Defining and Proving Invariants in Clean

Zoltán Horváth, *hz@inf.elte.hu*
Tamás Kozsik, *kto@inf.elte.hu*
Máté Tejfel, *matej@inf.elte.hu*

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Content

- Temporal properties in imperative programs.
- Why would we like to use **temporal** properties in **functional** programs ?
- How can we do this ?
- How can we prove them ?
- Examples …

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Temporal logic

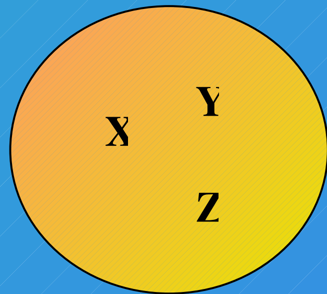
- Language for specifying properties of reactive and distributive systems.
- Widely used for reasoning about sequential and parallel ***imperative*** programs.
- Describe how the values of the program variables (the so-called program state) **vary in time**.
- Complex temporal logical operators can be constructed ("always", "sometimes",

Supported by OTKA T037742 and János Bolyai Research Scholarship.

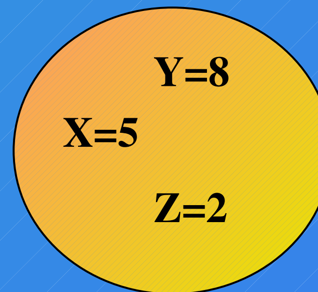
State Place

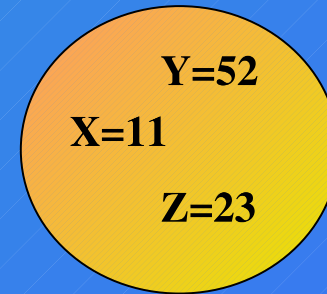
State Place:

INT * INT* INT



States:



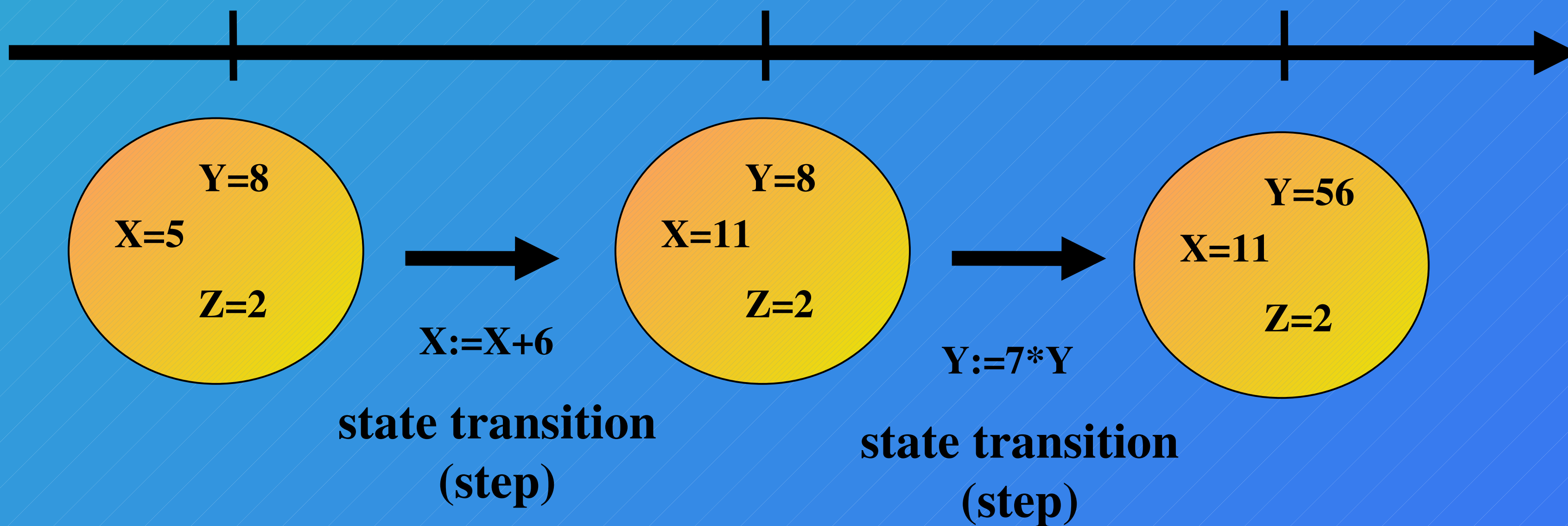


Supported by OTKA T037742 and János Bolyai Research Scholarship.

Temporality

(Imperative view)

Time (program execution)



Properties

X < Y	**Y < X**	**X < Y**
X > Z	**X > Z**	**X > Z**

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Temporal properties
(Unity)

Unity: Chandy, K. M., Misra, J.:
Parallel program design: a foundation. Addison-Wesley, 1989.

- P, Q are properties and **prog** is a program.

Invariant

P **INV prog** Q

Unless

P **UNLESS prog** Q

Ensures

P **ENSURES prog** Q

Leads-to

P **LEADSTO prog** Q

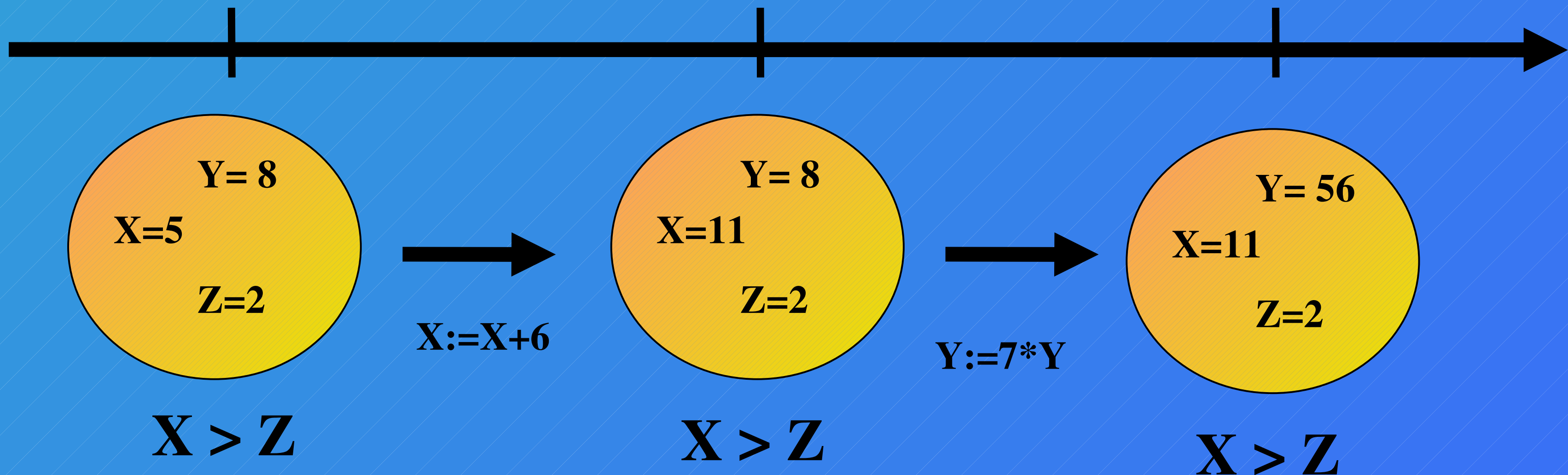
Supported by OTKA T037742 and János Bolyai Research Scholarship.

