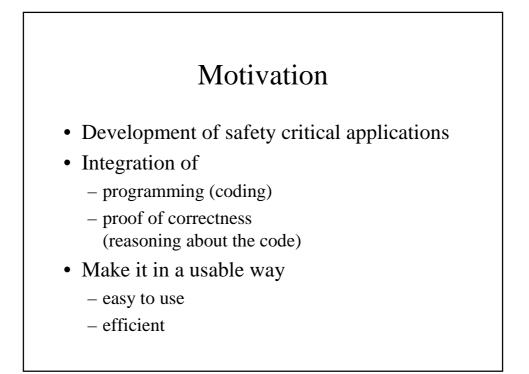
## Subtyping with Strengthening Type Invariants

**KOZSIK Tamás** Eötvös Loránd University, Budapest

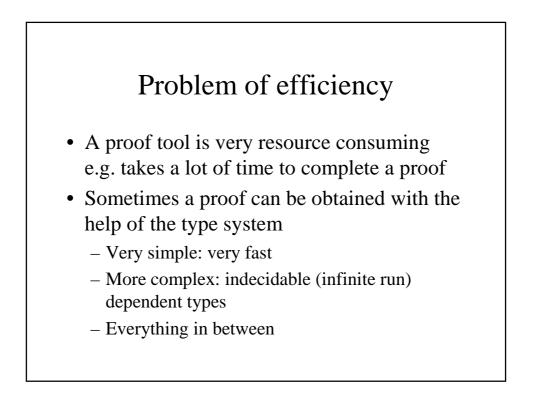
**Diederik VAN ARKEL, Rinus PLASMEIJER** "Clean group", University of Nijmegen, The Netherlands

Supported by OTKA T037742.



## Vision

- Integrate a proof tool in the Clean environment
  - into the programming environment (IDE)
     prove properties while writing the program (these are often very simploe properties)
  - into the run-time environment reason about programs during run-time enhance reliability of mobile code

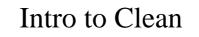


## Key idea

• Program properties expressed as type invariants

x: Natural x: Integer with  $x \ge 0$ 

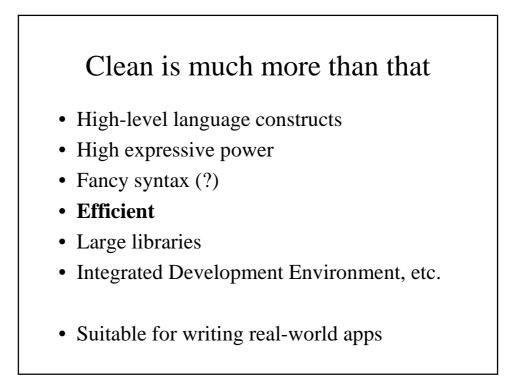
- Propagation of properties: verified by type system
  - If I add two Natural numbers, the result is also a Natural number
- Polymorphism is gained with subtyping
  - Natural is a special Integer, that is Natural ≤ Integer



- Functional programming language
  - lazy, pure, polymorphic, higher-order
  - semantics based on Term Graph Rewriting Systems
- Program = collection of function definitions + an expression to evaluate (khmm...)
- No assignment, no "imperative variables", only "mathematical" ones
  - variable: sg. that can hold an arbitrary value of a certain type
- Program execution: evaluation of the Start expression

## Why is it good?

- A program is an executable specification
- Just maths...
- Easy to learn FP, easy to do FP
- Referential transparency: no side effects
  - less error-prone
  - better quality software: understand/modify/reuse
- Easy to reason about programs formally
  - mathematical proofs use referential transparency



#### Some features

- Predefined type constructs: lists, tuples, arrays, records, functions
- Functions are first-class citizens – higher-order
- Flexible type system: algebraic types, parametric polymorphism, type classes, type constructors (higher-order types)
- Strictness annotations (evaluation order)
- Uniqueness attributes (destructive updates)

# Some more...

- Strong type checking
- Type inference
- Modules
- Block structure
- Abstract data types
- Generic programming
- Dynamic typing
- Object IO for the devel. of graphical apps

#### Example: quicksort

```
module qsort
```

```
qsort [] = []
qsort [x:xs] =
qsort [a \\ a <- xs | a < x]
++ [x] ++
qsort [a \\ a <- xs | a > x]
```

Start = qsort [42, 33, 100, 15]

```
Type declaration

qsort :: [a] -> [a] | < a

qsort [] = []

qsort [x:xs] =

qsort [a \\ a <- xs | a < x]

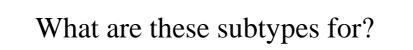
++ [x] ++

qsort [a \\ a <- xs | x > a]
```

