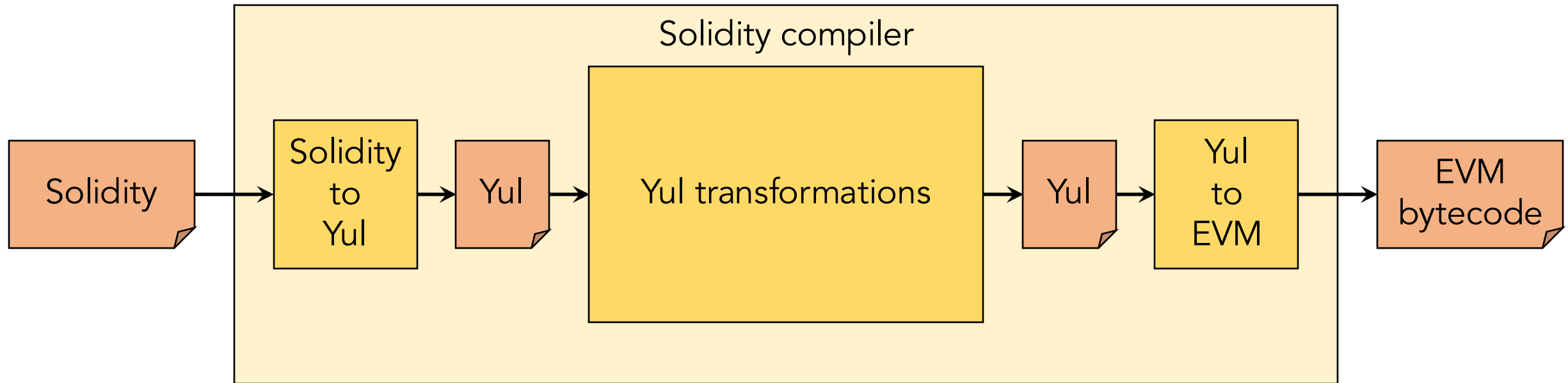


Workshop 2025

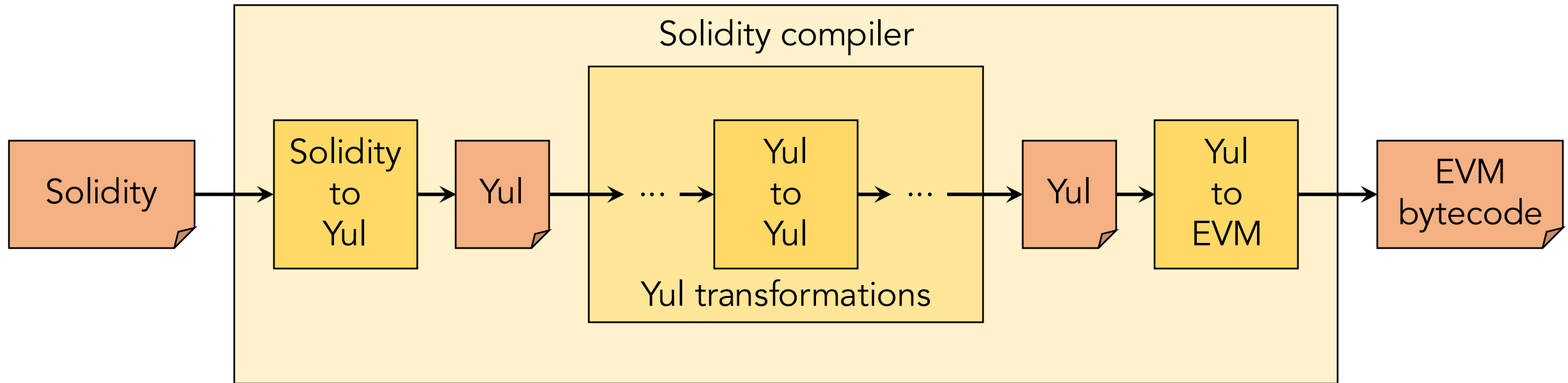
Yul is an intermediate language used in the Solidity compiler.



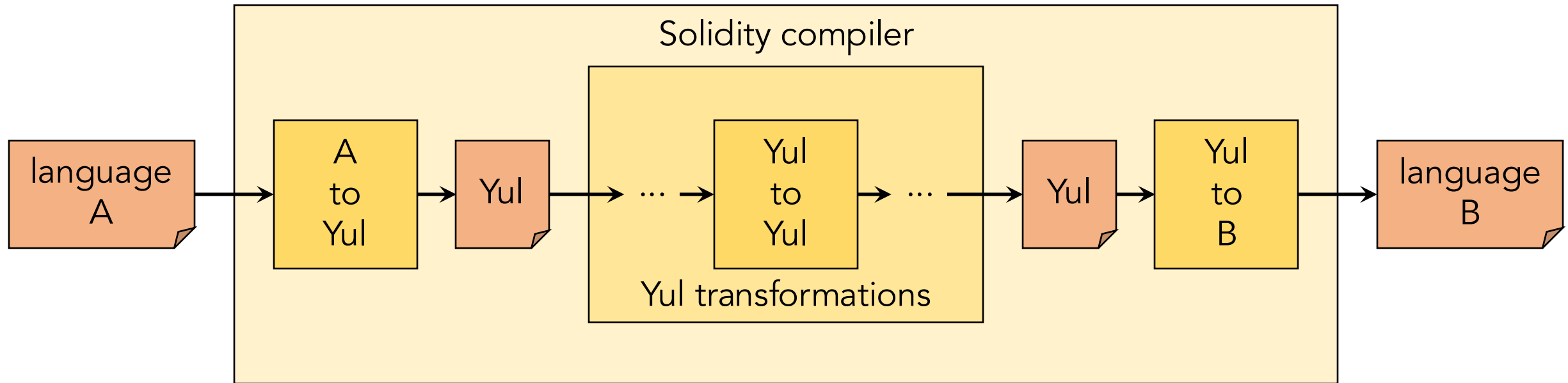
Yul is an intermediate language used in the Solidity compiler.



Yul is an intermediate language used in the Solidity compiler.



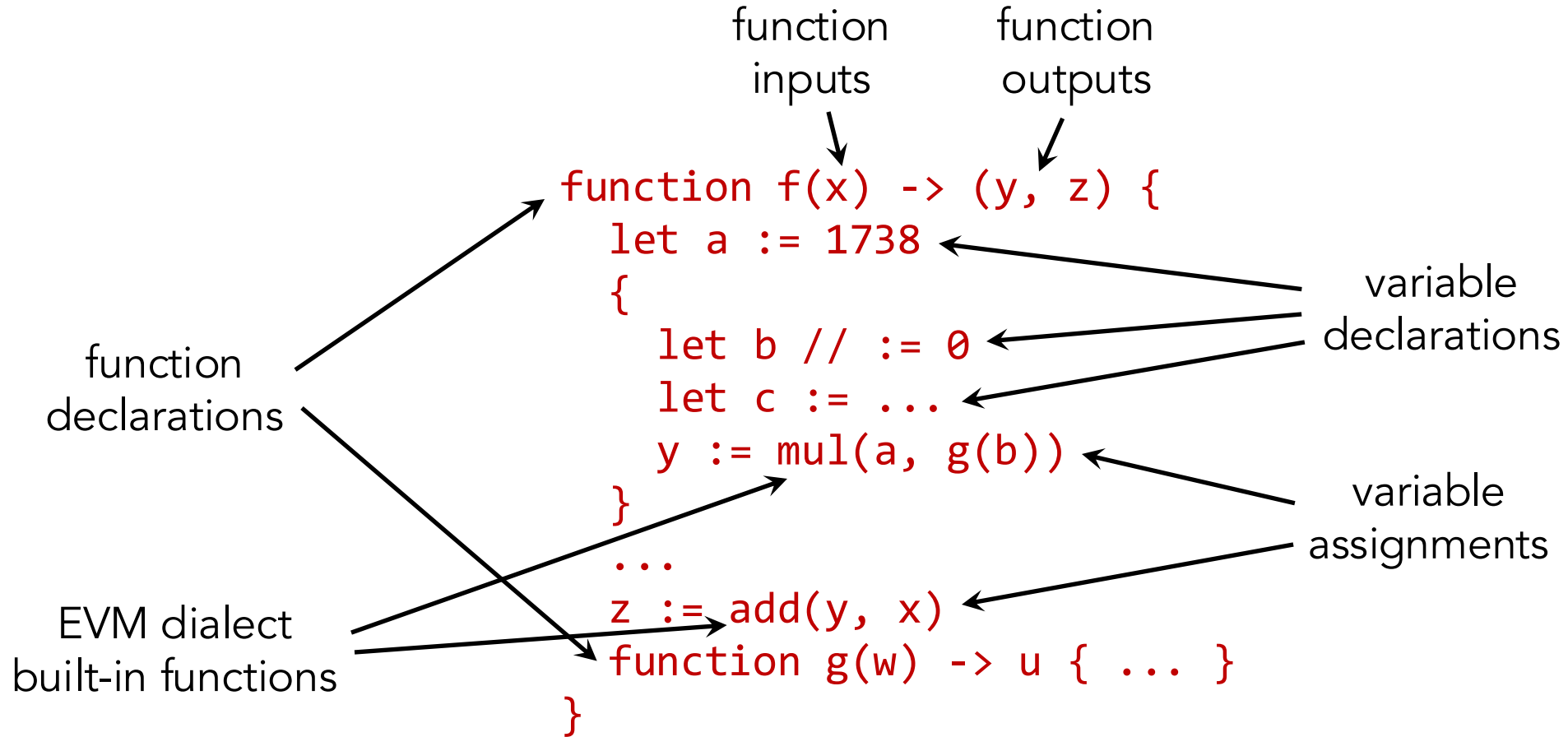
Yul is an intermediate language used in the Solidity compiler.



