Java Code Generation for the ACL2 Theorem Prover

Kestrel Institute

Alessandro Coglio
designer & implementor

Limei Gilham   David Hardin   Eric McCarthy
Eric Smith   Stephen Westfold
users & contributors

ACL2-2020 Workshop

A very preliminary version was described in the ACL2-2018 paper.

A Simple Java Code Generator for ACL2 Based on a Deep Embedding of ACL2 in Java

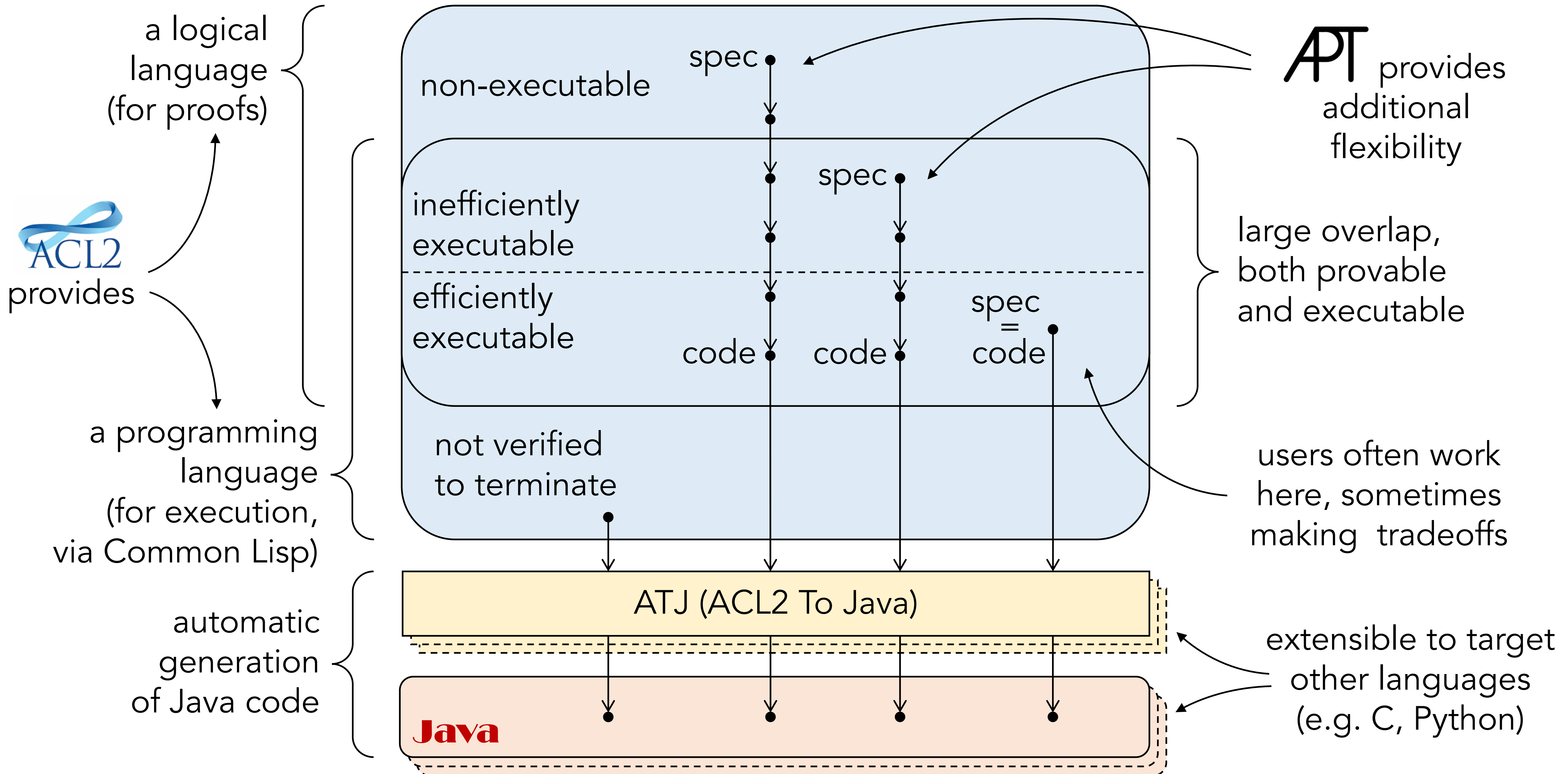
Alessandro Coglio

Kestrel Institute

`http://www.kestrel.edu`

AIJ (**A**CL2 **I**n **J**ava) is a deep embedding in Java of an executable, side-effect-free, non-stobj-accessing subset of the ACL2 language without guards. ATJ (**A**CL2 **T**o **J**ava) is a simple Java code generator that turns ACL2 functions into AIJ representations that are evaluated by the AIJ interpreter. AIJ and ATJ enable possibly verified ACL2 code to run as, and interoperate with, Java code, without much of the ACL2 framework or any of the Lisp runtime. The current speed of the resulting Java code may be adequate to some applications.