Invariant

P **INV prog** Q : P holds during the program execution, if initially Q holds.

X > Z INV prog (X=5 ∧ Z=2)

```
prog {
   X:=X+6;
   Y:=7*Y   }
```



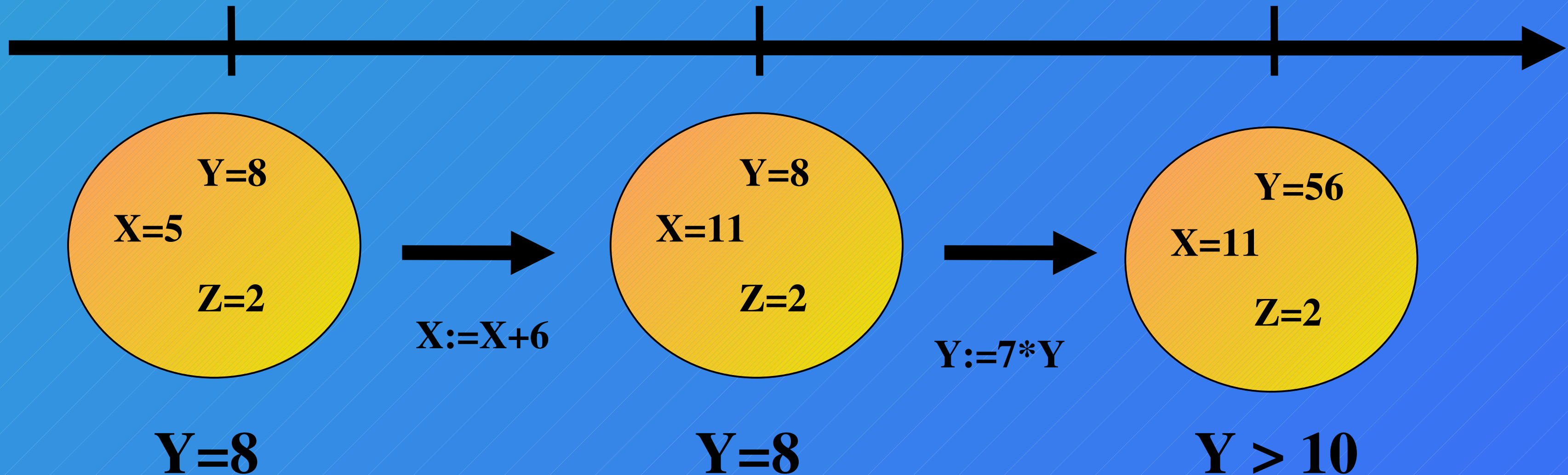
Supported by OTKA T037742 and János Bolyai Research Scholarship.

Unless

P **UNLESS prog** Q : during the execution of the program if once P holds, it remains to hold at least until Q holds.

(Y=8) UNLESS prog (Y>10)

```
prog {
   X:=X+6;
   Y:=7*Y   }
```



Supported by OTKA T037742 and János Bolyai Research Scholarship.

Unless

P UNLESS prog Q

Correct execution (the property holds) :

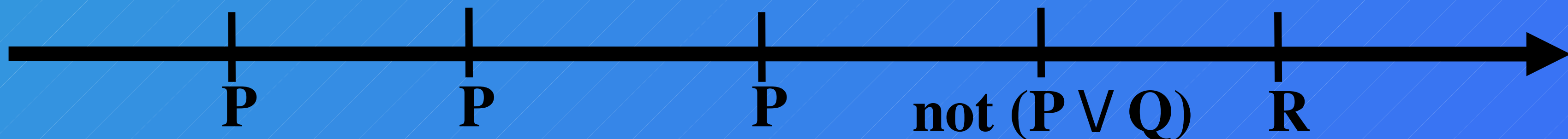
P P P P P

P P P Q Q

P Q not (P ∨ Q) P Q

not P not P not P not P not P

Incorrect execution (the property doesn't hold) :

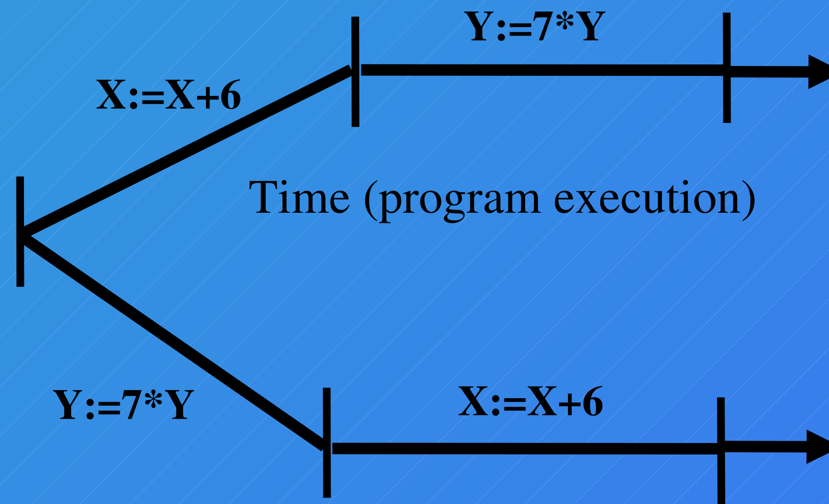


Supported by OTKA T037742 and János Bolyai Research Scholarship.

