The Language Designer's Workbench

automating the verification of language definitions

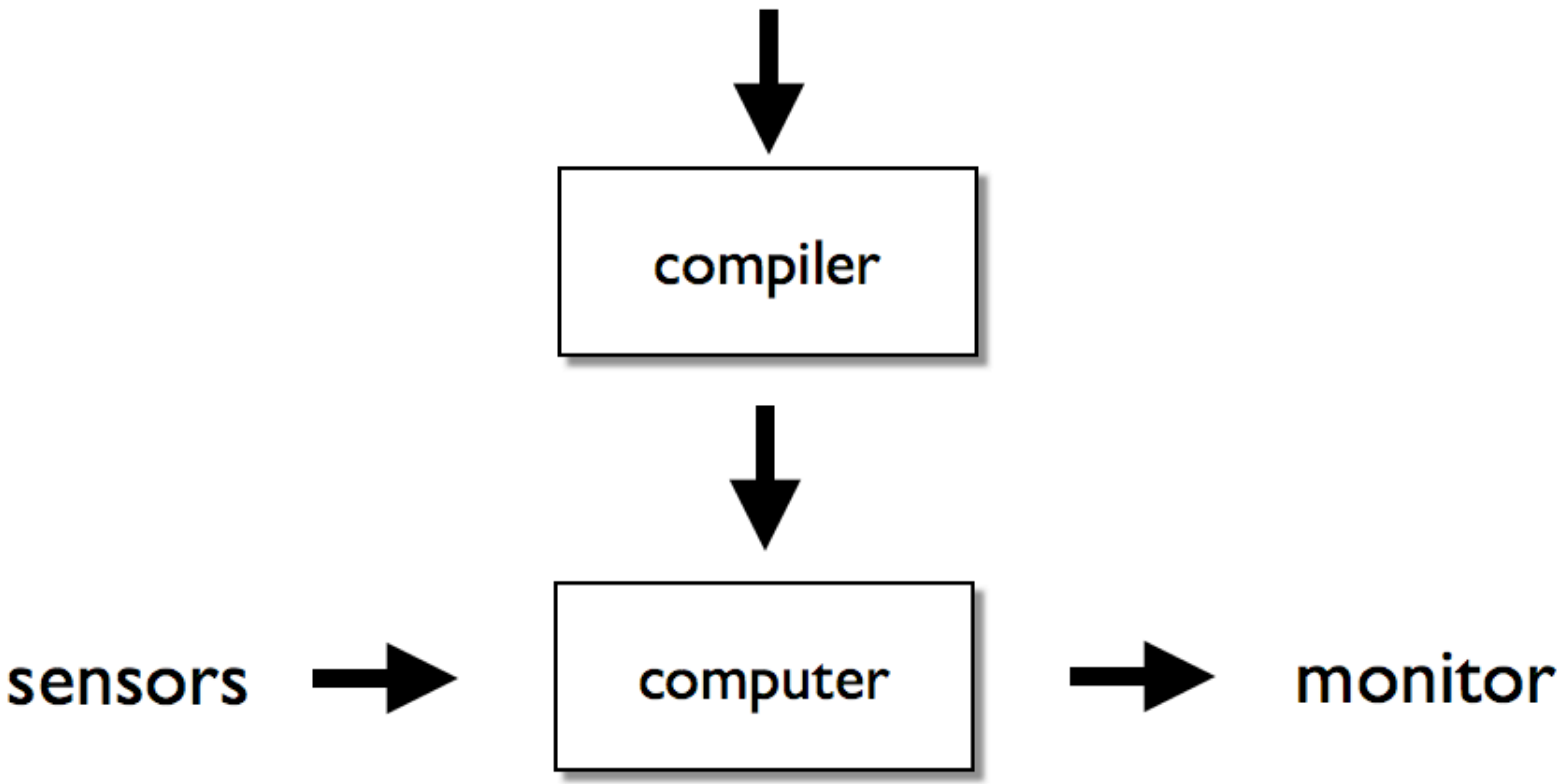
Eelco Visser

VICI 2012

TU Delft
Delft University of Technology

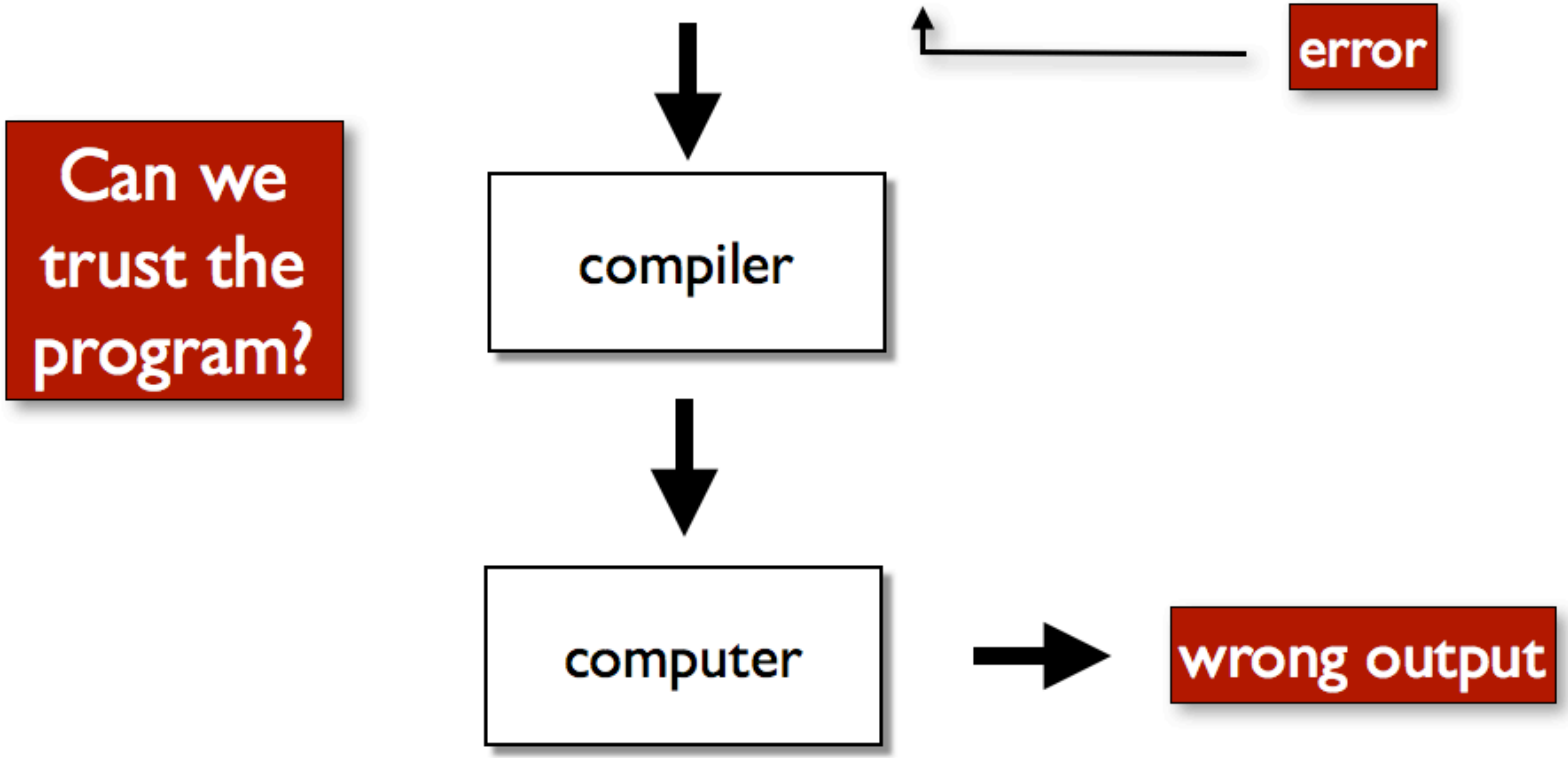
Encoding Application Knowledge in Programs

```
var distance : Float;
var duration : Float;
var speed : Float := duration / distance;
```



