A Proof-Generating C Code Generator for ACL2 Based on a Shallow Embedding of C in ACL2

Alessandro Coglio

Kestrel Institute

ACL2 Workshop 2022

proof-generating

code generator

shallow embedding

shallow embedding of C in ACL2 = representation of C code as ACL2 code

Some example C code…

```
int f(int x, int y, int z) {
    return (x + y) * (z – 3);
}
```

… and its representation as ACL2 code.

```
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard (and (sintp |x|)
                              (sintp |y|)
                              (sintp |z|)
                              ...)))
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```

shallow embedding of C in ACL2 = representation of C code as ACL2 code

The identifiers `f`, `x`, `y`, `z` are represented
by the symbols `|f|`, `|x|`, `|y|`, `|z|`.
The `symbol-name` is the identifier.
Note that the symbols `f`, `x`, `y`, `z`
would represent the identifiers `F`, `X`, `Y`, `Z`.

The type (`signed`) `int` is represented
by the predicate `sintp`,
which recognizes ACL2 integers
in the range of the type `int`
tagged with a type indication,
e.g. `(:sint 8)` and `(:sint -3)`.

shallow embedding of C in ACL2 = representation of C code as ACL2 code

The operation `+` on `int`s is represented
by the function `add-sint-sint`.

```
(define add-sint-sint ((x sintp) (y sintp))
  :guard (add-sint-sint-okp x y)
  (sint (+ (sint->get x) (sint->get y))))
```

The guard ensures that the result
is well-defined according to C18.

```
(define add-sint-sint-okp ((x sintp) (y sintp))
  (sint-integerp (+ (sint->get x) (sint->get y))))
```

shallow embedding of C in ACL2 = representation of C code as ACL2 code

```c
int f(int x, int y, int z) {
    return (x + y) * (z - 3);
}
```

```lisp
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard (and (sintp |x|)
                              (sintp |y|)
                              (sintp |z|)
                              (<= ... (sint->get |x|))
                              (<= (sint->get |x|) ...)
                              (<= ... (sint->get |y|))
                              (<= (sint->get |y|) ...)
                              (<= ... (sint->get |z|))
                              (<= (sint->get |z|) ...))))
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```

the parameters must be in ranges such that all the operations are well-defined

shallow embedding of C in ACL2 = representation of C code as ACL2 code

```c
int f(int x, int y, int z) {
    return (x + y) * (z - 3);
}
```

```lisp
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard ...))
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```

shallow embedding of C in ACL2 = representation of C code as ACL2 code

The constant `3` in base 10 of type `int`
is represented by `(sint-dec-const 3)`.

The return type `int` is determined by
the fact that `|f|` returns `mul-sint-sint`.

shallow embedding of C in ACL2 = representation of C code as ACL2 code

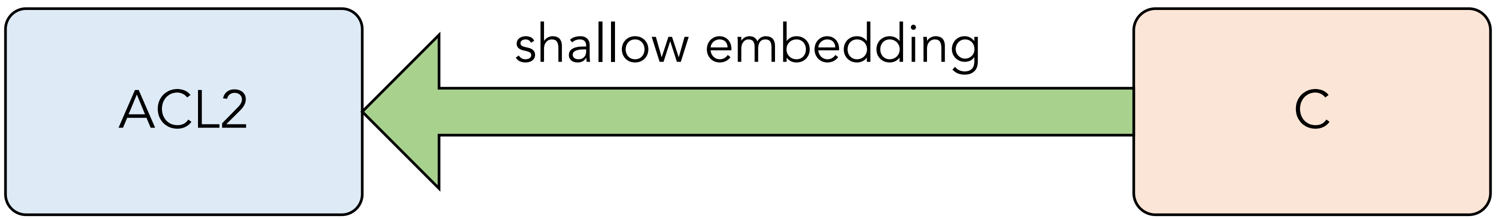
```c
int f(int x, int y, int z) {
    return (x + y) * (z - 3);
}
```

shallow embedding

```lisp
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard ...))
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```

shallow embedding of C in ACL2 = representation of C code as ACL2 code

This shallow embedding of C in ACL2
has more features than the example shows:
other integer types, arrays, structures,
local variables, conditionals, loops, etc.



