Type Checking and Type Constraints

Eelco Visser

TU Delft

CS4200 | Compiler Construction | September 16, 2021

This Lecture

Types

- kinds of types
- relations between types

Formalizing Type Systems

- judgments and inference rules

Testing Static Analysis

- in SPT

Statix

- Predicates and type constraints

Types

Why types?

Why types?

Why types?

- "guarantee absence of run-time type errors"

Why types?

Why types?

- "guarantee absence of run-time type errors"

What is a type system?

Why types?

Why types?

- "guarantee absence of run-time type errors"

What is a type system?

- A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [Pierce2002]

Why types?

Why types?

- "guarantee absence of run-time type errors"

What is a type system?

- A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [Pierce2002]

Discuss using a series of examples

- Do you consider the example correct or not, and why?
  - That is, do you think it should type-check?
- If incorrect: what types will disallow this program?
- If correct: what types will allow this program?

Preliminaries

```
class A {
    B b;
    int m(int i) {
        return i + b.f;
    }
}

class B {
    int f;
}
```

How do types show up in programs?

Preliminaries

```
class A {
    B b;
    int m(int i) {
        return i + b.f;
    }
}

class B {
    int f;
}
```

How do types show up in programs?

- Type literals describe types

Preliminaries

```
class A {
    B b;
    int m(int i) {
        return i + b.f;
    }
}

class B {
    int f;
}
```

How do types show up in programs?

- Type literals describe types
- Type definitions introduce new (named) types

Preliminaries

```
class A {
    B b;
    int m(int i) {
        return i + b.f;
    }
}

class B {
    int f;
}
```

How do types show up in programs?

- Type literals describe types
- Type definitions introduce new (named) types
- Type references refer to named types

Preliminaries

```
class A {
    B b;
    int m(int i) {
        return i + b.f;
    }
}

class B {
    int f;
}
```

How do types show up in programs?

- Type literals describe types
- Type definitions introduce new (named) types
- Type references refer to named types
- Declared variables have types (x : T)

Preliminaries

```
class A {
    B b;
    int m(int i) {
        return i + b.f;
    }
}

class B {
    int f;
}
```

How do types show up in programs?

- Type literals describe types
- Type definitions introduce new (named) types
- Type references refer to named types
- Declared variables have types (x : T)
- Expressions have types (e : T)

Preliminaries

```
class A {
    B b;
    int m(int i) {
        return i + b.f;
    }
}

class B {
    int f;
}
```

How do types show up in programs?

- Type literals describe types
- Type definitions introduce new (named) types
- Type references refer to named types
- Declared variables have types (x : T)
- Expressions have types (e : T)
  - Including all sub-expressions

Preliminaries

```
class A {
    B b;
    int m(int i) {
        return i + b.f;
    }
}

class B {
    int f;
}
```

How do types show up in programs?

- Type literals describe types
- Type definitions introduce new (named) types
- Type references refer to named types
- Declared variables have types (x : T)
- Expressions have types (e : T)
  - Including all sub-expressions

Types Example

```
4 / "four"
```

Types Example

```
4 / "four"
```

```
4      : number
"four" : string
/      : number * number → number
```

Types Example

```
4 / "four"
```

```
4      : number
"four" : string
/      : number * number → number
```

simple types

Types Example

```
4 / "four"
```

```
4      : number
"four" : string
/      : number * number → number
```

simple types

typing prevents undefined runtime behavior

Types Example

```
7 + (if (true) { 5 } else { "four" })
```

Types Example

```
7 + (if (true) { 5 } else { "four" })
```

```
7 : number          "four" : string
5 : number          if     : ?
```

Types Example

```
7 + (if (true) { 5 } else { "four" })
```

```
7 : number        "four" : string
5 : number        if     : ?
```

no simple type

Types Example

```
7 + (if (true) { 5 } else { "four" })
```

no simple type

```
7 : number          "four" : string
5 : number          if     : ?
```

- typing (over)approximates runtime behavior
- programs without runtime errors can be rejected

Types Example

```
function id(x) { return x; }
id(4); id(true);
```

Types Example

```
function id(x) { return x; }
id(4); id(true);
```

```
4       : number
true    : boolean
id      : ∀T.T→T
```

Types Example

```
function id(x) { return x; }
id(4); id(true);
```

```
4      : number
true   : boolean
id     : ∀T.T→T
```

polymorphic type

Types Example

```
function id(x) { return x; }
id(4); id(true);
```

polymorphic type

```
4     : number
true  : boolean
id    : ∀T.T→T
```

- richer types approximate behavior better
- depends on runtime representation of values

Types Example

```
if (a < 5) { 5 } else { "four" }
```

Types Example

```
if (a < 5) { 5 } else { "four" }
```

```
5      : number
"four" : string
if     : number|string
```

Types Example

```
if (a < 5) { 5 } else { "four" }
```

```
5      : number
"four" : string
if     : number|string
```

union type

Types Example

```
if (a < 5) { 5 } else { "four" }
```

```
5       : number
"four"  : string
if      : number|string
```

union type

```
- richer types approximate behavior better
- depends on runtime representation of values
```

Types Example

```
float distance = 12.0, time = 4.0
float velocity = time / distance
```

Types Example

```
float distance = 12.0, time = 4.0
float velocity = time / distance
```

```
distance : float<m>
time     : float<s>
velocity : float<m/s>
```

Types Example

unit-of-measure type

```
float distance = 12.0, time = 4.0
float velocity = time / distance
```

```
distance : float<m>
time     : float<s>
velocity : float<m/s>
```

Types Example

unit-of-measure type

```
float distance = 12.0, time = 4.0
float velocity = time / distance
```

```
distance : float<m>
time     : float<s>
velocity : float<m/s>
```

- no runtime problems, but not correct (v = d / t)
- types can enforce other correctness properties

What kind of types?

What kind of types?

- Simple: int, float→float, bool
- Named: class A, newtype Id
- Polymorphic: List<X>, ∀a.a→a
- Union/sum (one of): string|string[]
- Unit-of-measure: float<m>, float<m/s>

What kind of types?

- Simple — int, float→float, bool
- Named — class A, newtype Id
- Polymorphic — List<X>, ∀a.a→a
- Union/sum (one of) — string|string[]
- Unit-of-measure — float<m>, float<m/s>
- Structural — { x: number, y: number }

What kind of types?

– Simple	int, float→float, bool
– Named	class A, newtype Id
– Polymorphic	List<X>, ∀a.a→a
– Union/sum (one of)	string|string[]
– Unit-of-measure	float<m>, float<m/s>
– Structural	{ x: number, y: number }
– Intersection (all of)	Comparable&Serializable

What kind of types?

- Simple: int, float→float, bool
- Named: class A, newtype Id
- Polymorphic: List<X>, ∀a.a→a
- Union/sum (one of): string|string[]
- Unit-of-measure: float<m>, float<m/s>
- Structural: { x: number, y: number }
- Intersection (all of): Comparable&Serializable
- Recursive: μT.int|T*T (binary int tree)

What kind of types?

- Simple — int, float→float, bool
- Named — class A, newtype Id
- Polymorphic — List<X>, ∀a.a→a
- Union/sum (one of) — string|string[]
- Unit-of-measure — float<m>, float<m/s>
- Structural — { x: number, y: number }
- Intersection (all of) — Comparable&Serializable
- Recursive — μT.int|T*T (binary int tree)
- Ownership — &mut data

What kind of types?

- Simple `int, float→float, bool`
- Named `class A, newtype Id`
- Polymorphic `List<X>, ∀a.a→a`
- Union/sum (one of) `string|string[]`
- Unit-of-measure `float<m>, float<m/s>`
- Structural `{ x: number, y: number }`
- Intersection (all of) `Comparable&Serializable`
- Recursive `μT.int|T*T` (binary int tree)
- Ownership `&mut data`
- Dependent – values in types `Vector 3`

What kind of types?

- Simple int, float→float, bool
- Named class A, newtype Id
- Polymorphic List<X>, ∀a.a→a
- Union/sum (one of) string|string[]
- Unit-of-measure float<m>, float<m/s>
- Structural { x: number, y: number }
- Intersection (all of) Comparable&Serializable
- Recursive μT.int|T*T (binary int tree)
- Ownership &mut data
- Dependent – values in types Vector 3
- ... many more ...

Why types?

Why types?

Why types?

- Statically prove the absence of certain (wrong) runtime behavior

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - “Well-typed programs cannot go wrong.” [Reynolds1985]

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - “Well-typed programs cannot go wrong.” [Reynolds1985]
  - Also logical properties beyond runtime problems

Why types?

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - “Well-typed programs cannot go wrong.” [Reynolds1985]
  - Also logical properties beyond runtime problems

What are types?

Why types?

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - “Well-typed programs cannot go wrong.” [Reynolds1985]
  - Also logical properties beyond runtime problems

What are types?

- Static classification of expressions by approximating the runtime values they may produce

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - "Well-typed programs cannot go wrong." [Reynolds1985]
  - Also logical properties beyond runtime problems

What are types?

- Static classification of expressions by approximating the runtime values they may produce
- Richer types approximate runtime behavior better

Why types?

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - "Well-typed programs cannot go wrong." [Reynolds1985]
  - Also logical properties beyond runtime problems

What are types?

- Static classification of expressions by approximating the runtime values they may produce
- Richer types approximate runtime behavior better
- Richer types may encode correctness properties beyond runtime crashes

Why types?

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - “Well-typed programs cannot go wrong.” [Reynolds1985]
  - Also logical properties beyond runtime problems

What are types?

- Static classification of expressions by approximating the runtime values they may produce
- Richer types approximate runtime behavior better
- Richer types may encode correctness properties beyond runtime crashes

What is the difference between typing and testing?

Why types?

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - "Well-typed programs cannot go wrong." [Reynolds1985]
  - Also logical properties beyond runtime problems

What are types?

- Static classification of expressions by approximating the runtime values they may produce
- Richer types approximate runtime behavior better
- Richer types may encode correctness properties beyond runtime crashes

What is the difference between typing and testing?

- Typing is an over-approximation of runtime behavior (proof of absence)

Why types?

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
  - “Well-typed programs cannot go wrong.” [Reynolds1985]
  - Also logical properties beyond runtime problems

What are types?

- Static classification of expressions by approximating the runtime values they may produce
- Richer types approximate runtime behavior better
- Richer types may encode correctness properties beyond runtime crashes

What is the difference between typing and testing?

- Typing is an over-approximation of runtime behavior (proof of absence)
- Testing is an under-approximation of runtime behavior (proof of presence)

Types influence language design

Types and language design

Types influence language design

- Types abstract over implementation

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - As long as the public interface is implemented, dependent modules do not change

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - As long as the public interface is implemented, dependent modules do not change

Can we have our cake and eat it too?

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - As long as the public interface is implemented, dependent modules do not change

Can we have our cake and eat it too?

