Taalgebaseerde Softwareveiligheid

Eelco Visser

TU Delft

KNAW | October 31, 2019

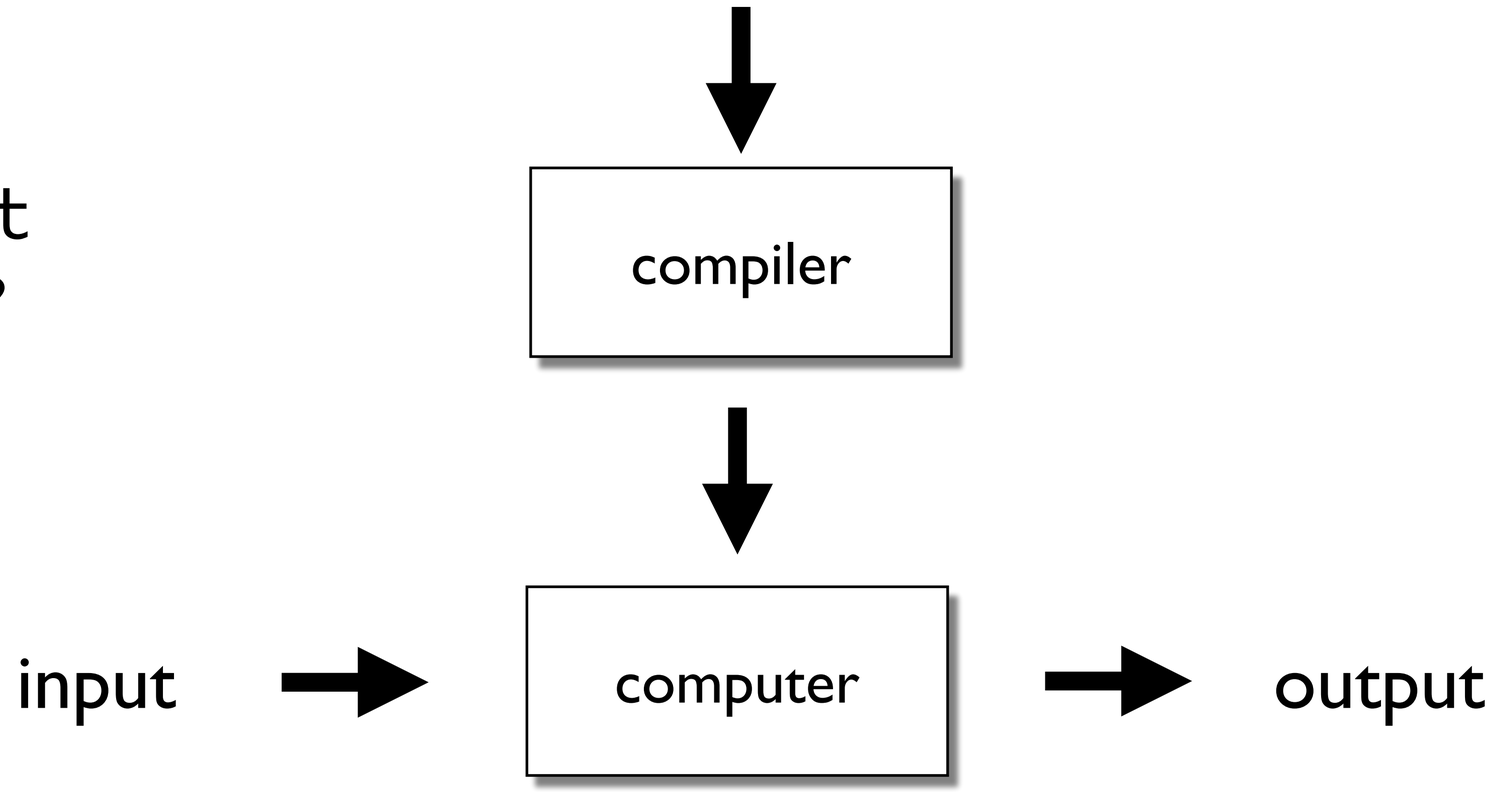
Kunnen we programmeerfouten voorkomen door middel van programmeertalen?

Kunnen we veiligheid van programmeertalen garanderen tijdens het ontwerp?

