

Typical

Taking the Tedium Out of Typing

Robert Grimm, Laune Harris, Anh Le
New York University

The Trouble with Type Checkers

- Complex

- Require significant engineering effort

GCC C	8,400
O'Caml	18,400
Java	23,200

- Hard to debug

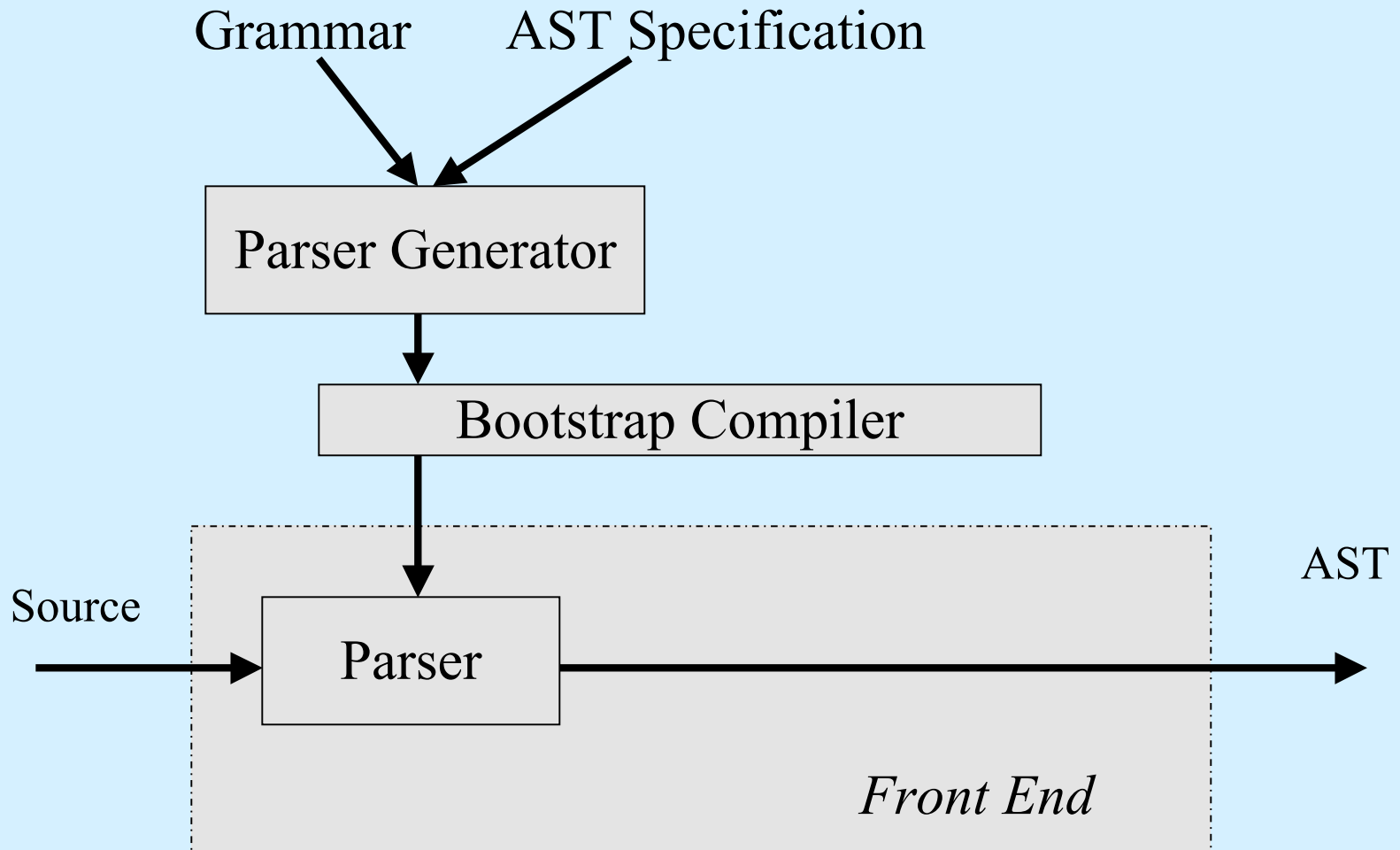