Application Assumptions not Checked

```
var distance : Float;
var duration : Float;
var speed : Float := duration / distance;
```



Impact of Software Errors

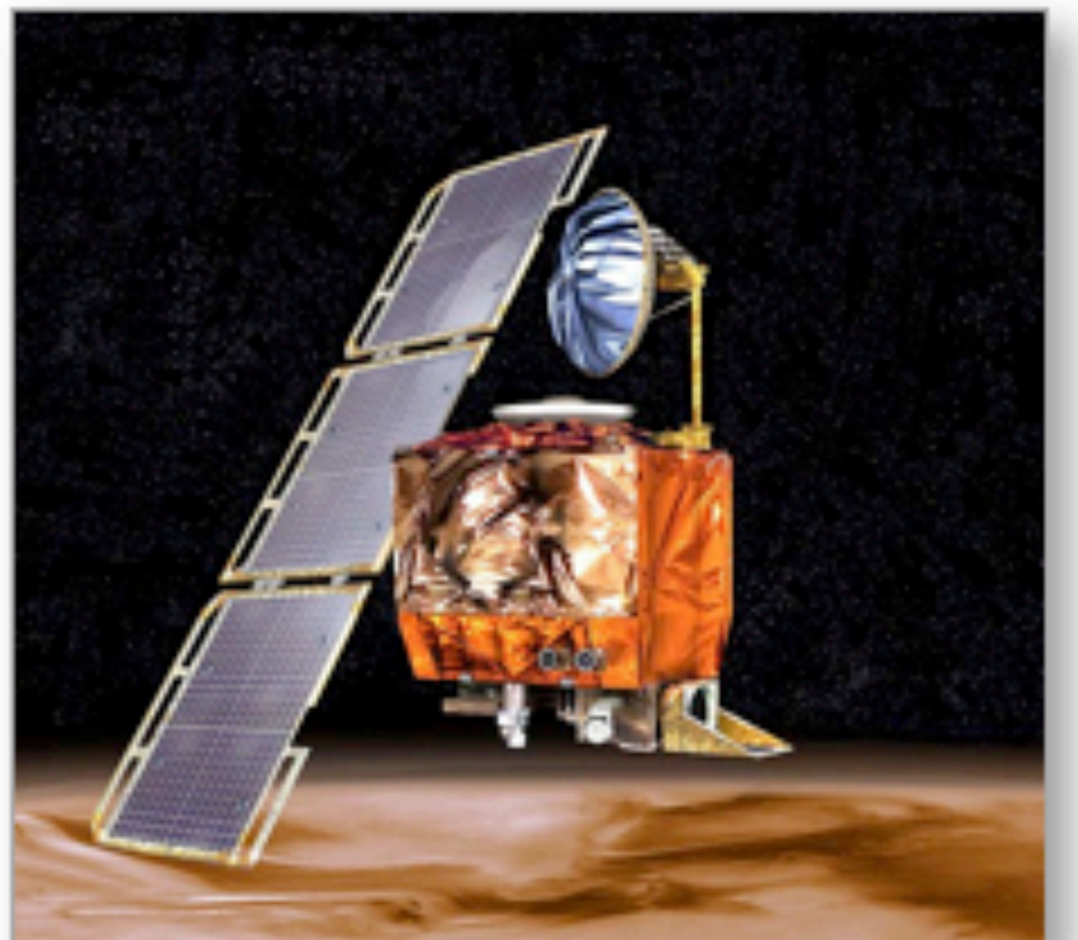
```
var distance : Float;
var duration : Float;
var speed : Float := duration / distance;
```

