# Typical: Taking the Tedium Out of Typing

Robert Grimm  Laune Harris  Anh Le

Technical Report TR2007-904
New York University
{rgrimm,lharris,anhlevn}@cs.nyu.edu

## Abstract

The implementation of real-world type checkers requires a non-trivial engineering effort. The resulting code easily comprises thousands of lines, which increases the probability of software defects in a component critical to compiler correctness. To make type checkers easier to implement and extend, this paper presents Typical, a domain-specific language and compiler that directly and concisely captures the structure of type systems. Our language builds on the functional core for ML to represent syntax trees and types as variants and to traverse them with pattern matches. It then adds declarative constructs for common type checker concerns, such as scoping rules, namespaces, and constraints on types. It also integrates error checking and reporting with other constructs to promote comprehensive error management. We have validated our system with two real-world type checkers written in Typical, one for Typical itself and the other for C.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Translator writing systems and compiler generators; D.3.3 [*Programming Languages*]: Language Constructs and Features—Constraints; Modules, packages

***General Terms*** design, languages

***Keywords*** type checker generator, compiler-compiler, extensible type systems, extensible compiler, Typical

## 1. Introduction

The implementation of type checkers for real-world languages requires a non-trivial engineering effort. For example, gcc 4.1's type checker for C alone comprises 8,400 lines of rather dense C code. O'Caml 3.10's type checker and supporting code comprise 18,600 lines of O'Caml code. And OpenJDK b23's semantic analysis, which includes data-flow analysis, constant folding, and type erasure for generic types, comprises 15,100 lines of Java code and depends on another 8,200 lines for its types and symbol table. Clearly, the sheer size of these components increases the probability of bugs. At the same time, type checker correctness is critical for program safety. After all, if a type checker bug causes the compiler to accept malformed code, the resulting program may unexpectedly misbehave or crash—which is exactly what a language's static typing discipline seeks to prevent.

Attribute grammar systems such as Eli [17], the synthesizer generator [35], and more recently JastAdd [13], Ruler [10], Silver [37], and xoc [9], can help reduce the complexity of semantic analyzers. Their primary benefit is that attribute values are declaratively specified through equations, which eliminates the need for manually scheduling syntax tree traversals. At the same time, attribute grammars complicate the implementation of components familiar to compiler writers. In particular, the symbol table needs to be implemented through several interdependent attributes per namespace, and even auxiliary data structures need to be implemented through productions [12]. This paper explores a different trade-off: We observe that type systems are highly structured—whether they are specified in prose as in the C standard [23] or formally as in the ML standard [29]—and we present a language that directly and concisely captures this structure, even though it does require explicit syntax tree traversals.

Our language is called Typical and is designed to provide four main properties. First, it is *expressive*, building on the functional core of ML to represent syntax trees and types as variants, i.e., type-safe unions, and traversing them with pattern matches. Second, constructs new to Typical are *declarative*, capturing aspects of type systems, such as their namespaces and scoping rules, instead of describing how to implement them. Third, Typical is *prescriptive* in that error checking and reporting are integrated with other constructs to promote comprehensive error management. Fourth, it encourages *correct* code by being strongly typed and by relying on constructs that prevent mistakes in the first place. Beyond this core language, Typical includes a module system, which treats Typical programs as collections of rules and facilitates the introduction of new rules—as when extending a language—and supplemental rules—as when implementing pluggable type systems [1, 4].

We have validated our design by implementing two real-world type checkers in Typical. First, the type checker for the Typical compiler is itself written in Typical. In particular, it supports higher-order functions, parametric polymorphism, parameterized data types, and Hindley-Milner type inference. Furthermore, it seamlessly integrates with the rest of the compiler, which is written in Java and targets Java—though nothing in the design prevents other target languages. Second, we have implemented a type checker for C, which supports the C99 standard [23] with common gcc extensions. C99, in turn, is a significant extension of Kernighan and Ritchie C [24], resulting in a significant body of type rules covering the interaction of the two major versions. The C type checker passes all regression tests for the gcc 4.1 front-end and also type checks the entire Linux 2.6 kernel.

This paper's contributions are threefold:

1. A domain-specific language for implementing type checkers, which is largely functional yet seamlessly integrates with other, imperative compiler code;
2. An extension language for type checkers, which enables the introduction of new type rules as well as supplemental rules;
3. Experiences with two real-world type checkers written in Typical, one for Typical itself and the other for C.

