A Complex Java Code Generator for ACL2 Based on a Shallow Embedding of ACL2 in Java

Alessandro Coglio

Kestrel Institute

ACL2 Workshop 2022

How can we run ACL2 code in Java?

```
(defun f (x) ; example of ACL2 code
  (declare (xargs :guard (integerp x)))
  (+ x 3))
```

We need a Java representation
of `integerp` and other types,
and of `+` and other operations.

We need a Java representation
of `integerp` and other types,
and of `+` and other operations.

```
public final class Acl2Integer extends Acl2Rational {

    // representation of integerp:
    private final BigInteger numericValue;

    // representation of +:
    Acl2Integer addInteger(Acl2Integer other) ...
    Acl2Rational addRational(Acl2Rational other) ...
    Acl2Number addNumber(Acl2Number other) ...
    Acl2Number addValue(Acl2Value other) ...

    ...
}
```

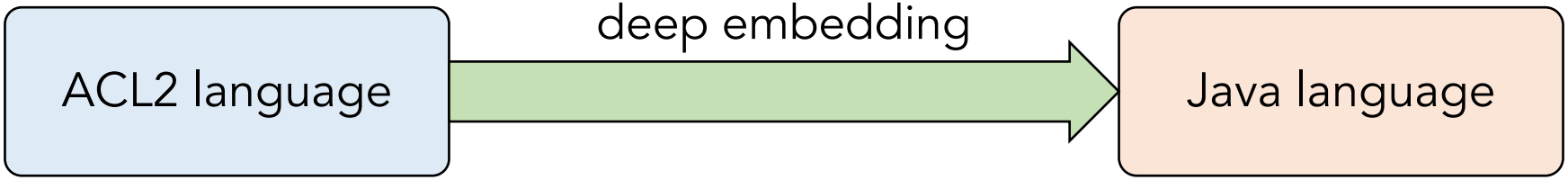
How can we run ACL2 code in Java?

```
(defun f (x) ; example of ACL2 code
  (declare (xargs :guard (integerp x)))
  (+ x 3))
```

Given the Java representation of `integerp`, `+`, etc.,
we can run the ACL2 code in Java in two main ways.

(1) Via an interpreter of ACL2 written in Java,
which uses `Acl2Integer`, `Acl2Rational`, etc.

The interpreter is a deep embedding of ACL2 in Java.



How can we run ACL2 code in Java?

```
(defun f (x) ; example of ACL2 code
  (declare (xargs :guard (integerp x)))
  (+ x 3))
```

Given the Java representation of `integerp`, `+`, etc.,
we can run the ACL2 code in Java in two main ways.

(1) Via an interpreter of ACL2 written in Java,
which uses `Acl2Integer`, `Acl2Rational`, etc.

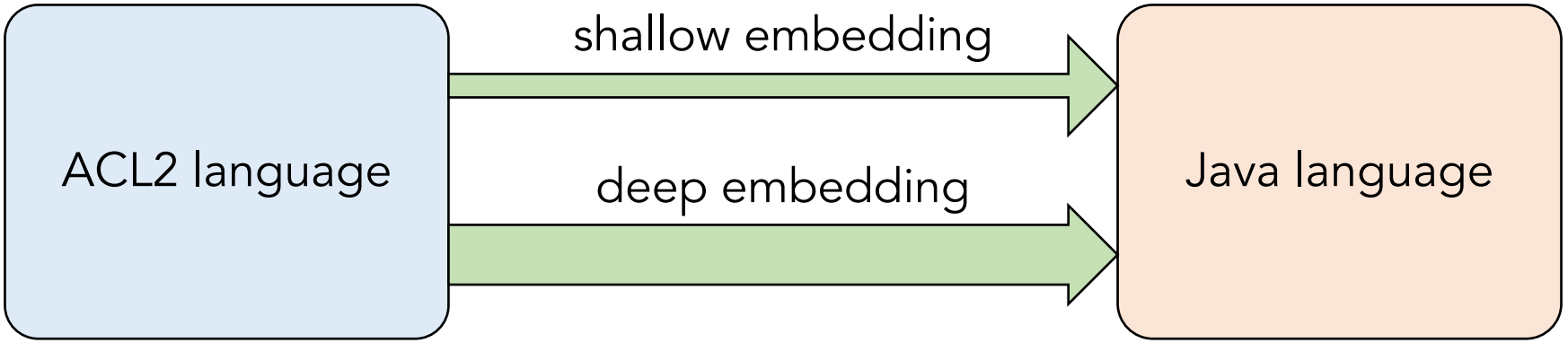
(2) By translating the ACL2 constructs
to suitably equivalent Java constructs.

```
(defun f (x) ; example of ACL2 code
  (declare (xargs :guard (integerp x)))
  (+ x 3))
```

