A Prototype of CPPCC - Safe Functional Mobile Code in Clean

Daxkobler K., Horváth Z., Kozsik T.

Eötvös Loránd University, Budapest, Hungary

E-mails: mzperx@inf.elte.hu, hz@inf.elte.hu, kto@inf.elte.hu

IFL 2002, Madrid

Checking applicability of mobile code

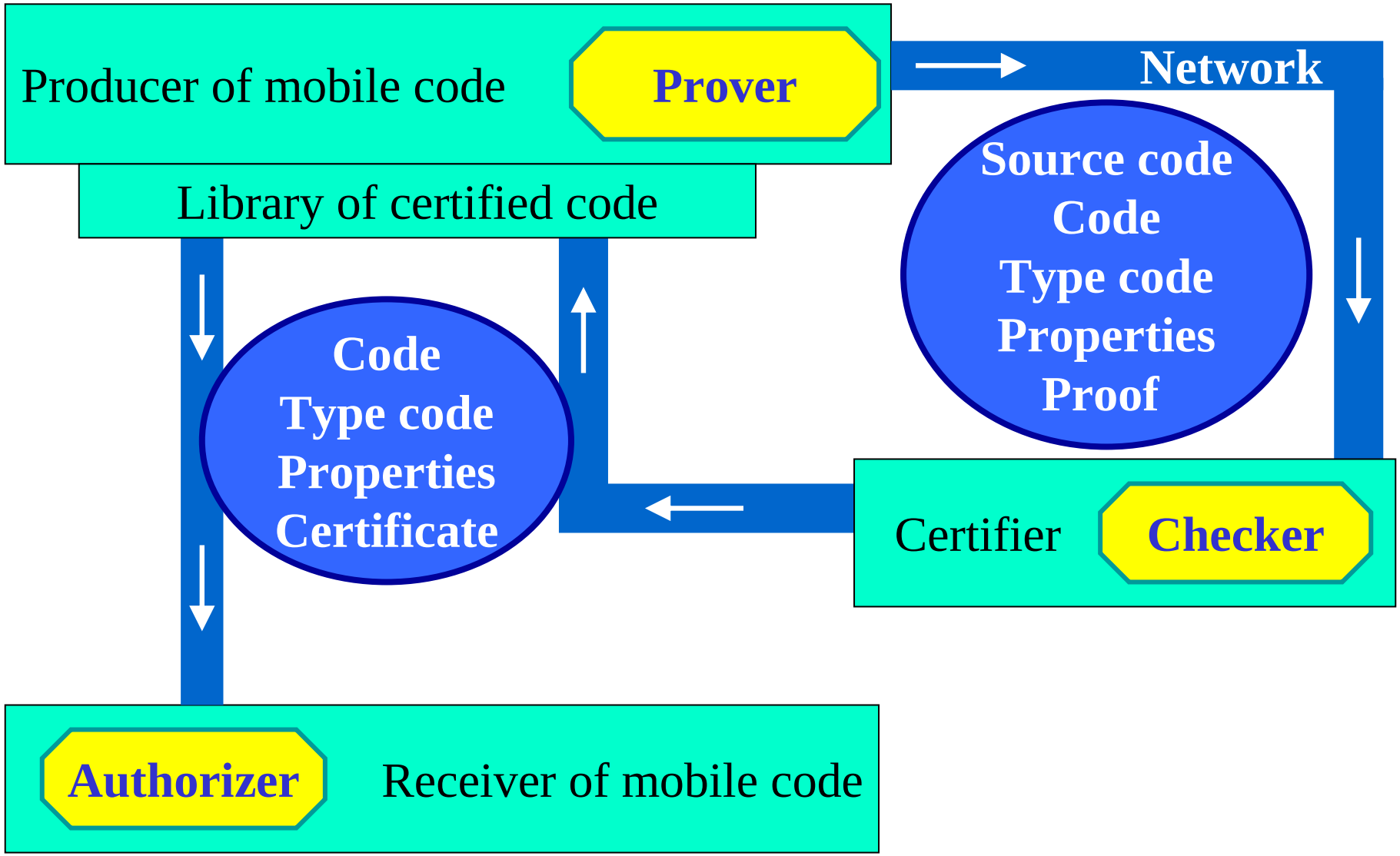
- Well-formed and well-typed code
- Resource usage bounds (memory and execution time)
- Termination
- Has the specified functionality (safety and progress)
  - ✉ Not malicious
  - ✉ Non-malicious code may still contain bugs