error

compiler

computer

wrong output



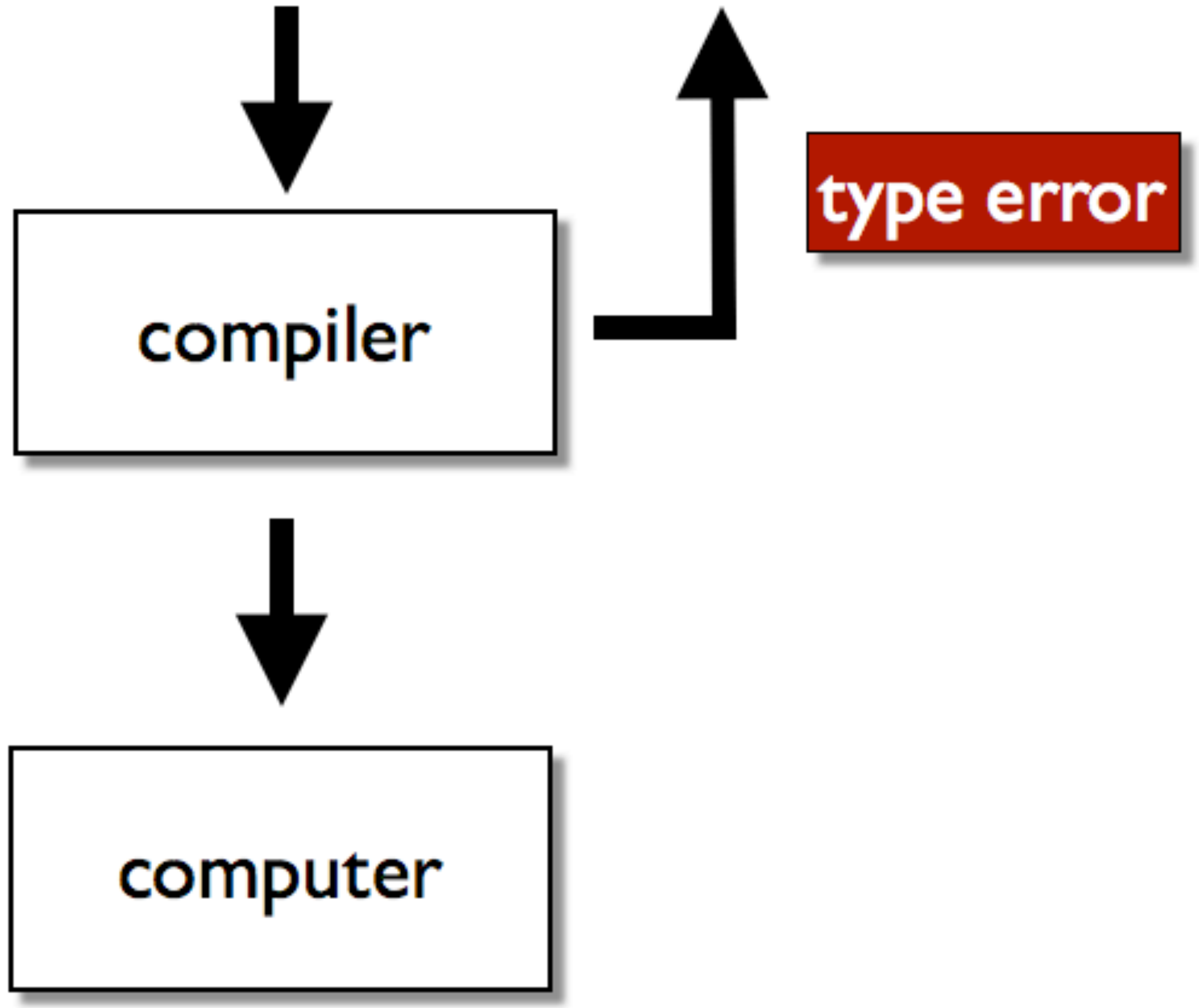
Mars Climate Orbiter

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

Damage: ~350 M$

Domain-Specific Languages

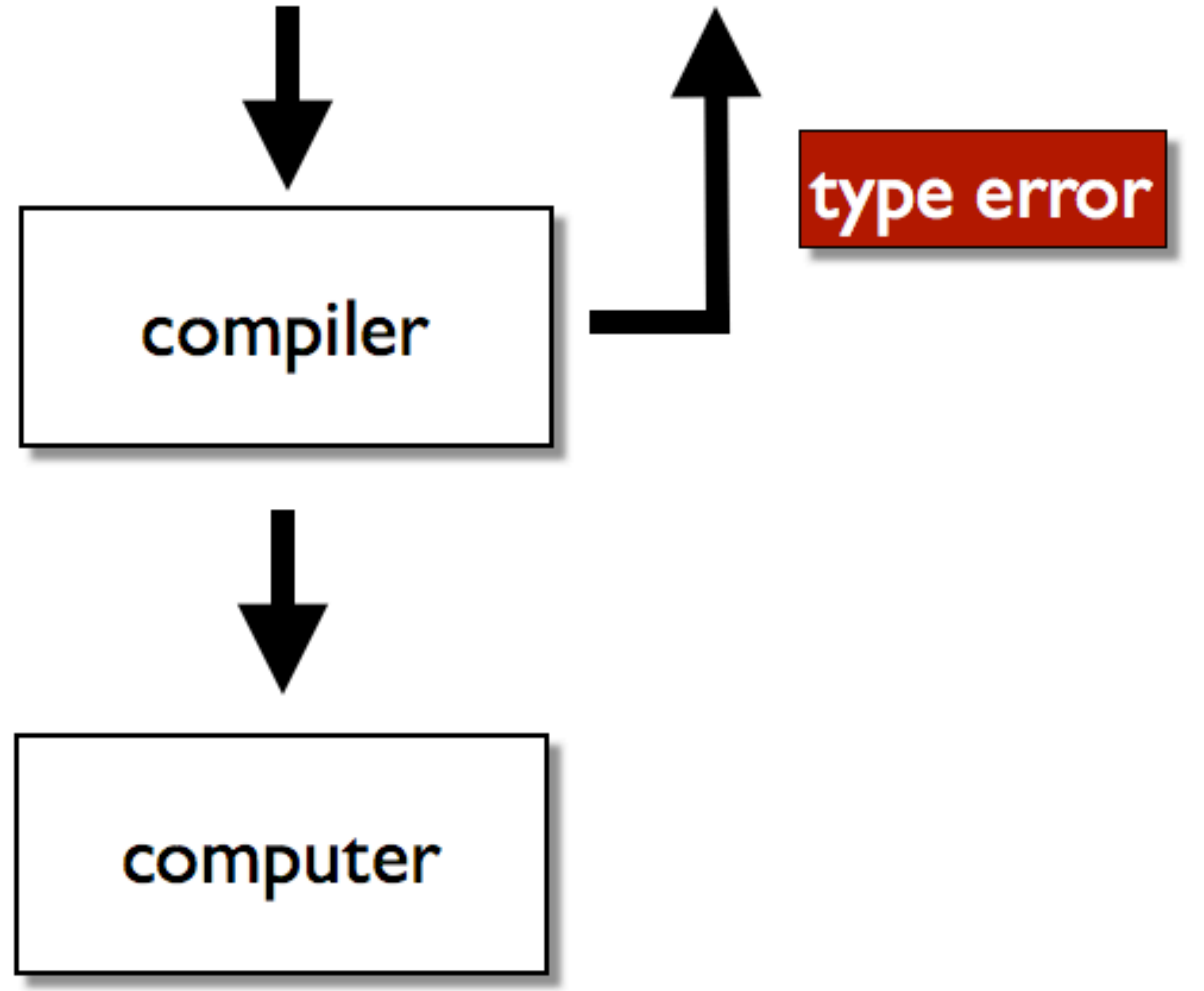
```
var distance : Meter;
var duration : Second;
var speed : Meter/Second := duration / distance;
```



formalize knowledge of application area in language

Domain-Specific Languages

```
var distance : Meter;
var duration : Second;
var speed : Meter/Second := duration / distance;
```