Yul = generic core + dialect-specific extensions

only the EVM dialect exists so far
(A = Solidity, B = EVM bytecode)

Yul is a statically typed, block-structured, imperative language.



Yul grammar from the Yul documentation:

```
Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue |
    Leave
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':' Expression )?
Assignment =
    IdentifierList ':' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
Leave = 'leave'
FunctionCall =
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_] [a-zA-Z_$0-9]*
IdentifierList = Identifier ( ',' Identifier )*
TypeName = Identifier
TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )? )*
Literal =
    (NumberLiteral | StringLiteral | TrueLiteral | FalseLiteral) ( ':' TypeName )?
NumberLiteral = HexNumber | DecimalNumber
StringLiteral = '"' ([^"r\n\\] | '\\' .)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

Yul semi-formal semantics from the Yul documentation:

```
E(G, L, <{St1, ..., Stn}>: Block) =
  let G1, L1, mode = E(G, L, St1, ..., Stn)
  let L2 be a restriction of L1 to the identifiers of L
  G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
  if n is zero:
    G, L, regular
  else:
    let G1, L1, mode = E(G, L, St1)
    if mode is regular then
      E(G1, L1, St2, ..., Stn)
    otherwise
      G1, L1, mode
E(G, L, FunctionDefinition) =
  G, L, regular
E(G, L, <let var_1, ..., var_n := rhs>: VariableDeclaration) =
  E(G, L, <var_1, ..., var_n := rhs>: Assignment)
E(G, L, <let var_1, ..., var_n>: VariableDeclaration) =
  let L1 be a copy of L where L1[$var_i] = 0 for i = 1, ..., n
  G, L1, regular
E(G, L, <var_1, ..., var_n := rhs>: Assignment) =
  let G1, L1, v1, ..., vn = E(G, L, rhs)
  let L2 be a copy of L1 where L2[$var_i] = vi for i = 1, ..., n
  G1, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
  if n >= 1:
    let G1, L1, mode = E(G, L, i1, ..., in)
    // mode has to be regular or leave due to the syntactic restrictions
    if mode is leave then
      G1, L1 restricted to variables of L, leave
    otherwise
      let G2, L2, mode = E(G1, L1, for {} condition post body)
      G2, L2 restricted to variables of L, mode
  else:
    let G1, L1, v = E(G, L, condition)
    if v is false:
      G1, L1, regular
    else:
      let G2, L2, mode = E(G1, L, body)
      if mode is break:
        G2, L2, regular
      otherwise if mode is leave:
        G2, L2, leave
      else:
        G3, L3, mode = E(G2, L2, post)
        if mode is leave:
          G3, L3, leave
        otherwise
          E(G3, L3, for {} condition post body)
```

```
E(G, L, break: BreakContinue) =
  G, L, break
E(G, L, continue: BreakContinue) =
  G, L, continue
E(G, L, leave: Leave) =
  G, L, leave
E(G, L, <if condition body>: If) =
  let G0, L0, v = E(G, L, condition)
  if v is true:
    E(G0, L0, body)
  else:
    G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
  E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
  let G0, L0, v = E(G, L, condition)
  // i = 1 .. n
  // Evaluate literals, context doesn't matter
  let _, _, v1 = E(G0, L0, l1)
  ...
  let _, _, vn = E(G0, L0, ln)
  if there exists smallest i such that vi = v:
    E(G0, L0, sti)
  else:
    E(G0, L0, st')
```

E(G, L, <name>: Identifier) =
G, L, L[\$name]

E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
G1, L1, vn = E(G, L, argn)
...
G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)
Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)
Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
be the function of name \$fname visible at the point of the call.
Let L' be a new local state such that
L'[\$parami] = vi and L'[\$reti] = 0 for all i.
Let G'', L'', mode = E(Gn, L', block)
G'', Ln, L''[\$ret1], ..., L''[\$retm]

E(G, L, l: StringLiteral) = G, L, str(l),
where str is the string evaluation function,
which for the EVM dialect is defined in the section 'Literals' above

E(G, L, n: HexNumber) = G, L, hex(n)
where hex is the hexadecimal evaluation function,
which turns a sequence of hexadecimal digits into their big endian value

E(G, L, n: DecimalNumber) = G, L, dec(n),
where dec is the decimal evaluation function,
which turns a sequence of decimal digits into their big endian value

List of Yul transformations from the Yul documentation:

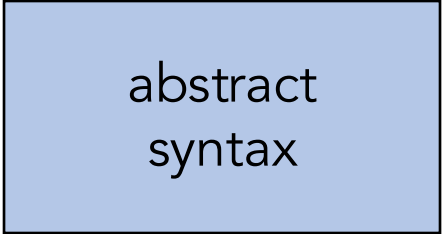
The exact sequence of Yul transformations can be customized.

Not all sequences are valid: some transformations expect the code to be in certain forms, as resulting from applying previous transformations

Abbreviation	Full name		
f	BlockFlattener	i	FullInliner
l	CircularReferencesPruner	g	FunctionGrouper
c	CommonSubexpressionEliminator	h	FunctionHoister
C	ConditionalSimplifier	F	FunctionSpecializer
U	ConditionalUnsimplifier	T	LiteralRematerialiser
n	ControlFlowSimplifier	L	LoadResolver
D	DeadCodeEliminator	M	LoopInvariantCodeMotion
E	EqualStoreEliminator	m	Rematerialiser
v	EquivalentFunctionCombiner	V	SSAReverser
e	ExpressionInliner	a	SSATransform
j	ExpressionJoiner	t	StructuralSimplifier
s	ExpressionSimplifier	r	UnusedAssignEliminator
x	ExpressionSplitter	p	UnusedFunctionParameterPruner
I	ForLoopConditionIntoBody	S	UnusedStoreEliminator
O	ForLoopConditionOutOfBody	u	UnusedPruner
o	ForLoopInitRewriter	d	VarDeclInitializer

```
solc --optimize --ir-optimized  
--yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul:fDnTOcmu'
```

Our Yul development in ACL2.



abstract
syntax

Our abstract syntax of Yul consists of algebraic fixtypes.

```
(fty::deftagsum statement
  (:block ((get block)))
  (:variable-single ((name identifier) (init expression-option)))
  (:variable-multi ((names identifier-list) (init funcall-optionp)))
  (:assign-single ((target path) (value expression)))
  (:assign-multi ((targets path-list) (value funcall)))
  (:funcall ((get funcall)))
  (:if ((test expression) (body block)))
  (:switch ((target expression) (cases swcase-list) (default block-option)))
  (:for ((init block) (test expression) (update block) (body block)))
  (:break ())
  (:continue ())
  (:leave ())
  (:fundef ((get fundef)))
  :pred statementp)
```

Our Yul development in ACL2.



Our concrete syntax of Yul consists of an ABNF grammar transcribed from the documentation.