The ACL2 theorem prover, the APT toolkit, and the ATJ tool.



The portion of the ACL2 language that ATJ translates to Java.

language (both logical and programming)

non-executable

executable, without side effects

executable, with side effects

this cannot be accepted by ATJ

side effects include printing, file I/O, stobj updates, etc. (realized in 'raw Lisp')

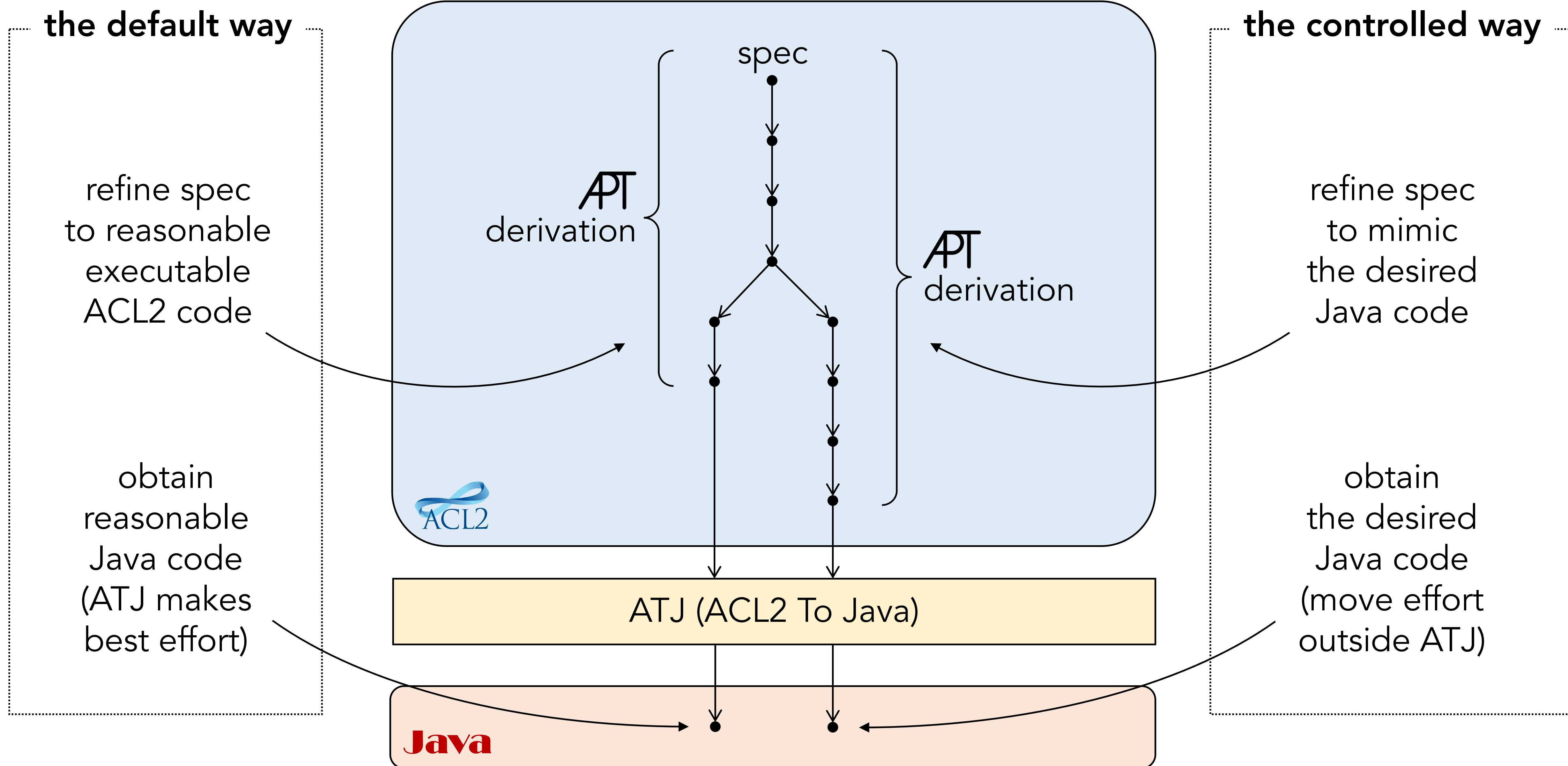
this is not accepted by ATJ currently, but it can be, by mimicking the side effects in Java (case by case)