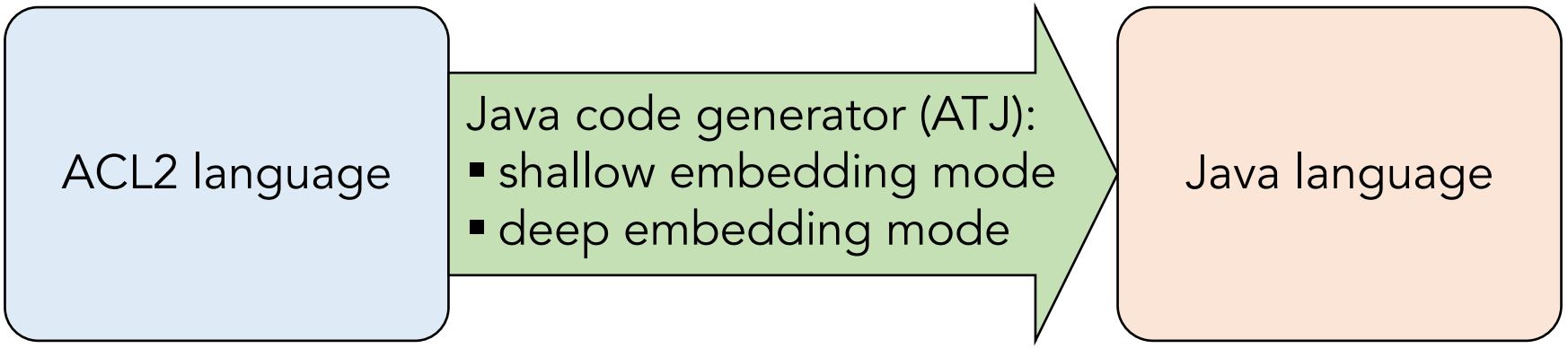
(2) By translating the ACL2 constructs
to suitably equivalent Java constructs.

```
public static Acl2Integer f(Acl2Integer x) {
    return binary_plus(x, $N_3);
}
```

The translation is a shallow embedding of ACL2 in Java.