CPPCC overview

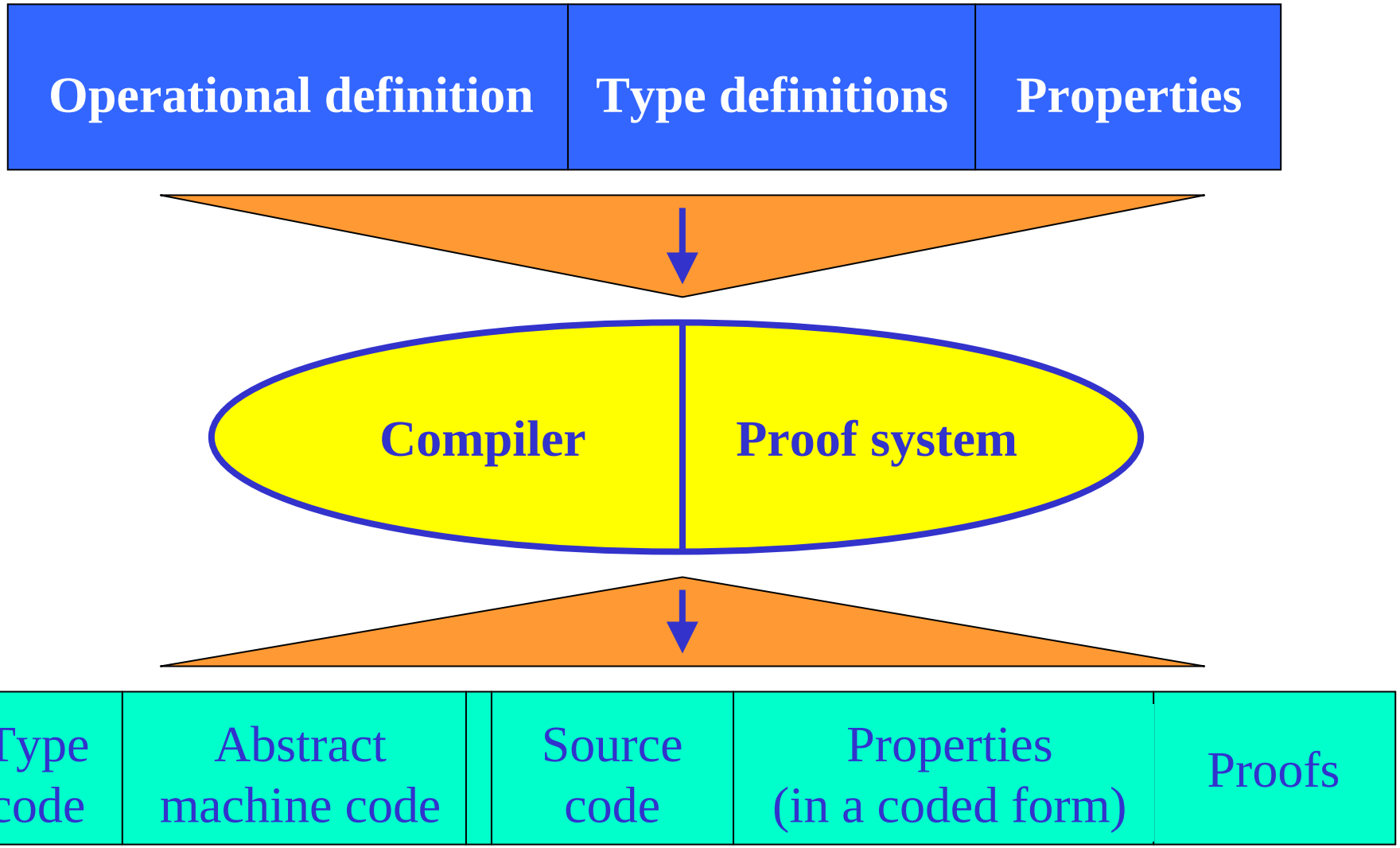
The Certified Proved-Property-Carrying Code (CPPCC): three main components