Parallel execution

```
prog {
   X:=X+6      Y:=7*Y   }
```

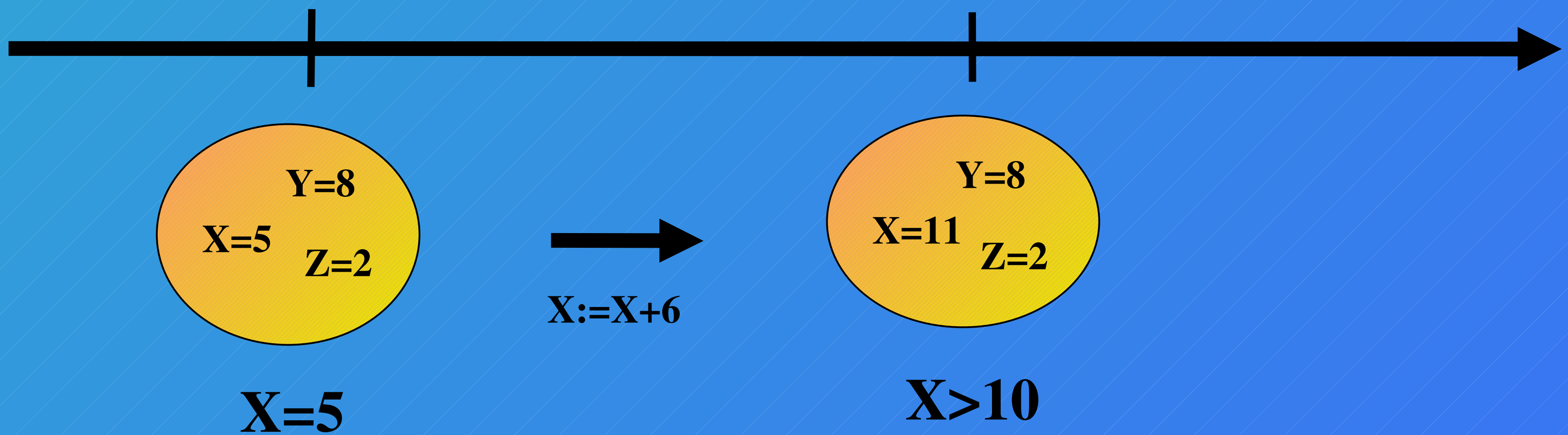
- We don't know the executive order.
- We have to analyse all possible sequence.



Supported by OTKA T037742 and János Bolyai Research Scholarship.

Weakest precondition operator

wp(s,R) : - it is a condition.
- if it holds in a state, then after the execution of **s** R will hold.



wp(X:=X+6, X>10) = ($X^{\leftarrow x+6}$ > 10) = (X+6 > 10) = X > 4

$X = 5 \Rightarrow X > 4$

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Weakest precondition operator

- P **INV prog** Q : $(Q \Rightarrow P) \wedge (\forall s \in$ **prog** $: P \Rightarrow wp(s, P))$

  for all state transitions of the **prog** program

- P **UNLESS prog** Q : $\forall s \in$ **prog** $: P \wedge \neg Q \Rightarrow wp(s, P \vee Q)$

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Basic problem

- In a functional language (like Clean) the values of the variables are ***constants***.
- **Don't** vary in time.
- It seems that temporality has **no** meaning.
- **Why** do we use temporal properties ???

BUT…

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Temporal logic in FP

- We have **uniqness** type
- We have **Object IO**, where
  - we have **reactive programs**
    - with **States**
- It is very similar as the imperative case

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Temporal logic in FP

- We calculate this values from each other
- After we calculated the **new** one we “throw” the **old** one
- We can create an **abstract object** and consider this values as different values of this object
- It is very similar as the imperative case

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Abstract objects

- Object abstraction

    - we consider a series of values computed from each other as different states of the same abstract object

- For this abstract object we can already define and prove **temporal** properties

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Modified Clean source

- Two additional syntax element to Clean:

  - \`.#.\` for steps (with similar syntax as \`#\` in Clean)

  - \`.|.\` for objects (with syntax :

    .|. object_identifier original_identifier)

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Abstract objects

- `exec1 x y z`

  `# x = x + 6`

  `# y = 7*y`

  `= (z, x, y)`

- exec1 x_1 y_1 z_1

  \# $x_2 = x_1 + 6$

  \# $y_2 = 7*y_1$

  $= (z_1, x_2, y_2)$

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Abstract objects

- exec1 x_1 y_1 z_1

  $\#$ x_2 = x_1 + 6

  $\#$ y_2 = 7*y_1

  = (z_1, x_2, y_2)

- exec1 (.|. Obj_x x_1) y_1 z_1

  .#. (.|. Obj_x x_2) = (.|. Obj_x x_1) + 6

  $\#$ y_2 = 7*y_1

  = (z_1, (.|. Obj_x x_2) , y_2)

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Abstract objects

- ```
  exec1 x₁ y₁ z₁
      # x₂ = x₁ + 6
      # y₂ = 7*y₁
      = (z₁, x₂, y₂)
  ```

- ```
  exec1 (.|. Obj_x x₁) (.|. Obj_y y₁) (.|. Obj_z z₁)
      .#. (.|. Obj_x x₂) = (.|. Obj_x x₁) + 6
      .#. (.|. Obj_y y₂) = 7*(.|. Obj_y y₁)
       = ((.|. Obj_z z₁), (.|. Obj_x x₂), (.|. Obj_y y₂))
  ```

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Sparkle

- theorem prover
- specially constructed for Clean
- properties are expressed in a basic logic:

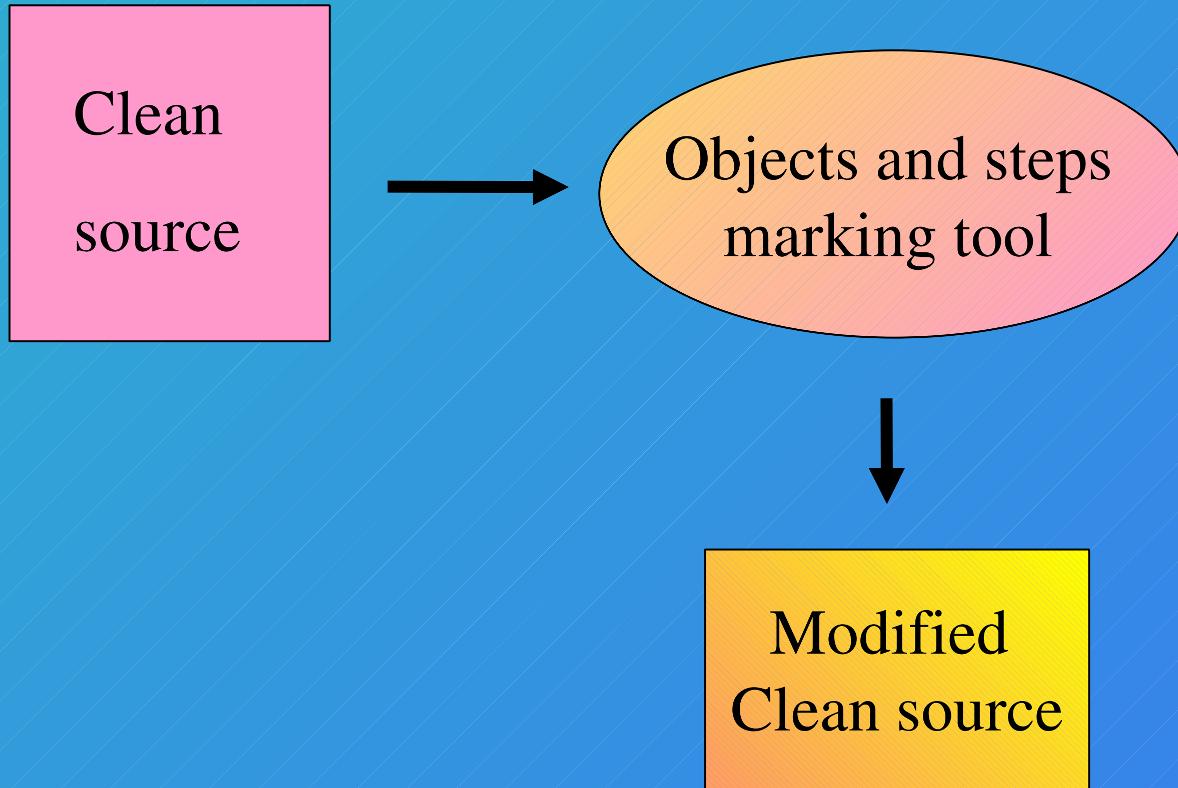
  equality (on expressions), negation, implication, conjunction, disjunction, equivalence (iff), universal quantification and existential quantification
- reasoning in Sparkle takes place on Core language (Core-Clean)
  - subset of Clean
  - application, sharing and case distinction
  - semantics based on lazy graph rewriting