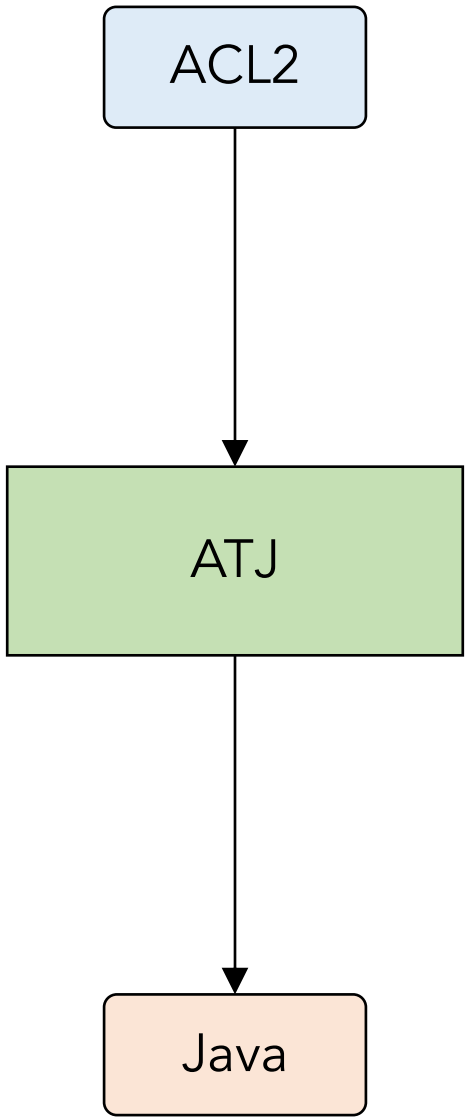


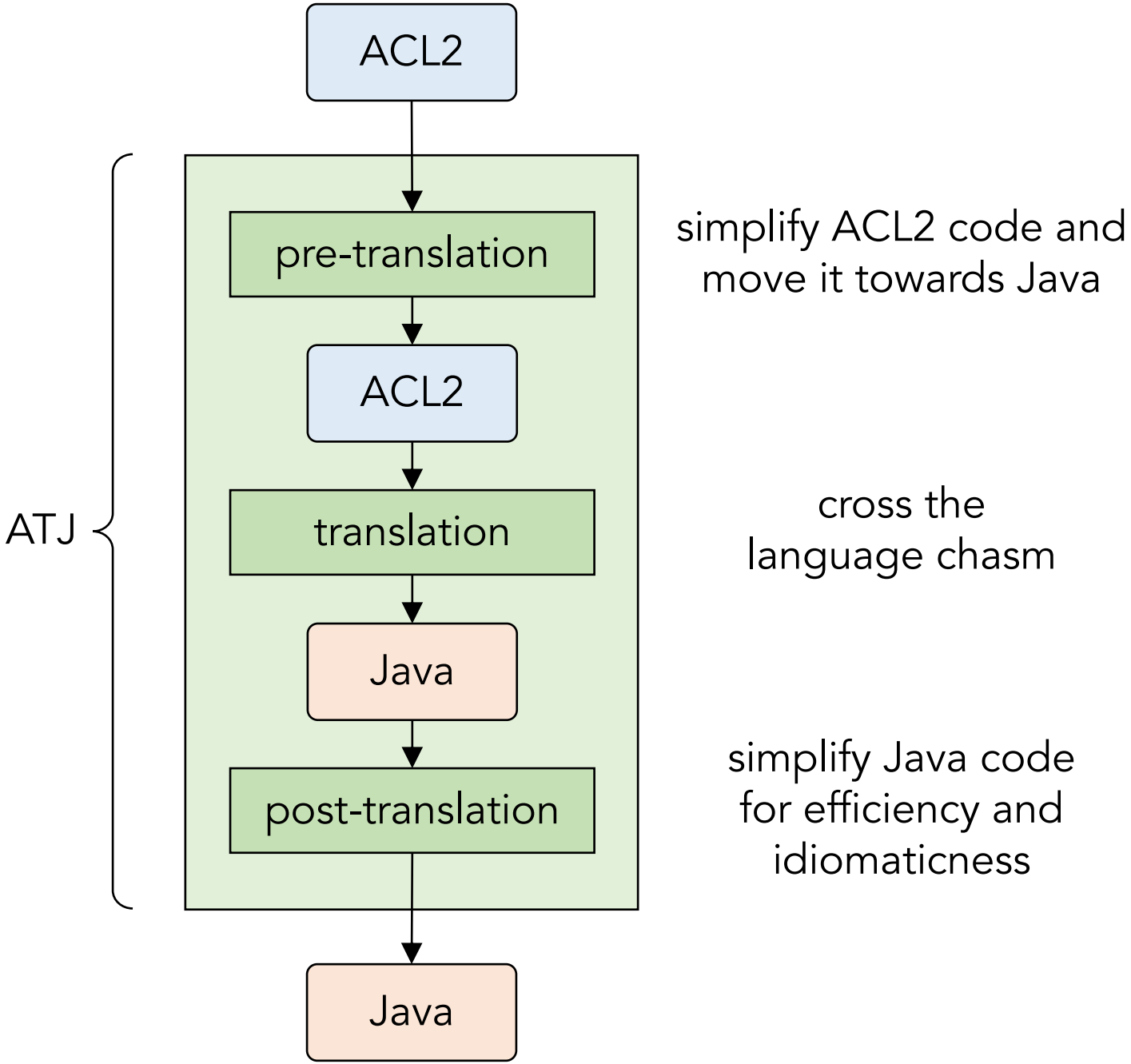
The shallow embedding mode is more conventional.
It is complex, but the generated code is more efficient and idiomatic.
It is the main subject of this ACL2-2022 paper and presentation.