- Ever more precise types lead to ever more correct programs

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - As long as the public interface is implemented, dependent modules do not change

Can we have our cake and eat it too?

- Ever more precise types lead to ever more correct programs
- What would be the most precise type you can give?

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - As long as the public interface is implemented, dependent modules do not change

Can we have our cake and eat it too?

- Ever more precise types lead to ever more correct programs
- What would be the most precise type you can give?
  - The exact set of values computed for a given input?

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - As long as the public interface is implemented, dependent modules do not change

Can we have our cake and eat it too?

- Ever more precise types lead to ever more correct programs
- What would be the most precise type you can give?
  - The exact set of values computed for a given input?
- Expressive typing problems become hard to compute

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - As long as the public interface is implemented, dependent modules do not change

Can we have our cake and eat it too?

- Ever more precise types lead to ever more correct programs
- What would be the most precise type you can give?
  - The exact set of values computed for a given input?
- Expressive typing problems become hard to compute
- Many are undecidable, if they imply solving the halting problem

Types and language design

Types influence language design

- Types abstract over implementation
  - Any value with the correct type is accepted
- Types enable separate or incremental compilation
  - As long as the public interface is implemented, dependent modules do not change

Can we have our cake and eat it too?

- Ever more precise types lead to ever more correct programs
- What would be the most precise type you can give?
  - The exact set of values computed for a given input?
- Expressive typing problems become hard to compute
- Many are undecidable, if they imply solving the halting problem
- Designing type systems always involves trade-offs

Relations between Types

Comparing Types

Comparing Types

```
interface Point2D { x: number, y: number }
interface Vector2D { x: number, y: number }
var p1: Point2D = { x: 5, y: -11 }
var p2: Vector2D = p1
```

Comparing Types

```
interface Point2D { x: number, y: number }
interface Vector2D { x: number, y: number }
var p1: Point2D = { x: 5, y: -11 }
var p2: Vector2D = p1
```

Is this program correct?

Comparing Types

```
interface Point2D { x: number, y: number }
interface Vector2D { x: number, y: number }
var p1: Point2D = { x: 5, y: -11 }
var p2: Vector2D = p1
```

Is this program correct?

- No, if types are compared by name

Comparing Types

```
interface Point2D { x: number, y: number }
interface Vector2D { x: number, y: number }
var p1: Point2D = { x: 5, y: -11 }
var p2: Vector2D = p1
```

Is this program correct?

- No, if types are compared by name
- Yes, if types are compared based on structure

Comparing Types

Comparing Types

```
interface Point2D { x: number, y: number }
interface Point3D { x: number, y: number, z: number }
var p1: Point3D = { x: 5, y: -11, z: 0 }
var p2: Point2D = p1
```

Comparing Types

```
interface Point2D { x: number, y: number }
interface Point3D { x: number, y: number, z: number }
var p1: Point3D = { x: 5, y: -11, z: 0 }
var p2: Point2D = p1
```

Is this program correct?

Comparing Types

```
interface Point2D { x: number, y: number }
interface Point3D { x: number, y: number, z: number }
var p1: Point3D = { x: 5, y: -11, z: 0 }
var p2: Point2D = p1
```

Is this program correct?

- No, if equal types are required

Comparing Types

```
interface Point2D { x: number, y: number }
interface Point3D { x: number, y: number, z: number }
var p1: Point3D = { x: 5, y: -11, z: 0 }
var p2: Point2D = p1
```

Is this program correct?

- No, if equal types are required
- Yes, if structural subtypes are allowed

Comparing Types

```
interface Point2D { x: number, y: number }
interface Point3D { x: number, y: number, z: number }
var p1: Point3D = { x: 5, y: -11, z: 0 }
var p2: Point2D = p1
```

Is this program correct?

- No, if equal types are required
- Yes, if structural subtypes are allowed
- When is `T` a subtype of `U`?

Comparing Types

```
interface Point2D { x: number, y: number }
interface Point3D { x: number, y: number, z: number }
var p1: Point3D = { x: 5, y: -11, z: 0 }
var p2: Point2D = p1
```

Is this program correct?

- No, if equal types are required
- Yes, if structural subtypes are allowed
- When is `T` a subtype of `U`?
  - When a value of type `T` can be used when a value of `U` is expected

Comparing Types

```
interface Point2D { x: number, y: number }
interface Point3D { x: number, y: number, z: number }
var p1: Point3D = { x: 5, y: -11, z: 0 }
var p2: Point2D = p1
```

Is this program correct?

- No, if equal types are required
- Yes, if structural subtypes are allowed
- When is `T` a subtype of `U`?
    - When a value of type `T` can be used when a value of `U` is expected
- What about nominal subtypes?

Comparing Types

```
interface Point2D { x: number, y: number }
interface Point3D { x: number, y: number, z: number }
var p1: Point3D = { x: 5, y: -11, z: 0 }
var p2: Point2D = p1
```

Is this program correct?

- No, if equal types are required
- Yes, if structural subtypes are allowed
- When is `T` a subtype of `U`?
  - When a value of type `T` can be used when a value of `U` is expected
- What about nominal subtypes?
  - `interface Point3D extends Point2D`

Combination Example: Generics and Subtyping

```
class A {}
class B extends A {}

B[] bs = new B[1];
A[] as = bs;
as[0]  = new A();
B b    = bs[0];
```

Combination Example: Generics and Subtyping

```
class A {}
class B extends A {}

B[] bs = new B[1];
A[] as = bs;
as[0]  = new A();
B b    = bs[0];
```

subtyping on arrays & mutable updates is unsound

- unsound = under-approximation of runtime behavior
- feature combinations are not trivial

Comparing Types

Comparing Types

```
int i = 12
float f = i
```

Comparing Types

```
int i = 12
float f = i
```

Is this program correct?

Comparing Types

```
int i = 12
float f = i
```

Is this program correct?

- No, floats and integers have different runtime representations

Comparing Types

```
int i = 12
float f = i
```

Is this program correct?

- No, floats and integers have different runtime representations
- Yes, possible by coercion

Comparing Types

```
int i = 12
float f = i
```

Is this program correct?

- No, floats and integers have different runtime representations
- Yes, possible by coercion
  - Coercion requires insertion of code to convert between representations

Comparing Types

```
int i = 12
float f = i
```

Is this program correct?

- No, floats and integers have different runtime representations
- Yes, possible by coercion
  - Coercion requires insertion of code to convert between representations
- How is this different than subtyping?

Comparing Types

```
int i = 12
float f = i
```

Is this program correct?

- No, floats and integers have different runtime representations
- Yes, possible by coercion
  - Coercion requires insertion of code to convert between representations
- How is this different than subtyping?
  - Subtyping says that the use of the unchanged value is safe

Type Relations

What kind of relations between types?

- Equality `T=T` – syntactic or structural
- Subtyping `T<:T` – nominal or structural
- Coercion – requires code insertion

Why Type Checking?

Why Type Checking? Some Discussion Points

Dynamically Typed vs Statically Typed

- Dynamic: type checking at run-time
- Static: type checking at compile-time (before run-time)

What does it mean to type check?

- Type safety: guarantee absence of run-time type errors

Why static type checking?

- Avoid overhead of run-time type checking
- Fail faster: find (type) errors at compile time
- Find all (type) errors: some errors may not be triggered by testing
- But: not all errors can be found statically (e.g. array bounds checking)

Formalizing Type Systems

(in the ChocoPy reference manual)

Formalizing Type Systems: Judgements and Inference Rules

$$\frac{\text{hypotheses/premises}}{\text{judgement}}$$

$$\frac{\vdots}{O, M, C, R \vdash e : T}$$

if the hypotheses/premises are true then the judgment below the bar is true

judgement: context ⊢ proposition

proposition (e : T): expression e has type T

Formalizing Type Systems: Examples

$$\frac{i \text{ is an integer literal}}{O, M, C, R \vdash i : int} \quad [\text{INT}]$$

Formalizing Type Systems: Examples

$$\frac{O, M, C, R \vdash e_1 : bool \qquad O, M, C, R \vdash e_2 : bool}{O, M, C, R \vdash e_1 \texttt{ and } e_2 : bool} \quad [\text{AND}]$$

Formalizing Type Systems: Examples

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : bool \\ O, M, C, R \vdash e_2 : bool \\ \bowtie \in \{\texttt{==}, \texttt{!=}\} \end{array}}{O, M, C, R \vdash e_1 \bowtie e_2 : bool} \quad [\text{BOOL-COMPARE}]$$

Formalizing Type Systems: Examples

$$\frac{O, M, C, R \vdash e_1 : int \quad O, M, C, R \vdash e_2 : int \quad \bowtie \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\}}{O, M, C, R \vdash e_1 \bowtie e_2 : bool} \quad [\text{INT-COMPARE}]$$

Intermezzo: Testing Static Analysis

Testing Name Resolution

```
test outer name [[
   let type t = u
       type [[u]] = int
       var x: [[u]] := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := 42
      end
end
]] resolve #2 to #1
```

```
test inner name [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type [[u]] = t
          var y: [[u]] := 0
      in
         y := 42
      end
end
]] resolve #2 to #1
```

Testing Type Checking

```
test integer constant [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[42]]
      end
end
]] run get-type to INT()
```