Supported by OTKA T037742 and János Bolyai Research Scholarship.

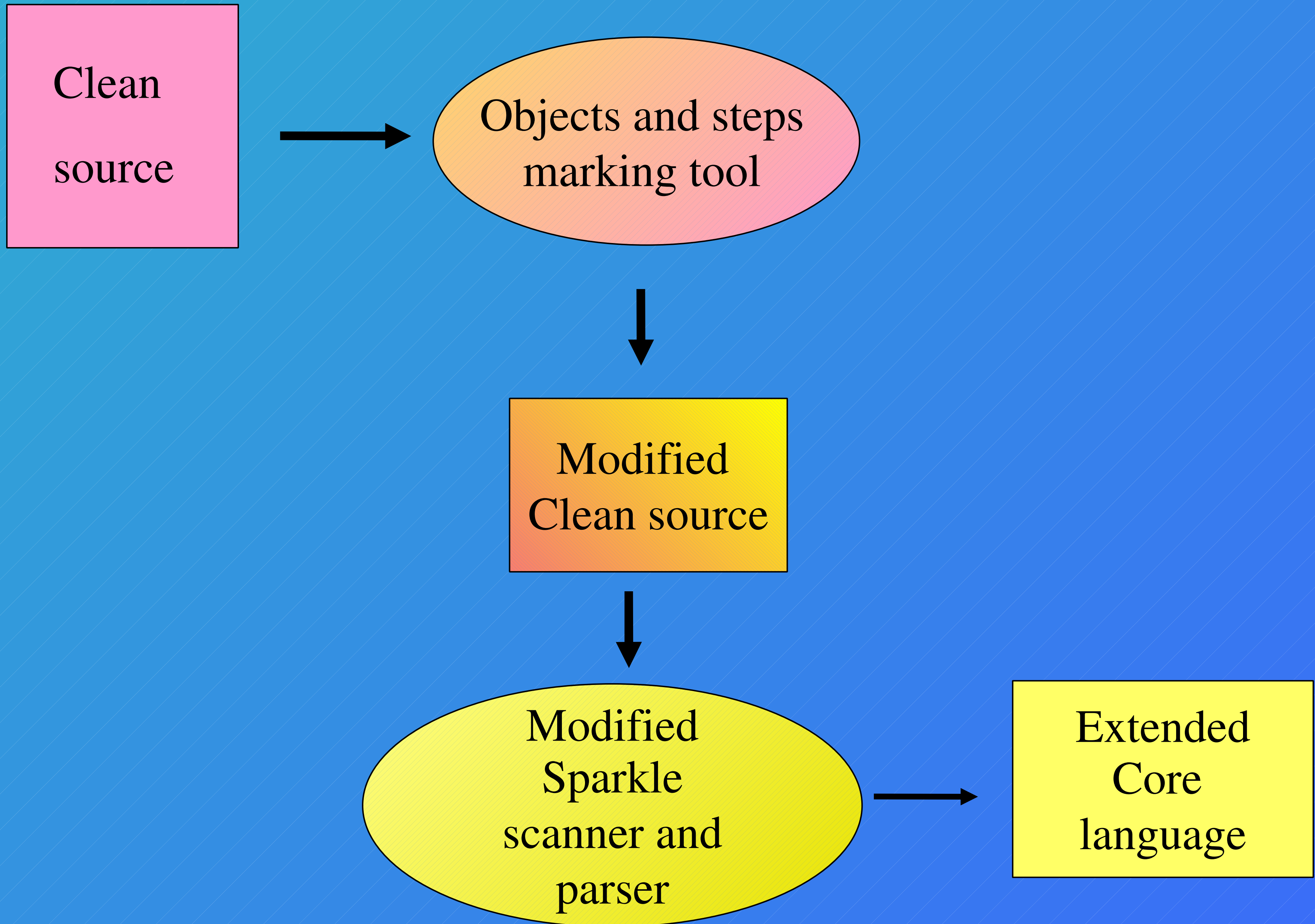
The modifications in Sparkle

- Modified scanner, parser
- New syntax element in Core language
  - CObj_Var
  - CStep
- Modified definition of functions
- New tactics for objects (ongoing work)

Supported by OTKA T037742 and János Bolyai Research Scholarship.



Supported by OTKA T037742 and János Bolyai Research Scholarship.



Supported by OTKA T037742 and János Bolyai Research Scholarship.

Invariant Example

The proved property:

$$(obj_x > obj_z)\ \ INV\ \ (exec1\ x\ y\ z)\ \ (z = 2 \wedge x = 5)$$

- ```
exec1 (.|. Obj_x x1) (.|. Obj_y y1) (.|. Obj_z z1)
  .#. (.|. Obj_x x2) = (.|. Obj_x x1) + 6
  .#. (.|. Obj_y y2) = 7*(.|. Obj_y y1)
   = ((.|. Obj_z z1), (.|. Obj_x x2), (.|. Obj_y y2))
```

Supported by OTKA T037742 and János Bolyai Research Scholarship.
```

The weakest precondition in functional case

- Calculating the weakest precondition in a functional environment is a simple rewrite rule (rewriting the postcondition according to the substitution defined by the step)

Supported by OTKA T037742 and János Bolyai Research Scholarship.

The proof

- Initially it holds:

$$(z = 2 \land x = 5) \rightarrow (obj_x_var_0 = x \land obj_y_var_0 = y \land obj_z_var_0 = z) \rightarrow (obj_x_var_0 > obj_z_var_0)$$

we replaced the objects (obj_x, obj_y and obj_z) with variables (obj_x_var0, obj_y_var0 and obj_z_var0)