1. Producer of the mobile code adds properties of the code and their proofs
2. Code receiver will execute the code only after all the checks have been done
3. Certifying authority reduces the work-load of the receiver

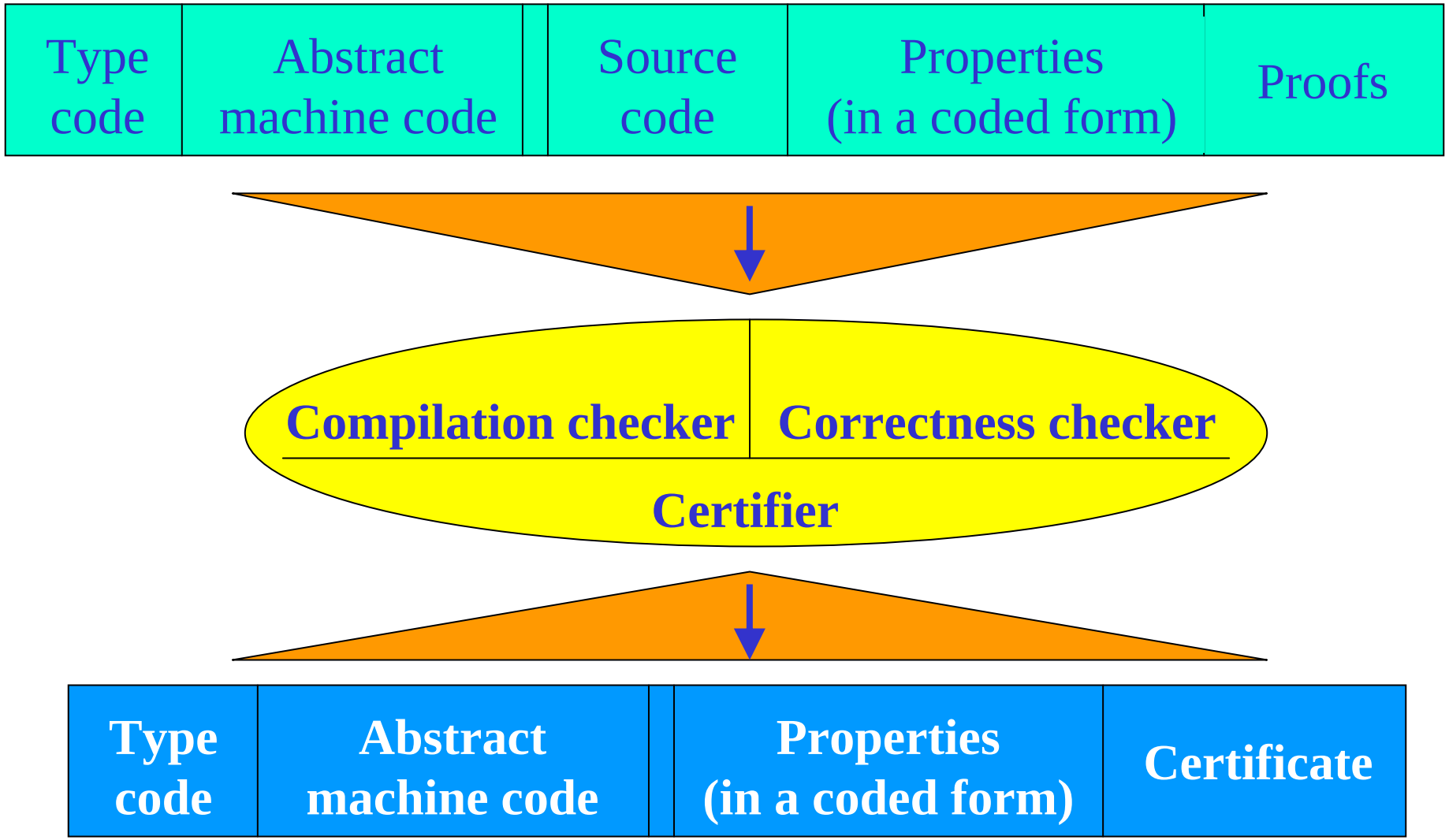
CPPCC architecture



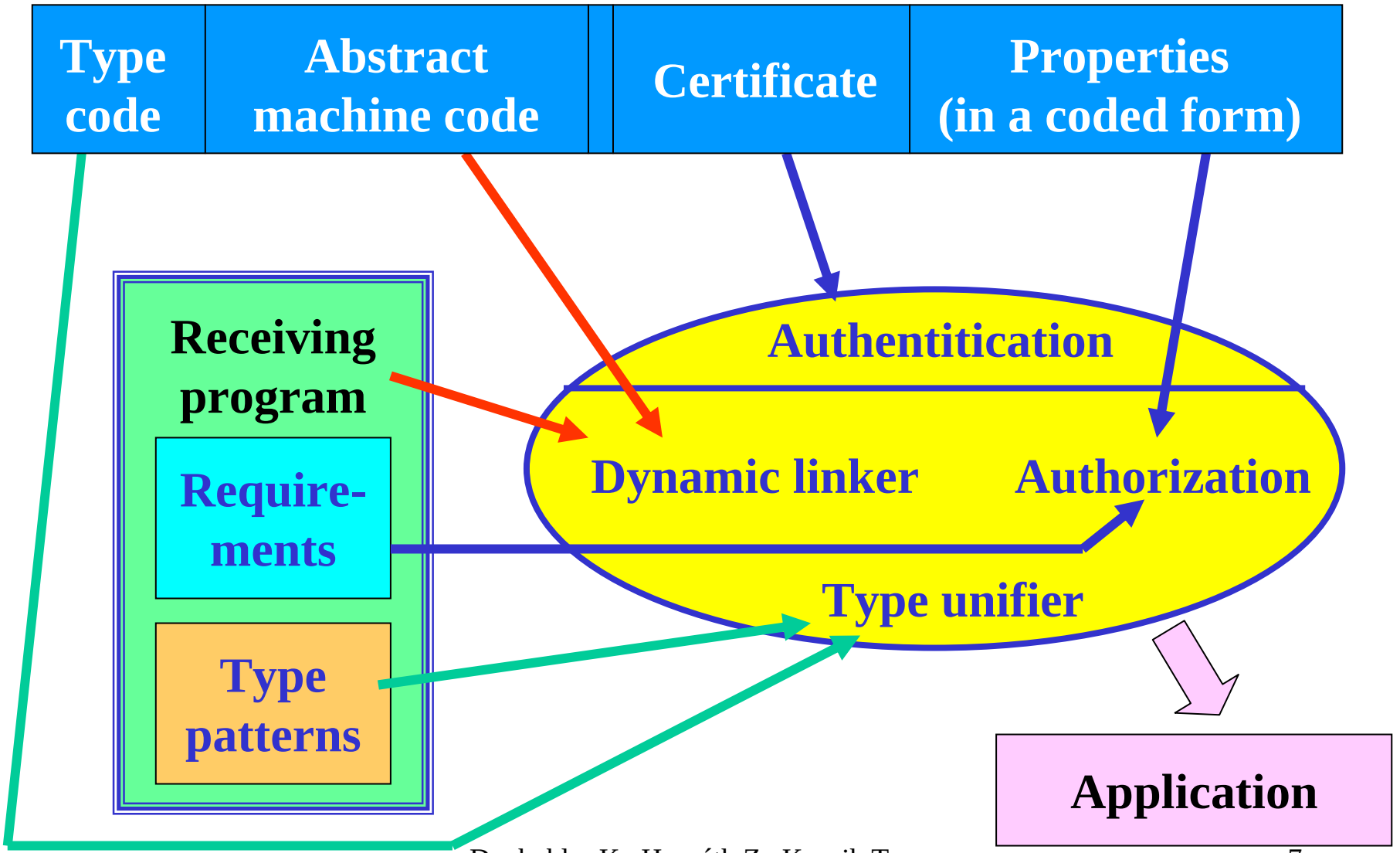
Code producer / sender component



Checker / certifier component



Receiver / authorizer component



CPPCC prototype: aims and tasks

- Demonstrate the feasibility of the architecture concept
- Analyze the relationship between the sender and the checker components (ignore the receiver for the time being)
- Adjust the architecture to the Clean environment
- Implement the sender and checker components
- Concentrate on checking functionality

The Clean environment

- Clean language
- Clean IDE
- Sparkle
- Dynamics (experimental)
- SendDynamic

An example (1)

- **Task**

  Implement the sweep function for bubble sort as dynamic code. Return if list is sorted and if not, also the indexes of a pair of elements in inversion

- **Specification**

  - **Type and variables**

    ```
    sweep :: [a] -> (Bool, Int, Int) | < a
    sweep list -> (isSorted, i, j)
    ```

  - **Specification**

$$\text{list} \neq \bot \rightarrow \text{isSorted} = \forall k \left(k \in [0..\text{list.dom} - 2] \rightarrow \text{list!!} k \leq \text{list!!} (k+1) \right)$$

$$\text{list} \neq \bot \rightarrow \left(\neg \text{isSorted} \rightarrow \text{i} < \text{j} \wedge \text{list!! j} < \text{list!!i} \right)$$

An example (2)