all of this is accepted by ATJ

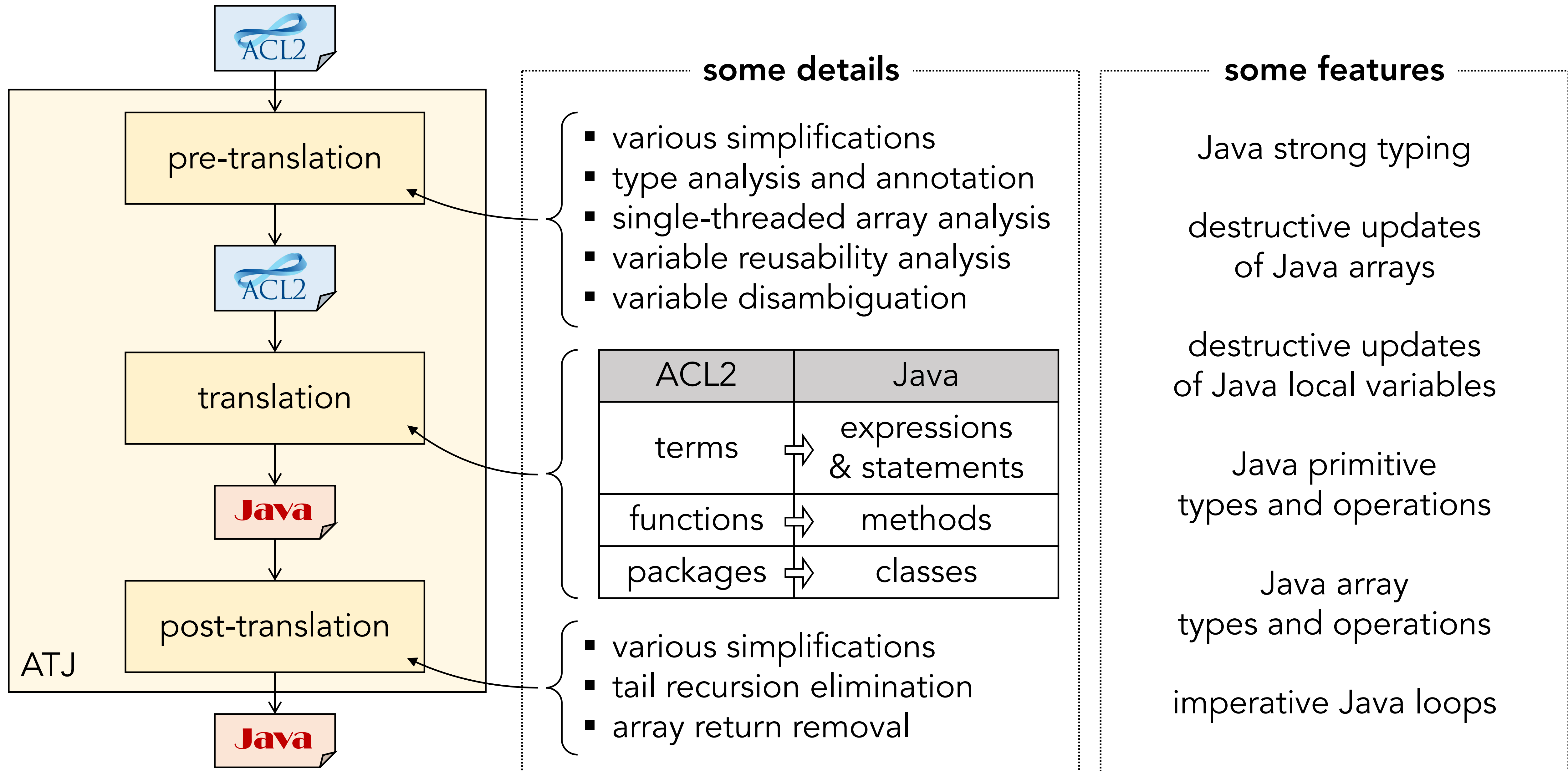
ATJ (ACL2 To Java)

Java

The two main ways to use ATJ, in conjunction with APT.



Some internal details of ATJ, and what they accomplish.



An example of Java code generated in ATJ's default way.

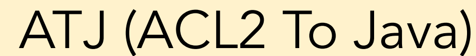
```
(defun fact (n)
  (declare (xargs :guard (natp n)))
  (if (zp n)
      1
    (* n (fact (1- n)))))
```

ATJ (ACL2 To Java)

```
public static Acl2Integer fact(Acl2Integer n) throws ... {
    if (zp(n) != NIL) {
        return $N_1;
    } else {
        return binary_star(n, fact(binary_plus($N_minus1, n)));
    }
}
```

Another example of Java code generated in ATJ's default way.