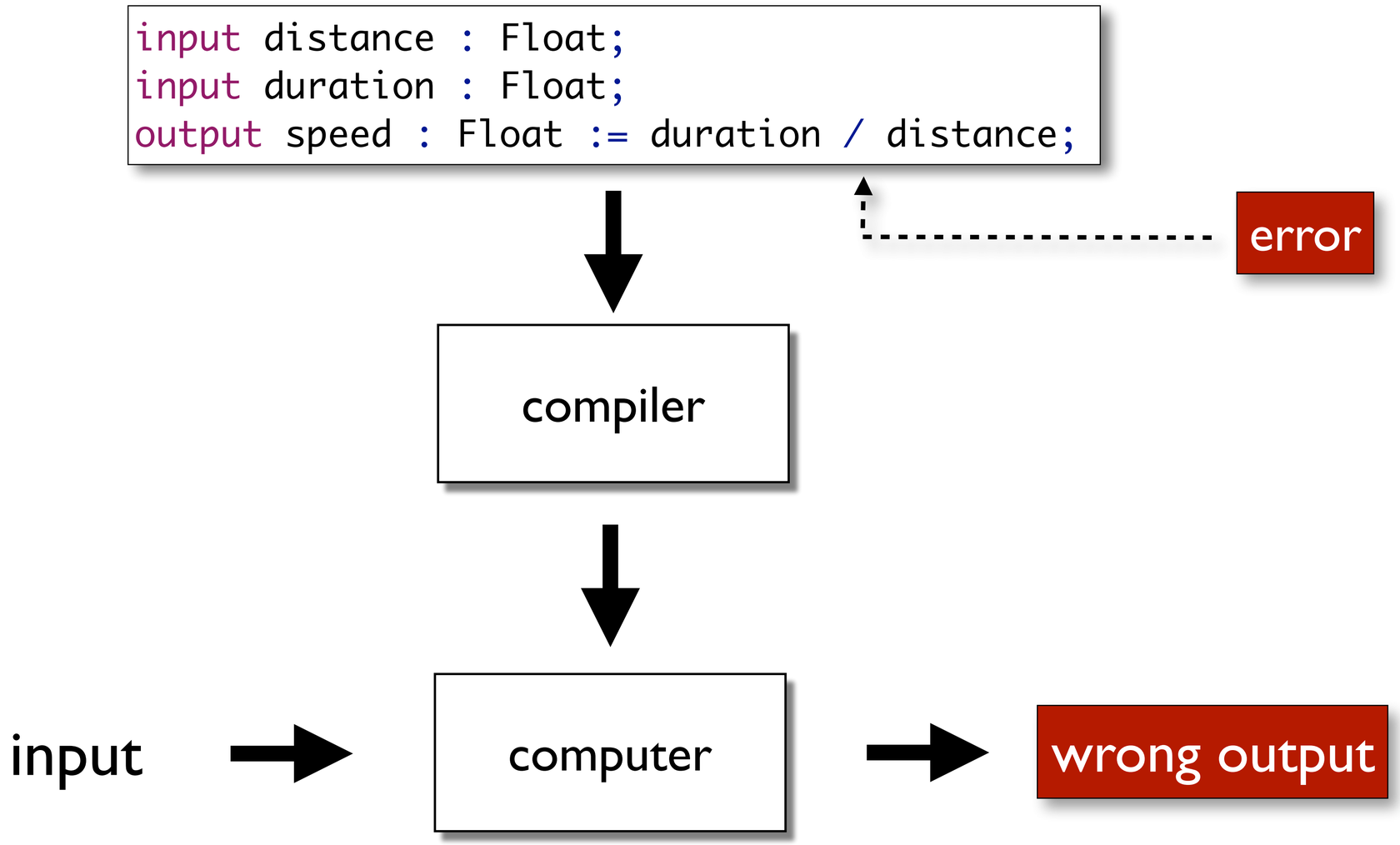
Programmeren met Fysieke Eenheden

```
input distance : Float;
input duration : Float;
output speed : Float := duration / distance;
```

Wat is het probleem?



Programmeren met Fysieke Eenheden



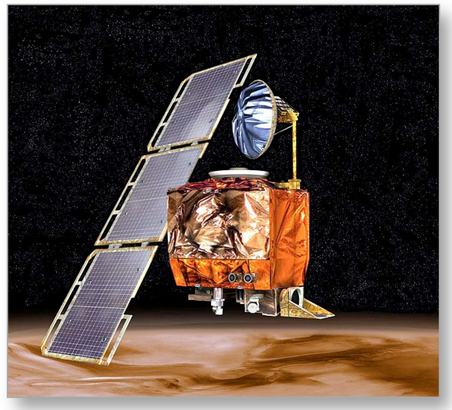
Impact van Programmeerfouten

```
input distance : Float;
input duration : Float;
output speed : Float := duration / distance;
```