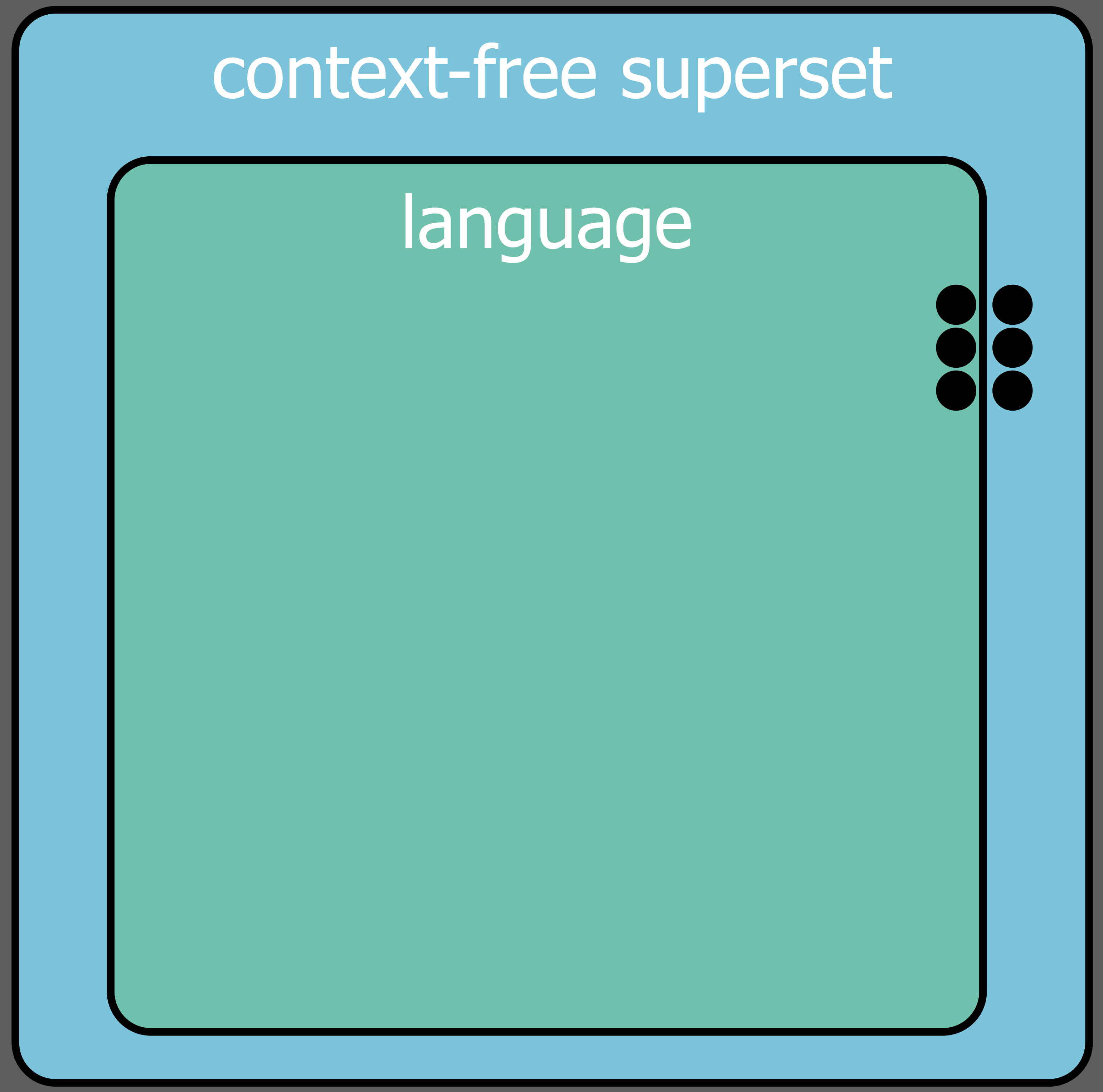
```
test variable reference [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[x]]
      end
end
]] run get-type to INT()
```

Testing Errors

```
test undefined variable [[
   let type t = u
       type u = int
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[z]]
      end
end
]] 1 error
```

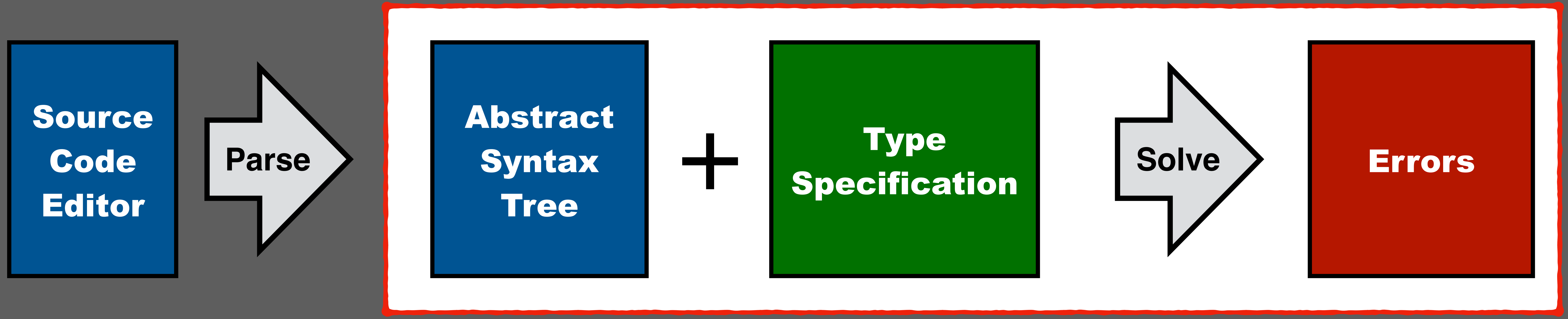
```
test type error [[
   let type t = u
       type u = string
       var x: u := 0
   in
      x := 42 ;
      let type u = t
          var y: u := 0
      in
         y := [[x]]
      end
end
]] 1 error
```

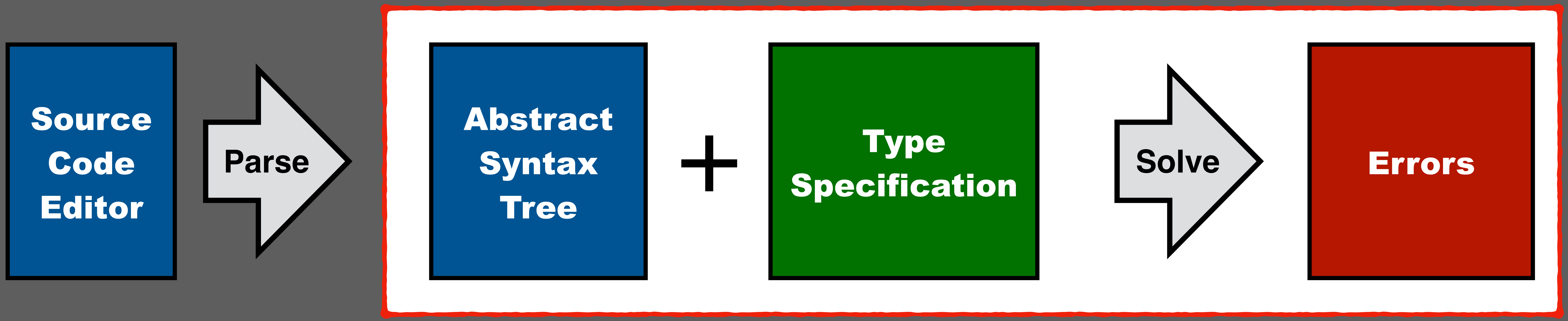
Test Corner Cases



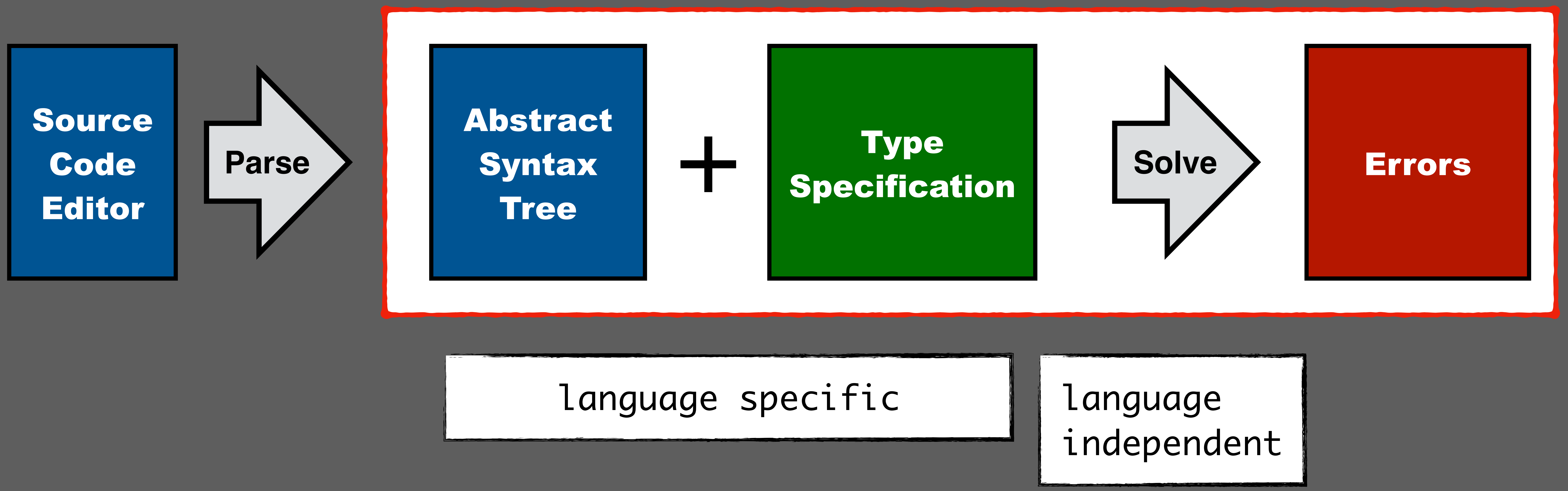
Type Checking using High-level Typing Rules

Check that names are used correctly and that expressions are well-typed


Source Code Editor
Parse
Abstract Syntax Tree
+
Type Specification
Solve
Errors



language specific


Source Code Editor
Parse
Abstract Syntax Tree
+
Type Specification
Solve
Errors
language specific
language independent

Separation of concerns

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism
- Language independent algorithm to interpret specification

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism
- Language independent algorithm to interpret specification
- Write specification, get an executable checker

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism
- Language independent algorithm to interpret specification
- Write specification, get an executable checker

Advantages

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism
- Language independent algorithm to interpret specification
- Write specification, get an executable checker

Advantages

- High-level, declarative specification

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism
- Language independent algorithm to interpret specification
- Write specification, get an executable checker

Advantages

- High-level, declarative specification
- Abstract over algorithmic concerns

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism
- Language independent algorithm to interpret specification
- Write specification, get an executable checker

Advantages

- High-level, declarative specification
- Abstract over algorithmic concerns
  - Execution order

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism
- Language independent algorithm to interpret specification
- Write specification, get an executable checker

Advantages

- High-level, declarative specification
- Abstract over algorithmic concerns
  - Execution order
  - Transparently support for inference

Type Checking with Specifications

Separation of concerns

- Language specific specification in terms of logical formalism
- Language independent algorithm to interpret specification
- Write specification, get an executable checker

Advantages

- High-level, declarative specification
- Abstract over algorithmic concerns
  - Execution order
  - Transparently support for inference
- Logical variables act as interface between different kinds of premises

Statix

What is Statix?

- Domain-specific specification language…
- … to write typing and name binding specification
- Comes with a solver to use for type checking

Statix

What is Statix?

- Domain-specific specification language…
- … to write typing and name binding specification
- Comes with a solver to use for type checking

What features does it support?

- Predicates defined by logical (Horn-clause) rules
- Rich binding structures using scope graphs
- Unification based inference

Statix

What is Statix?

- Domain-specific specification language…
- … to write typing and name binding specification
- Comes with a solver to use for type checking

What features does it support?