```
statement = block / variable-declaration / assignment / function-call  
           / if-statement / switch-statement / for-statement  
           / %s"leave" / %s"break" / %s"continue" / function-definition
```

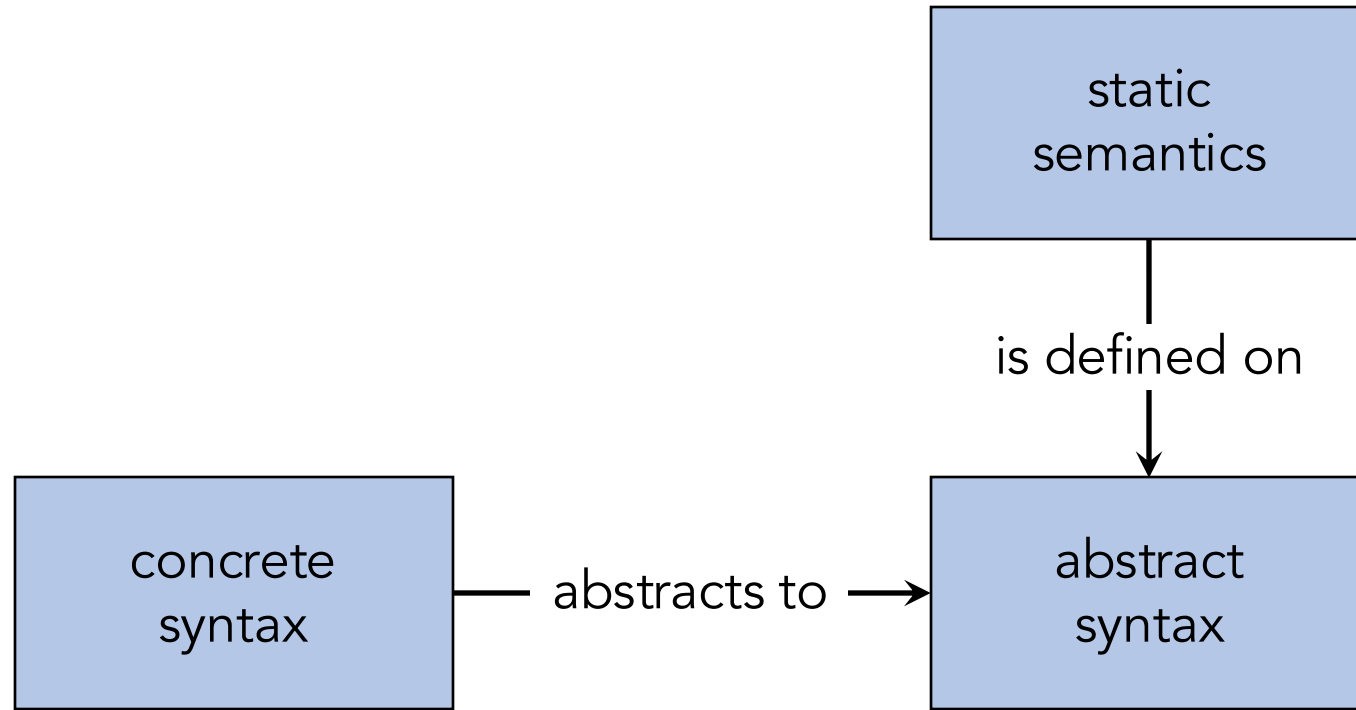
Which is turned into an ACL2 formal representation via our verified ABNF grammar parser.

We have not yet formalized:

- Extra-grammatical syntactic restrictions.
- Mapping from concrete to abstract syntax.

But we have developed an executable Yul parser, which provides a low-level specification of the extra-grammatical syntactic restrictions and the mapping from concrete to abstract syntax.

Our Yul development in ACL2.



Our static semantics of Yul consists of predicates that perform compiler-like checks, described informally in the Yul documentation:

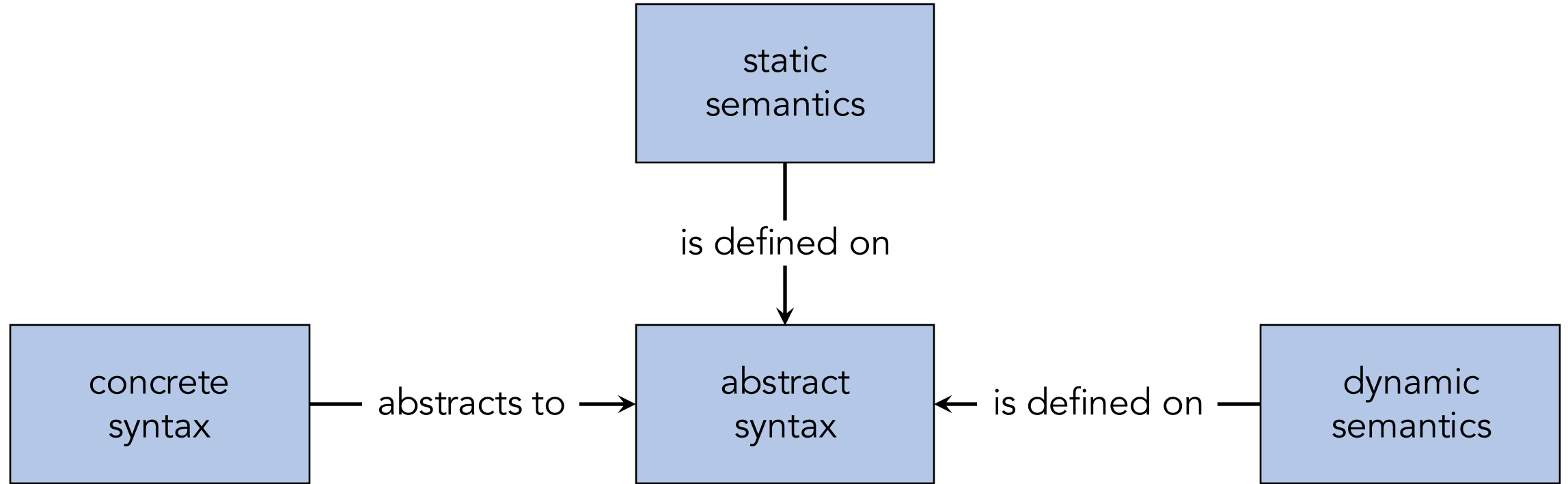
- Each referenced variable or function is in scope (Yul's scoping rules are a bit tricky).
- Each function is called with the right number of arguments.
- ...