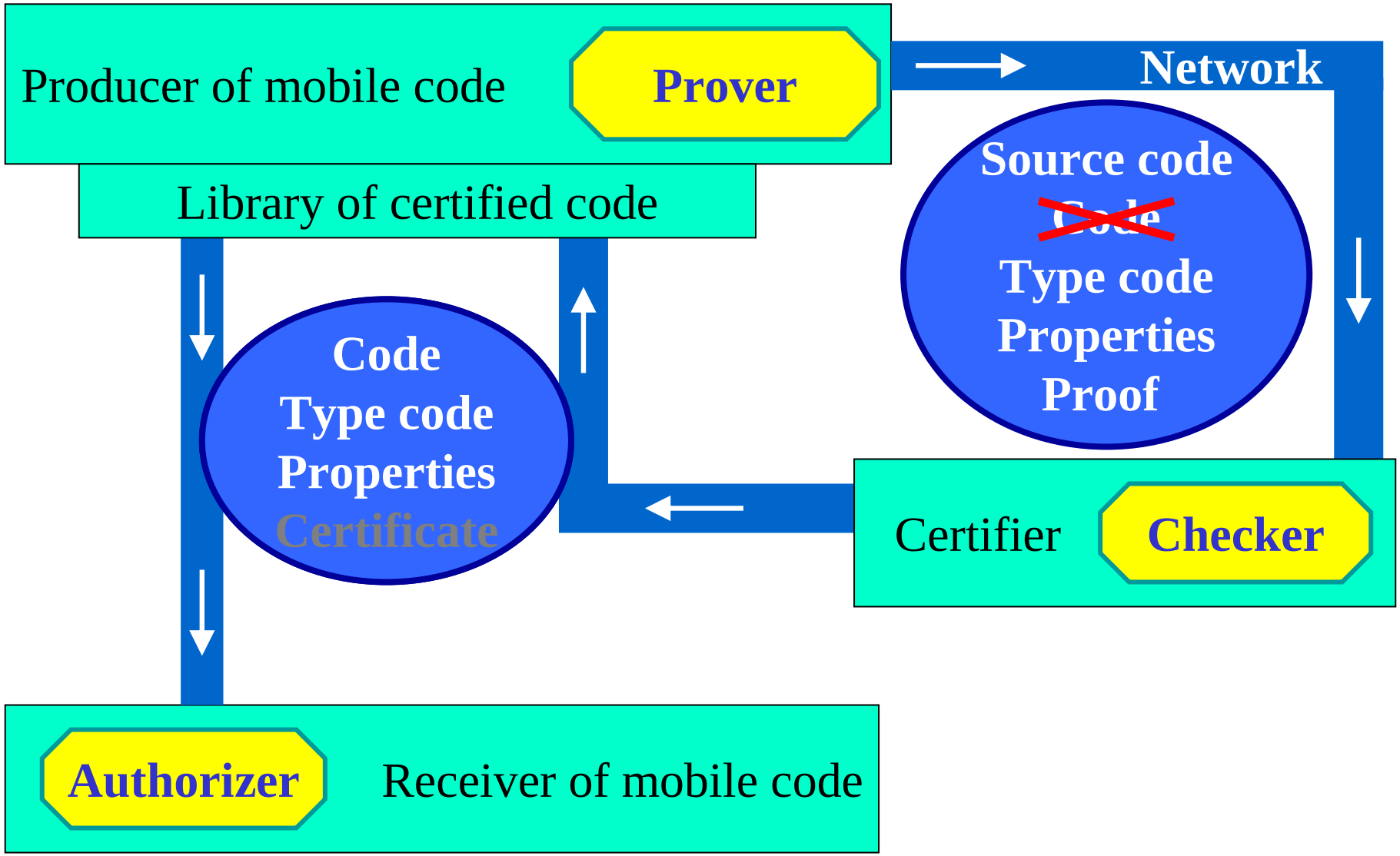
- **Proof**

  A simpler statement (unsorted list has min. 2 elements) required almost 80 tactic applications and 2 lemmas

