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?

Grammar → AST Specification

Parser Generator → Bootstrap Compiler

Source → Parser

Front End

AST
Use a Domain-Specific Language!

Parser Generator

Grammar

AST Specification

Type Checker Generator

Type Rules

Bootstrap Compiler

Source

Parser

AST

Type Checker

Annotated AST

Front End
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

\[ \lambda \]

30 + 7 lines

\begin{align*}
\text{mltype node =} & \quad \text{begin match tl with} \\
| \text{Abstraction of node * node * node} & \quad | \text{FunctionT (param, res) ->} \\
| \text{Application of node * node} & \quad \text{require param = tr} \\
| \text{Identifier of string} & \quad \text{error "argument type mismatch" in} \\
| \text{IntegerConstant of string} & \quad \text{res} \\
| \text{FunctionType of node * node} & \quad \_ \_ \rightarrow \text{error "applying non-function"} \\
| \text{IntegerType} & \quad \text{end} \\
\text{mltype type = IntegerT} & \quad | \text{Abstraction (id, type, body) ->} \\
| \text{FunctionT of type * type;} & \quad \text{let param = analyze type in} \\
\text{scope Abstraction _ as lambda ->} & \quad \text{let \_ = define id param in} \\
\quad \text{Scope(Anonymous("lambda"), [lambda]) ;} & \quad \text{let res = analyze body in} \\
\quad \text{namespace default =} & \quad \text{FunctionT (param, res)} \\
\quad \text{Identifier (id) -> SimpleName(id) ;} & \quad | \text{Identifier _ as id -> lookup id} \\
\quad \text{mlvalue analyze = function} & \quad | \text{IntegerConstant _ -> IntegerT} \\
\quad | \text{Application (lambda, expr) ->} & \quad | \text{FunctionType (parameter, result) ->} \\
\quad \quad \text{let tl = analyze lambda} & \quad \text{FunctionT (analyze parameter,} \\
\quad \quad \text{and tr = analyze expr in} & \quad \quad \text{analyze result)} \\
\quad \quad \text{end} & \quad \text{IntegerType -> IntegerT ;}
\end{align*}
Let’s Make This Concrete: Simply Typed Lambda Calculus

Machine-generated
AST specification

mltype node =
| Abstraction of node * node * node
| Application of node * node
| Identifier of string
| IntegerConstant of string
| FunctionType of node * node
| IntegerType;

mltype ·
| Function

Type representation

Scope Abstraction _ as lambda ->
  Scope(Anonymous("lambda"), [lambda])
namespace default = Identifier (id) -> SimpleName(id);

mlvalue 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
| _ ->
  "application or function"
end

Typing rules

and

Error management

| Abstraction _ as id -> lookup id
| IntegerConstant _ -> IntegerT
| FunctionType (parameter, result) ->
  FunctionT (analyze parameter,
  analyze result)
| IntegerType -> IntegerT;
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

```plaintext
mltype node = Abstraction of node * node * node
| Application node * node   | Identifier of string |
| IntegerConstant of string | IntegerType;         [	extit{Rats!}-generated AST]
```
Expressing Language Types: Representation

- Represent types with variants
  
  ```mltype
type = IntegerT | FunctionT of type * type
```

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

  ```mltype
raw_type = IntegerT | PointerT of type ...;
mltype qualifier = Const | Restrict | Volatile;
mltype 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.
a. Type rule
Abstraction (id, type, body) \rightarrow expr;

b. Scope rule
scope Abstraction _ a \rightarrow 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

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`.

a. Type rule
   Abstraction (id, type, body) -> `expr`;

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

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

- **C checker performance (on Linux kernel files)**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Handwritten</th>
<th>Typical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (s)</td>
<td>2.557</td>
<td>2.427</td>
</tr>
<tr>
<td>Memory Pressure (MB)</td>
<td>148.828</td>
<td>425.042</td>
</tr>
</tbody>
</table>
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/