- Safety critical

- Wrongly typed programs misbehave, even crash

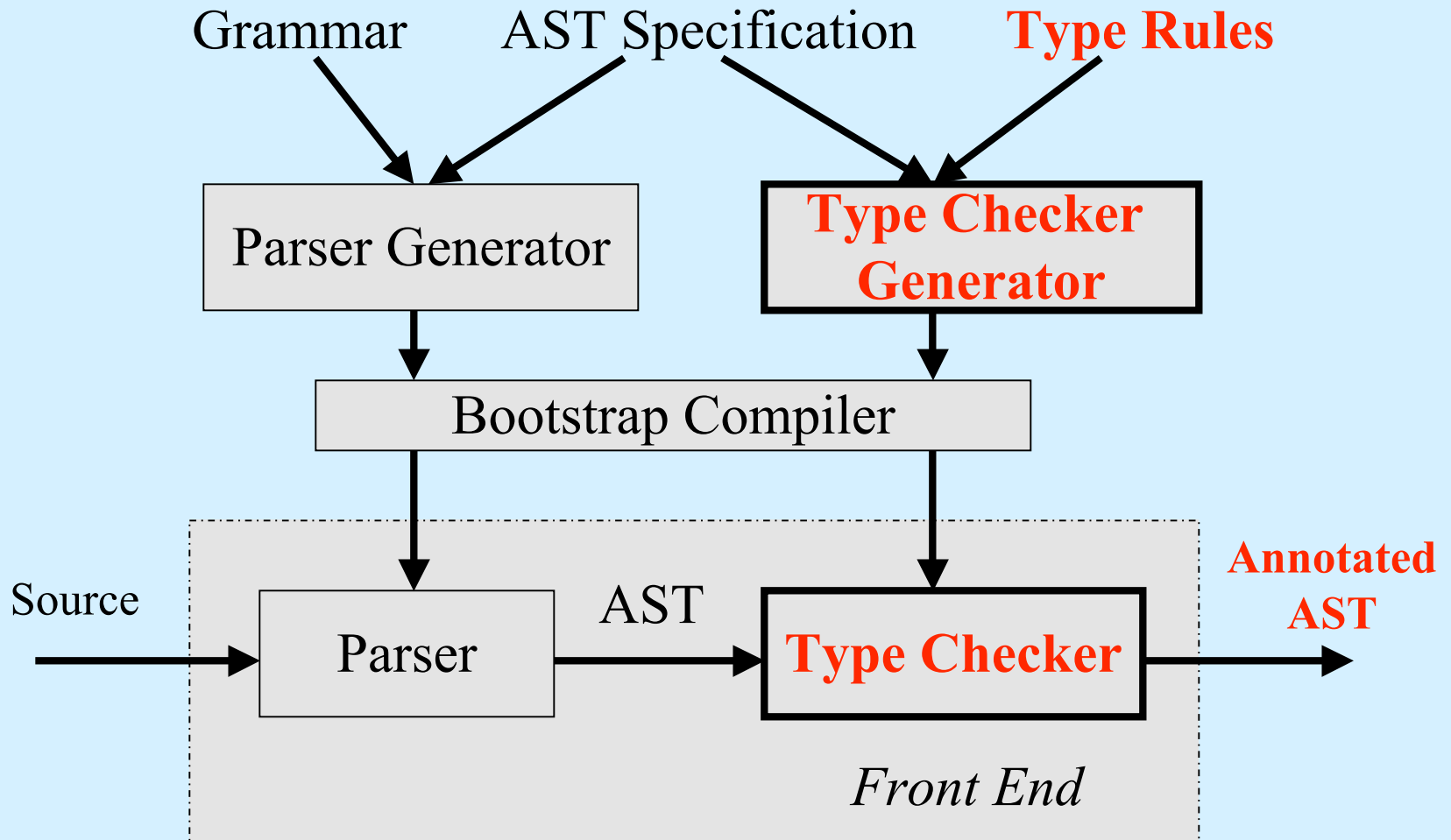
- Hard to extend

- Fundamentally new types (e.g., objects for C)
- Pluggable type systems (e.g., nonnull modifier for Java)

Can We Do Better?