Other Domains

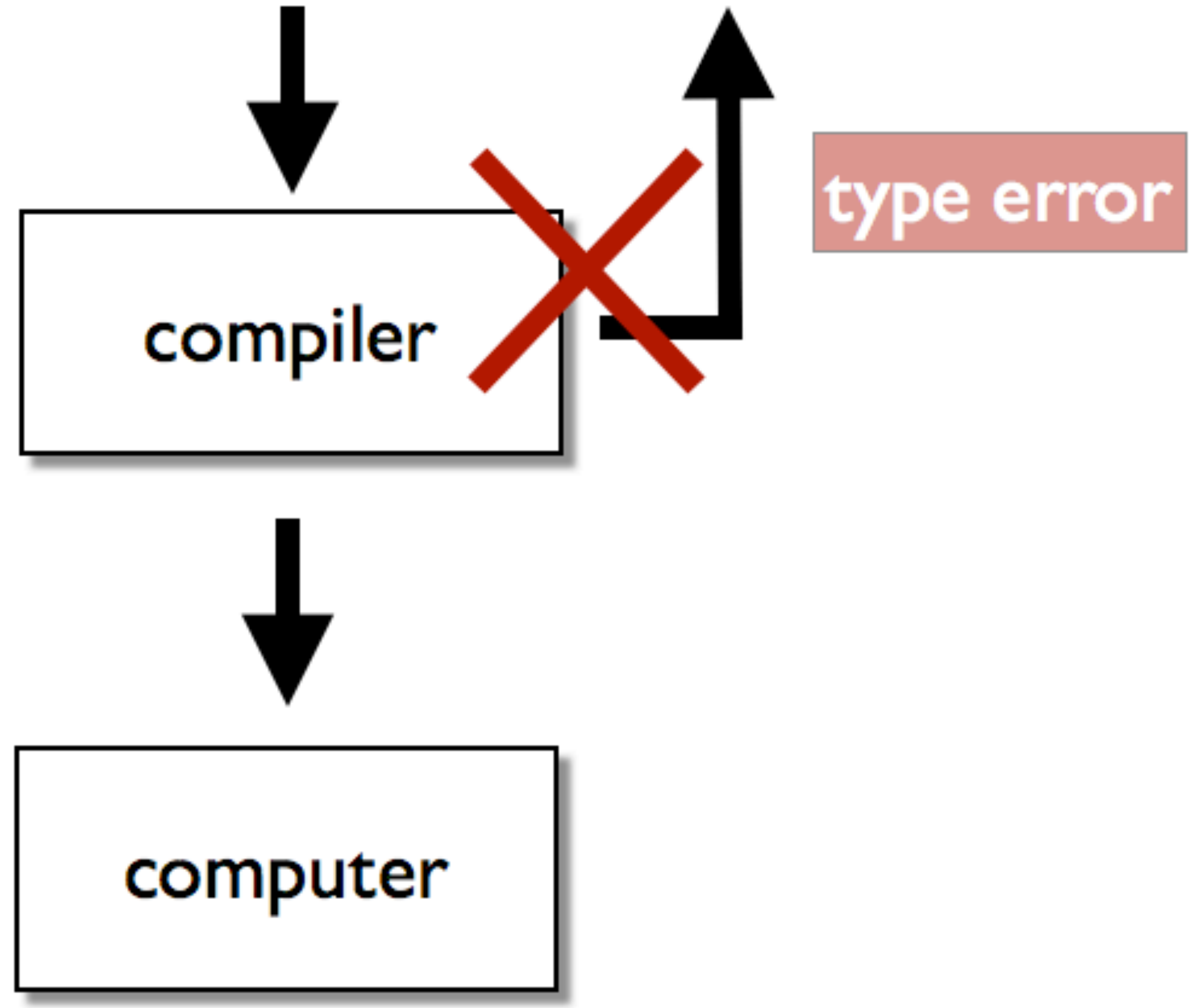
database query
web programming
finance (interest)
...

formalize knowledge of application area in language

Problem: Correctness of Language Definitions

```
var distance : Meter;
var duration : Second;
var speed : Meter/Second := duration / distance;
```

Can we trust the compiler?



Does the compiler report all errors?