error

compiler

input → computer → wrong output



Mars Climate Orbiter

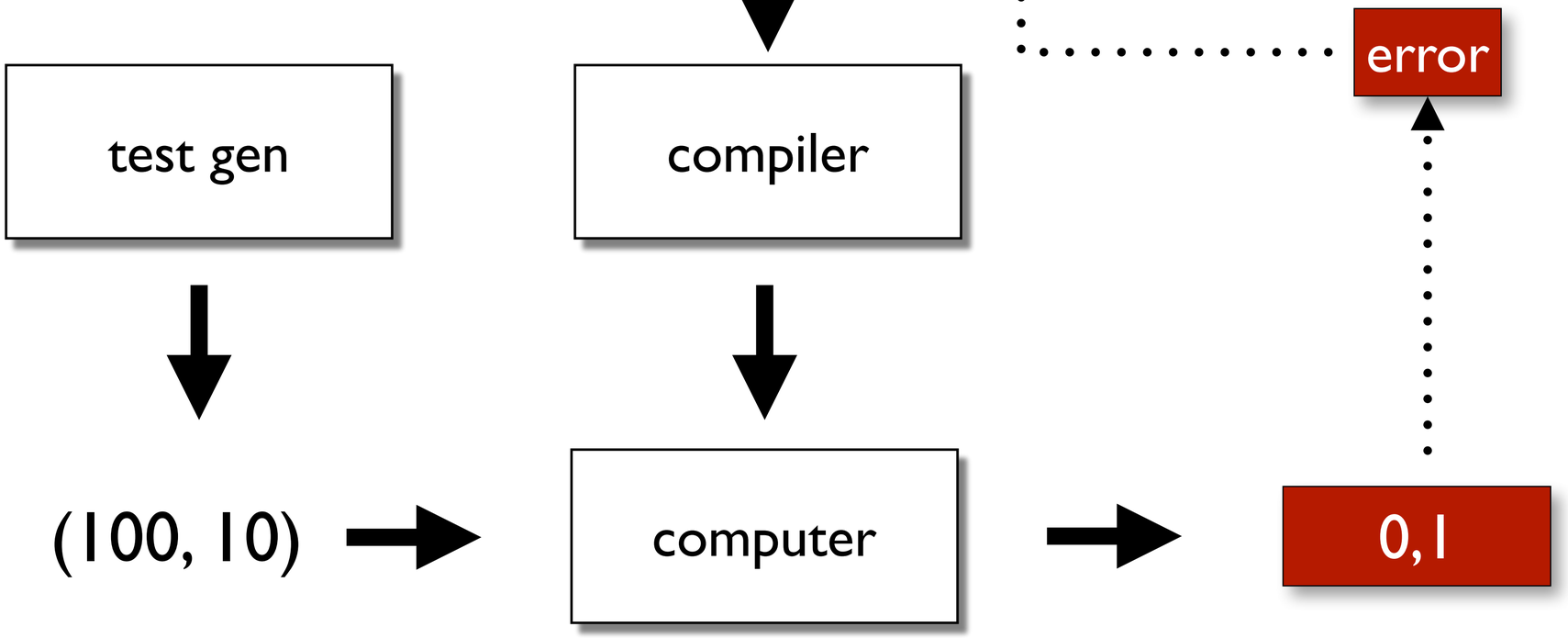
Unit mismatch: Orbiter variables in Newtons, Ground control software in Pound-force.

Damage: ~350 M$

Hoe kunnen we programmeerfouten ontdekken of zelfs voorkomen?

Fouten Vinden door Testen

```
input distance : Float;
input duration : Float;
output speed : Float := duration / distance;
```



Niet vinden van fouten garandeert niet dat programma correct is!

Fouten Voorkomen met Programma Verificatie

```
input distance : Float;
input duration : Float;
output speed : Float := duration / distance;
```

Bewijs dat programma voldoet aan specificatie

Bewijs slaagt => programma is correct
(voor alle executies)

Bewijs lukt niet => fout in het programma?

Alleen de programmatekst is nodig

Verificatie vereist werk voor ieder programma!

Fouten Voorkomen met Programmeertaal

```
input distance : Float;
input duration : Float;
output speed : Float := duration / distance;
```

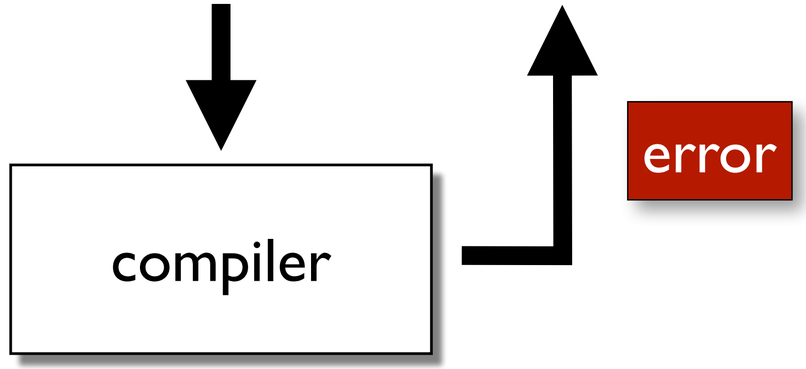
```
input distance : Meter;
input duration : Second;
output speed : Meter/Second := duration / distance;
```

Formaliseer semantische eigenschap in type systeem

Fysieke eenheden als types

Verificatie door Compilatie

```
input distance : Meter;
input duration : Second;
output speed : Meter/Second := duration / distance;
```



Formaliseer semantische eigenschap in type systeem

Fysieke eenheden als types

Verificatie wordt gedaan door compiler; automatisch correctheidsbewijs!