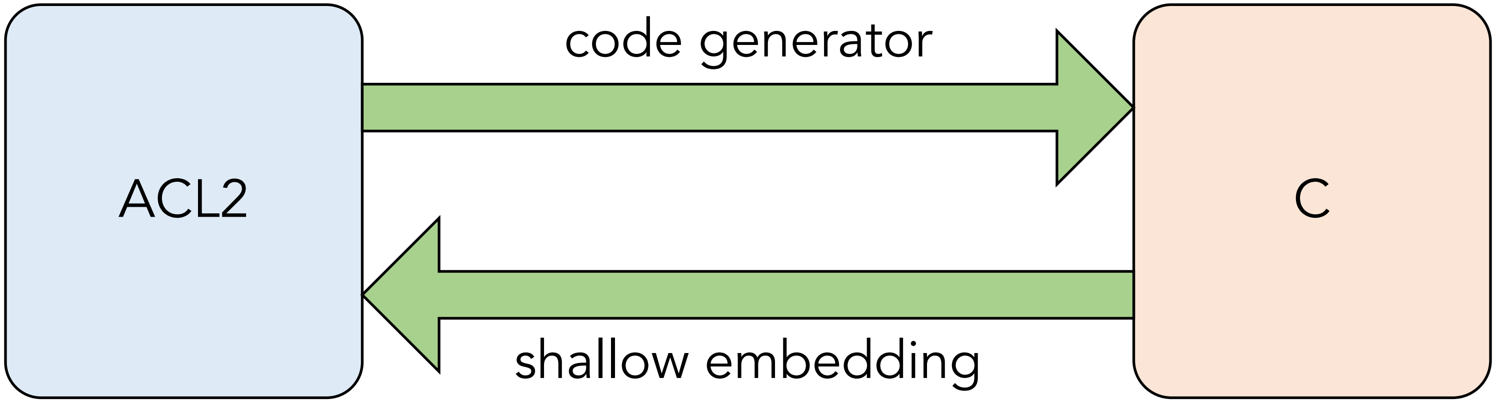
proof-generating

code generator

✔ shallow embedding

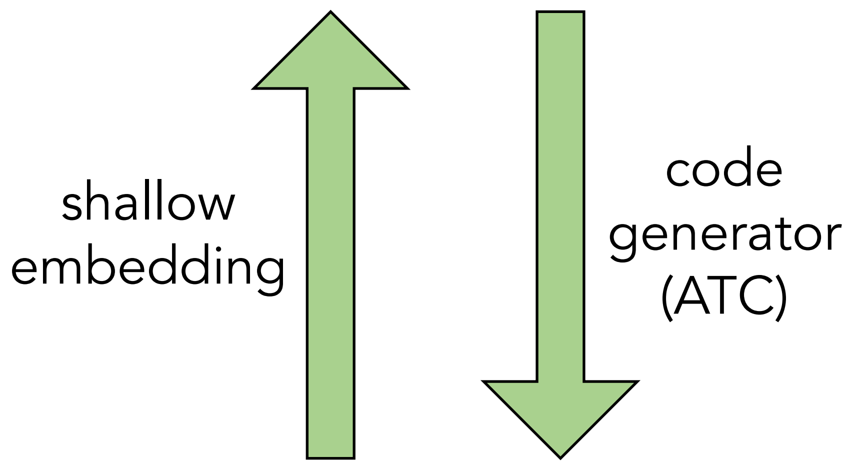
code generator = inverse of the shallow embedding

- recognizes the image of the shallow embedding
- translates the representation "back" to C code
- is implemented, unlike the shallow embedding



code generator = inverse of the shallow embedding

```
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard ...))
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```




```
int f(int x, int y, int z) {
    return (x + y) * (z - 3);
}
```

code generator = inverse of the shallow embedding

```
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard ...))
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```

non-idiomatic ACL2

code
generator
(ATC)

idiomatic C

```
int f(int x, int y, int z) {
    return (x + y) * (z - 3);
}
```

code/specification written in idiomatic ACL2

ATC is designed for program synthesis by stepwise refinement.