Use a Domain-Specific Language!



The Commons of Typing

- Common specification structure: case analysis
 - C and Java specifications in prose
 - ML specification in formal notation
- Common language idioms
 - Scopes and namespaces
- Common type checker operations
 - AST traversal and annotation
 - Symbol table access
 - Error detection, message generation, and reporting

Language Design Guidelines

- Expressive
 - Directly capture common type checker idioms
 - Support uncommon operations
- Declarative
 - Capture type system aspects, not implementation
- Prescriptive
 - Integrate error management with basic constructs
- Correct
 - Prevent type checker bugs in the first place
 - Enable automatic reasoning (future work)

Let's Make This Concrete: Simply Typed Lambda Calculus

30 + 7 lines

```
m1type node =  
  | Abstraction of node * node * node  
  | Application of node * node  
  | Identifier of string  
  | IntegerConstant of string  
  | FunctionType of node * node  
  | IntegerType ;  
m1type type = IntegerT  
  | FunctionT of type * type;  
scope Abstraction _ as lambda ->  
  Scope(Anonymous("lambda"), [lambda]) ;  
namespace default =  
  Identifier (id) -> SimpleName(id) ;  
m1value analyze = function  
  | Application (lambda, expr) ->  
    let tl = analyze lambda  
    and tr = analyze expr in
```

```
begin match tl with  
  | FunctionT (param, res) ->  
    require param = tr  
    error "argument type mismatch" in  
    res  
  | _ -> error "applying non-function"  
end  
| Abstraction (id, type, body) ->  
  let param = analyze type in  
  let _ = define id param in  
  let res = analyze body in  
    FunctionT (param, res)  
| Identifier _ as id -> lookup id  
| IntegerConstant _ -> IntegerT  
| FunctionType (parameter, result) ->  
  FunctionT (analyze parameter,  
             analyze result)  
| IntegerType -> IntegerT ;
```

λ

Let's Make This Concrete: Simply Typed Lambda Calculus

```
m1type node =  
  | Abstraction of node * node * node  
  | App  
  | Id  
  | Int  
  | FunctionType of node * node  
  | IntegerType ;
```

Machine-generated
AST specification

```
m1type t1  
  | Fun
```

Type representation

```
scope Abstraction _ as lambda ->  
  Scop  
  namesp  
  Identifier (id) -> SimpleName(id) ;
```

Scoping and namespaces

```
m1value analyze = function  
  | Application (lambda, expr) ->  
    let tl = analyze lambda  
    and tr = analyze expr in
```

```
begin match tl with  
  | FunctionT (param, res) ->  
    require param = tr  
    error "argument type mismatch" in  
    res
```

```
  | _ ->  
end
```

Typing rules

```
| Abstrac  
  let p  
  let _
```

and

Error management

```
  let res = analyze body in  
    FunctionT (param, res)  
| Identifier _ as id -> lookup id  
| IntegerConstant _ -> IntegerT  
| FunctionType (parameter, result) ->  
  FunctionT (analyze parameter,  
             analyze result)  
| IntegerType -> IntegerT ;
```

λ

Abstract Syntax Tree

- Goals
 - Represent AST in functional core
 - Facilitate AST sharing with other compiler phases
- Solution
 - Represent nodes as variant types
 - Abstract away implementation details, including source location
 - Specify type rules as pattern matches
 - Automatically generate AST declaration from grammar

```
m1type node = Abstraction of node * node * node  
  | Application node * node    | Identifier of string  
  | IntegerConstant of string | IntegerType;    [Rats!-generated AST]
```

λ

Expressing Language Types: Representation

- Represent types with variants

```
mctype type = IntegerT | FunctionT of type * type  $\lambda$ 
```

- Represent attributed types with variants and attribute declarations
 - Compiler combines attributes and raw_type

```
mctype raw_type = IntegerT | PointerT of type ...;  
mctype qualifier = Const | Restrict | Volatile;  
mctype storage_class = Auto | Extern | Static | Typedef;
```

```
attribute qualifiers : qualifier list;  
attribute storage : storage_class;
```