Problem: Correctness of Language Definitions

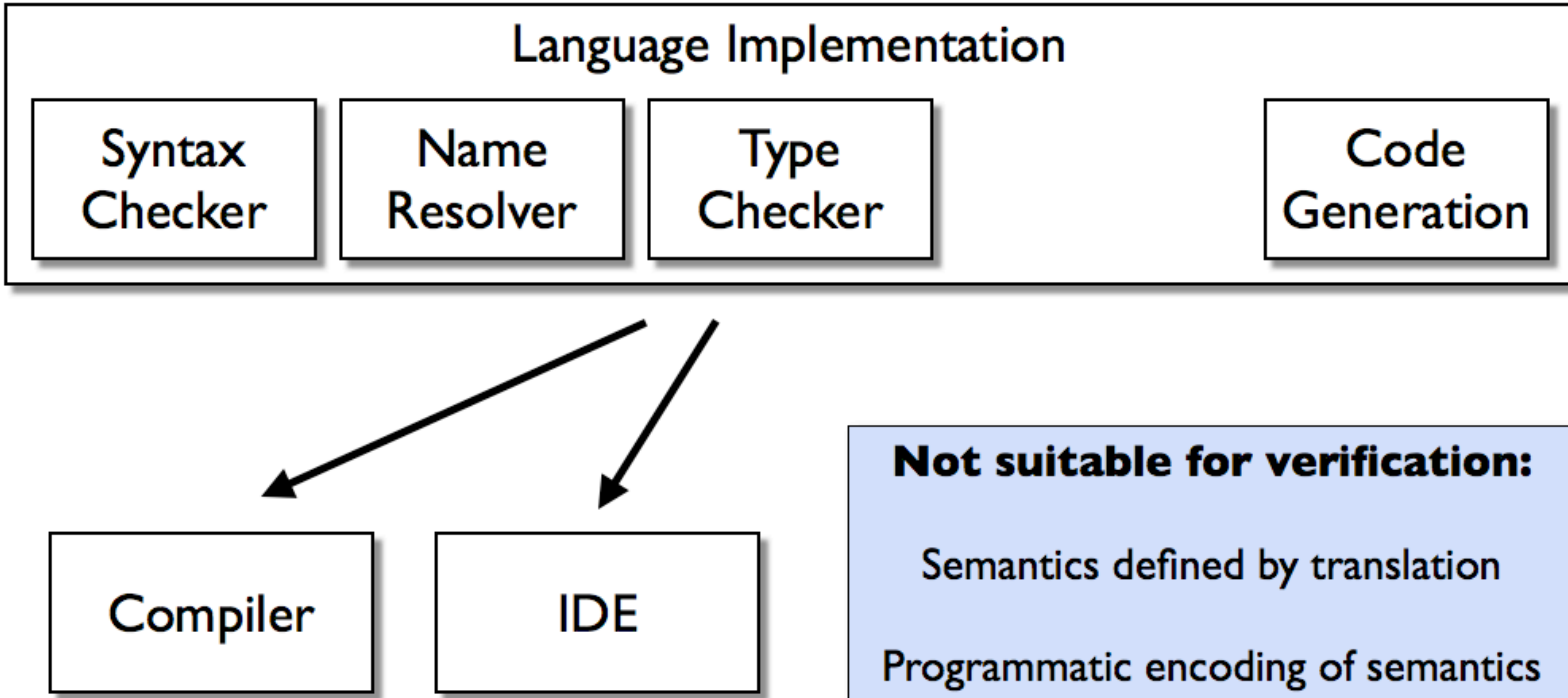
```
type system
  false : bool
  true : bool

  t1: bool, t2: ty, t3: ty
  => if(t1,t2,t3): ty

  zero : nat
  t: nat => succ(t): nat
  t: nat => pred(t): nat
  t: nat => iszero(t): bool
```

type preservation conflict!

```
dynamic semantics
  if(true,t2,t3) -> t2
  if(false,t2,t3) -> t3
  t1 -> t1' => if(t1,t2,t3) -> if(t1',t2,t3)
  t1 -> t1' => succ(t1) -> succ(t1')
  pred(zero) -> false
  pred(succ(nv)) -> nv
  t1 -> t1' => pred(t1) -> pred(t1')
  iszero(zero) -> true
  iszero(succ(nv)) -> false
  t1 -> t1' => iszero(t1) -> iszero(t1')
```

type soundness: consistency of type system and dynamic semantics

Language Engineering



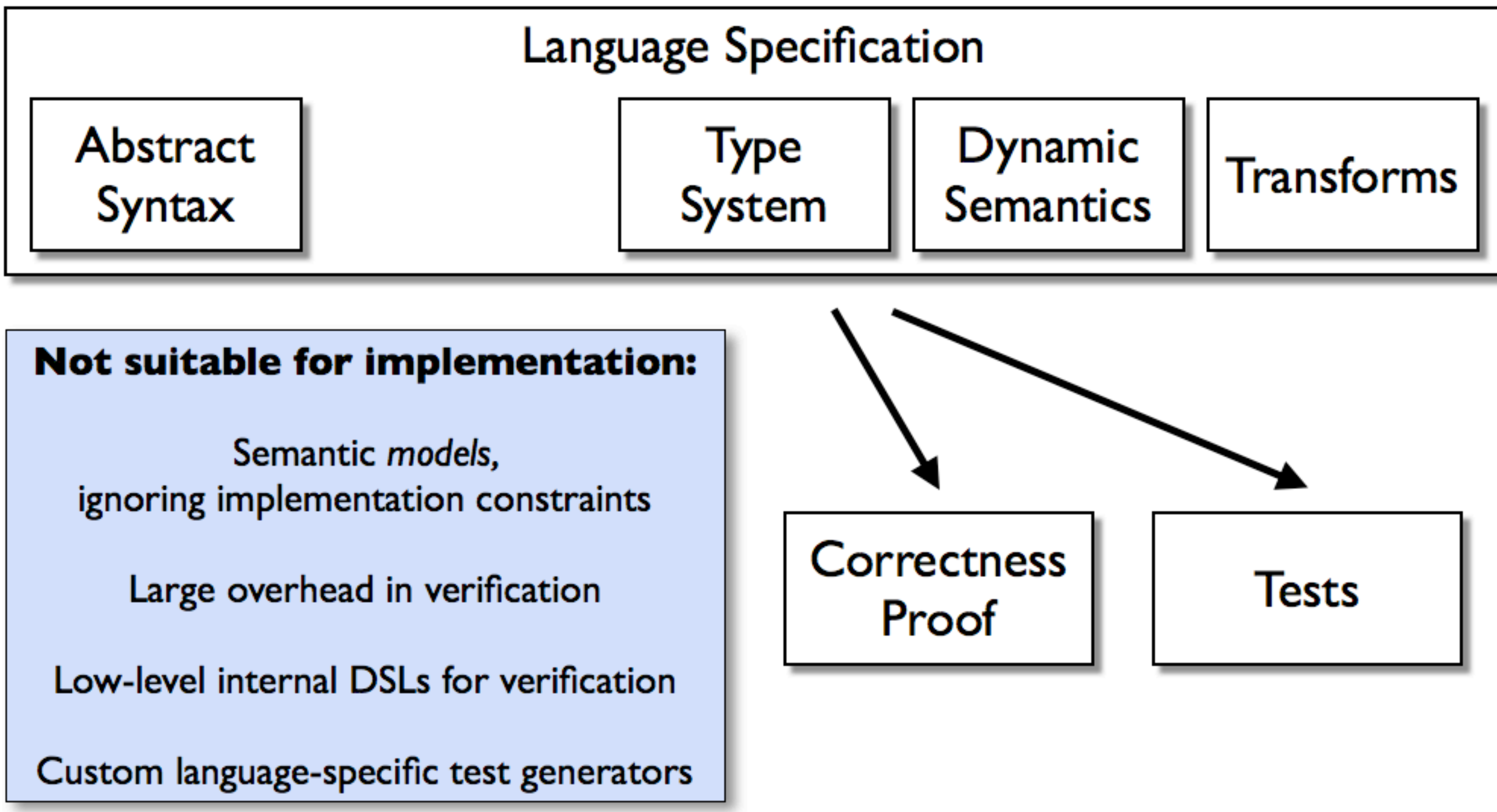
Not suitable for verification:

Semantics defined by translation

Programmatic encoding of semantics

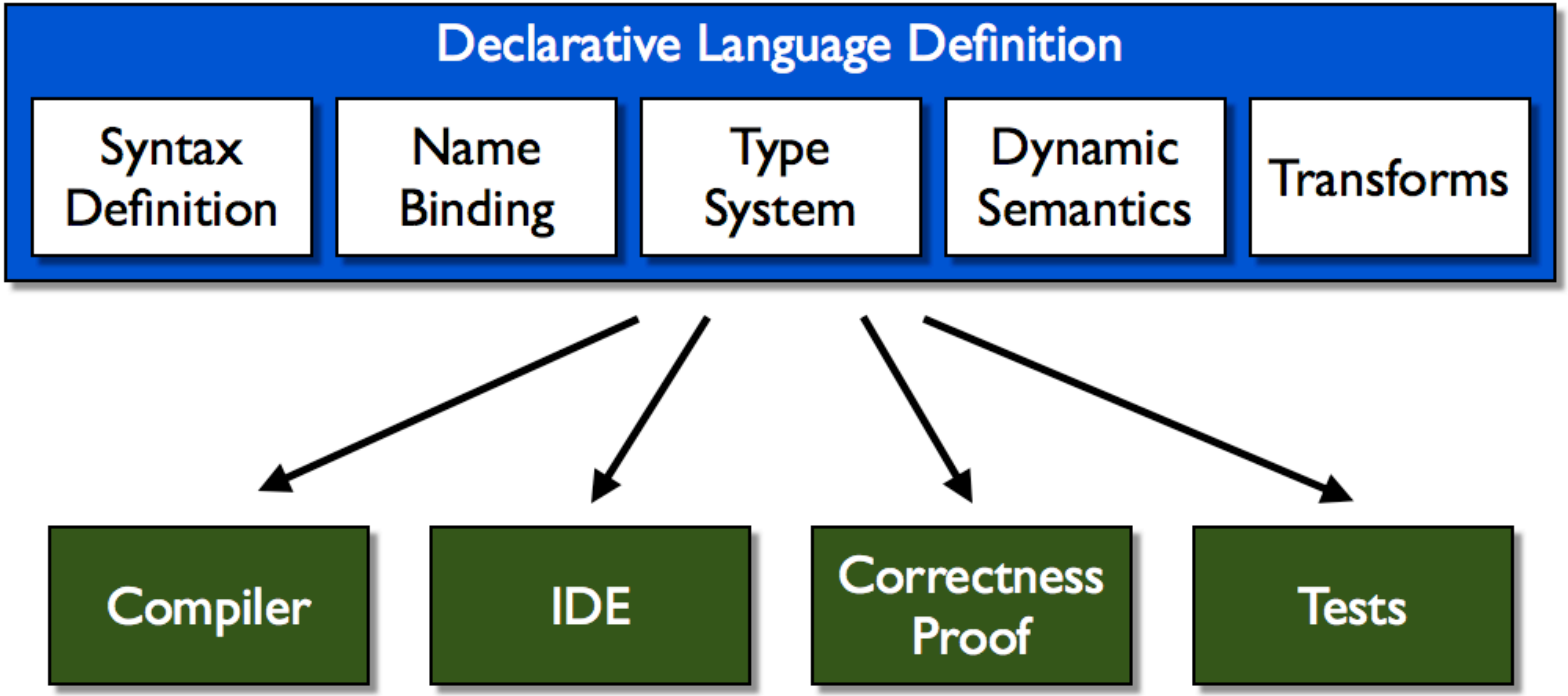
Complicated by incremental algorithms

Semantics Engineering



Challenge: Multi-Purpose Language Definitions

automate verification of language definitions



derive implementation and verification from single source

Intrinsically-Typed Definitional Interpreters: A Tutorial

```
Definitional interpreters in `DynSemSound’
that are type sound by construction

- Light weight dependent types
- Binding using scopes & frames
```

TU Delft

Syntactically Typed Interpreter

```
constructors // expressions
  Exp    : Sort
  IntC   : Int -> Exp
  Add    : Exp * Exp -> Exp
  True   : Exp
  False  : Exp
  If     : Exp * Exp * Exp -> Exp

constructors // values
  Val    : Sort
  IntV   : Int -> Val
  TrueV  : Val
  FalseV : Val

arrows
  Exp --> Val
  if(Val, Exp, Exp) --> Val
```

```
rules

  IntC(i) --> IntV(i).

  Add(e1, e2) --> IntV(k)
  where
    e1 --> IntV(i);
    e2 --> IntV(j);
    addI(i,j) --> k.

  True --> TrueV.

  False --> FalseV.

  If(e1, e2, e3) --> v
  where
    e1 --> v;
    if(v, e2, e3) --> v.

  if(TrueV, e2, e3) --> v
  where e2 --> v.

  if(FalseV, e2, e3) --> v
  where e3 --> v.
```

Syntactically Typed Interpreter

```
constructors // expressions
  Exp    : Sort
  IntC   : Int -> Exp
  Add    : Exp * Exp -> Exp
  True   : Exp
  False  : Exp
  If     : Exp * Exp * Exp -> Exp

constructors // values
  Val    : Sort
  IntV   : Int -> Val
  TrueV  : Val
  FalseV : Val

arrows
  Exp --> Val
  if(Val, Exp, Exp) --> Val
```

```
rules

  IntC(i) --> IntV(i).

  Add(e1, e2) --> IntV(k)
  where
    e1 --> IntV(i);
    e2 --> IntV(j);
    addI(i,j) --> k.
```

```
  Add(e1, e2) --> IntV(k)
  where
    e1 --> FalseV; // not an error
    e2 --> IntV(j);
    addI(i,j) --> k.
```