- Predicates defined by logical (Horn-clause) rules
- Rich binding structures using scope graphs
- Unification based inference

Limitations

- Restricted to the domain-specific (= restricted) model
  - Not all name binding patterns in the wild can be expressed
- Hypothesis is that all sensible patterns are expressible

Type System Specification in Statix

Constraint-based language with declarative semantics

- Understand type system without algorithmic reasoning

Name binding using scope graphs

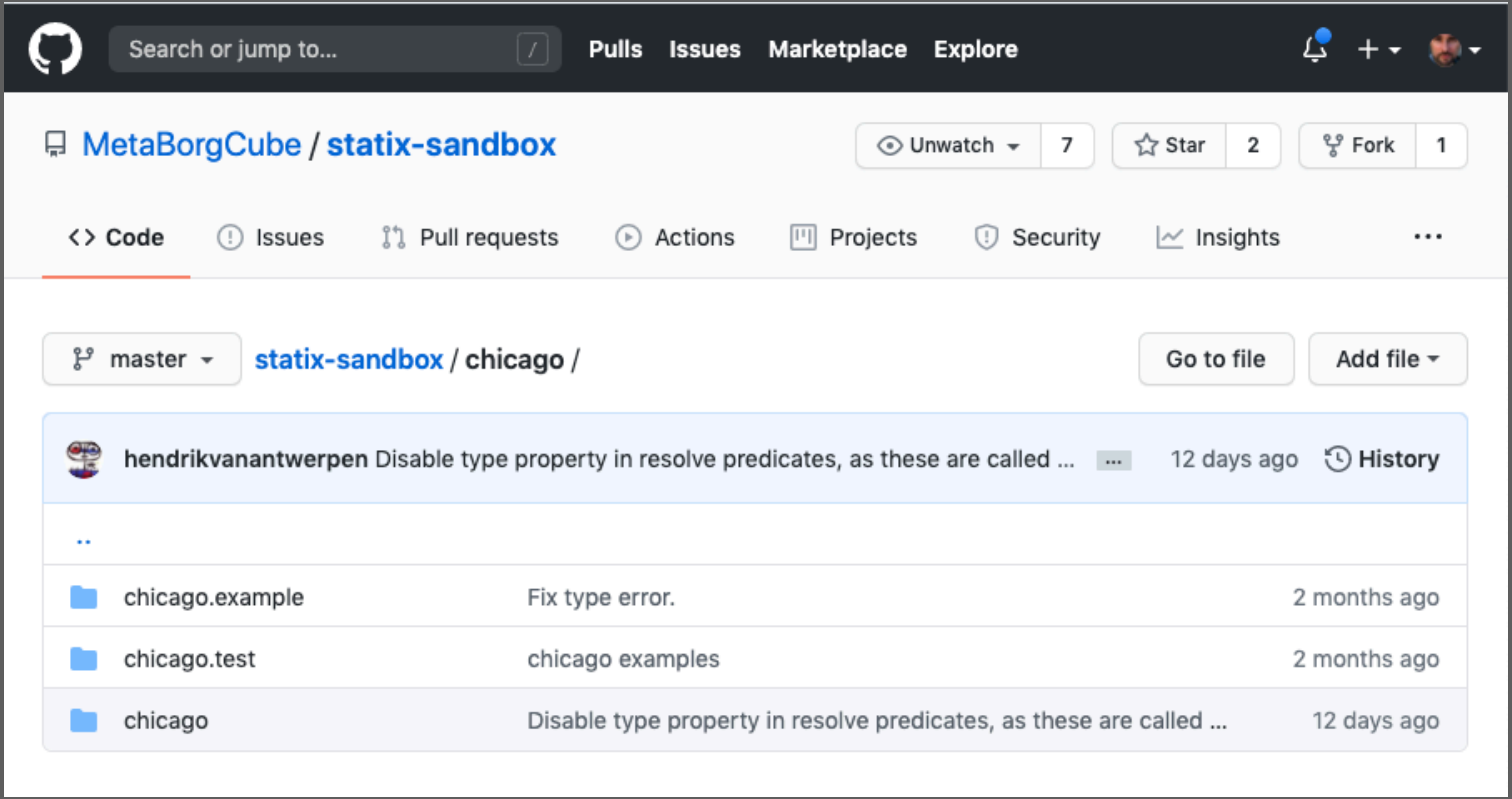
- *as part of constraint resolution*

Implementation

- Solver interprets specification as type checker
- Sound wrt declarative semantics
- Scheduling of constraint resolution based on language independent principles

Statix by Example

Example Project: statix-sandbox/chicago



Concrete and Abstract Syntax

From Concrete Syntax Definition to Abstract Syntax Signature

```
module base

imports lex

lexical sorts ID INT STRING
sorts Exp Type Val Decl Bind TYPE
context-free syntax
  Exp = <(<Exp>)> {bracket}
  Type = <(<Type>)> {bracket}
```

```
module signatures/base-sig

imports signatures/lex-sig

signature
  sorts
    ID = string
    INT = string
    STRING = string
    Exp Type Val Decl Bind TYPE
```

```
module arithmetic

imports base

context-free syntax
  Exp.Int   = <<INT>>
  Exp.Min   = [-[Exp]]
  Exp.Add   = <<Exp> + <Exp>> {left}
  Exp.Sub   = <<Exp> - <Exp>> {left}
  Exp.Mul   = <<Exp> * <Exp>> {left}
  Type.IntT = <Int>

context-free priorities
  Exp.Mul > {left: Exp.Add Exp.Sub}
```

```
module signatures/arithmetic-sig

imports signatures/base-sig

signature
  constructors
    Int : INT → Exp
    Min : Exp → Exp
    Add : Exp * Exp → Exp
    Sub : Exp * Exp → Exp
    Mul : Exp * Exp → Exp
    IntT : Type
```

From Concrete Syntax Definition to Abstract Syntax Signature

```
module arithmetic

imports base

context-free syntax
  Exp.Int   = <<INT>>
  Exp.Min   = [-[Exp]]
  Exp.Add   = <<Exp> + <Exp>> {left}
  Exp.Sub   = <<Exp> - <Exp>> {left}
  Exp.Mul   = <<Exp> * <Exp>> {left}
  Type.IntT = <Int>

context-free priorities
  Exp.Mul > {left: Exp.Add Exp.Sub}
```

```
module signatures/arithmetic-sig

imports signatures/base-sig

signature
  constructors
    Int : INT → Exp
    Min : Exp → Exp
    Add : Exp * Exp → Exp
    Sub : Exp * Exp → Exp
    Mul : Exp * Exp → Exp
    IntT : Type
```

```
1 + 2 * 3
```

```
Add(
  Int("1"),
  Mul(
    Int("2"),
    Int("3")))
```

From here we will use concrete syntax examples and abstract syntax rules

Predicates

Predicates Represent Program Properties

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules // type of ...

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Statix is a *pure logic programming language*

Use `maps` to apply a predicate to all elements of a list

Predicates Represent Program Properties

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules // type of ...

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Statix is a *pure logic programming language*

A Statix specification defines *predicates*

Use maps to apply a predicate to all elements of a list

Predicates Represent Program Properties

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules // type of ...

  typeOfType : scope * Type  → TYPE
  typeOfExp  : scope * Exp   → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Statix is a *pure logic programming language*

A Statix specification defines *predicates*

If a predicate *holds* for some term, the term has the *property* represented by the predicate

Use `maps` to apply a predicate to all elements of a list

Predicates Represent Program Properties

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules // type of ...

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Use `maps` to apply a predicate to all elements of a list

Statix is a *pure logic programming language*

A Statix specification defines *predicates*

If a predicate *holds* for some term, the term has the *property* represented by the predicate

`typeOfExp(s, e) == T`

expression `e` has type `T` in scope `s`

Predicates Represent Program Properties

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules // type of ...

  typeOfType : scope * Type  → TYPE
  typeOfExp  : scope * Exp   → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Use `maps` to apply a predicate to all elements of a list

Statix is a *pure logic programming language*

A Statix specification defines *predicates*

If a predicate *holds* for some term, the term has the *property* represented by the predicate

`typeOfExp(s, e) == T`

expression `e` has type `T` in scope `s`

`typeOfType(s, t) == T`

syntactic type `t` has semantic type `T` in scope `s`

Predicates Represent Program Properties

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules // type of ...

  typeOfType : scope * Type  → TYPE
  typeOfExp  : scope * Exp   → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

Use `maps` to apply a predicate to all elements of a list

Statix is a *pure logic programming language*

A Statix specification defines *predicates*

If a predicate *holds* for some term, the term has the *property* represented by the predicate

`typeOfExp(s, e) == T`
expression `e` has type `T` in scope `s`

`typeOfType(s, t) == T`
syntactic type `t` has semantic type `T` in scope `s`

`declOk(s, d)`
declaration `d` is well-defined (Ok) in scope `s`

Functional Notation vs Predicate Notation

```
rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

Functional Notation vs Predicate Notation

```
rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

```
rules

  typeOfType : scope * Type * TYPE
  typeOfExp  : scope * Exp  * TYPE
```

Functional Notation vs Predicate Notation

```
rules

  typeOfType : scope * Type -> TYPE
  typeOfExp  : scope * Exp  -> TYPE
```

```
rules

  typeOfType : scope * Type * TYPE
  typeOfExp  : scope * Exp  * TYPE
```

```
typeOfExp(s, e) == T
```

expression e has type T in scope s

Functional Notation vs Predicate Notation

```
rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

```
typeOfExp(s, e) == T
```

expression e has type T in scope s

```
rules

  typeOfType : scope * Type * TYPE
  typeOfExp  : scope * Exp  * TYPE
```

```
typeOfExp(s, e, T)
```

expression e has type T in scope s

Functional Notation vs Predicate Notation

```
rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

```
typeOfExp(s, e) == T
```

expression e has type T in scope s

One expression has one type

```
rules

  typeOfType : scope * Type * TYPE
  typeOfExp  : scope * Exp  * TYPE
```

```
typeOfExp(s, e, T)
```

expression e has type T in scope s

Functional Notation vs Predicate Notation

```
rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

```
typeOfExp(s, e) == T
```

expression e has type T in scope s

One expression has one type

```
rules

  typeOfType : scope * Type * TYPE
  typeOfExp  : scope * Exp  * TYPE
```

```
typeOfExp(s, e, T)
```

expression e has type T in scope s

One expression can have multiple types

Functional Notation vs Predicate Notation

```
rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

```
rules

  typeOfType : scope * Type * TYPE
  typeOfExp  : scope * Exp  * TYPE
```

`typeOfExp(s, e) == T`

expression e has type T in scope s

`typeOfExp(s, e, T)`

expression e has type T in scope s

One expression has one type

One expression can have multiple types

(Solver does not match on type argument)

Predicates are Defined by Rules

```
typeOfExp : scope * Exp → TYPE
```

Predicates are Defined by Rules

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Predicates are Defined by Rules

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT().
```

Predicates are Defined by Rules

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT().
```

Head

Predicates are Defined by Rules

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT().
```

Head

Premises

Predicates are Defined by Rules

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT().
```

Head

Premises

For all s, e1, e2

If the premises are true, the head is true

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Names

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Names

```
typeOfExp(e) == T
```

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Names

```
typeOfExp(e) == T
```