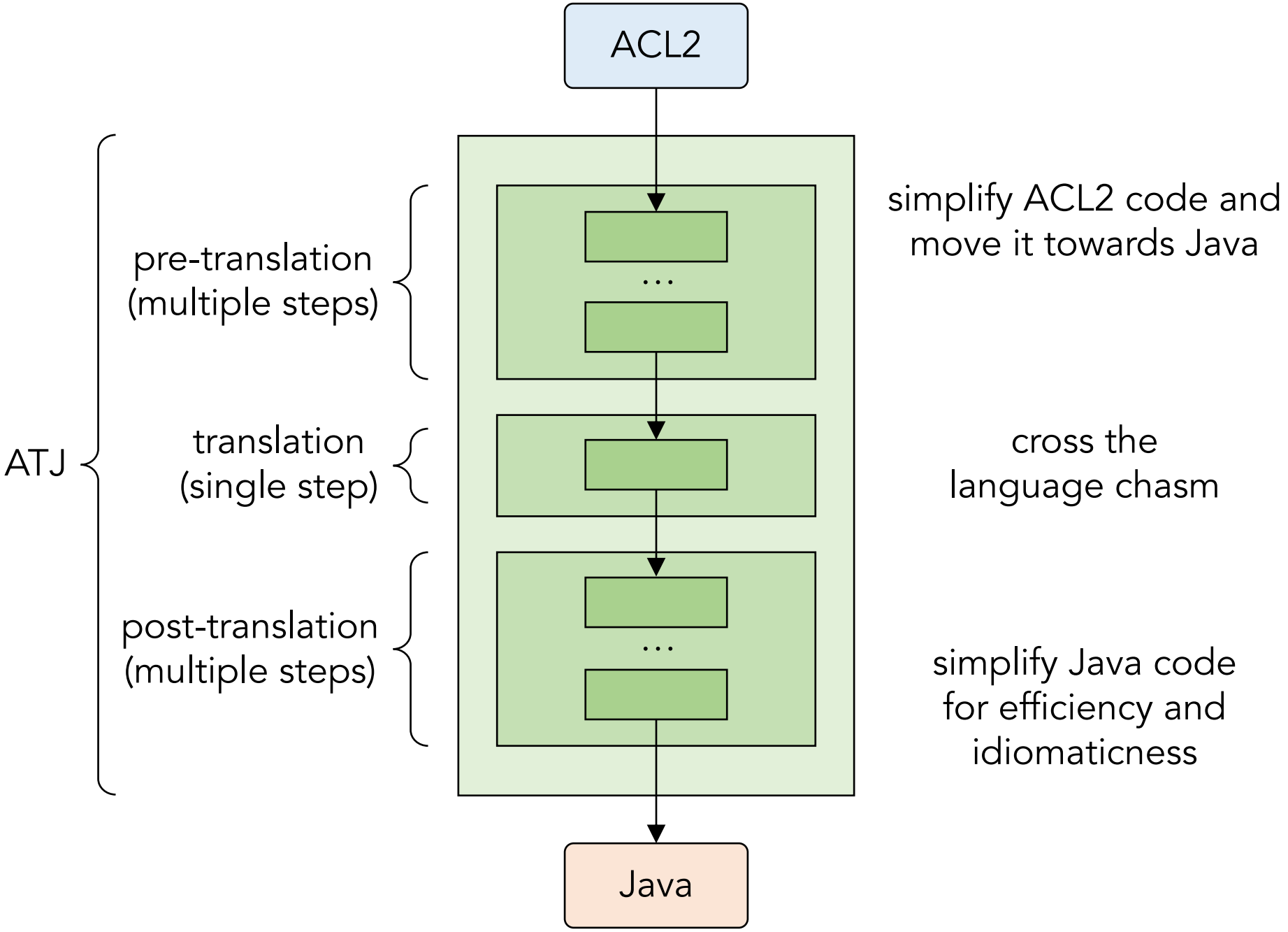


The deep embedding mode is somewhat unconventional.
It is simple, but the generated code is not very efficient or idiomatic.
It was described in the ACL2-2018 paper and presentation.

