# Adding a typing mechanism to ACL2

Vernon Austel

IBM

# Outline

What a typed ACL2 might look like

Vague proposal concerning
how to change ACL2 to allow
experimentation with type systems

- allow macros to take the ACL2 world and state as parameters
- this could also be used for other purposes

There is no time to describe my type system

- that would be boring anyway

# To type or not to type?

Let's avoid the question...

It is unreasonable for ACL2 to suddenly start requiring functions to be typed

It is probably not useful to have a typed and untyped mode (where either all functions are typed or none are).

Untyped functions and theorems must be interoperable with typed ones.

The internal representation must be the same (aside from some extra information).

# How would types affect me?

Most of the time, the type system will infer a type for your functions; you may occasionally have to add type-annotations or change functions to help the type system infer their type

If the type system infers hypotheses for theorems, you may be able to stop writing (true-listp l) and (integerp n) all the time

If the type system filters such type hypotheses from the proof output, you don't have to look at them

This is all up to you and your type system

# Function types

Most of the time, you shouldn't need to think about types, just as you don't think about type-prescription lemmas.

If the type system can't derive one for a function, you might have to tell it, perhaps using a new xargs keyword:

```
(defun myappend (x y)
   (declare (xargs :type ((list 0) (list 0) (list 0))))
   (if (endp x) y
      (cons (car x) (myappend (cdr x) y))))
```

Or perhaps with an event:

```
(deftype myappend ((list 0) (list 0) (list 0)))
```

# Adding type hypotheses to theorems

This event fails:

```
(defthm myappend-nil
    (equal (myappend x nil) x))
```

But as a typed theorem, it might succeed:

```
(defthmt myappend-nil
    (equal (myappend x nil) x))
```

That's because the type system
might generate this as the goal to prove:

```
  (implies (true-listp x)
                (equal (myappend x nil) x))
```

# That's it

Type-checking functions helps catch annoying little bugs quickly.

A type system takes care of adding type hypotheses to theorems, so you can forget about them.

This doesn't affect the proof engine or the logic. It can have no affect on soundness.

A typed theorem or function is no different from an untyped one; they are interoperable.

A type system may not be able to work with untyped functions, but you can always assign a type to a function after it was defined.

# The goal: allowing an untrusted type system

No one knows what kind of type system would be acceptable to the ACL2 community. It would probably take a lot of experimentation to find out.

The ACL2 maintainers don't have time for this.

The hope is that there is a way that the maintainers could modify ACL2 so that ordinary users could experiment with types without compromising the soundness of the system.

The system changes must not involve a lot of work.

# We can already almost do that

The problem is that a type system needs access to the world (and to state for efficient macro expansion).

That means the type system cannot be implemented just using macros; macros do not have access to the world or state, only functions do. (Macros can expand into function calls that take state as a parameter, though).

However, user-defined functions cannot occur within encapsulate and inside books; only embedded events (and macros) are permitted.

Let us review why this is.