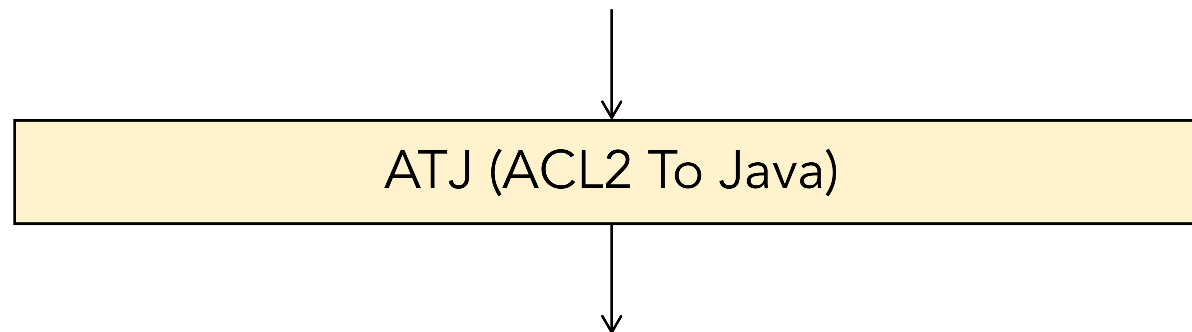
```
(defun fact-tr (n r)
  (declare (xargs :guard (and (natp n) (acl2-numberp r))))
  (if (zp n)
      r
    (fact-tr (+ -1 n) (* r n))))
```




```
public static Acl2Number fact_tr(Acl2Integer n, Acl2Number r) throws ... {
    while (zp(n) == NIL) {
        r = binary_star(r, n);
        n = binary_plus($N_minus1, n);
    }
    return r;
}
```

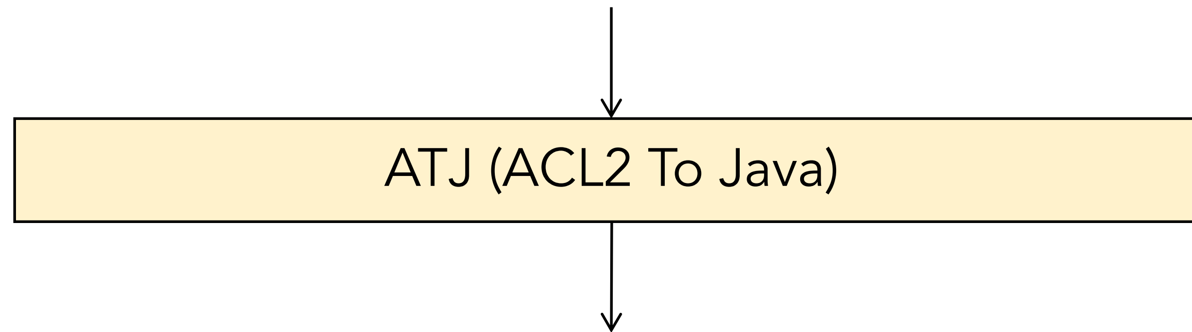
An example of Java code generated in ATJ's controlled way.

```
(define fact-mod-java ((n java::int-value-p))
  :guard (java::boolean-value->bool (java::int-greateq n (java::int-value 0)))
  :returns (result java::int-value-p)
  (if (mbt (and (java::int-value-p n)
                (java::boolean-value->bool
                 (java::int-greateq n (java::int-value 0)))))
      (if (java::boolean-value->bool (java::int-eq n (java::int-value 0)))
          (java::int-value 1)
        (java::int-mul n
                       (fact-mod-java (java::int-sub n
                                                     (java::int-value 1)))))
    (java::int-value 1)))
```



An example of Java code generated in ATJ's controlled way.