…

APT transformations

- `simplify` [ACL2-2017]
- `isodata` [ACL2-2020]
- many others
- some tailored to ATC

```
(defun |f| (|x| |y| |z|) ; non-idiomatic ACL2
  (declare (xargs :guard ...))
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```

ATC

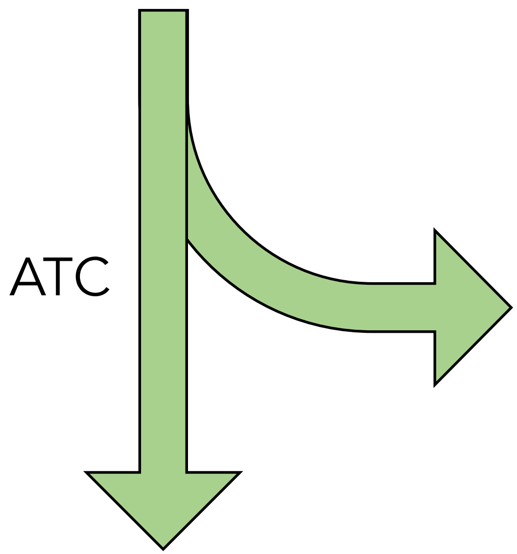
```
int f(int x, int y, int z) { // idiomatic C
    return (x + y) * (z - 3);
}
```

proof-generating

- ✓ code generator
- ✓ shallow embedding

proof-generating code generator

```
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard ...))
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```




```
(defthm |f|-correct ...)
```

```
int f(int x, int y, int z) {
    return (x + y) * (z - 3);
}
```

proof-generating code generator

```
(defthm |f|-correct
  (implies (and (compustatep compst)
                (equal fenv (init-fun-env *program*))
                (integerp limit)
                (>= limit ...)
                (and (sintp |x|)
                     (sintp |y|)
                     (sintp |z|)
                     ...))
           (equal (exec-fun (ident "f")
                            (list |x| |y| |z|)
                            compst
                            fenv
                            limit)
                  (b* ((result (|f| |x| |y| |z|)))
                    (mv result compst)))))
```

proof-generating code generator

ATC generates a named constant
for the generated C program
(currently, a single translation unit).

```
(defconst *program*
  <abstract syntax tree of the C program>)
```

This abstract syntax tree is
the one pretty-printed to file.

proof-generating code generator

The abstract syntax of (a subset of) C
is formalized via algebraic fixtypes, e.g.

```
(fty::deftagsum expr
  (:ident ((get ident)))
  (:const ((get const)))
  (:call ((fun ident)
          (args expr-list)))
  (:unary ((op unop)
           (arg expr)))
  (:binary ((op binop)
            (arg1 expr)
            (arg2 expr)))
    ...)
```

proof-generating code generator

ATC generates a named constant
for the generated C program
(currently, a single translation unit).

```
(defconst *program*
  <abstract syntax tree of the C program>)
```

This abstract syntax tree is
the one pretty-printed to file.

ATC also generates a theorem asserting
that the C program is well-formed
according to the C static semantics.

```
(defthm *program*-well-formed
    (equal (check-transunit *program*)
           :wellformed))
```