```
(define check-safe-statement ((stmt statementp)
                             (varset identifier-setp) ← variables in scope
                             (funtab funtablep)) ← functions in scope
:returns (varsmodes vars+modes-resultp)
(statement-case
 stmt
 :if
 (b* (((okf results) (check-safe-expression stmt.test varset funtab))
      ((unless (= results 1)) ...) ; error
      ((okf modes) (check-safe-block stmt.body varset funtab)))
  (make-vars+modes :vars (identifier-set-fix varset)
                   :modes (set::insert (mode-regular) modes)))
...)) ← handle other kinds of statements
```

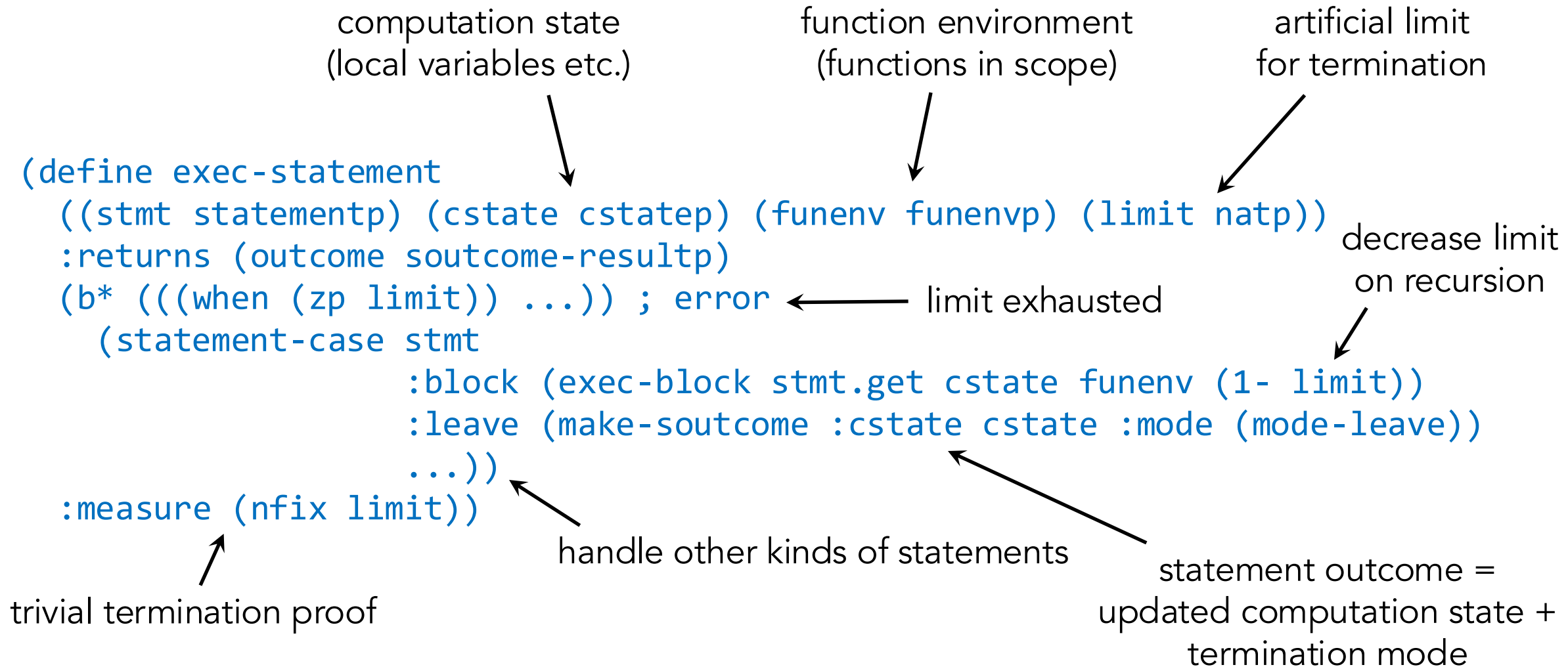
updated variables in scope ←

modes of termination ←

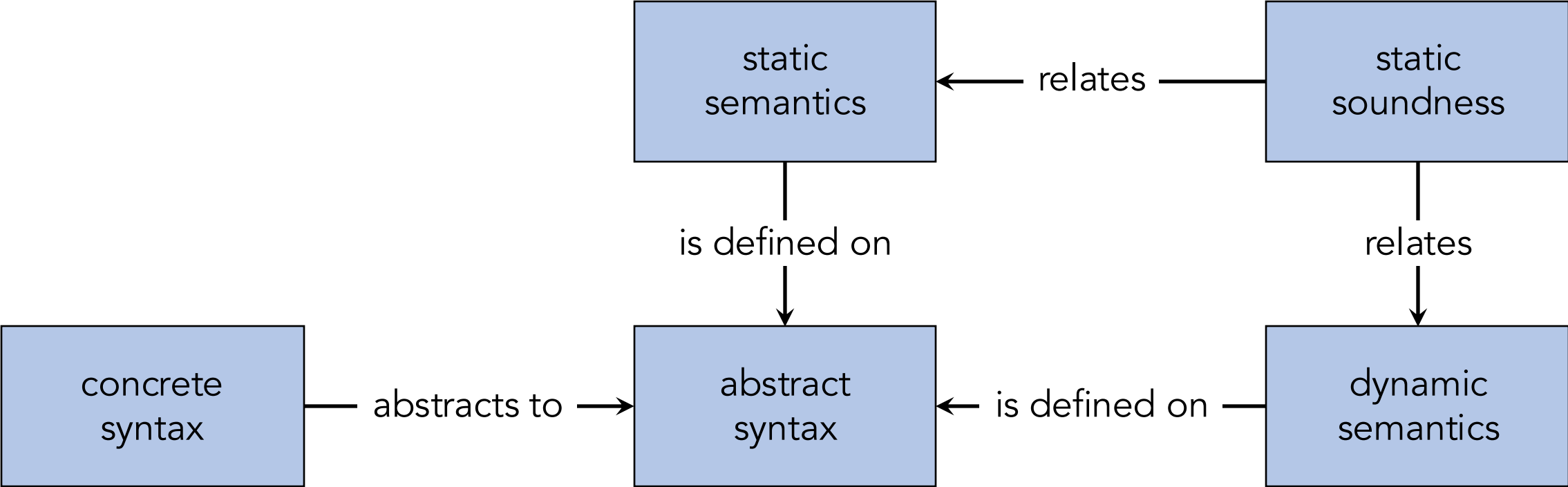
Our Yul development in ACL2.



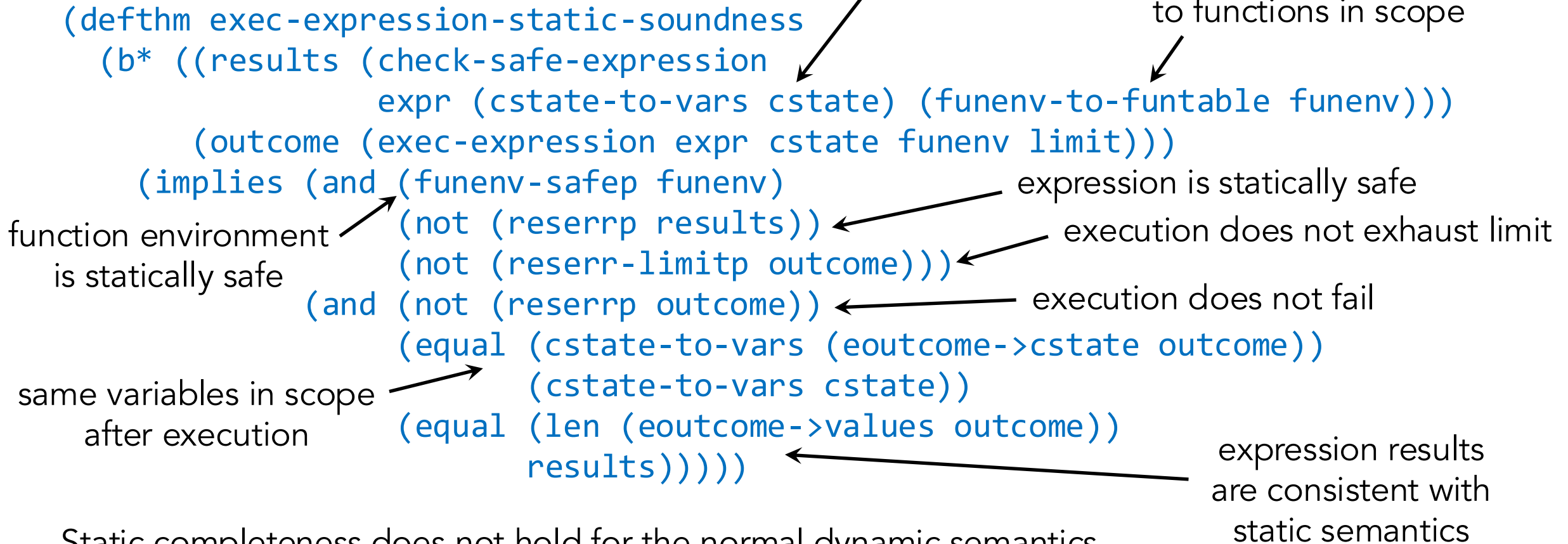
Our dynamic semantics of Yul consists of a big-step defensive interpreter, based on the one in the Yul documentation.