Intrinsically-Typed Interpreter

```
constructors // types
  Type   : Sort
  INT    : Type
  BOOL   : Type
constructors // expressions
  Exp    : Type -> Sort
  IntC   : Int -> Exp(INT)
  Add    : Exp(INT) * Exp(INT) -> Exp(INT)
  True   : Exp(BOOL)
  False  : Exp(BOOL)
  If     : Exp(BOOL) * Exp(t) * Exp(t) -> Exp(t)
constructors // values
  Val    : Type -> Sort
  IntV   : Int -> Val(INT)
  TrueV  : Val(BOOL)
  FalseV : Val(BOOL)
arrows
  Exp(t) --> Val(t)
```

```
rules

  IntC(i) --> IntV(i).

  Add(e1, e2) --> IntV(k)
  where
    e1 --> IntV(i);
    e2 --> IntV(j);
    addI(i,j) --> k.
```

```
  Add(e1, e2) --> IntV(k)
  where
    e1 --> FalseV; // error!
    e2 --> IntV(j);
    addI(i,j) --> k.
```

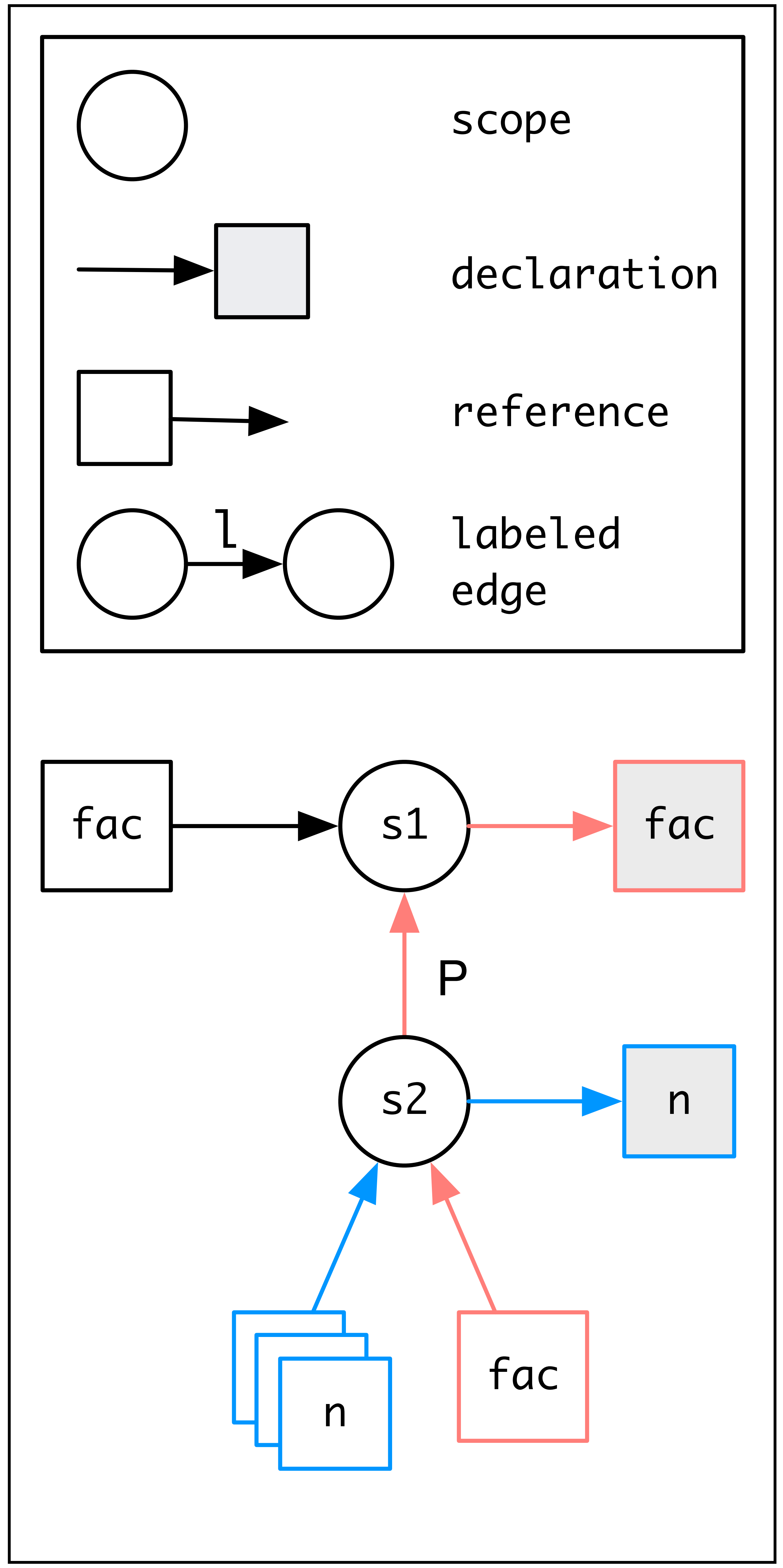
dependent types folklore

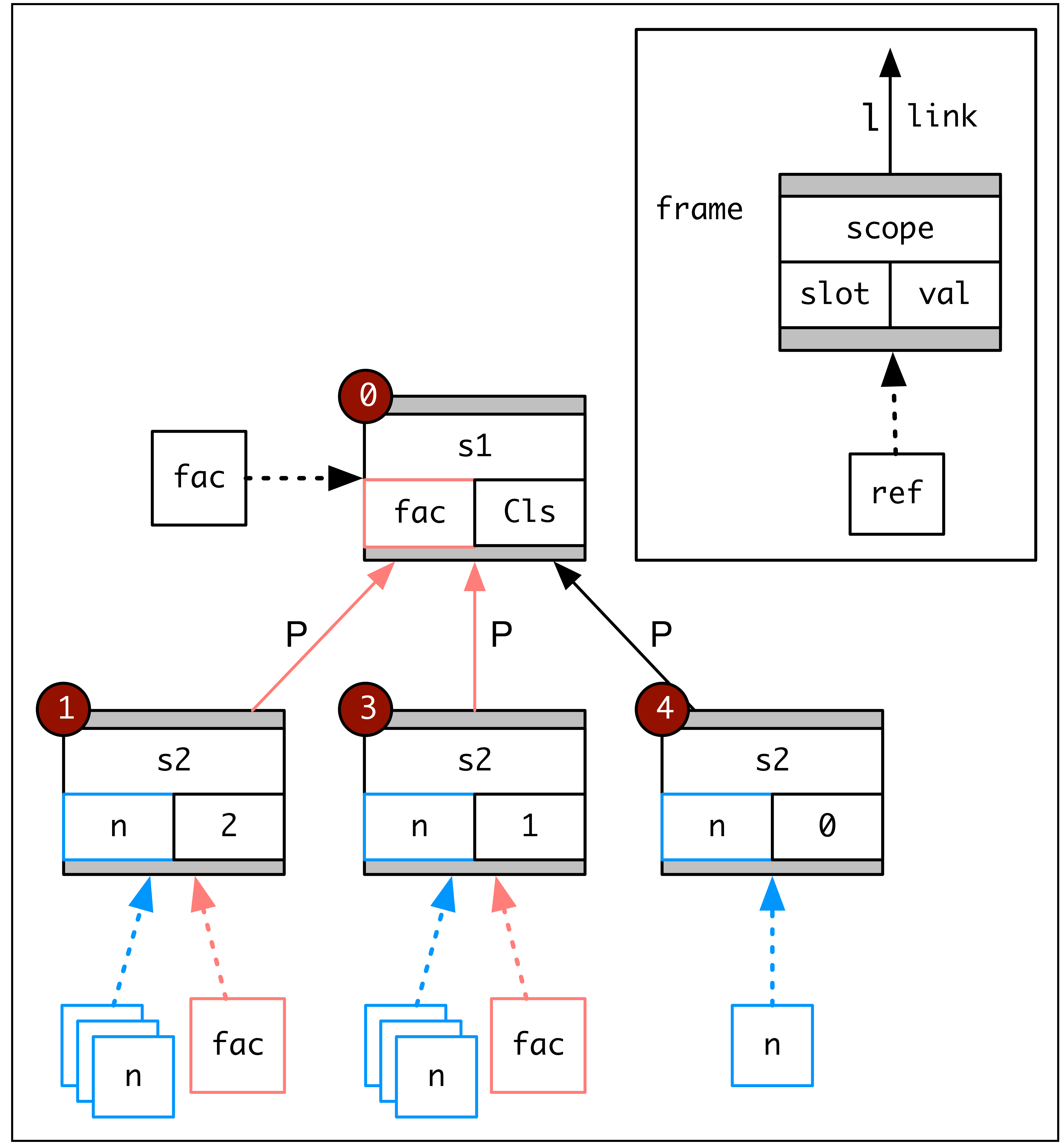
Name Binding

TU Delft

Scopes Describe Frames

```
letrec fac = 0                              s1
  fun (n : Int) : Int { 2                   s2
    if (n == 0) {
      1 5
    } else {
      n * fac(n - 1) 3 4
    }
  }
in
  fac(2) 1
```





All memory units are typed by scopes

Representing Bindings

```
constructors
  Exp     : Type -> Sort
arrows
  Exp(t) --> Val(t)
```

⇨

```
constructors
  Exp        : Scope * Type -> Sort
arrows
  Frame(s) |- Exp(s, t) --> Val(t)
```

Functions and Variables

```
constructors
  FUN    : Type * Type -> Type

  Var    : Ref(s, t) -> Exp(s, t)

  Fun    : Dec(s1, t1) * Exp(s1, t2) -> Exp(s2, FUN(t1, t2))
           where N(s1), E(s1, P, s2)

  App    : Exp(s, FUN(t1, t2)) * Exp(s, t1) -> Exp(s, t2)

  ClosV  : Dec(s1, t1) * Exp(s1, t2) * Frame(s2) -> Val(FUN(t1, t2))
           where E(s1, P, s2)
```

```
constructors
  Exp        : Scope * Type -> Sort
arrows
  Frame(s) |- Exp(s, t) --> Val(t)
```

Functions and Variables

```
constructors
  FUN   : Type * Type -> Type

  Var   : Ref(s, t) -> Exp(s, t)

  Fun   : Dec(s1, t1) * Exp(s1, t2) -> Exp(s2, FUN(t1, t2))
          where N(s1), E(s1, P, s2)

  App   : Exp(s, FUN(t1, t2)) * Exp(s, t1) -> Exp(s, t2)