‘Ownership’ in Rust Programmeertaal

```rust
fn as_str(data: &u32) -> &str {
    // compute the string
    let s = format!("{}", data);

    // OH NO! We returned a reference to something that
    // exists only in this function!
    // Dangling pointer! Use after free! Alas!
    // (this does not compile in Rust)
    &s
}
```

Type systeem garandeert:
geen ‘dangling pointers’

Type systeem garandeert:
geen ‘data races’

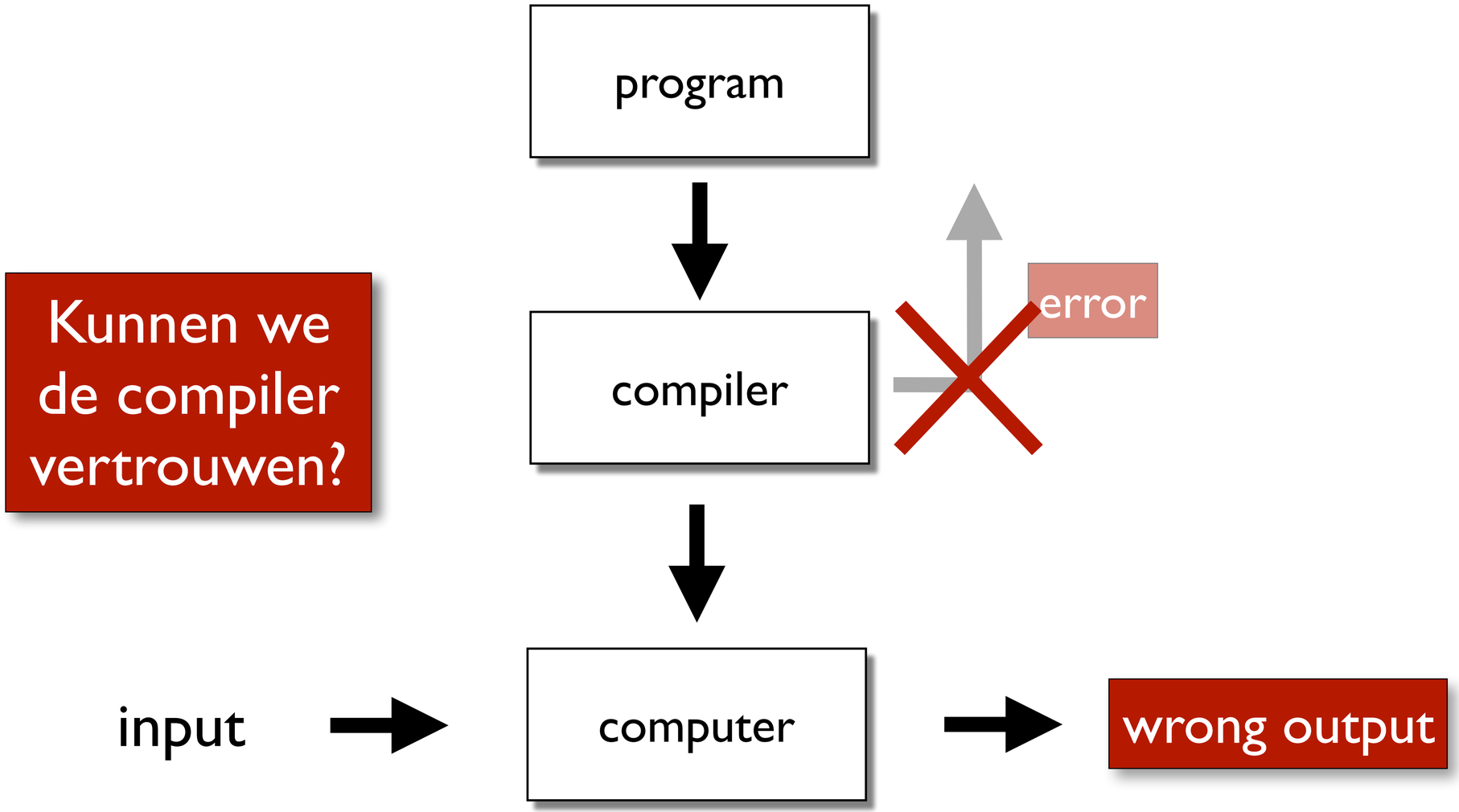
```rust
let mut data = vec![1, 2, 3];
// get an internal reference
let x = &data[0];

// OH NO! `push` causes the backing storage of `data`
to be reallocated.
// Dangling pointer! Use after free! Alas!
// (this does not compile in Rust)
data.push(4);

println!("{}", x);
```