```
exec1 (.|. obj_x x1) (.|. obj_y y1) (.|. obj_z z1)
   ...
```

Supported by OTKA T037742 and János Bolyai Research Scholarship.

The proof

- The first step preserves it:

$$(obj_x_var1_old > obj_z_var1_old) \rightarrow (obj_x_var1 = obj_x_var1_old + 6) \rightarrow (obj_y_var1 = obj_y_var1_old) \rightarrow (obj_z_var1 = obj_z_var1_old) \rightarrow (obj_x_var1 > obj_z_var1)$$

```
...
   .#. (.|. obj_x x2) = (.|. obj_x x1) + 6
...
```

Supported by OTKA T037742 and János Bolyai Research Scholarship.

The proof

- The second step preserves it:

(obj_x_var2_old > obj_z_var2_old)
→ (obj_y_var2 = 7 * obj_y_var2_old)
→ (obj_x_var2 = obj_x_var2_old)
→ (obj_z_var2 = obj_z_var2_old)
→ (obj_x_var2 > obj_z_var2)

```
...
 .#. (.|. obj_y y2) = 7*(.|. obj_y y1)
...
```

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Invariant Example 2.

- Simple database of financial transactions
- Transaction abstraction
  the date and the sum of the financial transaction

```
::ListData :== (!Int, !Int)
```

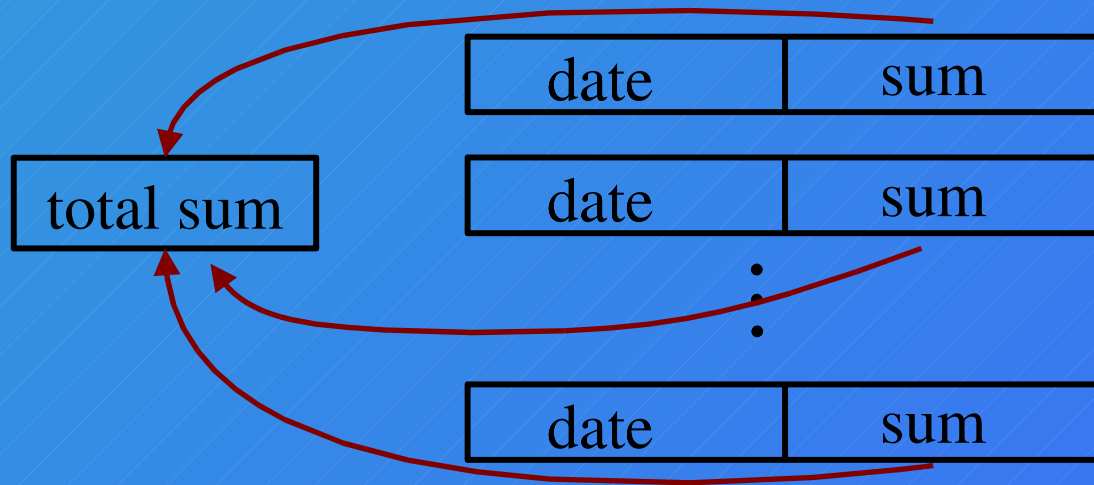
date	sum

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Invariant Example 2.

- Database abstraction
  the total sum of the sum of the transactions and
  the list of the transactions

```
:: DB :== (!Int, !List)
:: List = Nil | Cons !(!Int, !Int) !List
```



Supported by OTKA T037742 and János Bolyai Research Scholarship.

The operations of the example

- Creating a new database from the old one inserting a new transaction to it

```
insertDB:: !(!Int,!Int) !DB -> DB
insertDB (x1,x2) (sum,list) =
                (sum + x2, Cons (x1,x2) list)
```

- Computing the sorted version of the database (by date)

```
sortDB:: !DB -> DB
sortDB (x,list) = (x, sort_ins list)
```

Supported by OTKA T037742 and János Bolyai Research Scholarship.

The operations of the example

- Creating a new database from the old one by removing the first transaction

```
removeFirst:: !DB -> DB
removeFirst (x,Nil) = (x,Nil)
removeFirst (x, Cons (y1,y2) ys) = (x - y2, ys)
```

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Object abstraction

- The original function:

```
ex1 :: !DB !(!Int,!Int) -> DB
ex1 db p
    # db1 = insertDB p db
    # db2 = sortDB db1
    # db3 = removeFirst db2
    = db3
```

- The object abstraction:

```
ex1_o (.|. db_o db) p
 .#. (.|. db_o db1) = insertDB p (.|. db_o db)
 .#. (.|. db_o db2) = sortDB (.|. db_o db1)
 .#. (.|. db_o db3) = removeFirst (.|. db_o db2)
  = (.|. db_o db3)
```

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Proved property

- Our invariant property for function ex1_o:
  the sum field of the database always contains
  the total sum of the sum of transactions and the

  database is evaluable

- Initial condition for function ex1_o :
  in our special example it is the same as the
  previous property and additionally the second
  parameter is also evaluable

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Proved property

$P \text{ inv } (S, Q)$

where

P = (fst db_o = sumList (snd db_o) ∧ eval db_o)

S = ex1_o db_o p

Q = (fst db_o = sumList (snd db_o) ∧ eval db_o ∧ eval p)

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Proved property

The `sumList` function calculates the sum of the second components of the elements of the list

```
sumList:: !List -> Int
sumList Nil = 0
sumList (Cons (x1,x2) Nil) = x2
sumList (Cons (x1,x2) xs) = x2 + sumList xs
```

Supported by OTKA T037742 and János Bolyai Research Scholarship.

The steps of the proof

- Invariant tactic determine the steps, which have to prove by Sparkle
- In current example:

  - initially the property holds

```
fst db = sumList (snd db) ∧ eval db ∧ eval p
        →   fst db = sumList (snd db) ∧ eval db
```

which is trivial (5 lines in Sparkle)

Supported by OTKA T037742 and János Bolyai Research Scholarship.

The steps of the proof

- the atomic steps keep the property
  - the first step

```
fst db = sumList (snd db) ∧ eval p
    ∧ eval db ∧ db1 = insertDB p db →

        fst db1 = sumList (snd db1) ∧ eval db1
```

(114 lines and 2 additional theorem ≈ 130 lines in Sparkle)

Supported by OTKA T037742 and János Bolyai Research Scholarship.

The steps of the proof

- the second step