proof-generating code generator

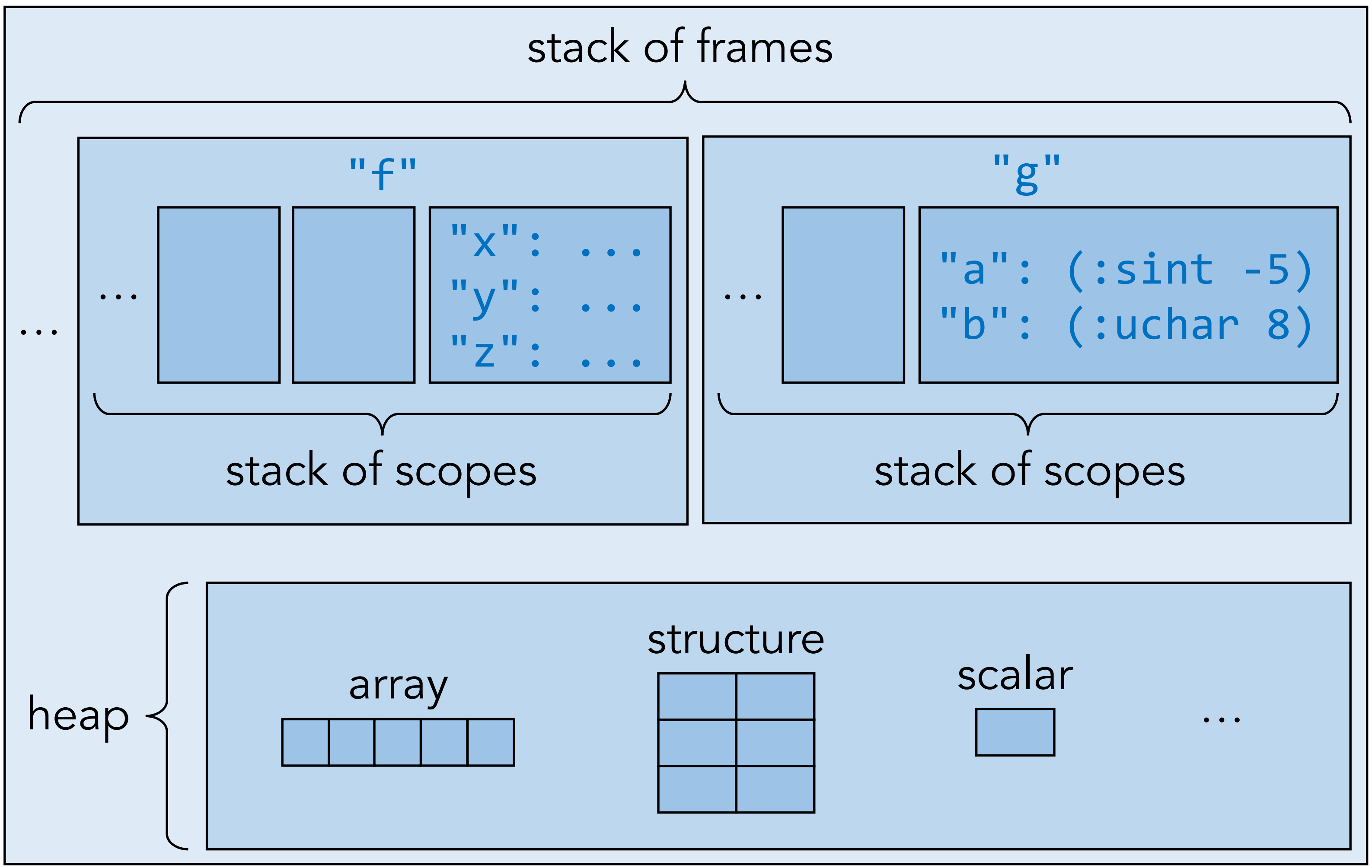
`check-transunit`,
along with `check-expr`
and other `check-...` functions,
formalize a static semantics of C,
i.e. the constraints on the code
necessary for its execution/compilation,
documented in the C18 standard.

proof-generating code generator

```
(defthm |f|-correct
  (implies (and (compustatep compst)
                (equal fenv (init-fun-env *program*))
                (integerp limit)
                (>= limit ...)
                (and (sintp |x|)
                     (sintp |y|)
                     (sintp |z|)
                     ...))
           (equal (exec-fun (ident "f")
                            (list |x| |y| |z|)
                            compst
                            fenv
                            limit)
                  (b* ((result (|f| |x| |y| |z|)))
                    (mv result compst)))))
```

function environment for the program, i.e. information (body etc.) about the functions proof-generating code generator

`compustatep` formalizes C computation states.



proof-generating code generator

```
(defthm |f|-correct
  (implies (and (compustatep compst)
                (equal fenv (init-fun-env *program*))
                (integerp limit)
                (>= limit ...)
                (and (sintp |x|)
                     (sintp |y|)
                     (sintp |z|)
                     ...))
           (equal (exec-fun (ident "f")
                            (list |x| |y| |z|)
                            compst
                            fenv
                            limit)
                  (b* ((result (|f| |x| |y| |z|)))
                    (mv result compst)))))
```

termination is expressed via a sufficiently large limit of the recursion of the `exec-...` functions, calculated by ATC

`exec-fun` and other `exec-...` functions formalize a big-step interpretive operational defensive dynamic semantics of C proof-generating code generator

```
(defthm |f|-correct
  (implies (and (compustatep compst)
                (equal fenv (init-fun-env *program*))
                (integerp limit)
                (>= limit ...)
                (and (sintp |x|)
                     (sintp |y|)
                     (sintp |z|)
                     ...))
           (equal (exec-fun (ident "f")
                            (list |x| |y| |z|)
                            compst
                            fenv
                            limit)
                  (b* ((result (|f| |x| |y| |z|)))
                    (mv result compst)))))
```

guard of |f|

execution of f in C

execution of |f| in ACL2

proof-generating code generator

The formulation of `|f|-correct` is more complicated in the presence of loops and/or array/structure updates.