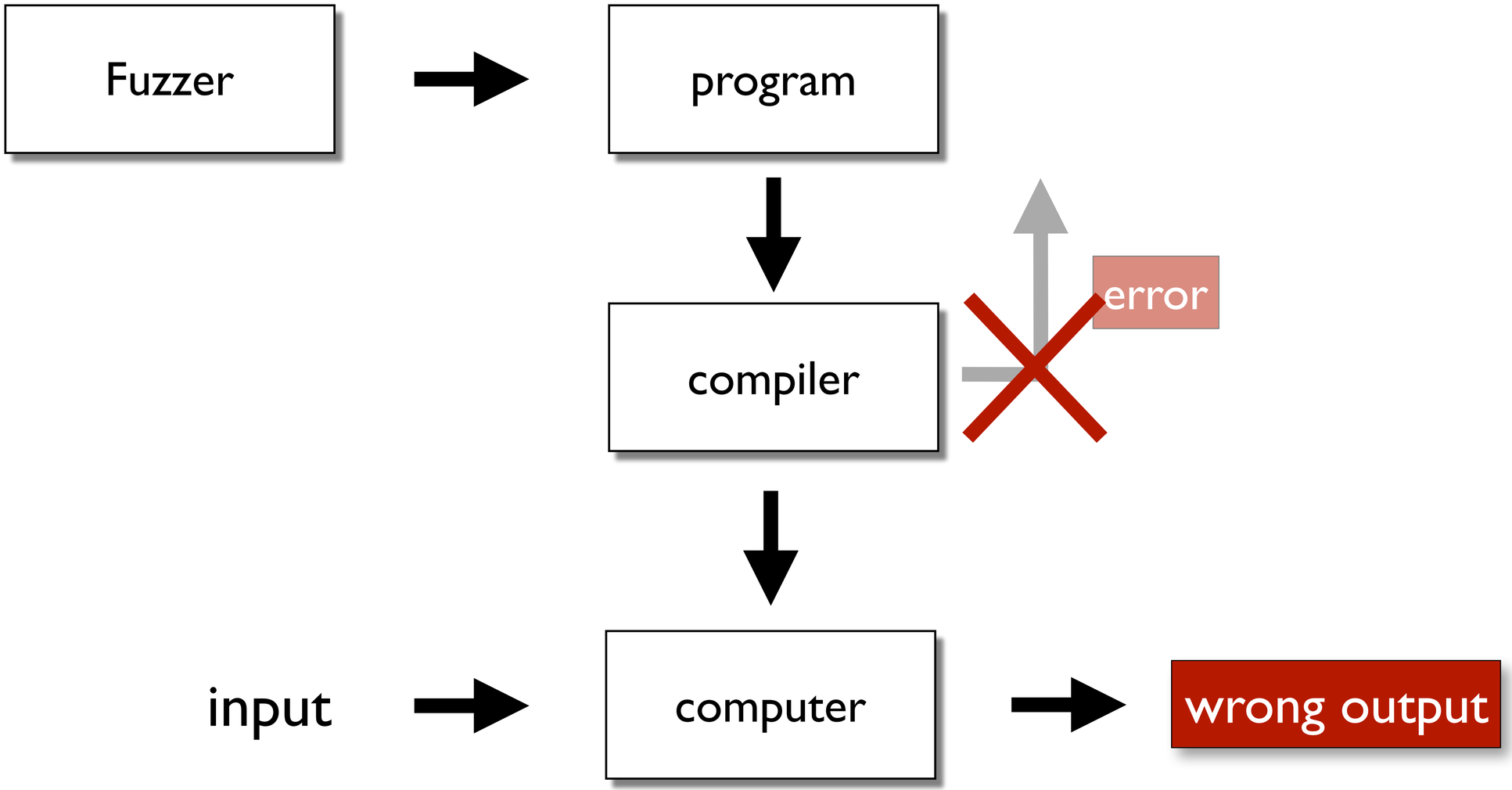
Grote klassen van programmeerfouten kunnen worden voorkomen door talen met sterkere type systemen

Probleem: Correctheid van Compiler



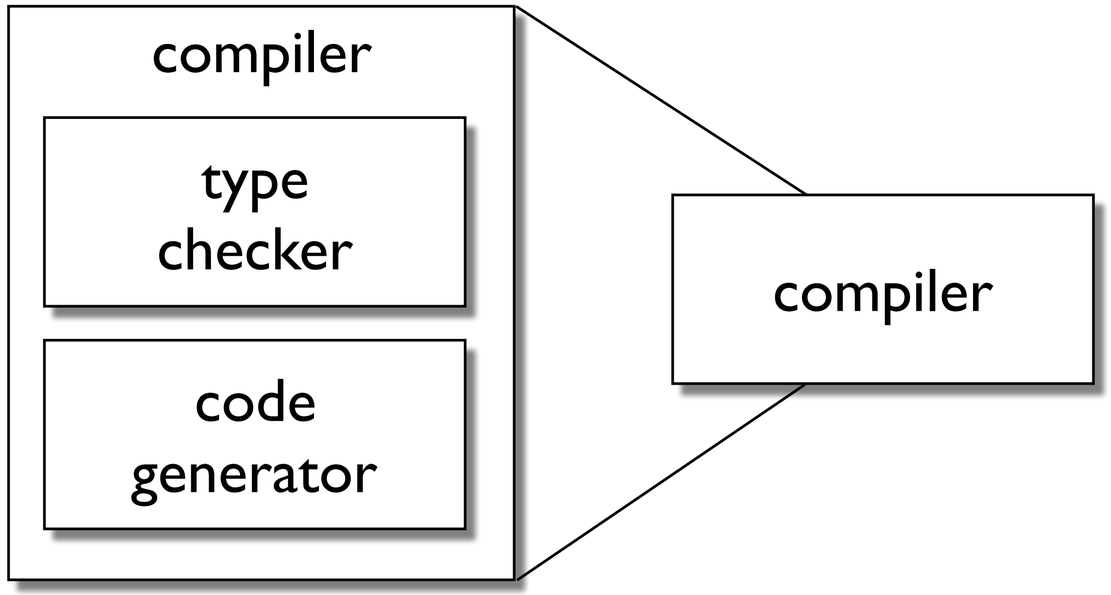
type safety: well-typed programs don't go wrong

Fuzzing: Compiler Fouten vinden door Testen



Niet vinden van fouten garandeert niet dat compiler correct is!