It could become more interesting by combining it with
partial evaluation (first Futamura projection; see paper).


ACL2
ATJ
Java


ACL2
pre-translation
ACL2
translation
Java
post-translation
Java
ATJ
simplify ACL2 code and
move it towards Java
cross the
language chasm
simplify Java code
for efficiency and
idiomaticness


ACL2
ATJ
pre-translation
(multiple steps)
translation
(single step)
post-translation
(multiple steps)
simplify ACL2 code and
move it towards Java
cross the
language chasm
simplify Java code
for efficiency and
idiomaticness
Java

Here is a somewhat artificial example of ACL2 code.

Let us see how ATJ turns it into Java, step by step.

```
(defun f (n a)
  (declare (xargs :guard (and (natp n) (natp a))))
  (let ((__function__ 'f))
    (if (mbt (natp n))
        (if (equal n 0)
            a
          (let ((n1 (1- n)))
            (let ((a (* n a)))
              (f n1 a))))
      0)))
```

ATJ has an option to assume guards or not.

Assuming guards is the normal thing to do.

Not assuming guard is execution "in the logic".

If guards are assumed, an ATJ pre-translation step turns `mbe`s into their `:exec` parts, and `mbt`s into `t`.

```
(defun f (n a)
  (declare (xargs :guard (and (natp n) (natp a))))
  (let ((__function__ 'f))
    (if t
        (if (equal n 0)
            a
          (let ((n1 (1- n)))
            (let ((a (* n a)))
              (f n1 a))))
      0)))
```

This function is no longer admissible in the ACL2 logic, but it is correct according to the ACL2 evaluation semantics.

An ATJ pre-translation step removes dead if branches
(which may arise from the previous transformation step).

```
(defun f (n a)
  (declare (xargs :guard (and (natp n) (natp a))))
  (let ((__function__ 'f))
    (if (equal n 0)
        a
      (let ((n1 (1- n)))
        (let ((a (* n a)))
          (f n1 a))))))
```

An ATJ pre-translation step removes unused bound variables (which may arise from macros like `define`).

```
(defun f (n a)
  (declare (xargs :guard (and (natp n) (natp a))))
  (if (equal n 0)
      a
    (let ((n1 (1- n)))
      (let ((a (* n a)))
        (f n1 a)))))
```

An ATJ pre-translation step adds type annotations.

Each variable is annotated with a type,
e.g. `[AI]` for ACL2 integers and `[AB]` for ACL2 boolean.

Each term is annotated with a type conversion `[from>to]`
(mostly identities in this example; see paper for other examples).

```
(defun f ([AI]n [AI]a)
  (declare (xargs :guard (and (natp [AI]n) (natp [AI]a))))
  ([AI>AI]
   (if ([AB>AB] (equal ([AI>AV] [AI]n)
                       ([AI>AV] 0)))
       ([AI>AI] [AI]a)
     ([AI>AI]
      (let (([AI]n1 ([AI>AI] (1- ([AI>AI] [AI]n)))))
        ([AI>AI]
         (let (([AI]a ([AI>AI] (* ([AI>AI] [AI]n)
                                  ([AI>AI] [AI]a)))))
           ([AI>AI] (f ([AI>AI] [AI]n1)
                       ([AI>AI] [AI]a)))))))))))
```

An ATJ pre-translation step adds type annotations.

Each variable is annotated with a type,
e.g. `[AI]` for ACL2 integers and `[AB]` for ACL2 boolean.

For illustrative purposes, we elide the type conversions `[from>to]`
in this example, where they do not play a big role anyhow.

```
(defun f ([AI]n [AI]a)
  (declare (xargs :guard (and (natp [AI]n) (natp [AI]a))))
  (if (equal [AI]n 0)
      [AI]a
    (let (([AI]n1 (1- [AI]n)))
      (let (([AI]a (* [AI]n [AI]a)))
        (f [AI]n1 [AI]a)))))
```

An ATJ pre-translation step marks variables for reuse or not.