# include-book

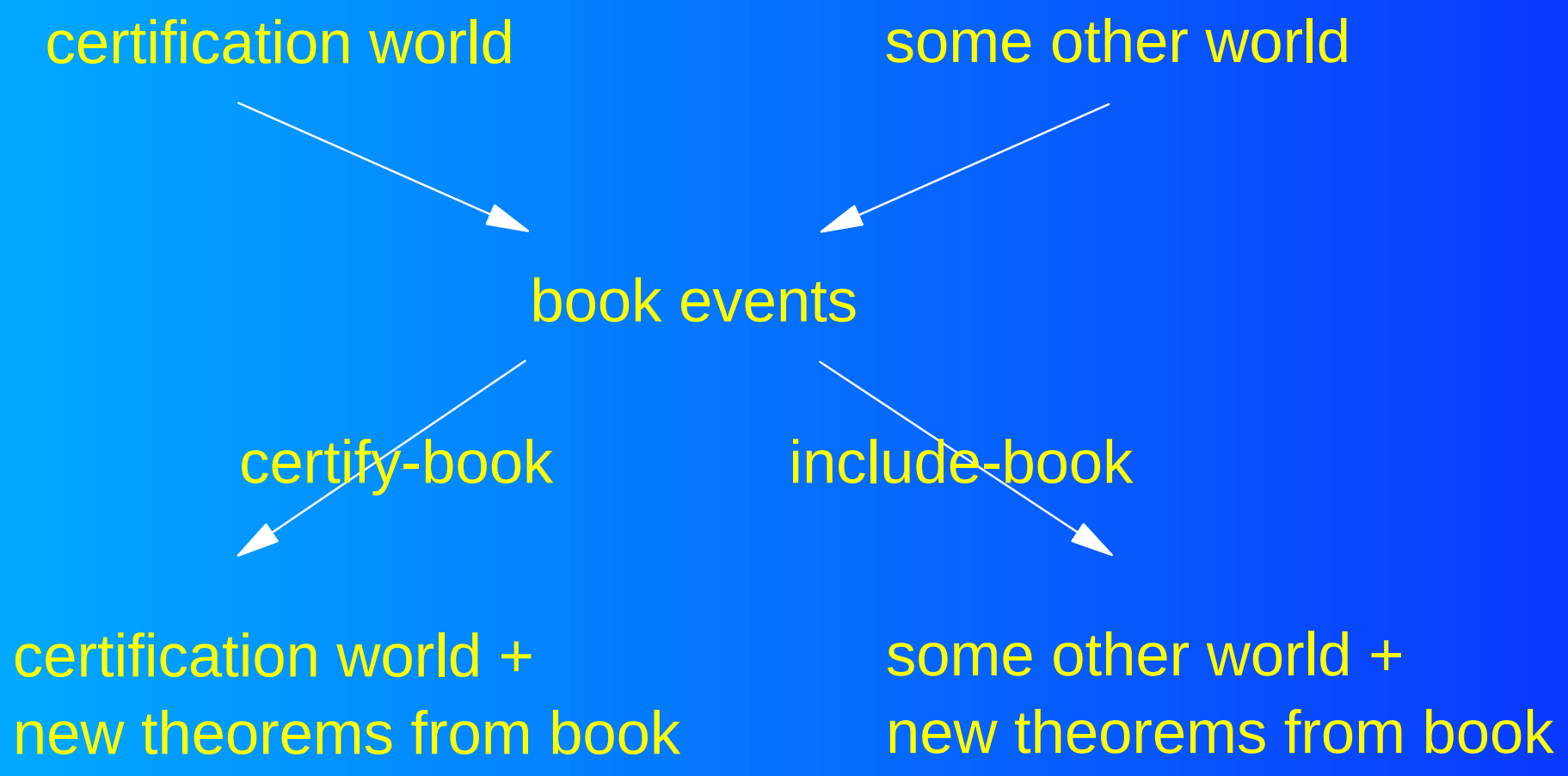
events inside a book are processed twice:

- during certification (in certify-book)
- during execution of include-book

Roughly speaking, certify-book checks that the theorem events are true, and include-book just modifies the current world by processing the events in the book.

include-book will execute using a different world than certify-book; the system must ensure that the same functions and theorems are added when include-book is executed.

# Picture of include-book processing



# The problem - code may do different things in the two passes

This is slightly modified from the example in the ACL2 documentation. It must not be allowed.

```
(if (ld-skip-proofsp state)
    (defthm thm-for-second-pass nil) ;; include-book
  (defthm thm-for-first-pass t))        ;; certify-book
```

So: we can only use code that is trusted to do the right thing in both passes. Embedded events are trusted to do this. (Other code may as well, but how would we be sure?)

# What about macros?

The ACl2 documentation does not say so, but macros are not allowed access to state or to the world for the same reasons.

```
(defmacro unsound (state)
  (if (ld-skip-proofsp state)
      '(defthm thm-for-second-pass nil)
    '(defthm thm-for-first-pass t)))
```

This would cause problems because

```
  (unsound state)
```

would expand to different things in the two passes.