Compiler Fouten Voorkomen met Programma Verificatie



Bewijs dat compiler type safe is

Bewijs slaagt => compiler is type safe

Bewijs lukt niet => fout in de compiler?

Alleen de programmatekst van de compiler is nodig

Eigenschap geldt voor alle programma's!

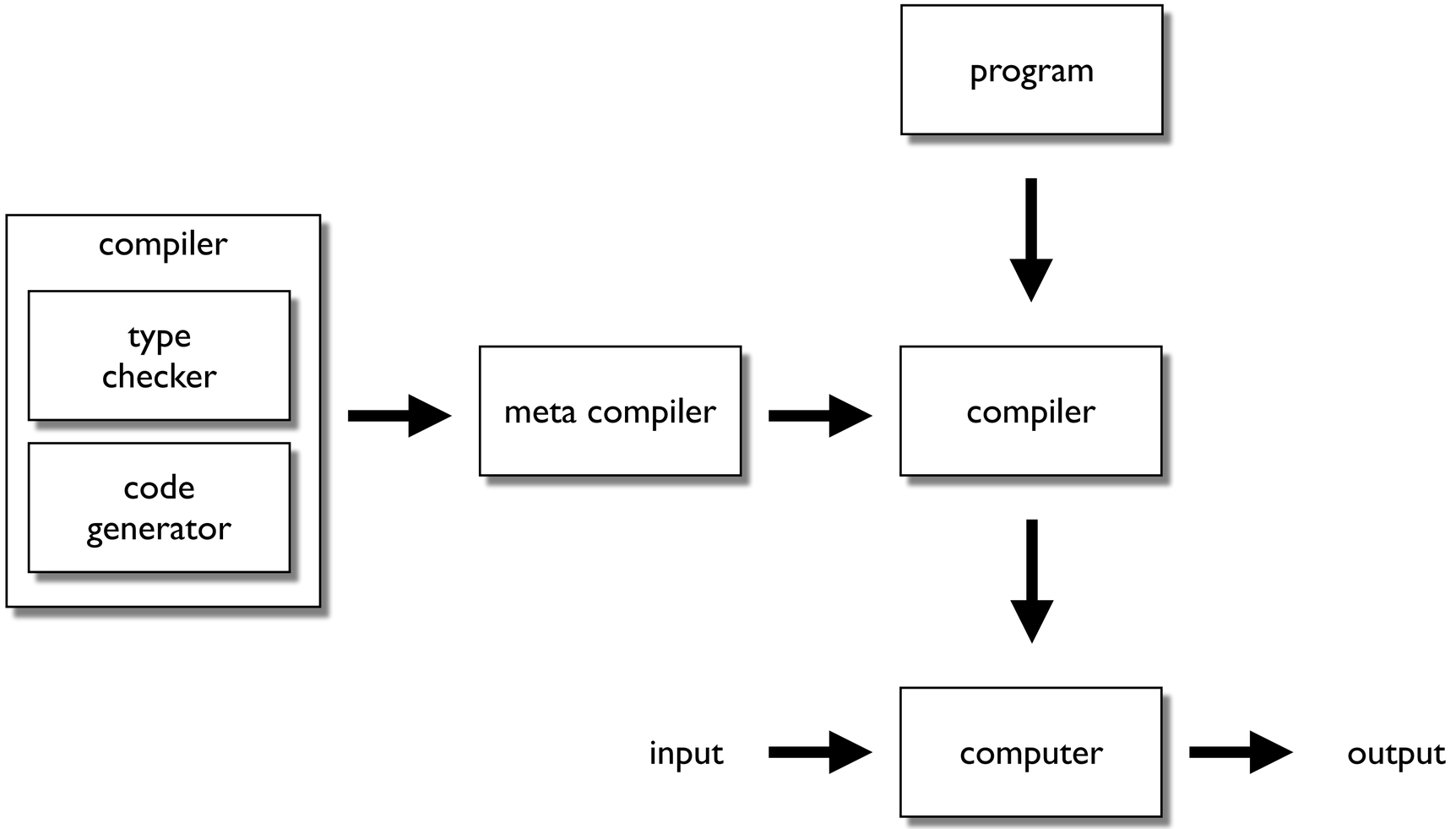
Verificatie vereist werk voor iedere compiler!