### What am I doing?

- Modify the type system of Clean
- Add subtyping with type invariants
- Clean 2.0 compiler offered by KUN

   source code is available
  - ... in Clean ... :-)
- Theory + implementation
- Hoping to do sg. useful, practical

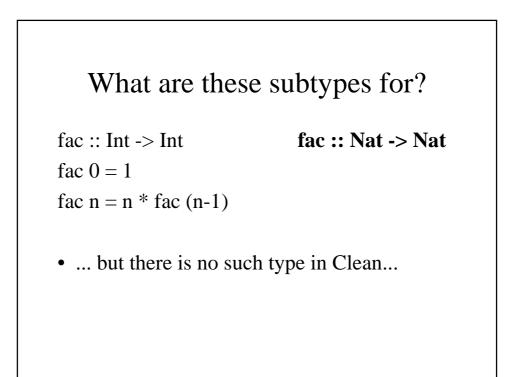


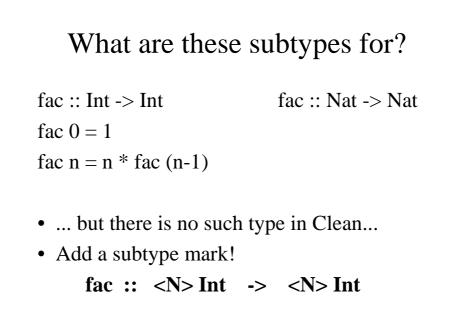
fac :: Int -> Int fac 0 = 1fac n = n \* fac (n-1)

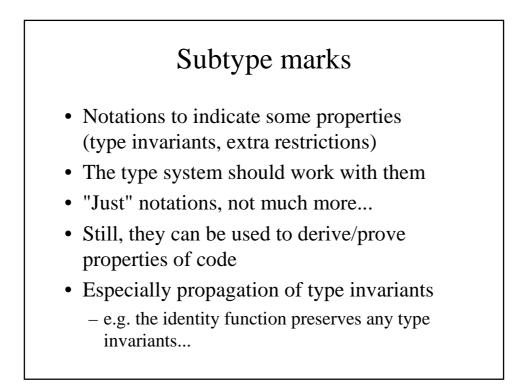
#### What are these subtypes for?

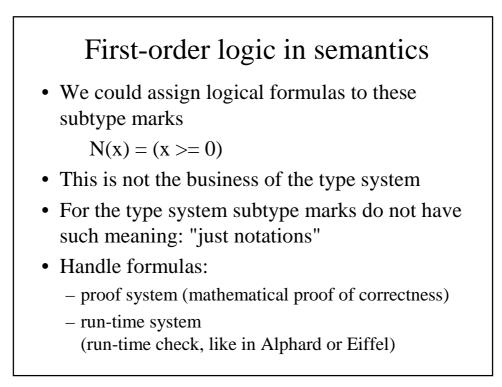
fac :: Int -> Int // only for non-negative arg.
fac 0 = 1
fac n = n \* fac (n-1)

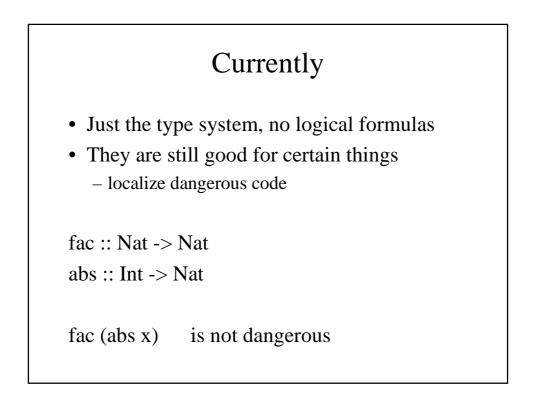
- Here the program aborts for negative numbers
- Things can be worse (do harmful computation)