Our Yul development in ACL2.



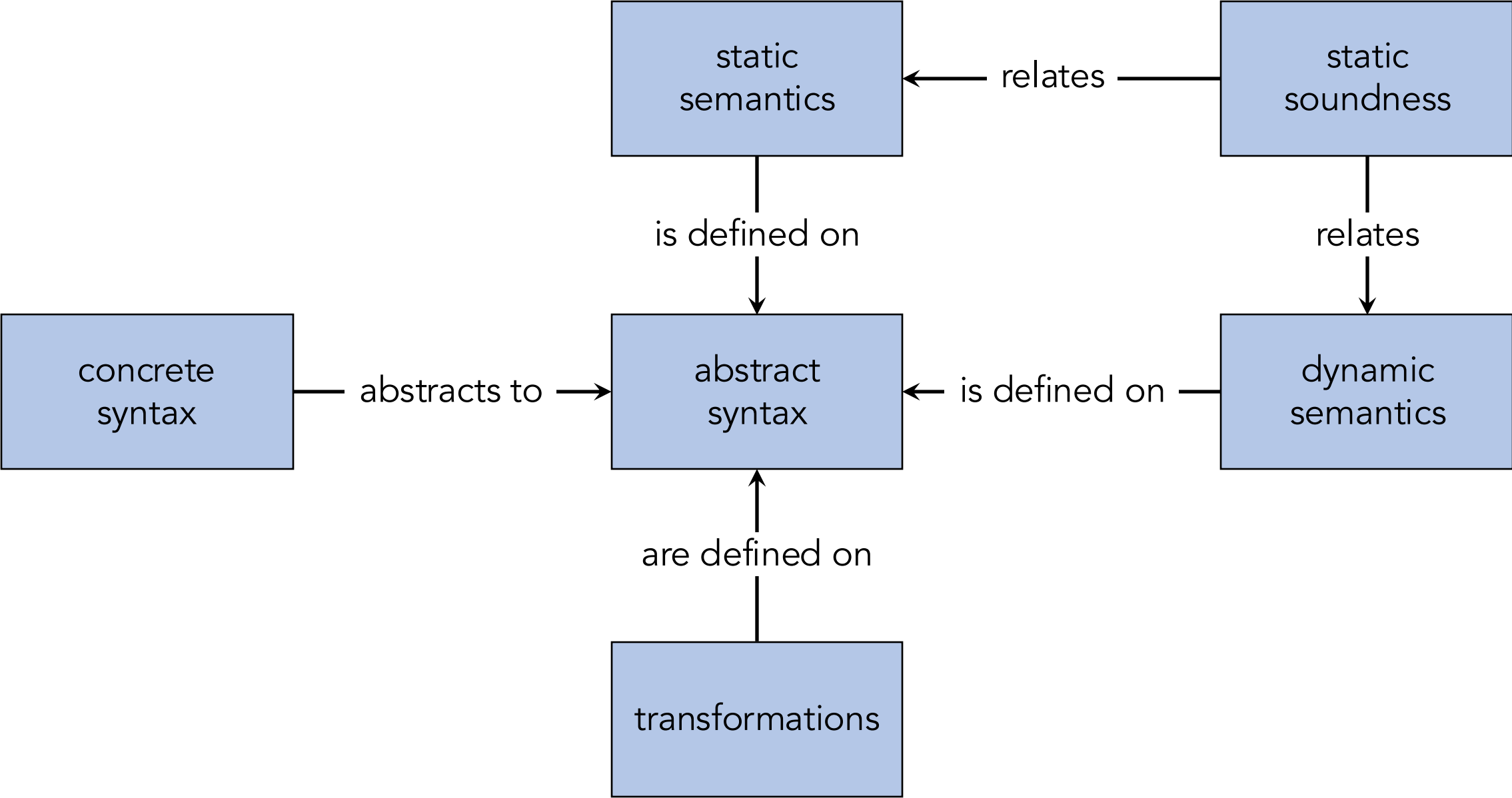
Our static soundness theorems says that:
if the static semantic checks succeed,
then the corresponding dynamic checks succeed.



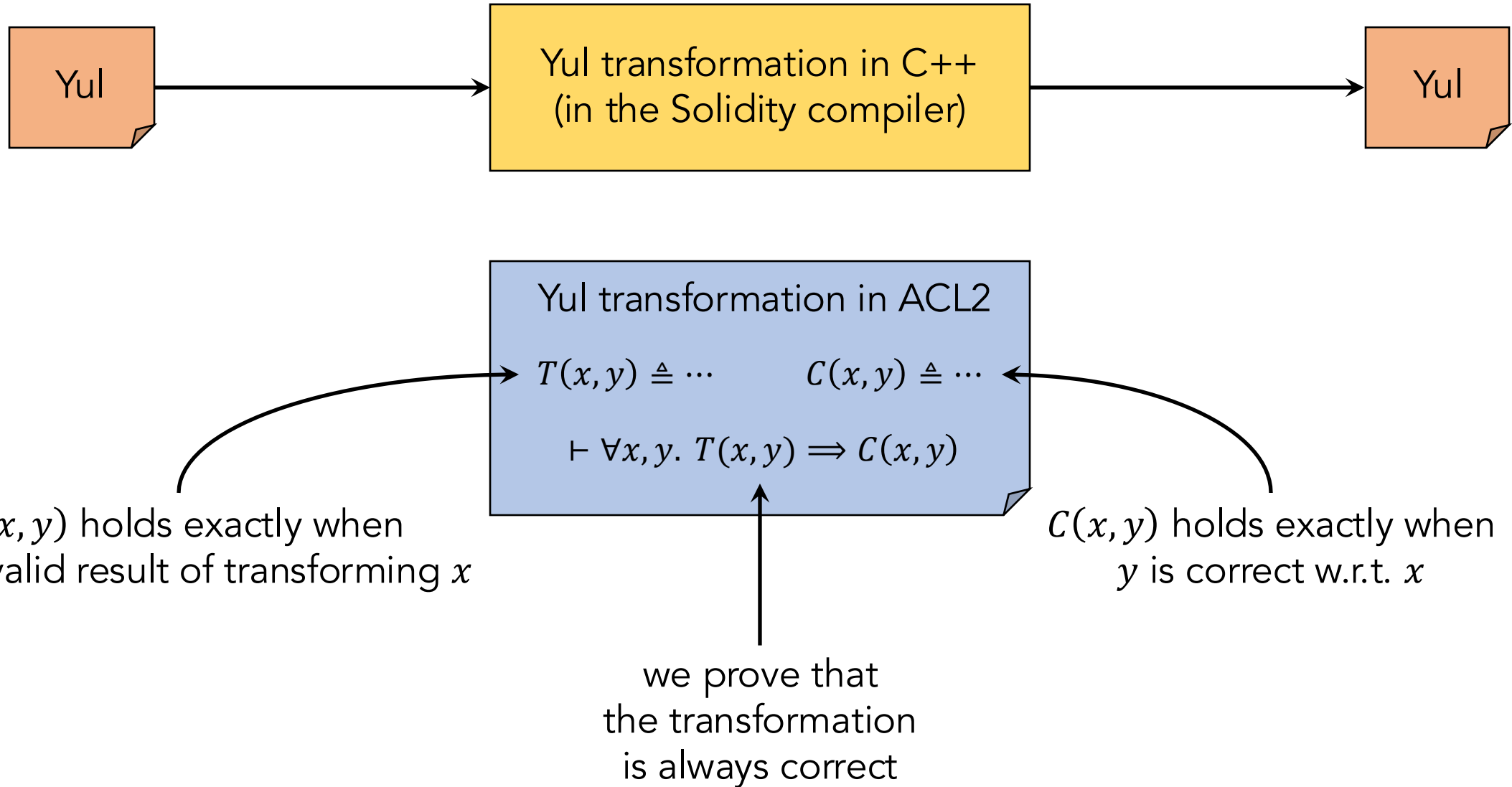
Static completeness does not hold for the normal dynamic semantics.

It should hold for an extended dynamic semantics that
nondeterministically chooses any branch regardless of the test value.

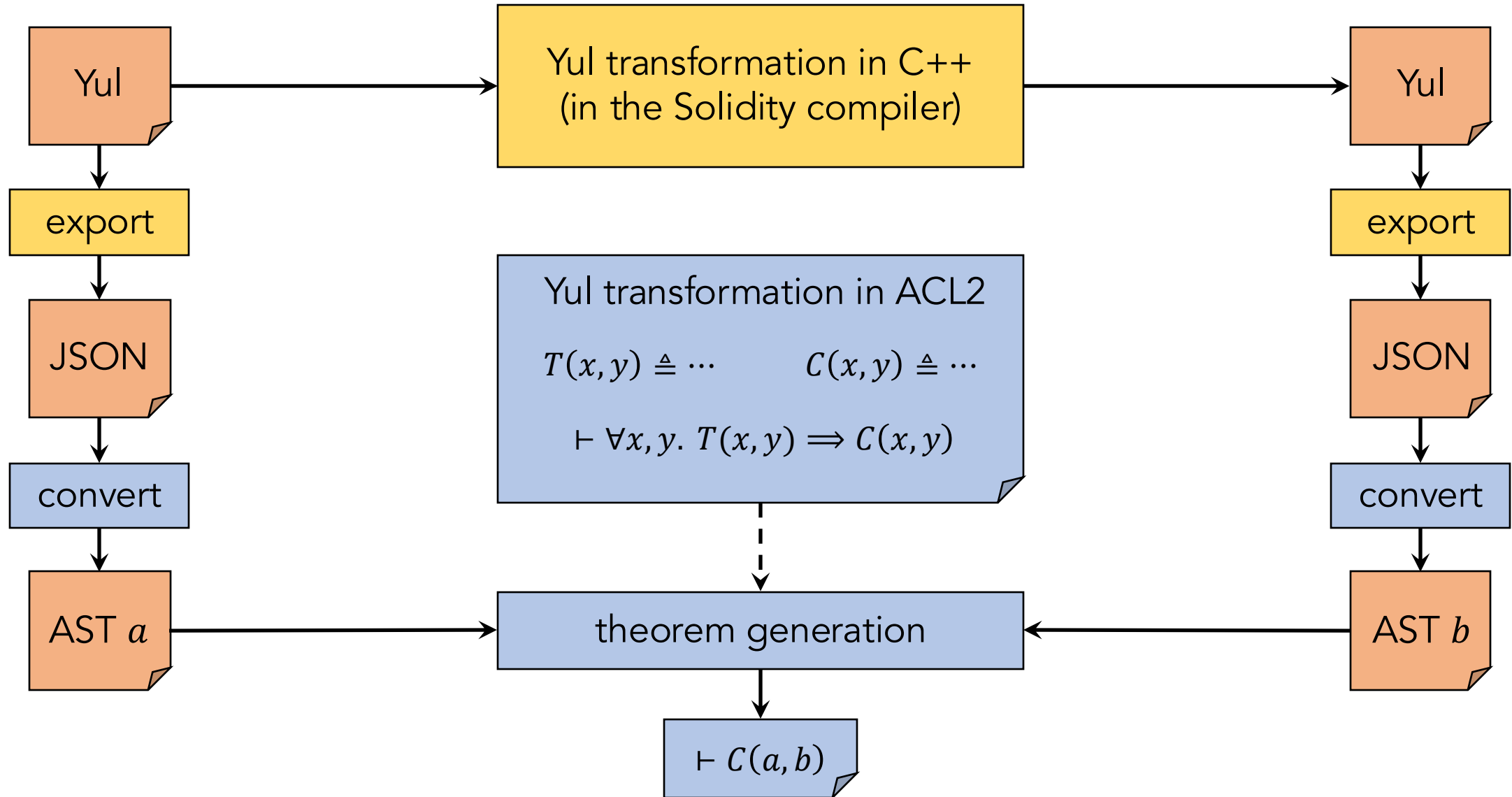
Our Yul development in ACL2.



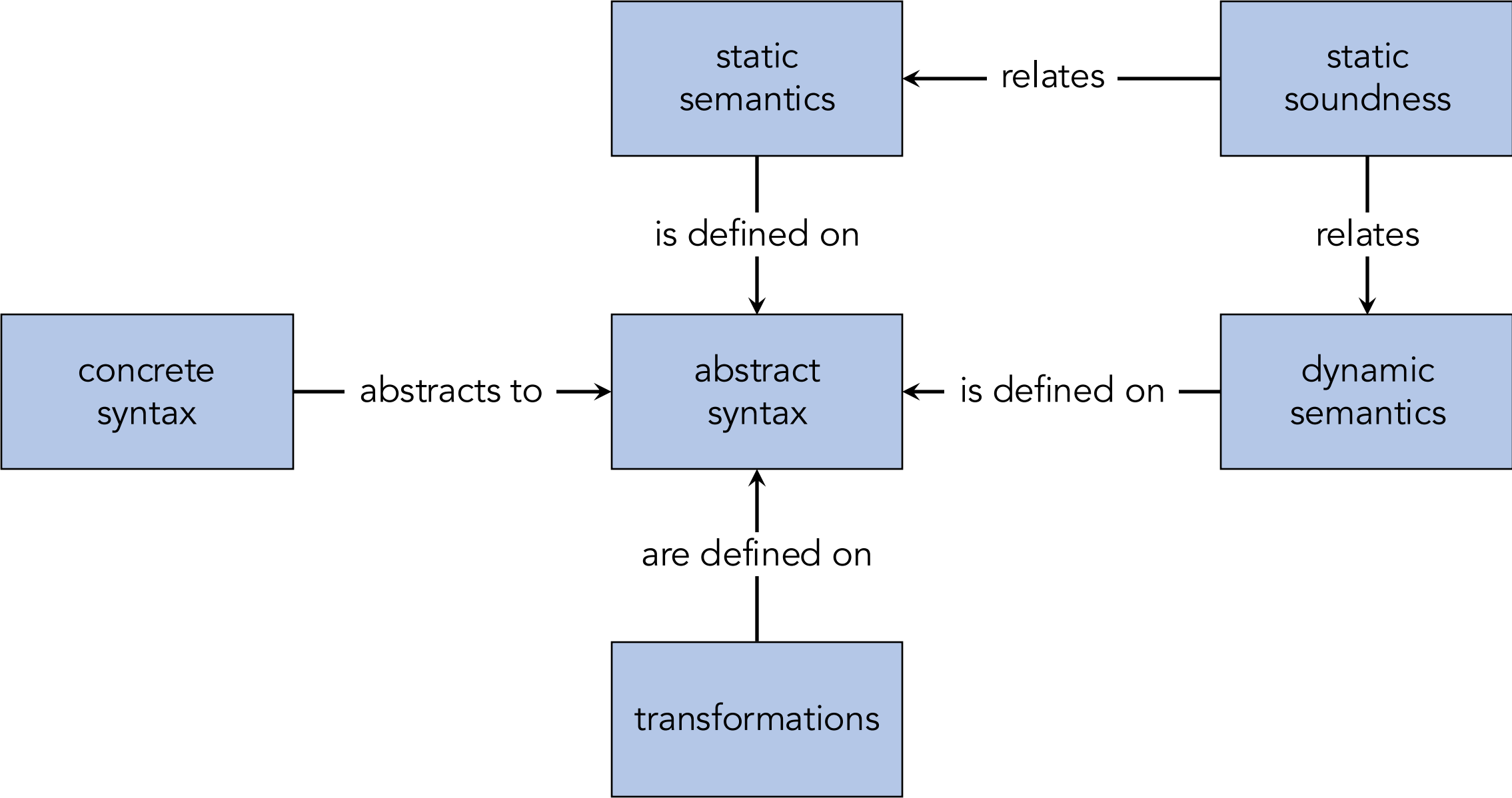
How do we ensure that the Yul transformations in the Solidity compiler are correct?



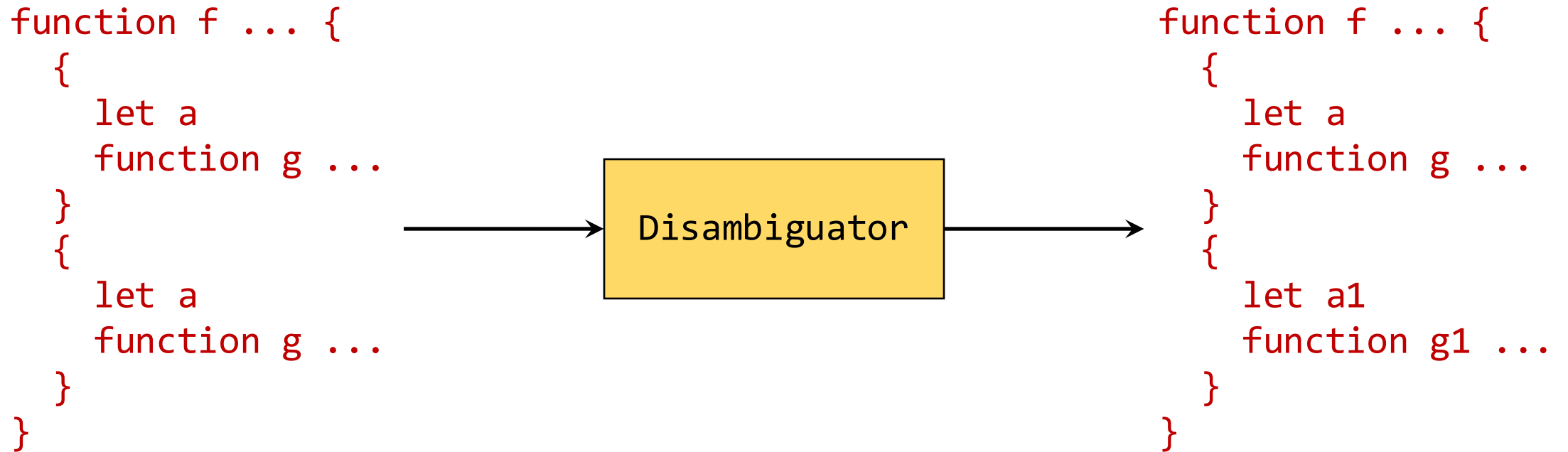
How do we ensure that the Yul transformations in the Solidity compiler are correct?



Our Yul development in ACL2.



We have formalized and verified a few transformations.
The most interesting one is Disambiguator.



It renames variables and functions so that they are globally unique,
to facilitate subsequent transformations.

There are many ways to rename variables and functions apart.

We formalized **Disambiguator** as a relation between old and new code, which holds when old and new code differ only in variable and function names, and the variables and functions in the new code have unique names.

Our formalization of **Disambiguator** consists of four predicates:

- Variable renaming over old and new code.
- Function renaming over old and new code.
- Unique variable names over new code.
- Unique function names over new code.

This “isolates” our formalization and proofs from peculiarities of, and changes to, the variable and function renaming algorithm used in the Solidity compiler.

This is a part of the variable renaming transformation.

alist with unique keys and values