The type of expression `e` is T

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Names

```
typeOfExp(e) == T
```

The type of expression e is T

Type system defines a (functional) relation

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Names

Operational Names

```
typeOfExp(e) == T
```

The type of expression `e` is T

Type system defines a (functional) relation

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Names	Operational Names
`typeOfExp(e) == T`	`typeCheck(e) = T`
The type of expression `e` is T	
Type system defines a (functional) relation	

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Names	Operational Names
`typeOfExp(e) == T`	`typeCheck(e) = T`
The type of expression `e` is `T`	Type checking expression `e` produces type `T`
Type system defines a (functional) relation	

Declarative Reading vs Operational Reading

Predicate

```
typeOfExp : scope * Exp → TYPE
```

Rule

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Head

Premises

Declarative Names	Operational Names
`typeOfExp(e) == T`	`typeCheck(e) = T`
The type of expression `e` is `T`	Type checking expression `e` produces type `T`
Type system defines a (functional) relation	Type checking is a process

Syntax-Directed Definitions: One Rule per Language Construct

```
module statix/base

imports signatures/base-sig

rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

```
module signatures/arithmetic-sig

imports signatures/base-sig

signature
  constructors
    Int : INT → Exp
    Min : Exp → Exp
    Add : Exp * Exp → Exp
    Sub : Exp * Exp → Exp
    Mul : Exp * Exp → Exp
    IntT : Type
```

```
module statics/arithmetic

imports statics/base
imports signatures/arithmetic-sig

signature
  constructors
    INT : TYPE

rules
  typeOfType(s, IntT()) = INT().

rules
  typeOfExp(s, Int(i)) = INT().

  typeOfExp(s, Min(e)) = INT() :-
    typeOfExp(s, e) == INT().

  typeOfExp(s, Add(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Sub(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Mul(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().
```

From Now: No Module Headers

```
rules

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE
```

```
signature
  constructors
    Int : INT → Exp
    Min : Exp → Exp
    Add : Exp * Exp → Exp
    Sub : Exp * Exp → Exp
    Mul : Exp * Exp → Exp
    IntT : Type
```

```
signature
  constructors
    INT : TYPE

rules
  typeOfType(s, IntT()) = INT().

rules
  typeOfExp(s, Int(i)) = INT().

  typeOfExp(s, Min(e)) = INT() :-
    typeOfExp(s, e) == INT().

  typeOfExp(s, Add(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Sub(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Mul(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().
```

Types Are Just Terms

```
signature
  constructors
    BoolT     : Type
    BOOL      : TYPE
    True      : Exp
    False     : Exp
    Not       : Exp → Exp
    And       : Exp * Exp → Exp
    Or        : Exp * Exp → Exp
    If        : Exp * Exp * Exp → Exp
    Eq        : Exp * Exp → Exp
```

```
rules // operations on types

  subtype   : Exp * TYPE * TYPE
  equitype : TYPE * TYPE
  lub       : TYPE * TYPE → TYPE

  subtype(_, T, T).
  equitype(T, T).
  lub(T, T) = T.
```

```
rules

  typeOfType(s, BoolT()) = BOOL().

rules

  typeOfExp(s, True()) = BOOL().

  typeOfExp(s, False()) = BOOL().

  typeOfExp(s, And(e1, e2)) = BOOL() :-
    typeOfExp(s, e1) == BOOL(),
    typeOfExp(s, e2) == BOOL().

  typeOfExp(s, If(e1, e2, e3)) = lub(T1, T2) :-
    typeOfExp(s, e1) == BOOL(),
    typeOfExp(s, e2) == T1,
    typeOfExp(s, e3) == T2,
    equitype(T1, T2).

  typeOfExp(s, Eq(e1, e2)) = BOOL() :- {T1 T2}
    typeOfExp(s, e1) == T1,
    typeOfExp(s, e2) == T2,
    equitype(T1, T2).
```

From Declarative Definition to Type Checker

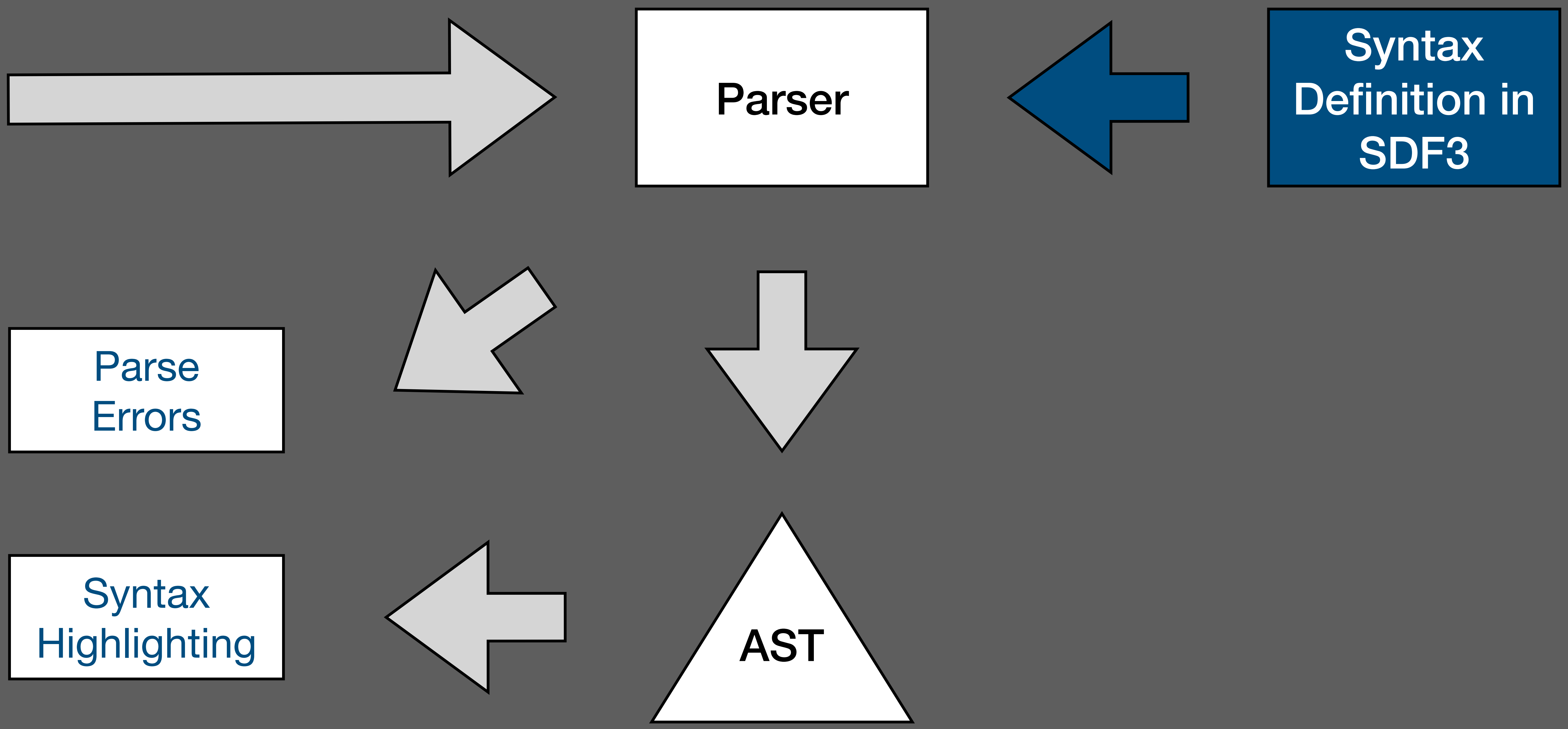
```
> 1 + 2 * 3

> true && false

> 1 ^ 2

> true + 4

> 1 && (true || false)

> if 1 == 1 then
    true
  else
    1 == 3

> if 1 == 1 then
    true
  else
    2
```

From Declarative Definition to Type Checker

```
> 1 + 2 * 3

> true && false

> 1 ^ 2

> true + 4

> 1 && (true || false)

> if 1 == 1 then
    true
  else
    1 == 3

> if 1 == 1 then
    true
  else
    2
```

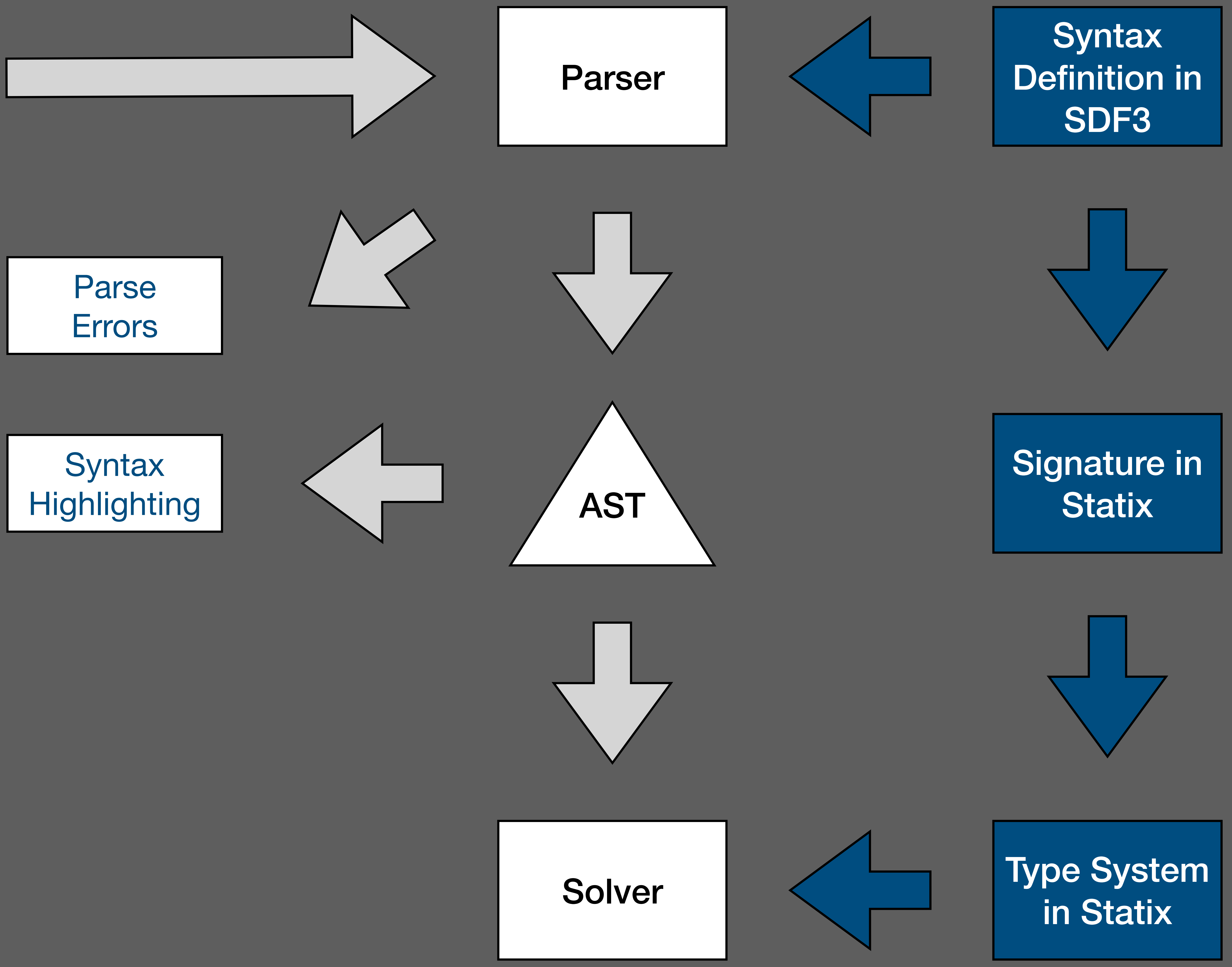
From Declarative Definition to Type Checker