Een compiler is ook maar een programma!

Grote klassen van programmeerfouten kunnen worden voorkomen door talen met sterkere type systemen

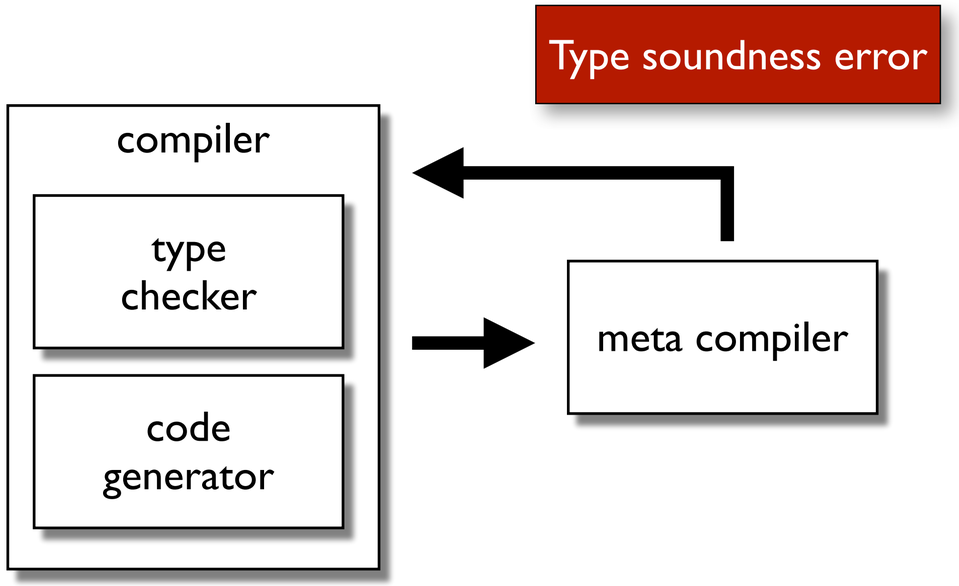
Is dat ook van toepassing op compilerfouten?!

Automatische Verificatie?



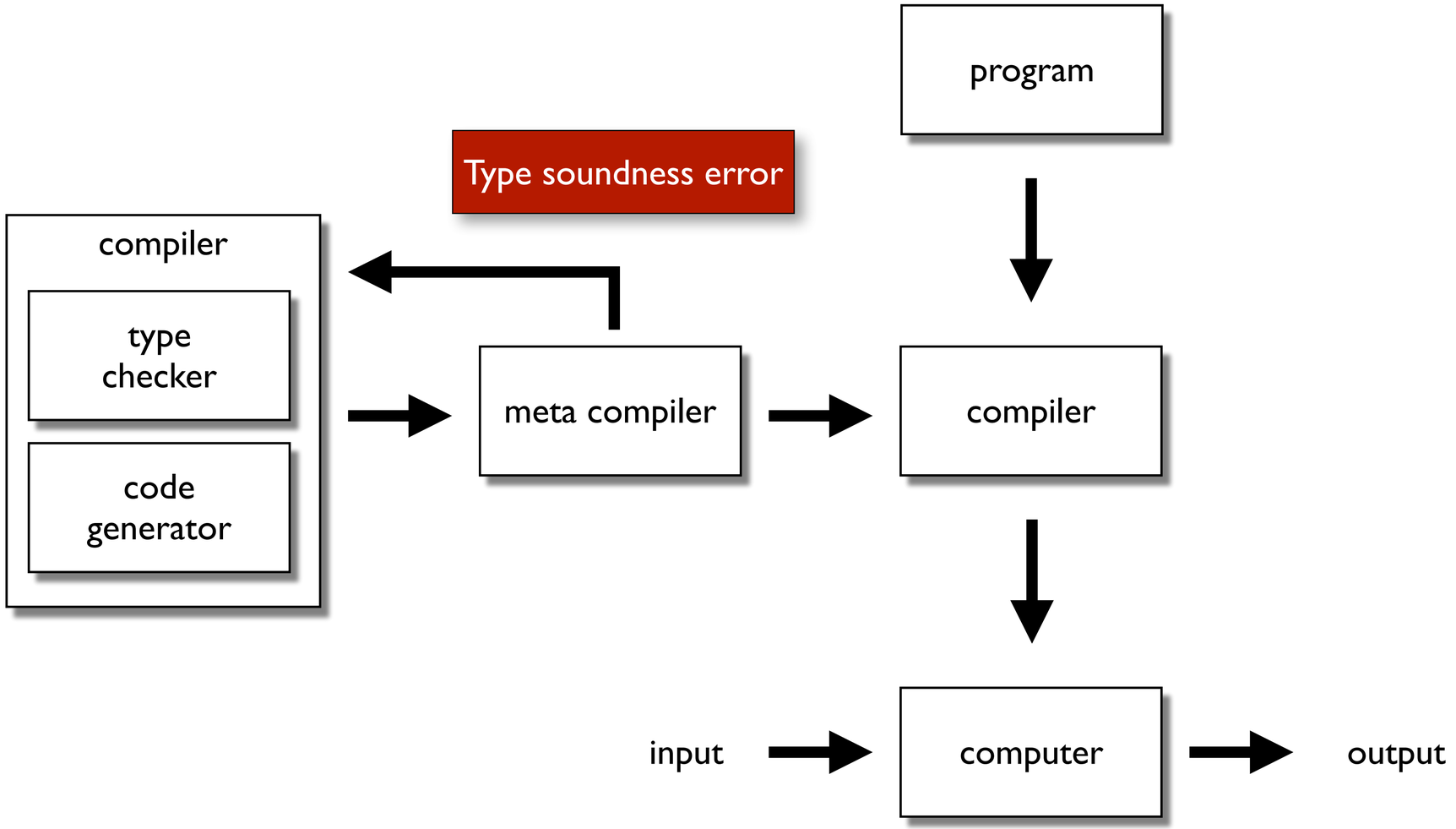
Formaliseer type soundness in type systeem van meta-taal