  ClosV : Dec(s1, t1) * Exp(s1, t2) * Frame(s2) -> Val(FUN(t1, t2))
          where E(s1, P, s2)

rules
  f |- Var(r) --> v where lookup(f, r) --> v.

  f |- Fun(d, e) --> ClosV(d, e, f).

  f |- App(e1, e2) --> v
  where
    f |- e1 --> ClosV(d, e_clos : Exp(s, t), f_clos);
    f |- e2 --> v2;
    initFrame(s) --> f_app;
    setLink(f_app, P, f_clos) --> U;
    setSlot(f_app, d, v2) --> U;
    f_app |- e_clos --> v.
```

```
constructors
  Exp       : Scope * Type -> Sort
arrows
  Frame(s) |- Exp(s, t) --> Val(t)
```

Intrinsically-Typed Definitional Interpreters for Imperative Languages

Casper Bach Poulsen
Arjen Rouvoet
Andrew Tolmach
Robbert Krebbers
Eelco Visser

POPL 2018

```
Definitional interpreters in Agda
that are type sound by construction

- Dependent types
- Binding using scopes & frames
- Strong monad for monotone state
- Case study: MJ.agda
```

TU Delft

Side Effect: Better LangDev Tools

DynSem

- dynamic semantics specification and interpreter generation based on IMSOS [RTA15]

NaBL

- declarative name binding rules [SLE12]
- incremental evaluation [SLE13]

Scope Graphs

- theory of name resolution [ESOP15]
- constraint language based on scope graphs [PEPM16]

Scopes describe Frames

- uniform model for memory based on scope graphs [ECOOP16]
- systematic soundness proof, but still manual

Intrinsically-Typed Definitional Interpreters

- automatic type soundness checking for evaluation rules [POPL18]

Challenges

More expressive intrinsically-typed interpreters

- more sophisticated type systems (generics)
- more (sophisticated) effects, concurrency, …

Verification of other language properties

- Type preservation of transformations
- Semantics preservation of transformations (compilers)

Integration in language workbench

- Hide boilerplate, efficient interpreters, custom dep-typed meta-language?

Verification of language workbench components

- Correctness of parsing algorithm
- Soundness and completeness of Statix solver
- Correctness of DynSem meta-interpreter and partial evaluator
- etc.