```
(define statement-renamevar ((old statementp) (new statementp) (ren renamingp))
  :returns (new-ren renaming-resultp)
  (statement-case
    old
    :block
    (b* (((unless (statement-case new :block)) ...) ; return error
          ((statement-block new) new)
          ((okf &) (block-renamevar old.get new.get ren))))
      (renaming-fix ren))
    :variable-single
    (b* (((unless (statement-case new :variable-single)) ...) ; return error
          ((statement-variable-single new) new)
          ((okf &) (expression-option-renamevar old.init new.init ren))))
      (add-var-to-var-renaming old.name new.name ren))
    ...))
```

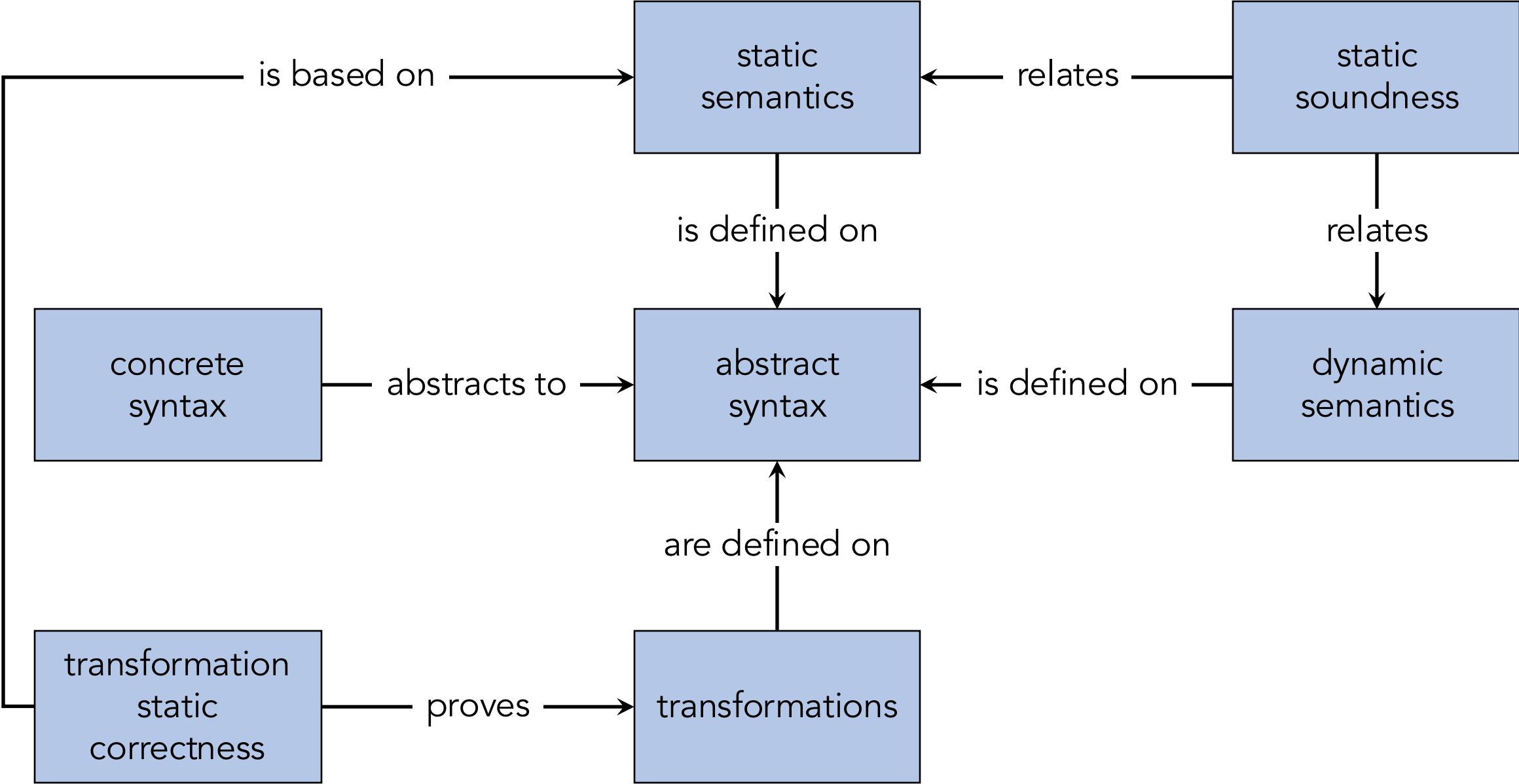
possibly updated alist, or error

same renaming after block

extend renaming

handle other kinds of statements

Our Yul development in ACL2.



We proved that the variable renaming transformation preserves the static semantics, e.g. if the old statement is statically safe, so is the new statement, and the results agree.

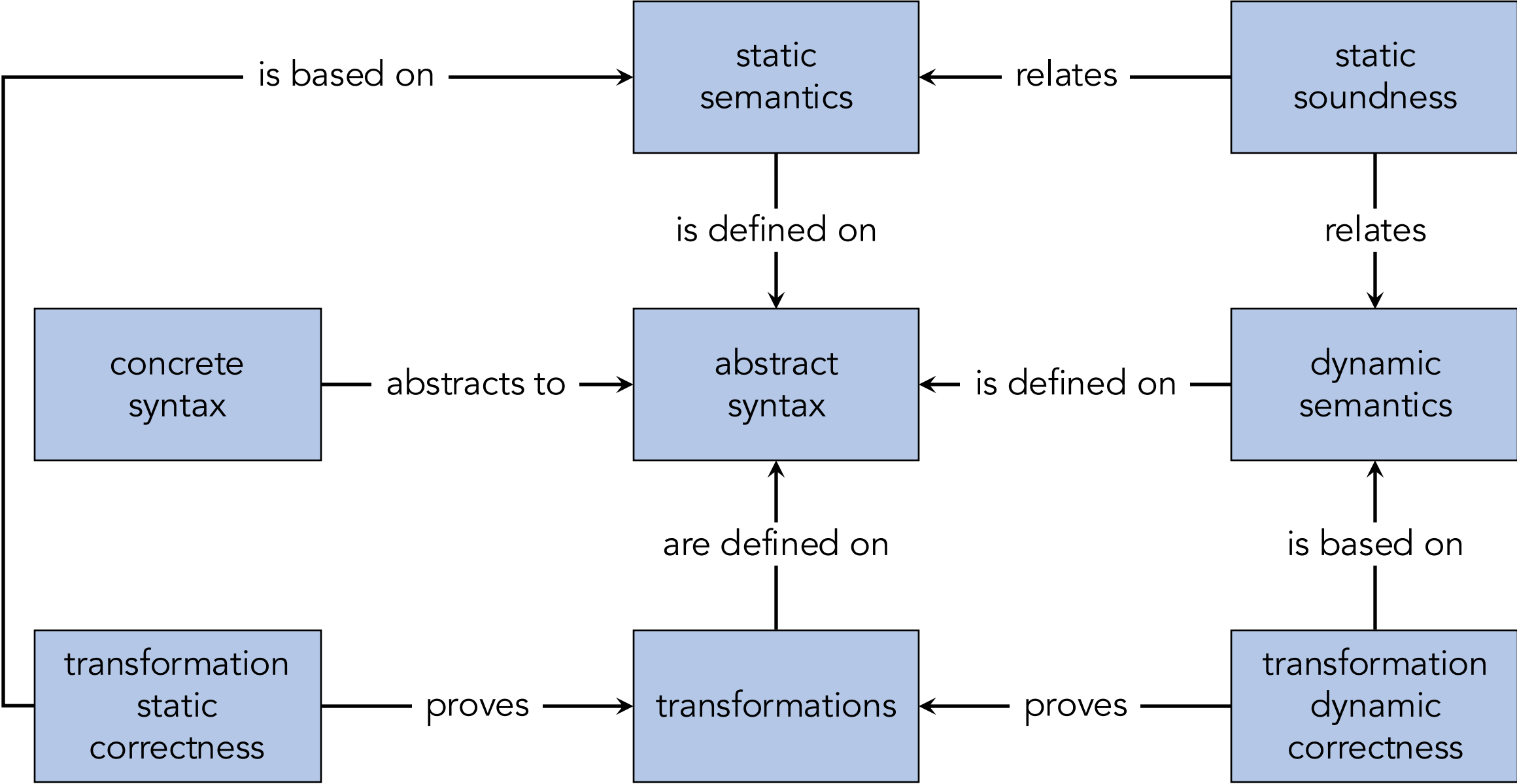
```
(defthm check-safe-statement-when-renamevar
  (b* ((ren1 (statement-renamevar stmt-old stmt-new ren))
      (varmodes-old (check-safe-statement stmt-old (varset-old ren) funtab))
      (varmodes-new (check-safe-statement stmt-new (varset-new ren) funtab)))
    (implies (and (not (reserrp ren1))
                  (not (reserrp varmodes-old)))
              (and (not (reserrp varmodes-new))
                    (equal (vars+modes->vars varmodes-old)
                           (varset-old ren1))
                    (equal (vars+modes->vars varmodes-new)
                           (varset-new ren1))
                    (equal (vars+modes->modes varmodes-old)
                           (vars+modes->modes varmodes-new)))))))
```

old and new statements
related by variable renaming

old statement is safe
new statement is safe

results agree

Our Yul development in ACL2.



We proved that the variable renaming transformation preserves the dynamic semantics, e.g. if the old and new statement execute without error, the results agree.