```
> 1 + 2 * 3

> true && false

> 1 ^ 2

> true + 4

> 1 && (true || false)

> if 1 == 1 then
    true
  else
    1 == 3

> if 1 == 1 then
    true
  else
    2
```



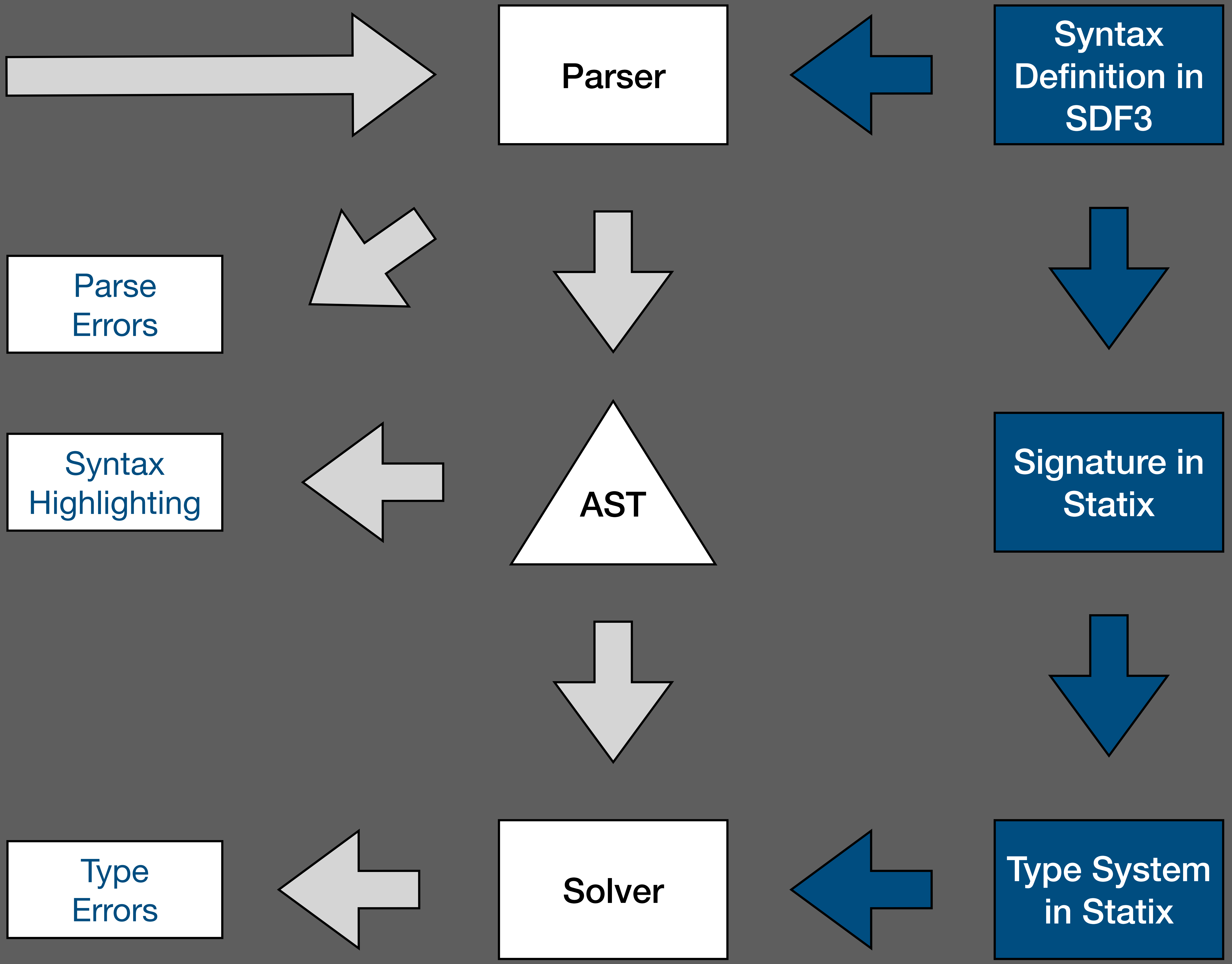
From Declarative Definition to Type Checker

```
> 1 + 2 * 3

> true && false

> 1 ^ 2

> true + 4

> 1 && (true || false)

> if 1 == 1 then
    true
  else
    1 == 3

> if 1 == 1 then
    true
  else
    2
```

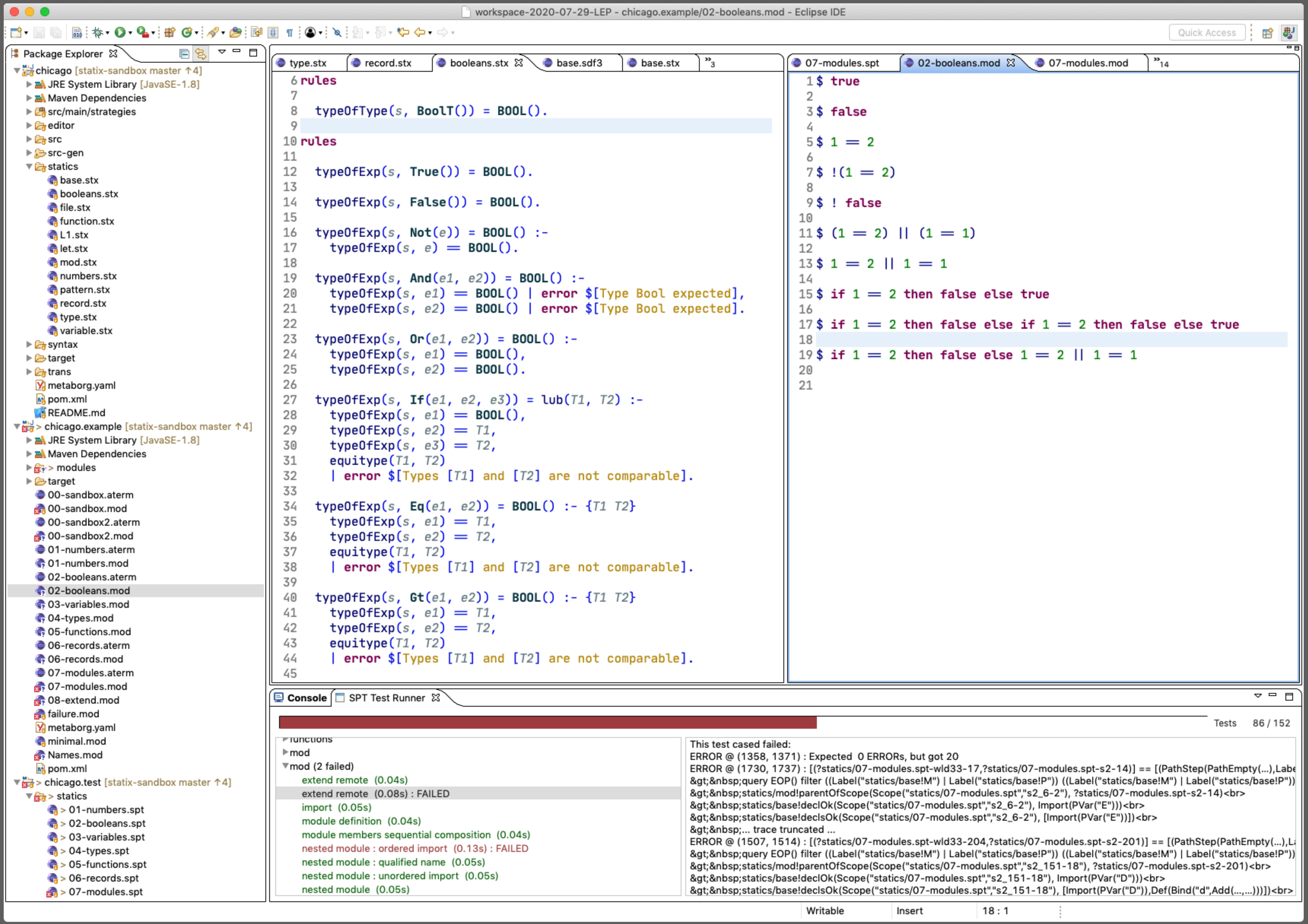


From Declarative Definition to Type Checker

```
> 1 + 2 * 3

> true && false

> 1 ^ 2

> true + 4

> 1 && (true || false)

> if 1 == 1 then
    true
  else
    1 == 3

> if 1 == 1 then
    true
  else
    2
```



Statix in Spoofax



Programs with Names

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x) {
         if x == 0 then true else odd(x - 1)
        }
  }

  module Odd {
    import Even
    def odd = fun(x) {
          if x == 0 then false else even(x - 1)
        }
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x) {
           Result{ input = x, output = Odd@odd x }
        }
  }

}
```

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x) {
         if x == 0 then true else odd(x - 1)
        }
  }

  module Odd {
    import Even
    def odd = fun(x) {
          if x == 0 then false else even(x - 1)
        }
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x) {
           Result{ input = x, output = Odd@odd x }
        }
  }

}
```

Name binding key in programming languages

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x) {
         if x == 0 then true else odd(x - 1)
        }
  }

  module Odd {
    import Even
    def odd = fun(x) {
          if x == 0 then false else even(x - 1)
        }
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x) {
           Result{ input = x, output = Odd@odd x }
        }
  }

}
```

Name binding key in programming languages

Many name binding patterns

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x) {
         if x == 0 then true else odd(x - 1)
        }
  }

  module Odd {
    import Even
    def odd = fun(x) {
          if x == 0 then false else even(x - 1)
        }
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x) {
           Result{ input = x, output = Odd@odd x }
        }
  }

}
```

Name binding key in programming languages

Many name binding patterns

Deal with erroneous programs

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x) {
         if x == 0 then true else odd(x - 1)
        }
  }

  module Odd {
    import Even
    def odd = fun(x) {
          if x == 0 then false else even(x - 1)
        }
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x) {
           Result{ input = x, output = Odd@odd x }
        }
  }

}
```