A variable marked with `[N]` is new, i.e. not reused in Java.

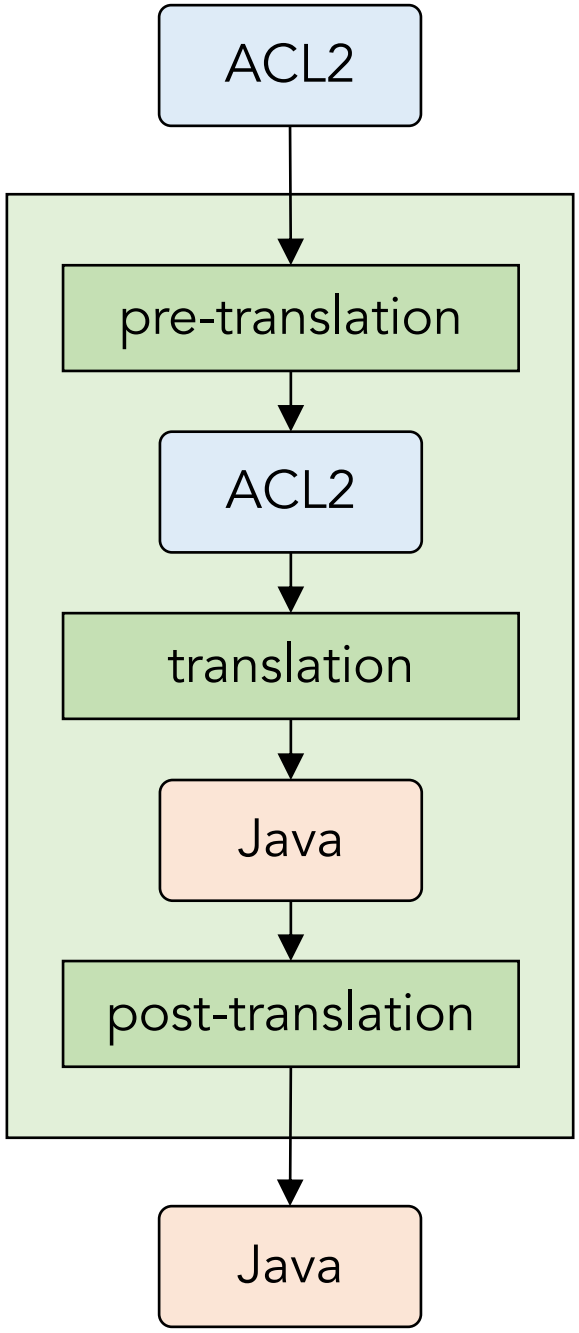
A variable marked with `[O]` is old, i.e. reused in Java.

This is quite complicated in general,
because ACL2 and Java have different scoping rules.

```
(defun f ([AI]n [AI]a)
  (declare (xargs :guard (and (natp [AI]n) (natp [AI]a))))
  (if (equal [AI]n 0)
      [AI]a
    (let (([N][AI]n1 (1- [AI]n)))
      (let (([O][AI]a (* [AI]n [AI]a)))
        (f [N][AI]n1 [O][AI]a)))))
```

ATJ has other pre-translation steps,
but they do not apply to this example.

This is the final result of pre-translation.



ATJ translates this ACL2 function to a Java method.

The input and output types are derived from the type annotations.

The ACL2 `if` is turned into the Java `if`.

`let` bindings of `[N]`ew variables are turned into declarations.