```
(defthm exec-statement-when-renamevar
  (b* ((ren1 (statement-renamevar stmt-old stmt-new ren)))
    (implies (and (not (reserrp ren1))
      (cstate-renamevarp cstate-old cstate-new ren)
      (funenv-renamevarp funenv-old funenv-new))
      (b* ((outcome-old
        (exec-statement stmt-old cstate-old funenv-old limit))
        (outcome-new
        (exec-statement stmt-new cstate-new funenv-new limit)))
        (implies (and (not (reserr-nonlimitp outcome-old))
          (not (reserr-nonlimitp outcome-new)))
          (soutcome-result-renamevarp outcome-old
            outcome-new
            ren1)))))))
```

old and new statements are
related by variable renaming

variable renaming on
computation states
and functions

old and new statement
execute without error

results agree

We proved that the variable renaming transformation preserves the dynamic semantics, e.g. if the old and new statement execute without error, the results agree.

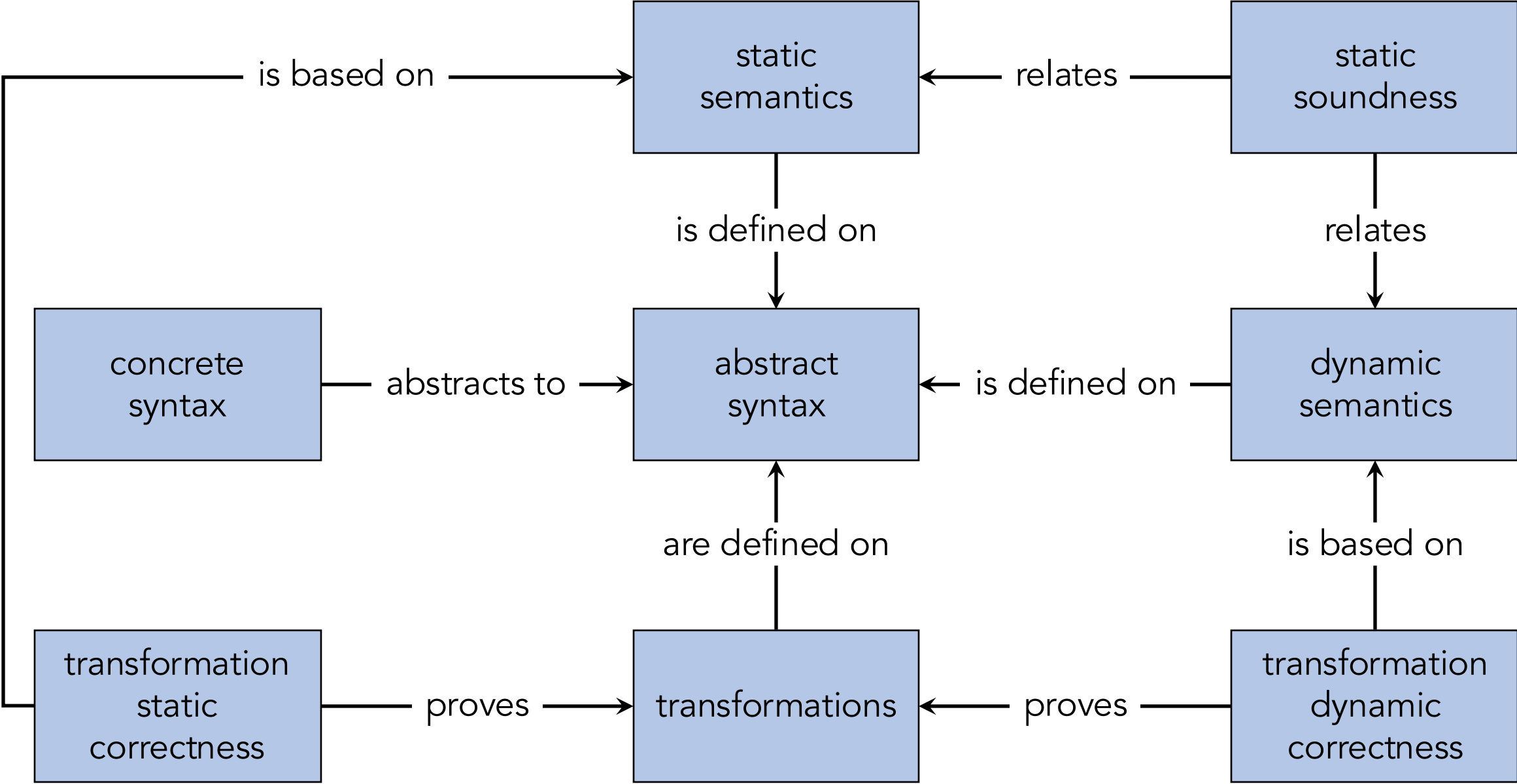
Should this be a conclusion instead of a hypothesis?

```
(not (reserr-nonlimitp (exec-statement stmt-new cstate-new funenv-new limit)))
```

Yes, but we already proved that the transformation preserves the static semantics, and we know that the static semantics guarantees the absence of execution errors, because of our general static soundness theorem.

So we can make that a hypothesis, which slightly simplifies the proof, and combine that with the other theorems to obtain the desired formulation.

Our Yul development in ACL2.



Status:

- Our model of generic Yul is complete, but it could be parameterized better over the dialect.
- Our model of the EVM dialect is quite minimal.
- Our formalization and verification of Yul transformations has just scratched the surface.

Outlook:

- Modeling the EVM dialect is doable but would take significant effort.
- Formalizing and verifying all the Yul transformations is doable but would take significant effort. Some are EVM-dialect-specific.

Remarks:

- Even seemingly simple transformations take effort to verify.
- The proofs were not difficult but a bit laborious.
- Computed hints with `(ac12::occur-1st '(ac12::flag-is '...) clause)` conditions were useful to apply different hints to different cases of the large induction proofs.