Name binding key in programming languages

Many name binding patterns

Deal with erroneous programs

Name resolution complicates type checkers, compilers

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x) {
         if x == 0 then true else odd(x - 1)
        }
  }

  module Odd {
    import Even
    def odd = fun(x) {
          if x == 0 then false else even(x - 1)
        }
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x) {
           Result{ input = x, output = Odd@odd x }
        }
  }

}
```

Name binding key in programming languages

Many name binding patterns

Deal with erroneous programs

Name resolution complicates type checkers, compilers

Ad hoc non-declarative treatment

Programs with Names

```
module Names {

  module Even {
    import Odd
    def even = fun(x) {
         if x == 0 then true else odd(x - 1)
        }
  }

  module Odd {
    import Even
    def odd = fun(x) {
          if x == 0 then false else even(x - 1)
        }
  }

  module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x) {
           Result{ input = x, output = Odd@odd x }
        }
  }

}
```

Name binding key in programming languages

Many name binding patterns

Deal with erroneous programs

Name resolution complicates type checkers, compilers

Ad hoc non-declarative treatment

A systematic, uniform approach to name resolution?

Name Resolution with Scope Graphs

Program

```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

Name Resolution with Scope Graphs

Program

```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

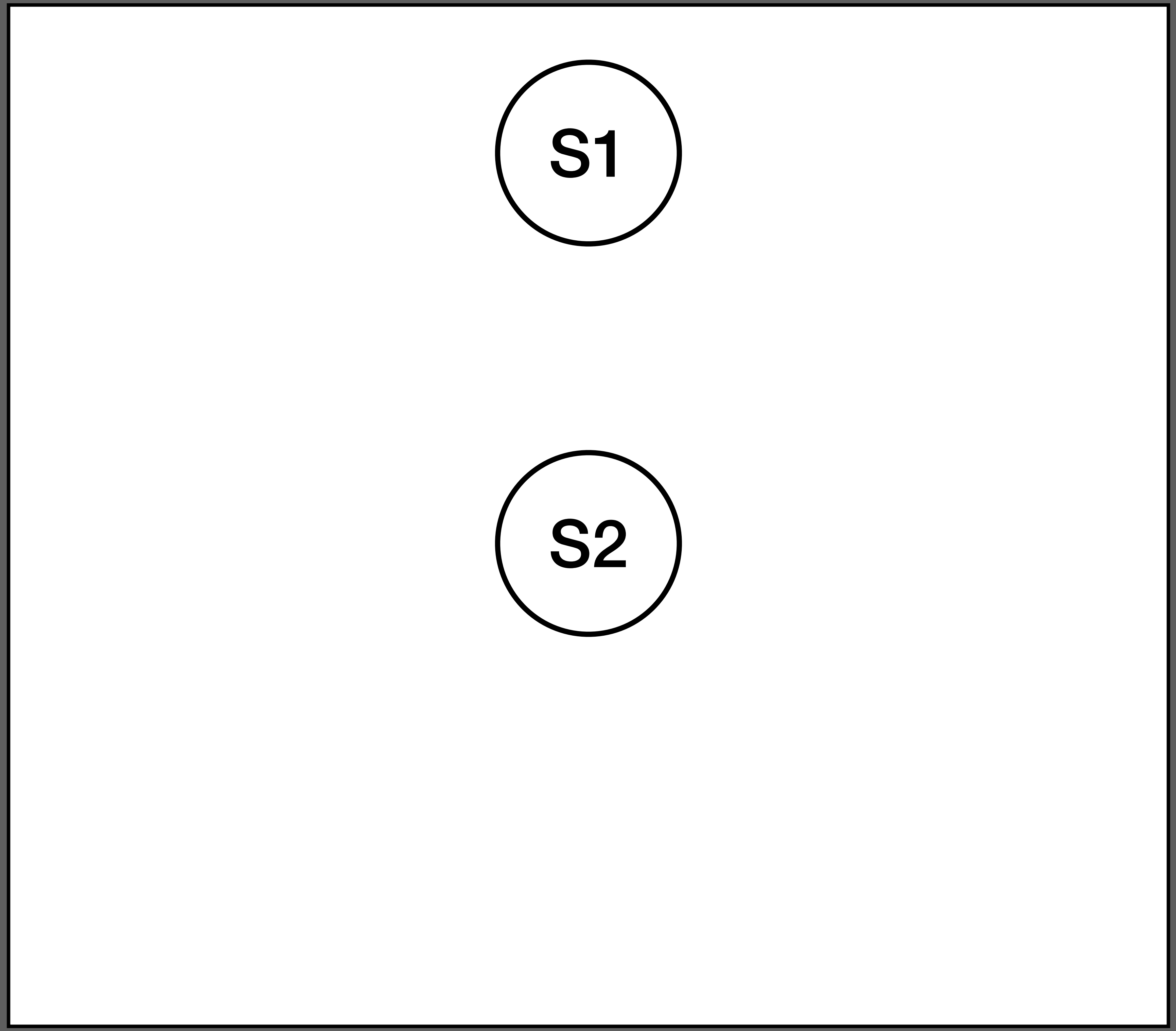
Scope Graph

Name Resolution with Scope Graphs

Program