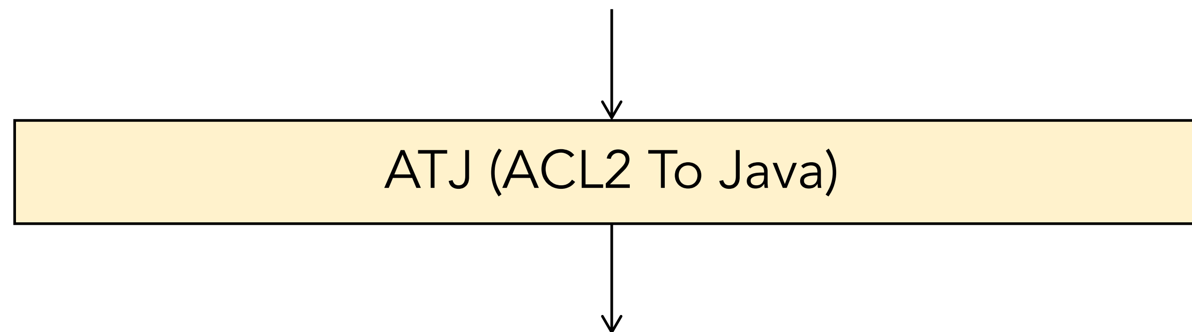
```
      (if (java::boolean-value->bool (java::int-eq n (java::int-value 0)))
          (java::int-value 1)
        (java::int-mul n
                      (fact-mod-java (java::int-sub n
                                                    (java::int-value 1)))))
    (java::int-value 1))
```




```
public static int fact_mod_java(int n) throws ... {
    if (n == 0) {
        return 1;
    } else {
        return n * fact_mod_java(n - 1);
    }
}
```

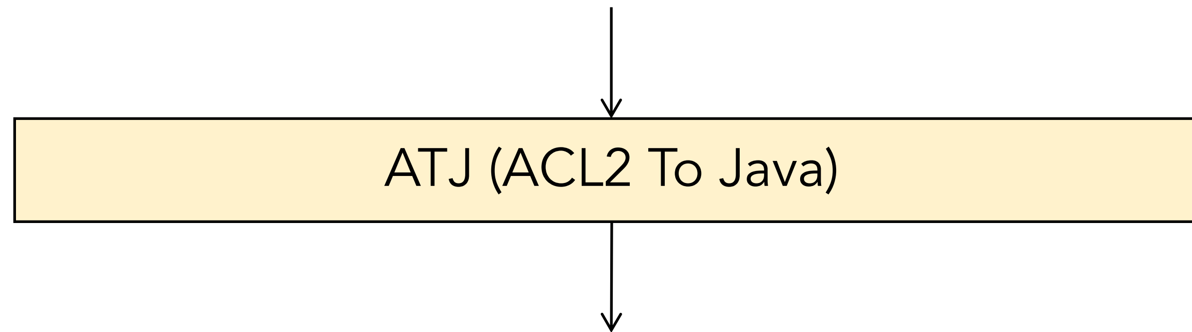
Another example of Java code generated in ATJ's controlled way.

```
(defun swap (bytes1 bytes2 i1 i2)
  (declare (xargs :guard (and (java::byte-array-p bytes1)
                              (java::byte-array-p bytes2)
                              (java::int-value-p i1)
                              (java::int-value-p i2)
                              (java::byte-array-index-in-range-p bytes1 i1)
                              (java::byte-array-index-in-range-p bytes2 i2))))
  (let* ((x1 (java::byte-array-read bytes1 i1))
         (x2 (java::byte-array-read bytes2 i2))
         (bytes1 (java::byte-array-write bytes1 i1 x2))
         (bytes2 (java::byte-array-write bytes2 i2 x1)))
    (mv bytes1 bytes2)))
```



Another example of Java code generated in ATJ’s controlled way.

```
                             (java::byte-array-index-in-range-p bytes2 i2))))
  (let* ((x1 (java::byte-array-read bytes1 i1))
         (x2 (java::byte-array-read bytes2 i2))
         (bytes1 (java::byte-array-write bytes1 i1 x2))
         (bytes2 (java::byte-array-write bytes2 i2 x1)))
    (mv bytes1 bytes2)))
```




```
public static MV_bytearray_bytearray
              swap(byte[] bytes1, byte[] bytes2, int i1, int i2) {
    byte x1 = bytes1[i1];
    byte x2 = bytes2[i2];
    bytes1[i1] = x2;
    bytes2[i2] = x1;
    return MV_bytearray_bytearray.make(bytes1, bytes2);
}
```

Some highlights of the planned future work for ATJ.

Improve 'default-way' Java code:

- More direct, unwrapped Java representations of (some of) the ACL2 values.
- Reduce the use of Java big integers (but inherently hard to match Lisp's speed).

Improve 'controlled-way' Java code:

- Provide constructs to mimic user-defined Java classes in ACL2 code.
- Translate the above constructs to Java classes, with destructive field updates.

Support side effects:

- Add support case by case, also motivated by user needs.
- Support stobjs (i.e. ACL2's single-threaded objects), with destructive updates.

Formal proofs of correctness:

- Modular proofs for each pre-translation, translation, and post-translation (sub)step.
- Based on evaluation semantics of Java and ACL2, being formalized in the ACL2 logic.