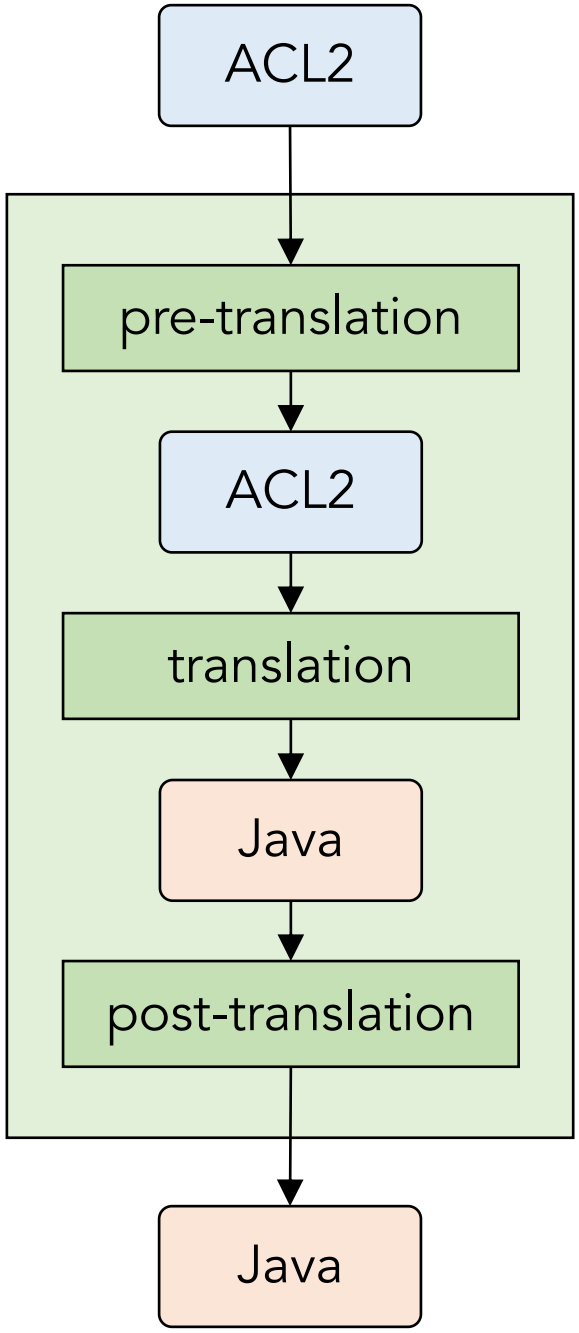
`let` bindings of `[O]`ld variables are turned into assignments.

Temporaries are introduce to hold results of statements, leading to a compositional translation of ACL2 terms to Java expressions "prepared" by Java statements.

```
static Acl2Integer f(Acl2Integer n, Acl2Integer a) {
    Acl2Integer $tmp1;
    if (equal(n, $N_0)) {
        $tmp1 = a;
    } else {
        Acl2Integer n1 = binary_plus($N_minus1, n);
        a = binary_star(n, a);
        $tmp1 = f(n1, a);
    }
    return $tmp1;
}
```

ATJ's translation step also handles other ACL2 constructs (e.g. `mv-let`) but they are not in this example.

This is the final result of the translation step.



An ATJ post-translation step eliminates temporaries
(in this example and in other cases, but not in all cases),
by folding the `return`s into the preceding statements.

```
static Acl2Integer f(Acl2Integer n, Acl2Integer a) {
    if (equal(n, $N_0)) {
        return a;
    } else {
        Acl2Integer n1 = binary_plus($N_minus1, n);
        a = binary_star(n, a);
        return f(n1, a);
    }
}
```

Since this method is tail-recursive, ATJ does tail recursion elimination.

An ATJ post-translation step surrounds the body with a loop
and replaces the `return`(ed) recursive calls
with `continue`(s) preceded by parallel assignments.

```
static Acl2Integer f(Acl2Integer n, Acl2Integer a) {
    while (true) {
        if (equal(n, $N_0)) {
            return a;
        } else {
            Acl2Integer n1 = binary_plus($N_minus1, n);
            a = binary_star(n, a);
            n = n1;
            continue;
        }
    }
}
```

A parallel assignment is generated
via topological sort when possible,
otherwise via temporaries.

An ATJ post-translation step lifts termination tests from `if`s to `while`s, when possible.