# Can we avoid that problem?

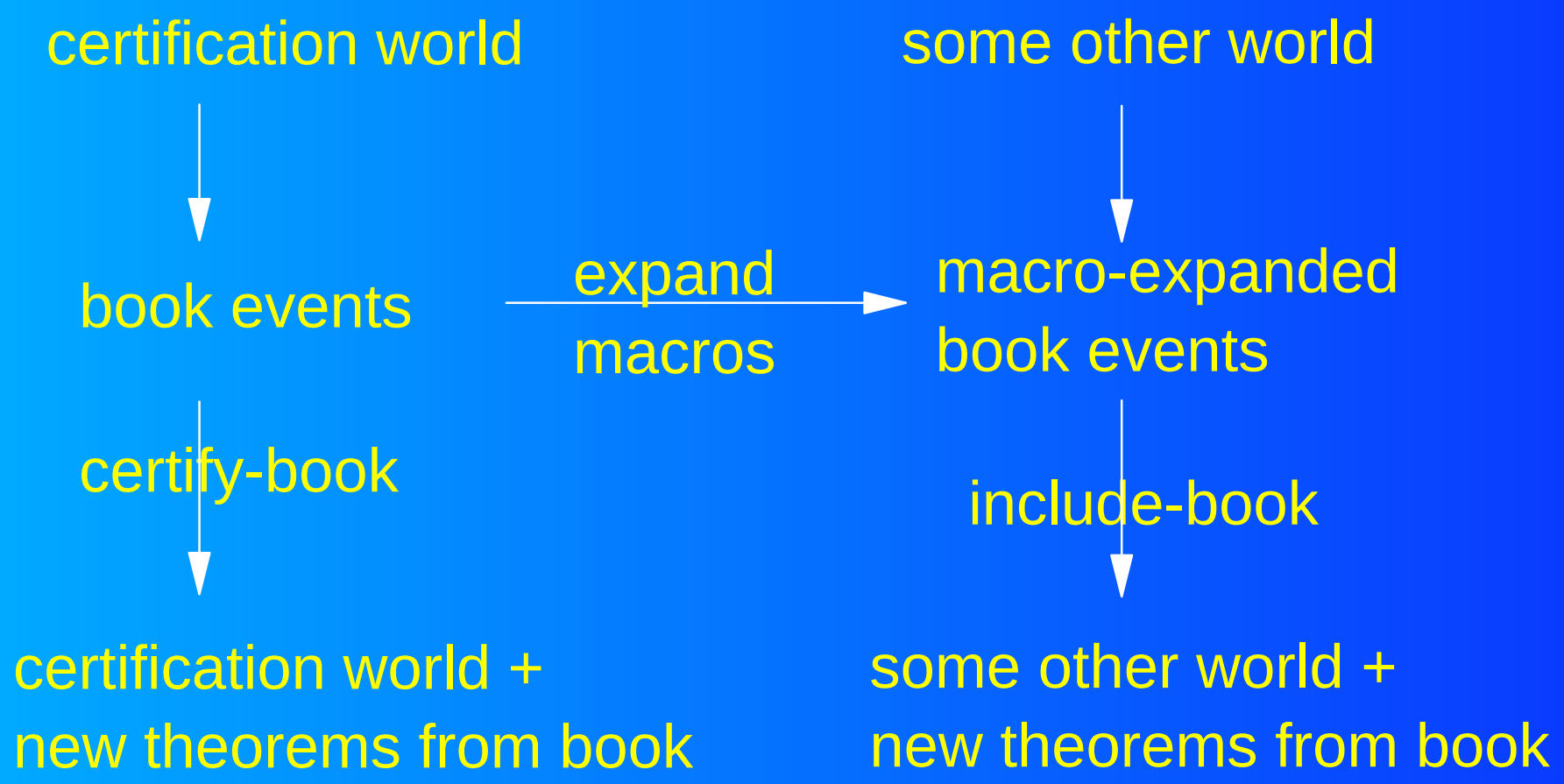
Yes. The question is how much work it would require, and how clean the result would be.

The central problem is that macros might expand to different things in the two passes.

We could avoid that if we save the expansion from the first pass and re-use it in the second pass.

include-book would not read the events in the book, but rather a file that certify-book generates.

# Re-using the macro expansion



# What's the problem?

The second pass imposes the same restrictions as the first pass; the macro-expanded version will fail these syntactic checks (embedded events are themselves macros).

```
(defthm prop1 t)
```

expands to:

```
(defthm-fn 'prop1 t state ...)
```

which is not an embedded event.

# Possible solutions

- Drop these checks in the second pass?
After all, the code passed the check once already.
There remains the sensative problem of ensuring
that the macro expansion saved from certification
really does correspond to the book (file system security).

- Just use the expansions as a check in the second pass?
That is, expand macros as usual in the second pass,
but check the result of the expansion with
the saved expansion; if there is a difference, fail.
There must be a 1-1 correspondence. All event macros
must expand the same way in both passes ("local").

- Don't expand embedded event macros?
Complicates macro expansion.

# Sounds like a lot of work...

...for something you really don't care about?
This would be useful for any system that needs access to function bodies to generate theorems.

Example: inferring measure theorems

```
(defun parse-type-pointers (basetype input)
  (if (eq (car input) '*)
      (parse-type-pointers (mk-ptr-type basetype) (cdr input))
    (mv basetype input)))

(infer-measure-thm parse-type-pointers)
==>
(defthm acl2-count-parse-type-pointers
  (<= (acl2-count (mv-nth 1 (parse-type-pointers types input)))
      (acl2-count input))
 :rule-classes (:linear))
```

# Summary

- Adding a type system would inconvenience no one
- Typed and untyped code would be interoperable
- The type system need not be trusted
- It only poses a soundness concern for embedded events
- non-trivial changes to include-book and encapsulate would be required
- the change would also be useful for other purposes
- I don't know what kind of change would be acceptable