Name Management

- Implicit, block structured, imperative symbol table
 - Simplifies repeated passes over AST
- Namespace declarations
 - Introduce namespace/values pairs
 - Map AST to names
- Scope declarations
 - Specify scope kind and range
 - Type checker syncs symbol table with program scope

Scope Example

a. Type rule

Abstraction (id, type, body) ->
expr;

b. Scope rule

scope Abstraction _ a ->
Scope(Anonymous("lambda"), [a]);

1. Match node with a.
2. Match node with b.

Scope Example

a. Type rule

Abstraction (id, type, body) ->
expr;

b. Scope rule

scope Abstraction _ a ->
Scope(Anonymous("lambda"), [a]);

1. Match node with a.
2. Match node with b.
3. Annotate node with scope info
4. Create new scope.
5. Push node onto traversal path.

Scope Example

a. Type rule

Abstraction (id, type, body) ->
expr;

b. Scope rule

scope Abstraction _ a ->
Scope(Anonymous("lambda"), [a]);

1. Match node with a.
2. Match node with b.
3. Annotate node with scope info
4. Create new scope.
5. Push node onto traversal path.
6. Evaluate *expr*

Scope Example

a. Type rule

Abstraction (id, type, body) ->
expr;

b. Scope rule

scope Abstraction _ a ->
Scope(Anonymous("lambda"), [a]);

1. Match node with a.
2. Match node with b.
3. Annotate node with scope info
4. Create new scope.
5. Push node onto traversal path.
6. Evaluate *expr*
7. Pop node from traversal path.
8. Restore previous scope.

Scope Example

a. Type rule

Abstraction (id, type, body) ->
expr;

b. Scope rule

scope Abstraction _ a ->
Scope(Anonymous("lambda"), [a]);

1. Match node with a.
2. Match node with b.
3. Annotate node with scope info
4. Create new scope.
5. Push node onto traversal path.
6. Evaluate *expr*
7. Pop node from traversal path.
8. Restore previous scope.
9. Return value of *expr*

Error Management

- Goals
 - Represent failures
 - Reduce notational clutter
 - Avoid cascading error messages
- Approach
 - System wide no-information monad
 - Bottom value injected into all types
 - Primitive operations produce bottom on error
 - Integrate detection and reporting into basic constructs

```
require param = arg error "argument type mismatch" in ret
```

Other Features

- List processing
 - ‘reduce’ declaratively enforces list constraints
- Constant propagation
 - Bottom for no compile time constant
 - Unlimited precision integer
 - Provides library of masking functions and primitive
- Module system supports extensibility and reuse
 - Add, modify, remove rules
 - Add, modify, remove types and attributes

Experimental Evaluation

- Questions
 - How expressive is Typical?
 - How concise is Typical?
 - How do generated checkers perform?
- Methodology
 - Write Typical type checkers for both Typical and C
 - Translate Typical code to Java (our bootstrap language)
 - Compare generated to handwritten type checkers
 - Lines of code (LoC)
 - Performance: speed, memory usage

Experimental Results: Typical Is Concise & Fast Enough

- Conciseness (C checker comparison)

Functionality	Java (NCSS)	Typical
AST Declaration	0	140
Type Declaration	3,480	50
Namespaces and Scope	0	60
AST analysis	2,060	540
List reductions	620	65
Helper functions	650	340
Total size	6,810	1,195

- C checker performance (on Linux kernel files)

Metric	Handwritten	Typical
Latency (s)	2.557	2.427
Memory Pressure (MB)	148.828	425.042

Conclusions

- Developed Typical
 - Type checker generator for ‘real world’ languages
 - Functional core plus declarative features
 - Type attributes, scoping rules, namespaces, type constraints, list processing
 - Module system for fined-grained extensibility
- Evaluated Typical
 - Expressivity: implemented checkers for C and Typical
 - Conciseness: clear improvement over handwritten (C)
 - Performance: comparable to hand written checker
- To be released at <http://cs.nyu.edu/rgrimm/xtc/>