The generated proof of `|f|-correct` is via symbolic execution, with induction for loops, in a precisely defined theory `:in-theory '(...)`.

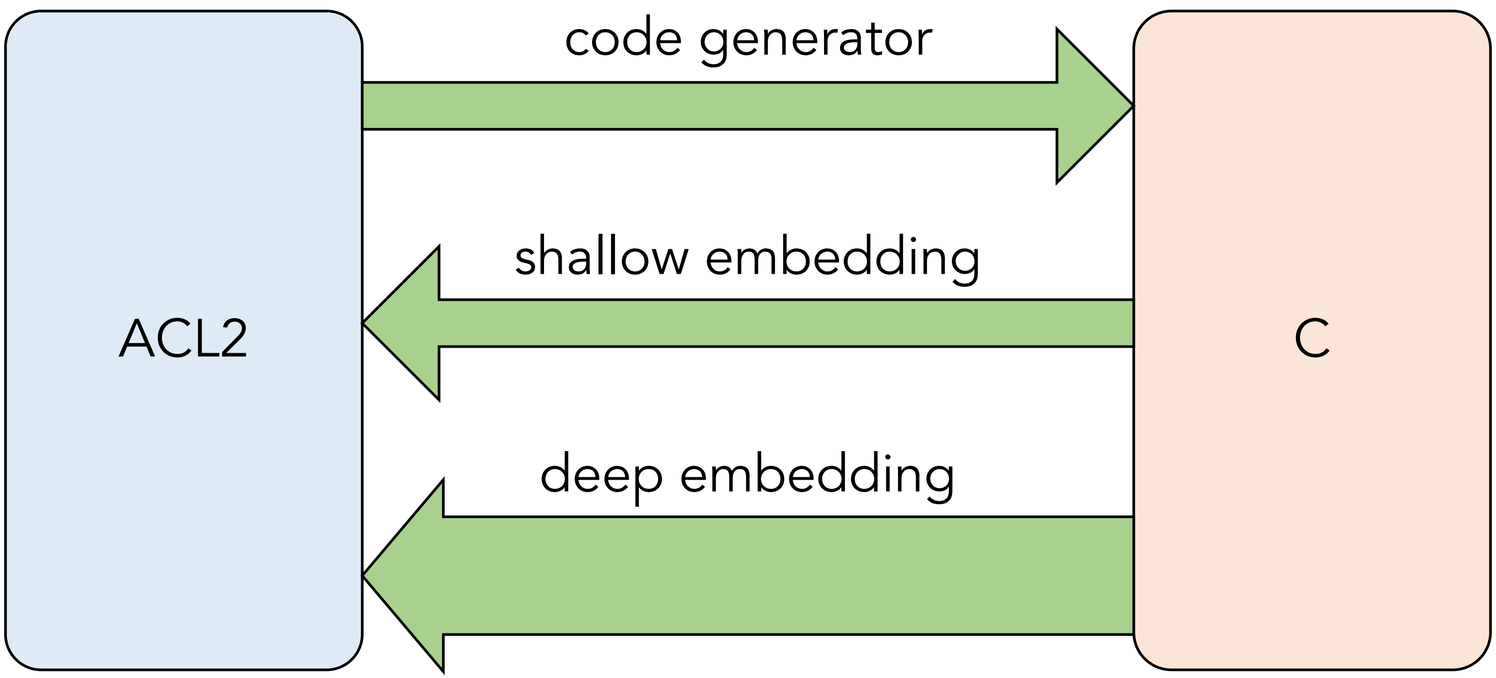
The symbolic execution turns the `exec-...` calls into the shallowly embedded C constructs (e.g. `add-sint-sint`) used in `|f|`.

The symbolic execution is fairly elaborate.
See the ATC developer documentation for details.

- proof-generating
- code generator
- shallow embedding

The generated proofs are based on
a formalization of (a subset of) C in ACL2,
exemplified earlier, consisting of
abstract syntax, static semantics,
and dynamic semantics.

This formalization is a deep embedding of C in ACL2.



See section on language embedding and code generation in the paper.