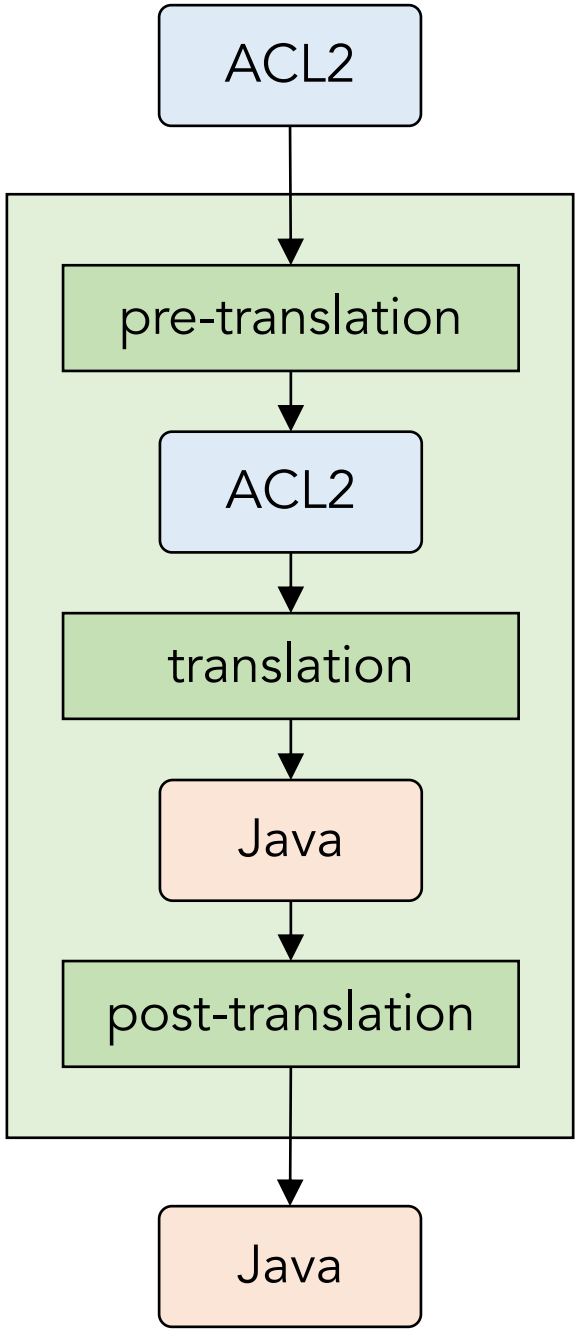
```
static Acl2Integer f(Acl2Integer n, Acl2Integer a) {
    while (!equal(n, $N_0)) {
        Acl2Integer n1 = binary_plus($N_minus1, n);
        a = binary_star(n, a);
        n = n1;
        continue;
    }
    return a;
}
```

An ATJ post-translation removes any unnecessary `continue`s.

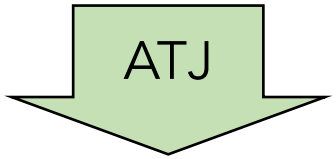
```
static Acl2Integer f(Acl2Integer n, Acl2Integer a) {
    while (!equal(n, $N_0)) {
        Acl2Integer n1 = binary_plus($N_minus1, n);
        a = binary_star(n, a);
        n = n1;
    }
    return a;
}
```

ATJ has other post-translation steps,
but they do not apply to this example.

This is the final result of post-translation.



```
(defun f (n a)
  (declare (xargs :guard (and (natp n) (natp a))))
  (let ((__function__ 'f))
    (if (mbt (natp n))
        (if (equal n 0)
            a
          (let ((n1 (1- n)))
            (let ((a (* n a)))
              (f n1 a))))
      0)))
```




```
static Acl2Integer f(Acl2Integer n, Acl2Integer a) {
    while (!equal(n, $N_0)) {
        Acl2Integer n1 = binary_plus($N_minus1, n);
        a = binary_star(n, a);
        n = n1;
    }
    return a;
}
```

The performance of ATJ's generated Java code
(in the shallow embedding mode assuming guards)
is somewhat competitive with the ACL2 code:
about 3–4x slower, on the ABNF grammar parser.

In the deep embedding mode assuming guards,
the code is 60–80x slower than the code
in the shallow embedding mode assuming guards;
this is due to the interpretation overhead.