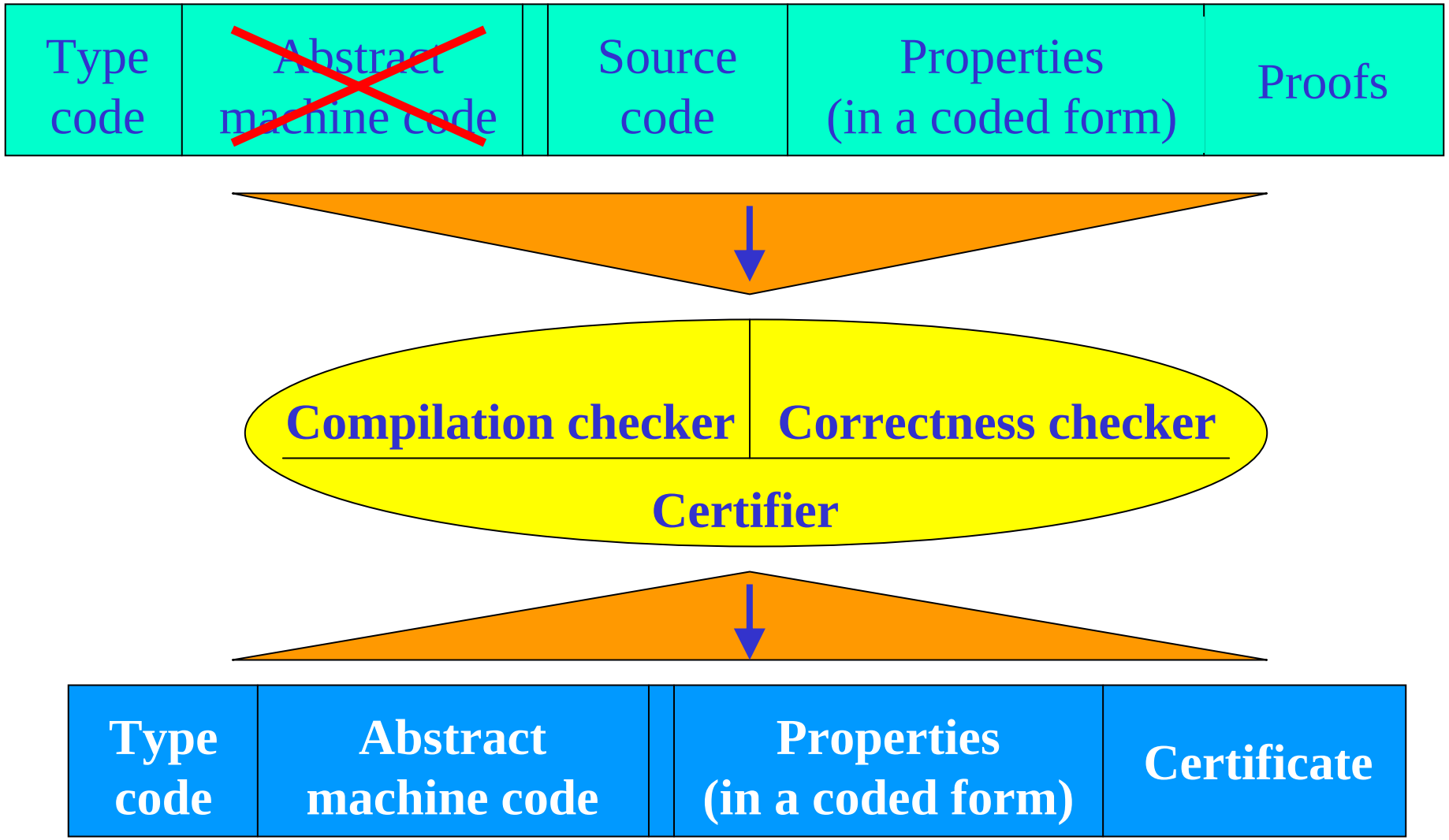
- **Specification in Clean** (not yet implemented)

```
specification = Spec (
  \list -> (isSorted, i, j)
where
  list <> undef -> (isSorted = (forall k in
    0..list.dom-2): list!!k <= list!!(k+1)
  list <> undef -> (not isSorted -> i < j and
    list!!j < list!!i)
)
[...]
semanticCheck specification typeCheckedFunction
```