```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

Scope Graph

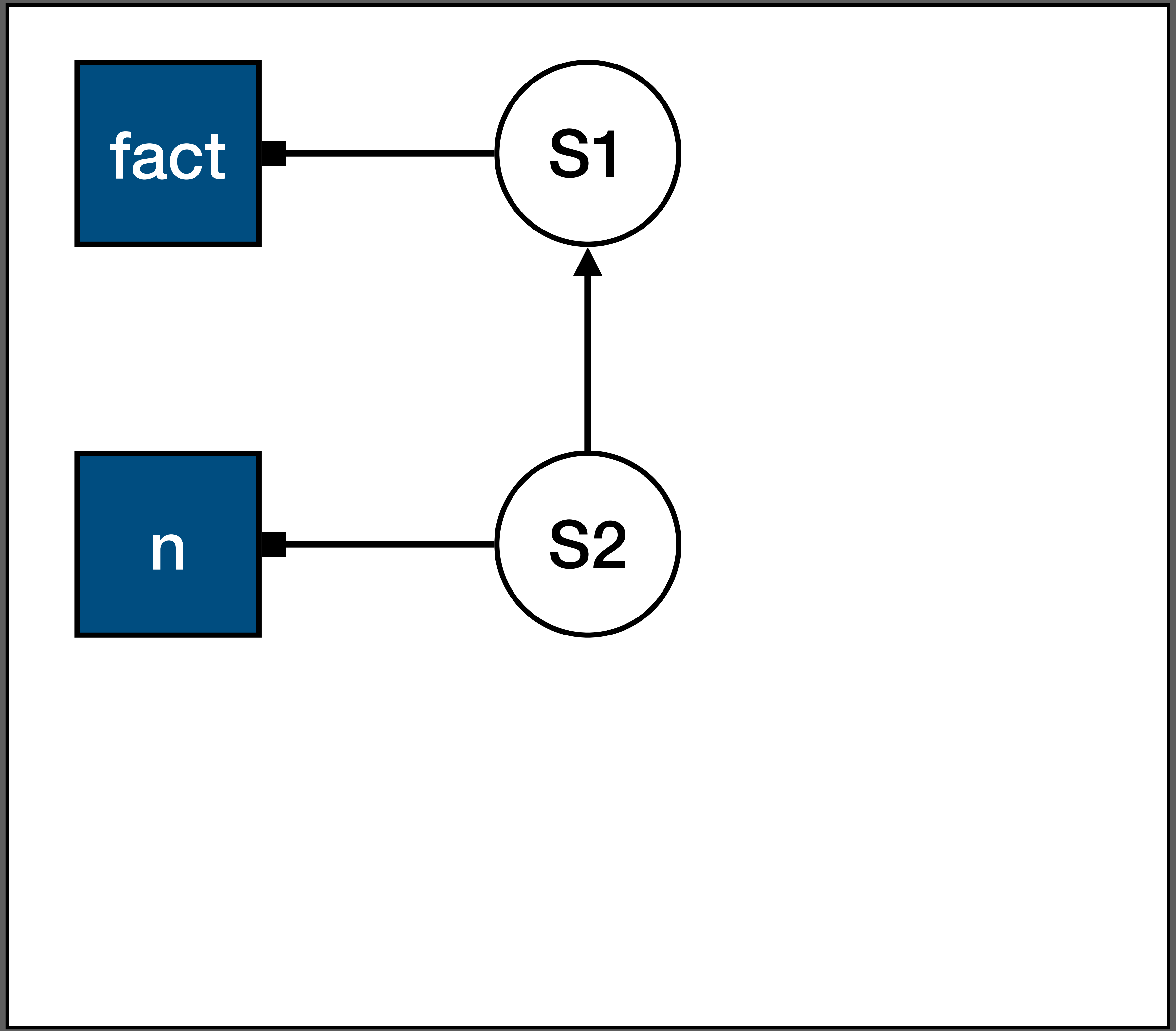


Name Resolution with Scope Graphs

Program

```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

Scope Graph

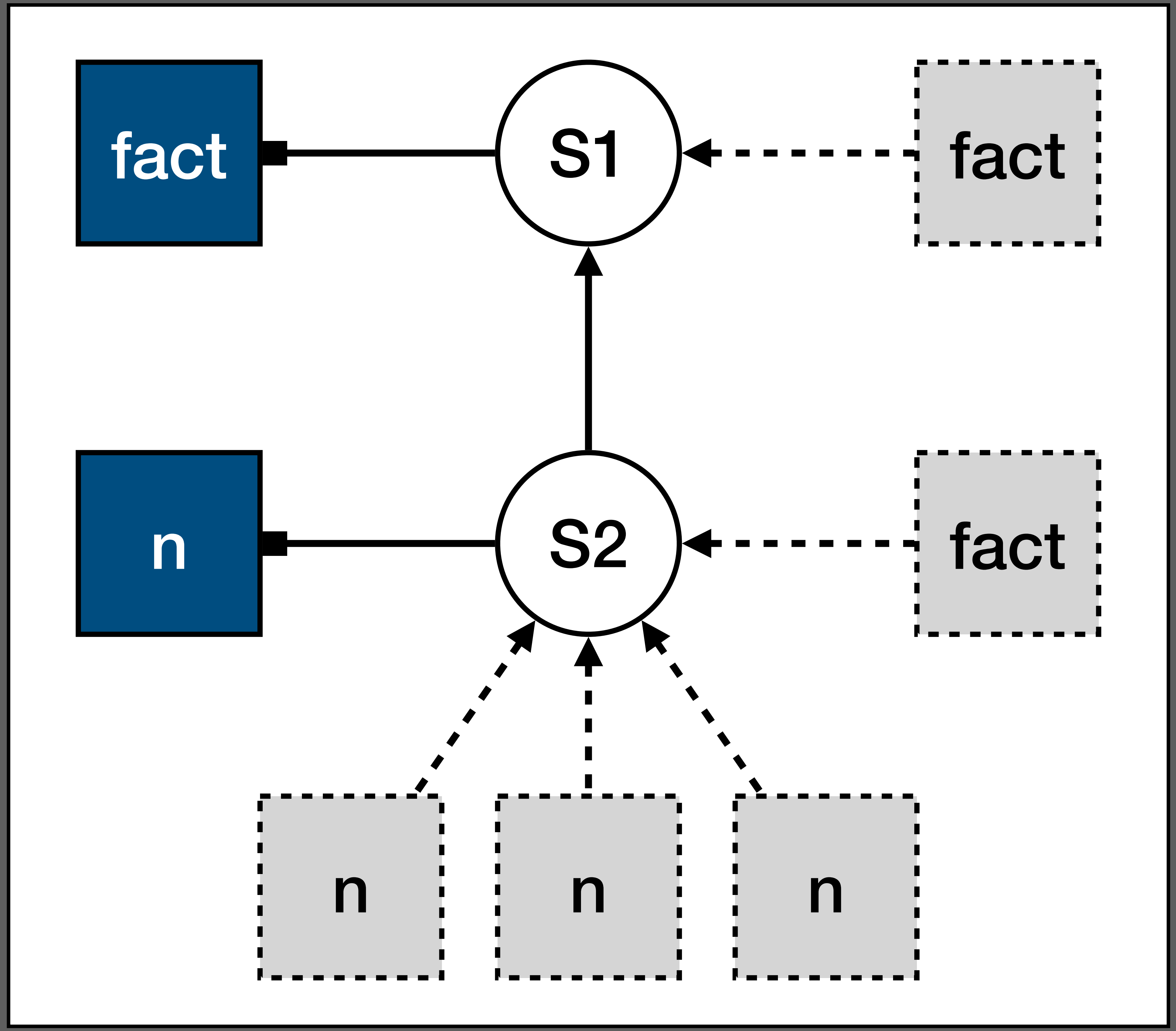


Name Resolution with Scope Graphs

Program

```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

Scope Graph

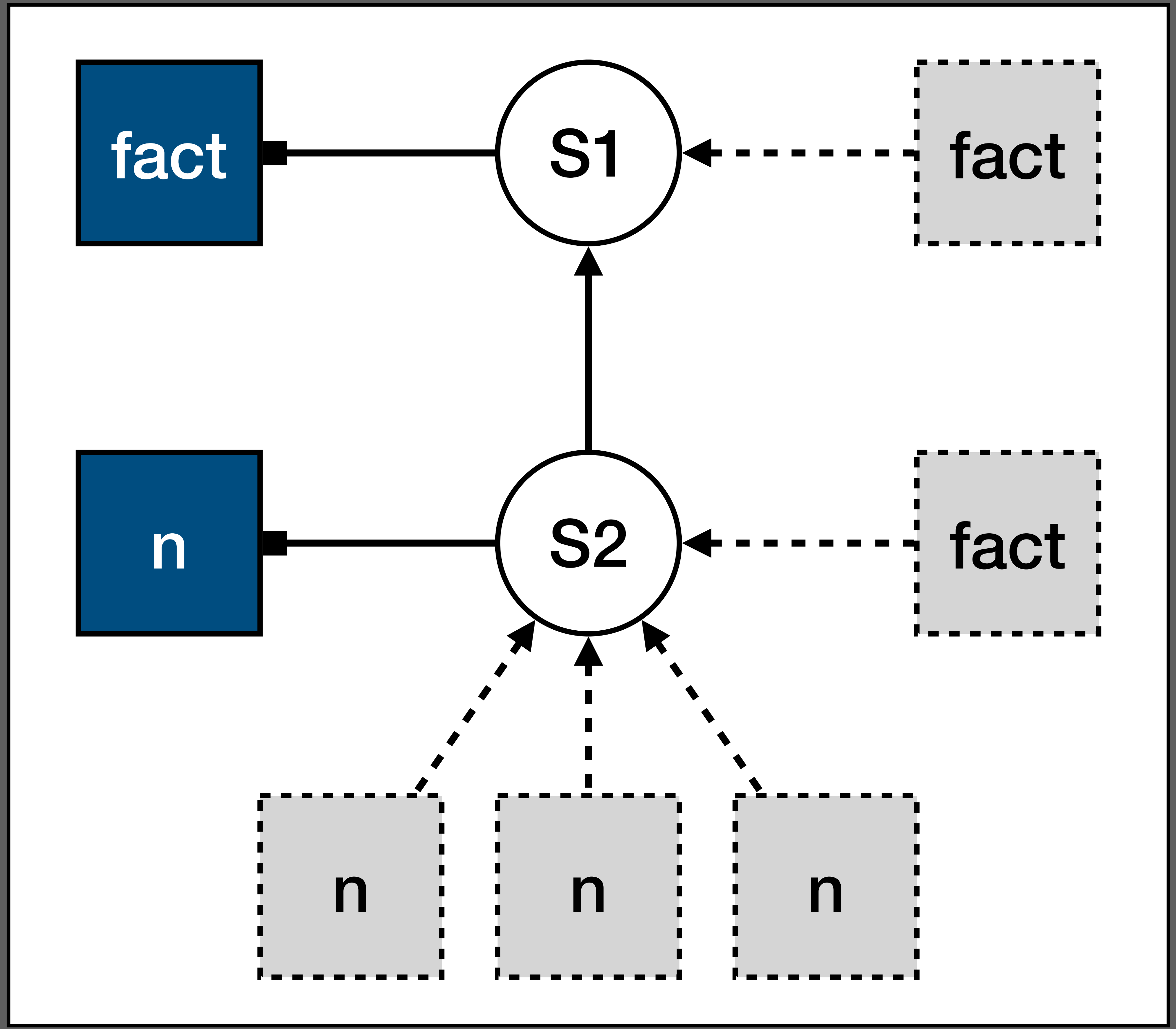


Name Resolution with Scope Graphs

Program

```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

Scope Graph



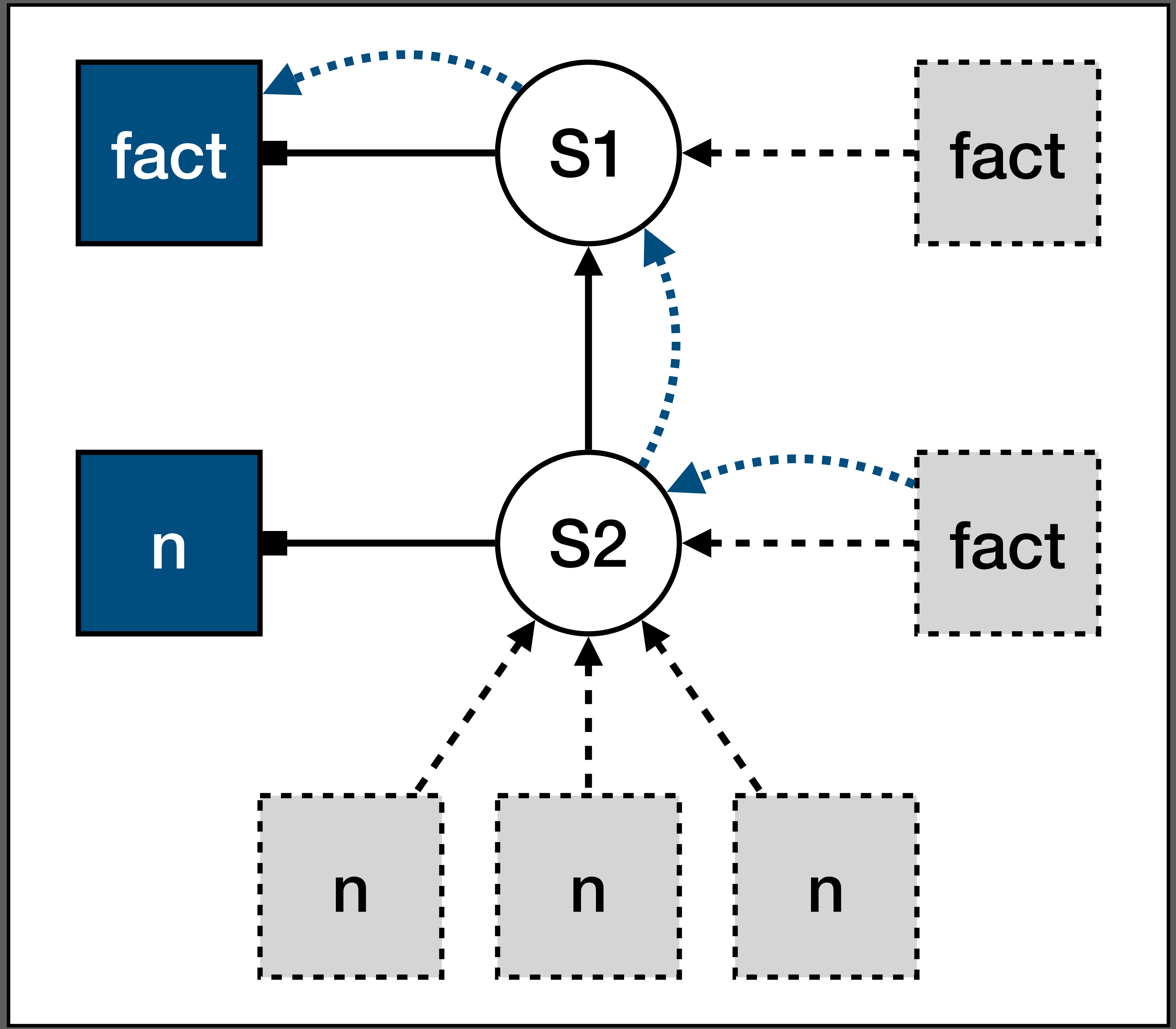
Name Resolution

Name Resolution with Scope Graphs

Program

```
let function fact(n : int) : int =
      if n < 1 then
        1
      else
        n * fact(n - 1)
 in
    fact(10)
end
```

Scope Graph



Name Resolution

Name Resolution with Scope Graphs in Statix

Declarations and References

Lexical Scope

Records

Modules

Permission to Extend

Scheduling Resolution

Reading Material

Publications on Statix

A Theory of Name Resolution

- Néron, Tolmach, Visser, Wachsmuth
- ESOP 2015

A constraint language for static semantic analysis based on scope graphs

- van Antwerpen, Néron, Tolmach, Visser, Wachsmuth
- PEPM 2016

Scopes as Types

- Van Antwerpen, Bach Poulsen, Rouvoet, Visser
- OOPSLA 2018

Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications

- Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, Eelco Visser.
- PACMPL 4(OOPSLA) 2020

Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers

- Hendrik van Antwerpen, Eelco Visser.
- ECOOP 2021

Next: Name Binding and Name Resolution