```
fst db1 = sumList (snd db1) ∧
    eval db1 ∧ db2 = sortDB db1 →

      fst db2 = sumList (snd db2) ∧ eval db2
```

(54 lines and 40 additional theorem ≈ 3100 lines in Sparkle)

Supported by OTKA T037742 and János Bolyai Research Scholarship.

The steps of the proof

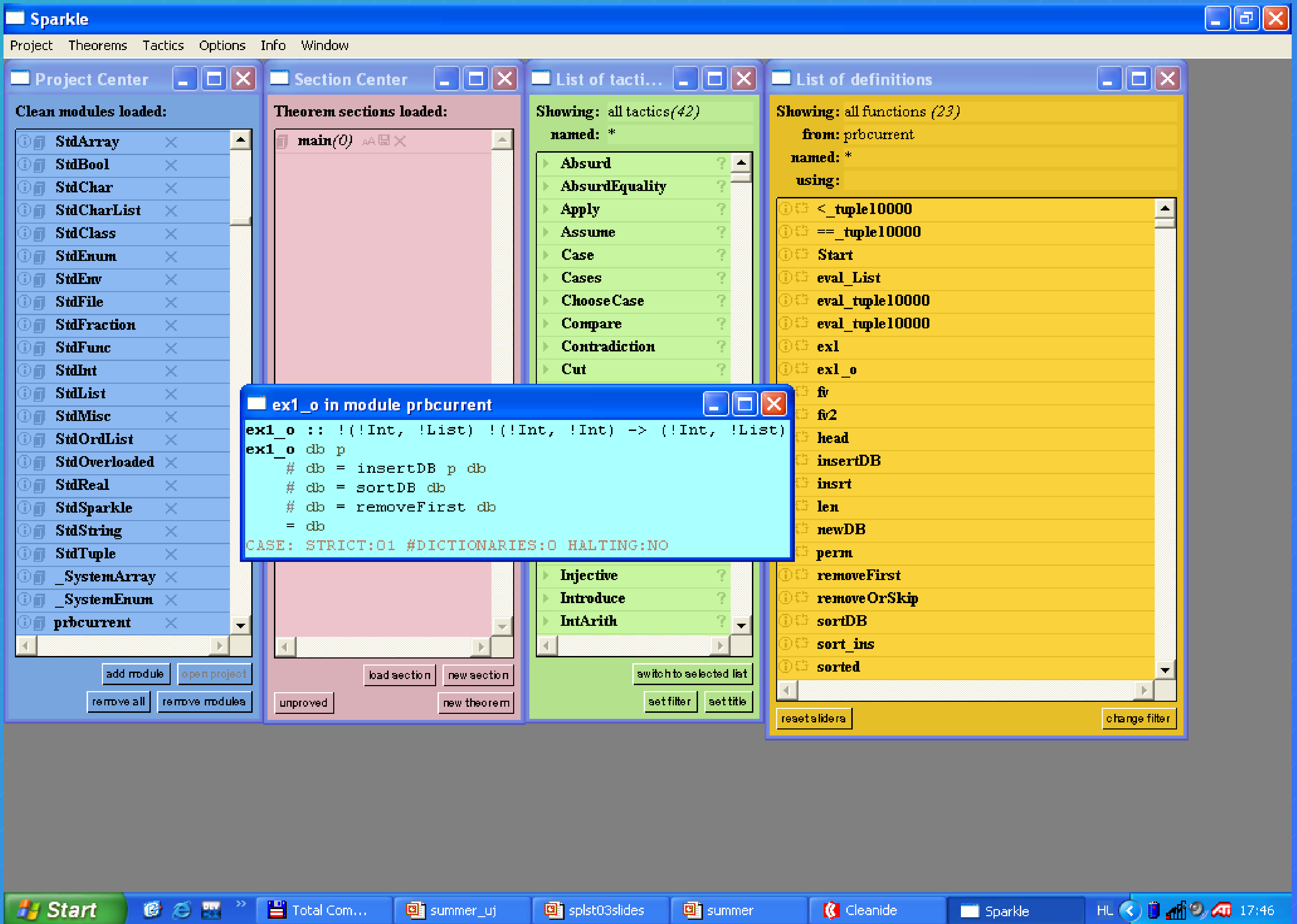
- the third step