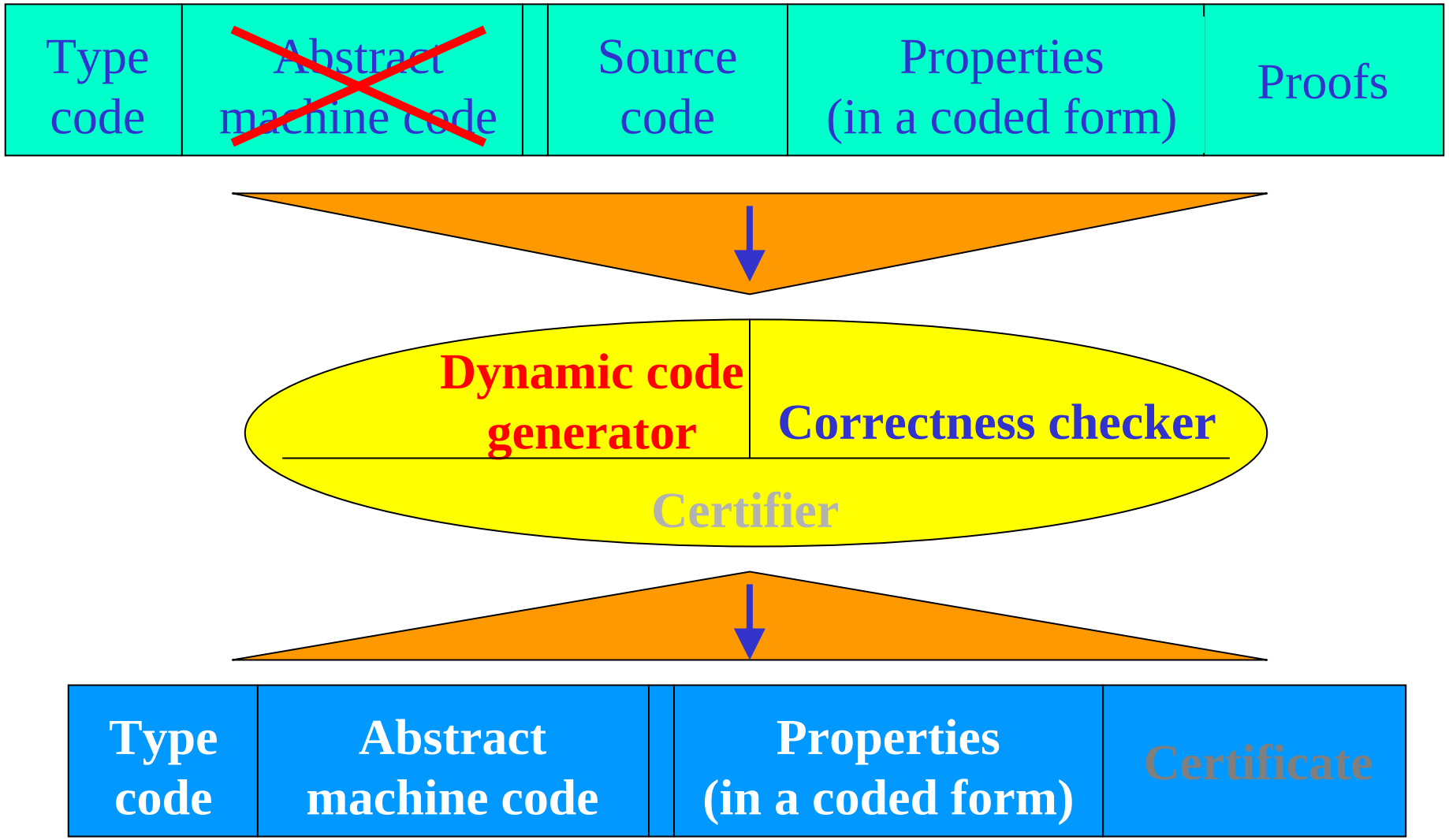
Changes to the architecture



Changes to the checker component



Changes to the checker component



Data format and communication

Types of data:

- Source code, dynamic code, proofs (as stored by Sparkle): large amount of data arranged in several files
- Proved code properties, additional information (names, etc.): small amount of data, stored in simple text files
- SendDynamic also uses a fixed set of files

⇧ Data are sent in a single compressed archive file

Implementation details

Code producer

- Started by the programmer
- Uses the CleanIDE project information to collect files

Certifier

- (Should be) triggered by the arrival of the archive
- Property checker uses Sparkle's source code to parse the input and check the proofs
- a generating program is "written" and compiled on the fly

Related work

PCC by George. C. Necula

Comparison

- VCGen component depends on the safety policy
- Typical implementation: memory and time bound checks vs. semantic checks using first-order logic reasoning
- Machine-code level vs. source-code level reasoning
- Fully automatic PCC implementations exist (certifying compiler generates proof)

Conclusions

- Proof-of-concept implementation only
- Wide range of programme properties allowed
- Proof checking seems to be indeed efficient

Future work

Complete the prototype and...

- Integrate the code properties and proofs into the dynamic code
- Check the dynamic code efficiently instead of generating (Certificate/source code checksum added by the compiler, for instance).
- Check resource bounds.
- Case studies and extensive performance tests