Automatische Verificatie?



Formaliseer type soundness in type systeem van meta-taal

Automatische Verificatie?



Formaliseer type soundness in type systeem van meta-taal

Type Soundness in het Meta Type Systeem

```
data Expr (Γ : Ctx) : Ty → Set where
  bool   :  Bool → Expr Γ bool
  num    :  ℤ → Expr Γ int
  var    :  ∀ {t} → t ∈ Γ → Expr Γ t
  if     :  ∀ {t} → Expr Γ bool →
            Expr Γ t → Expr Γ t →
            Expr Γ t
  plus   :  Expr Γ int → Expr Γ int →
            Expr Γ int

data Ty : Set where
  bool : Ty
  int  : Ty

Ctx = List Ty
```

```
data Val : Ty → Set where
  bool   :  Bool → Val bool
  num    :  ℤ → Val int

Env : Ctx → Set
Env Γ = All Val Γ

eval : ∀ {Γ t} → Expr Γ t → Env Γ → Val t
eval (bool b)    E    =  bool b
eval (num x)     E    =  num x
eval (var x)     E    =  lookup E x
eval (if c t e)  E    =  case (eval c E) of λ{ (bool b) →
                         if b then (eval t E) else (eval e E)}
eval (plus e₁ e₂) E   =  case (eval e₁ E) of λ{ (num z₁) →
                         case (eval e₂ E) of λ{ (num z₂) →
                         num (z₁ + z₂) }}
```

Voor Complexe Talen

$$\mathsf{eval} \;:\; \mathbb{N} \to \forall\,\{s\ t\ \Sigma\} \to \mathsf{Expr}\ s\ t \to \mathsf{M}\ s\ (\mathsf{Val}\ t)\ \Sigma$$

```
data Expr (s : Scope) : Ty → Set where
  var     : ∀ {t} → (s ↦ v^t t) → Expr s t
  new     : ∀ {s^r s'} → s ↦ c^t s^r s' → Expr s (ref s')
  get     : ∀ {s' t} → Expr s (ref s') → (s' ↦ v^t t) → Expr s t
  call    : ∀ {s' ts t} → Expr s (ref s') → (s' ↦ (m^t ts t)) →
            All (Expr s) ts → Expr s t
  upcast  : ∀ {s' s''} → s' <: s'' → Expr s (ref s') → Expr s (ref s'')
```

Intrinsically-Typed Definitional Interpreters for Imperative Languages. POPL 2018. Bach Poulsen, Rouvoet, Tolmach, Krebbers, Visser

```
eval (suc k) (new x)          = getv x »= λ{ (c^t class f) →
                                usingFrame f (init-obj k class) »= λ{ f' →
                                return (ref [] f') }}
eval (suc k) (get e p)        = eval k e »= λ{ null → raise ; (ref p' f) →
                                usingFrame f (getv (prepend p' p)) »= λ{ (v^t v) →
                                return v }}
eval (suc k) (call e p args)  = eval k e »= λ { null → raise ; (ref p' f) →
                                usingFrame f (getv (prepend p' p)) »= λ{ (m^t f' (meth s b)) →
                                (eval-args k args ^ f') »= λ{ (slots , f') →
                                init s slots (f' :: []) »= λ f'' →
                                usingFrame f'' (eval-body k b) }}}
eval (suc k) (upcast p e)     = eval k e »= λ v → return (upcastRef p v)
```

Intrinsically-Typed, Linear, and Monadic: Safe Interpreters for Session-typed Languages. Under submission 2019. Bach Poulsen, Rouvoet, Krebbers, Visser

Conclusie

Kunnen we programmeerfouten voorkomen door middel van programmeertalen?

Grote klassen van programmeerfouten kunnen worden voorkomen door talen met sterkere type systemen

Kunnen we veiligheid van programmeertalen garanderen bij het ontwerp?

We maken voortgang met
‘type veiligheid als een type eigenschap’

Toekomst: makkelijker om nieuwe programmeertalen te maken die gegarandeerd programmeerfouten voorkomen