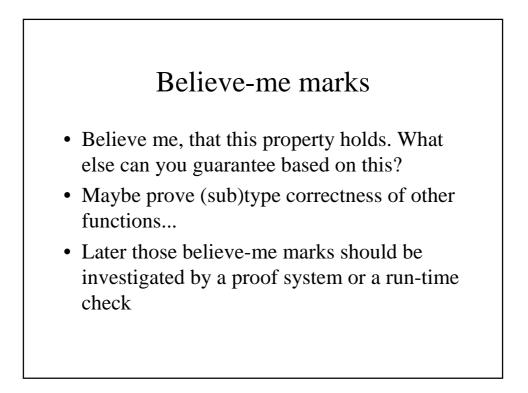




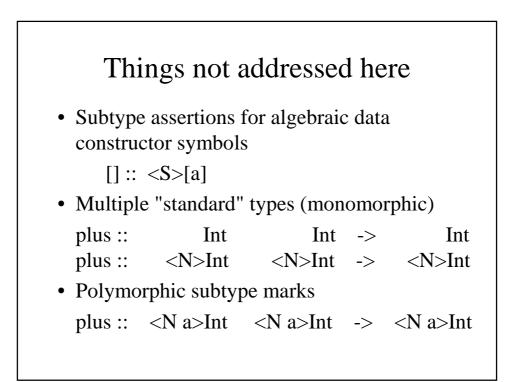


### Later

- Generate code that checks type invariants run-time, namely before and after evaluating a function (several examples...)
- Use a proof system to argue about type invariants
  - Special proof system (dedicated to Clean):
     Sparkle (formerly *C*lean *P*rover *S*ystem)
    - reason about Clean progs, no transformation
    - integrated with IDE

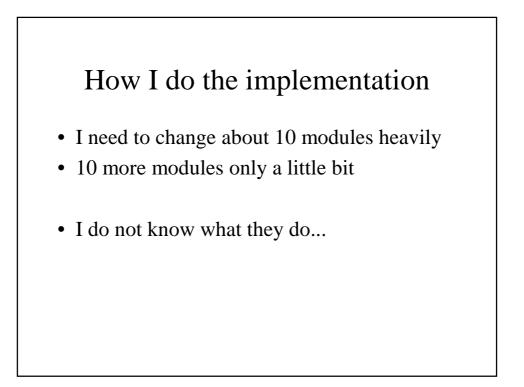


## For example, sorting...



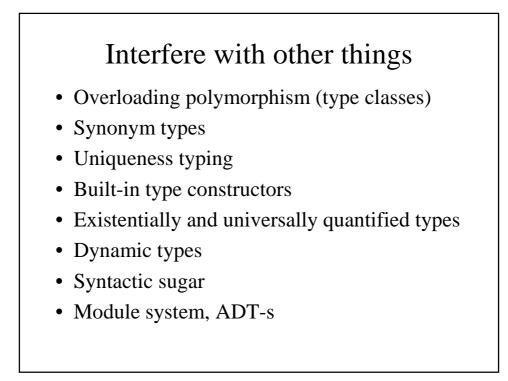
## Implementation difficulties

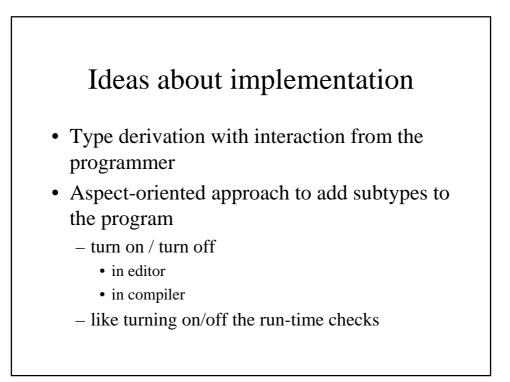
- The Clean compiler is written in Clean
- The front-end is about 50.000 lines (2.500.000 characters)
- Clean programs are shorter than corresponding C programs
  - Rinus says: only one tenth
- Actually, it is not a very nice code... (hacking, not too much abstraction, no comments, no documentation)

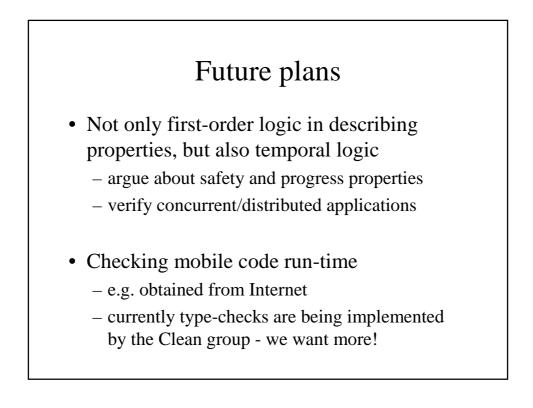


## Main activities

- Scanning
- Parsing
- Collect info
  - syntax tree
  - symbol tables
- Check visibility, etc.
- Type checking / inferencing (unification)







## Plans for me

- Finish this implementation (catch up with theory)
- Increase expressive power
- Eliminate interference with other language concepts not addressed in theory
- Develop large examples (case studies)
- Integrate with proof tool, do run-time checks
- Get the PhD