```
fst db2 = sumList (snd db2) ∧
     eval db2 ∧ db3 = removeFirst db2 →

          fst db3 = sumList (snd db3) ∧ eval db3
```

(155 lines and 7 additional theorem ≈ 200 lines in Sparkle)

Supported by OTKA T037742 and János Bolyai Research Scholarship.



Supported by OTKA T037742 and János Bolyai Research Scholarship.

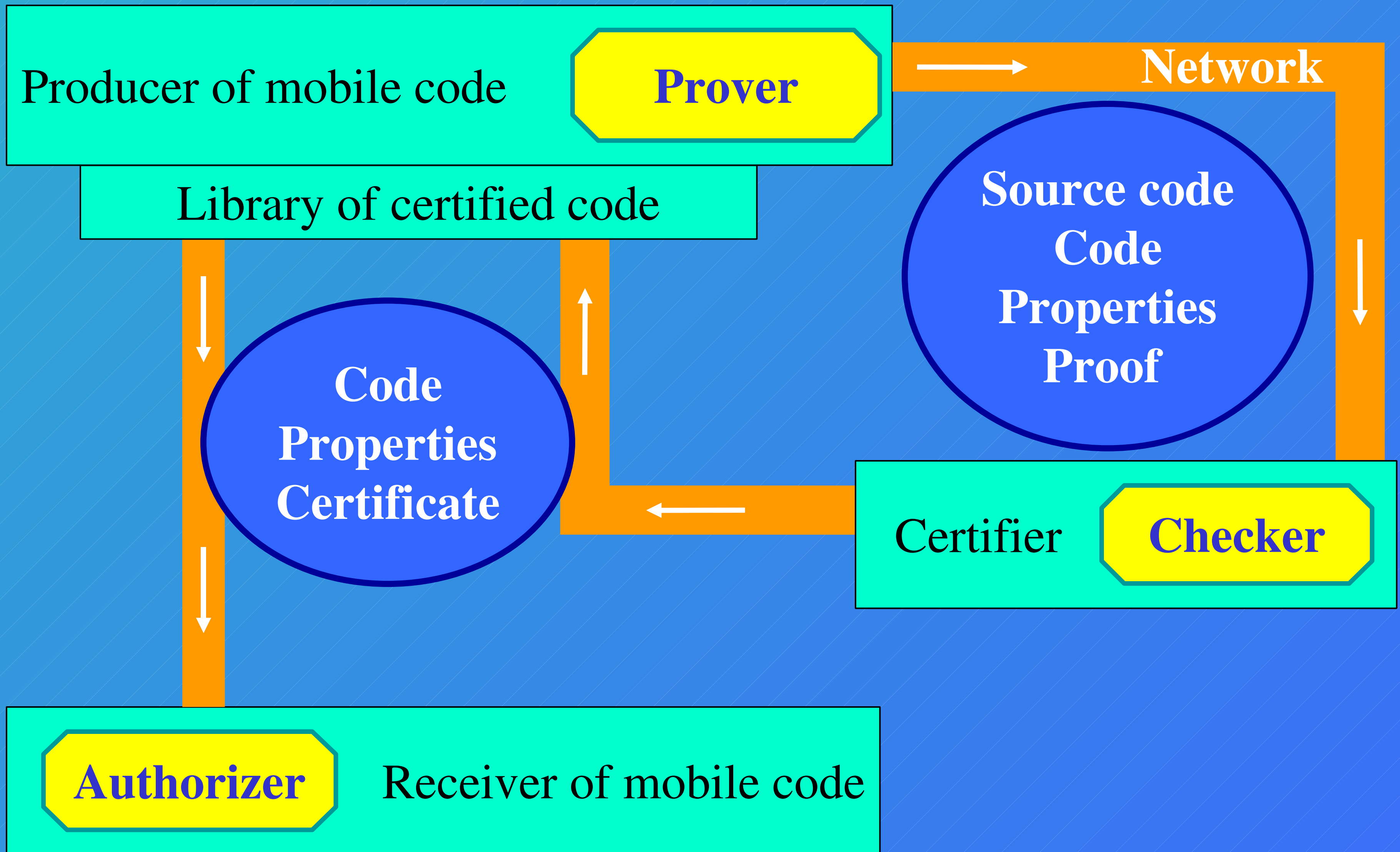
CPPCC overview

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.

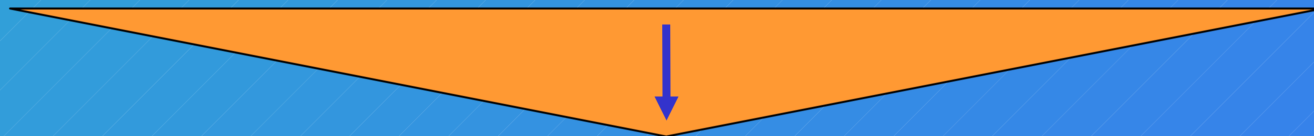
Supported by OTKA T037742 and János Bolyai Research Scholarship.

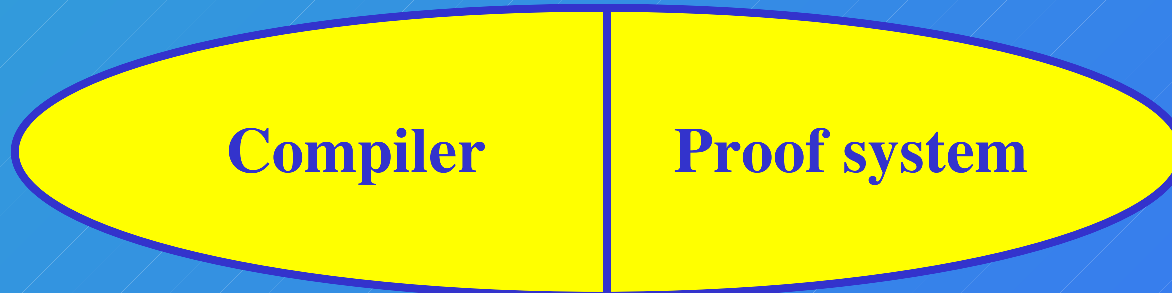
CPPCC architecture

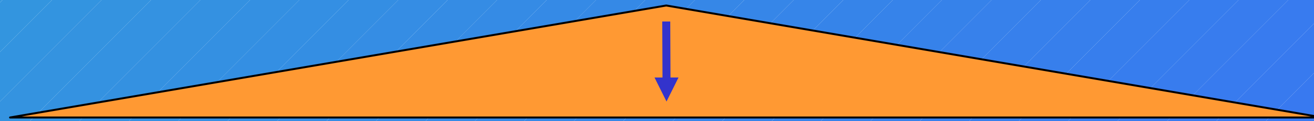


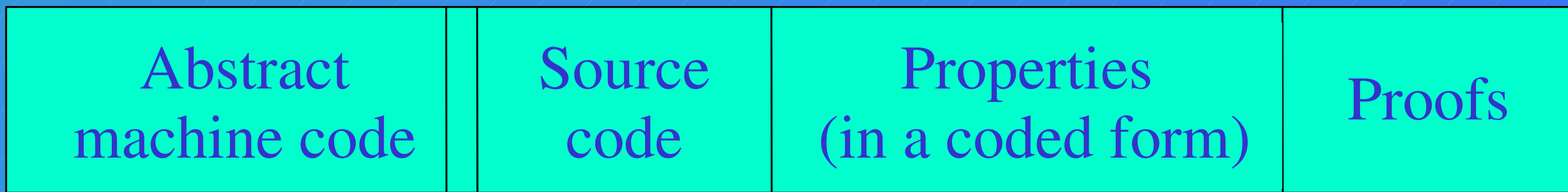
Supported by OTKA T037742 and János Bolyai Research Scholarship.

Code producer / sender component



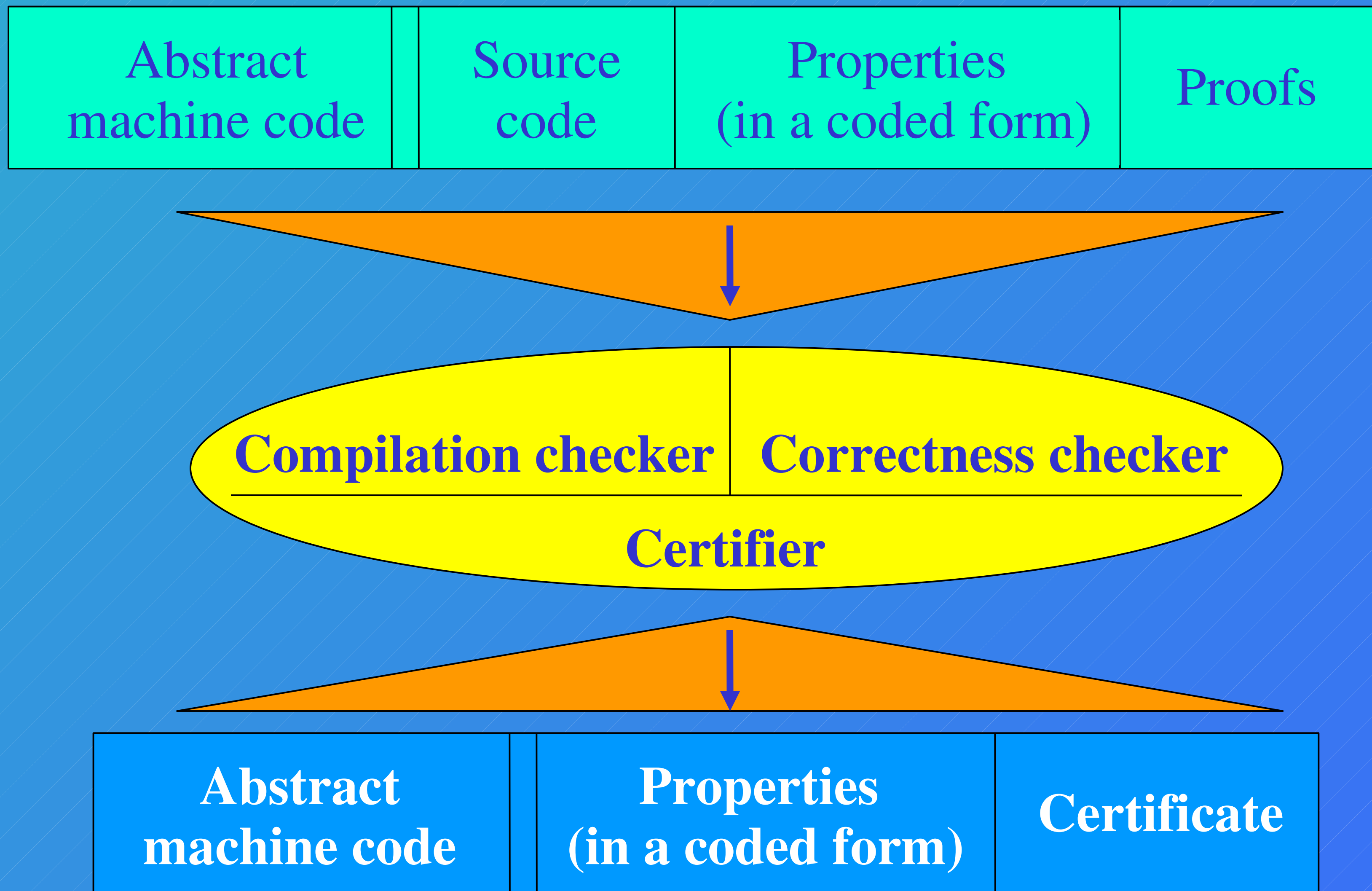






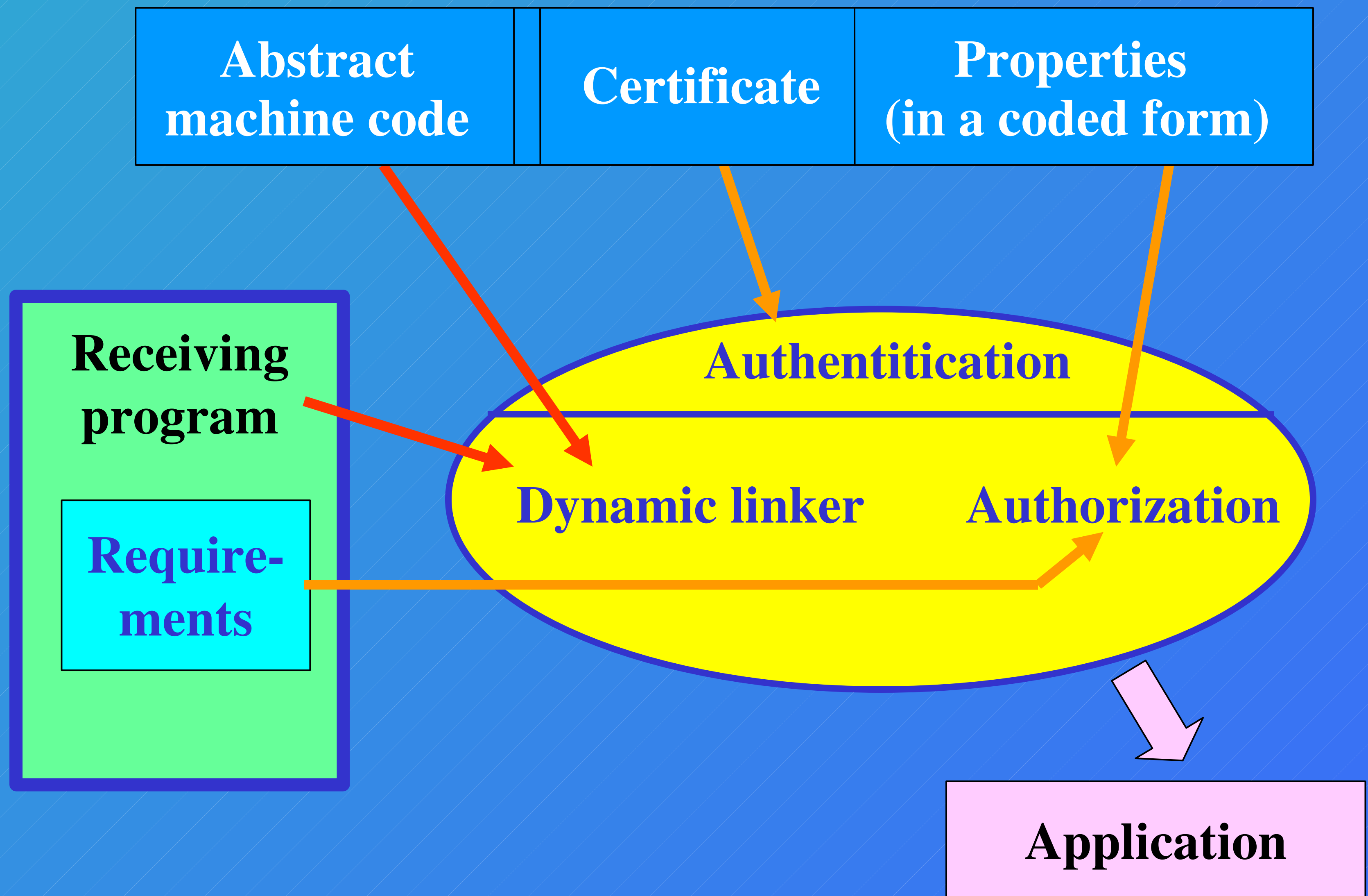
Supported by OTKA T037742 and János Bolyai Research Scholarship.

Checker / certifier component



Supported by OTKA T037742 and János Bolyai Research Scholarship.

Receiver / authorizer component



Supported by OTKA T037742 and János Bolyai Research Scholarship.

Summary

- Temporal logical operators are useful in proving properties of functional programs.
- Object abstraction is introduced for representing state as series of values.
- Temporal properties can be expressed and easy to calculate based on wp.
- Dedicated theorem prover Sparkle is applicable.
- Invariant tactic is implemented.

Supported by OTKA T037742 and János Bolyai Research Scholarship.

Some future work ...

- other temporal properties (progress)
- extension of Sparkle with some new tactics for proof of temporal properties (implementation)

Supported by OTKA T037742 and János Bolyai Research Scholarship.

For Lab…

- Download the following stuffs:
  - **Sections** directory
  - **Sparkle_obj.exe**
  - **inv_lab.icl**

  from **http://plc.inf.elte.hu**

  to **c:\Clean 2.1.1\Tools\Sparkle**

Supported by OTKA T037742 and János Bolyai Research Scholarship.