Our evaluation compares the C type checker written in Typical with one written in Java and another written in O'Caml; it demonstrates that Typical enables the concise specification of type checkers, while also resulting in reasonable performance.

## 2. Point of Departure: xtc

The design of Typical is informed by our analysis of the xtc compiler toolkit [22], which is available as open source and written in Java. As in other compilers, semantic analyzers for xtc traverse a program's abstract syntax tree (AST), populating the symbol table and reporting any typing errors. They also annotate AST nodes with their scopes, so that repeated traversals can easily synchronize with the appropriate scopes, and annotate nodes with their types, so that typing information is readily available for all constructs and not just the identifiers contained in the symbol table.

In xtc, ASTs are implemented as dynamically typed trees, consisting of instances of a single class `GNode`, which provides a name and zero or more children. Tree traversals are implemented through dynamically dispatched visitors, which use Java reflection to select the closest matching visit methods. Types are represented by a class hierarchy, whose common interface is defined by the base class `Type`. Leafs correspond to distinct kinds of types, such as `IntegerT` and `PointerT`, and capture each kind's inherent properties, such as the integral kind for `IntegerT` or the pointed-to type for `PointerT`. Every type instance can also have annotations representing, for example, the source location, memory shape, or C qualifiers such as `const`. The hierarchy of types is supplemented by language-specific classes, whose methods directly capture the corresponding language standard's instructions such as "If both the operands have arithmetic type, the usual arithmetic conversions are performed" [23, § 6.5.8].

Overall, this design facilitates a relatively clean and stylized implementation of type checkers. In particular, it has enabled the scalable composition of separate type checkers for Java and C into the semantic analyzer for Jeannie, which combines both languages to replace Java's explicit foreign function interface [22]. At the same time, xtc also suffers from the following short-comings:

1. Dynamic typing of AST nodes and visitors provides considerable flexibility but also forgoes static program safety, making it hard to find bugs, especially when the AST changes.
2. Annotating nodes with their types and synchronizing tree traversal with the symbol table requires snippets of boiler-plate code throughout a type checker.
3. Many type operations require manual decomposition of type instances and their properties, which is tedious at best and error-prone at worst.
4. Type errors, which are represented as instances of class `ErrorT`, need to be explicitly threaded through a type checker to avoid cascading error conditions.
5. The type framework itself is rather large, comprising 2,600 non-commenting source statements, which makes it hard to introduce new types.

Other compilers, which also rely on visitors and symbol tables, have similar issues. The challenge addressed in this paper is how to encode best practices from xtc, while also overcoming its short-comings and seamlessly integrating with the rest of the compiler.

## 3. Overview of the Typical System

As illustrated in Figure 1, a compiler's front-end consists of a parser, which converts source code to an abstract syntax tree, and a type checker, which ensures that the AST is well-formed. The front-end's output is either an AST annotated with its types or an error notification. Neither parser nor type checker are written by hand. Rather, the parser's source code is generated from a grammar by a parser generator. And the type checker's source code is generated from type rules by our type checker generator, i.e., the Typical compiler. Both compiler-compilers share the description of the syntax tree, since the type checker consumes the AST produced by the parser. Both compiler-compilers also rely on the bootstrap compiler to convert source code into executable components. To put it differently, the Typical system encompasses three different programming languages and their compilers. The *source language* is the language whose compiler front-end is being implemented. Its type rules are written in the *meta-language*, i.e., Typical. The Typical compiler, in turn, generates source code in the *bootstrap language*, i.e., Java in our implementation, which is then compiled into executable code by the bootstrap compiler.

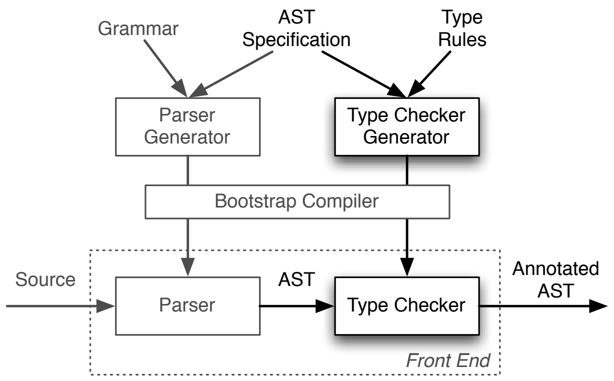

**Figure 1.** Conceptual overview of the Typical system.

| | | | |
|---|---|---|---|
| *expression* | ::= | \ *identifier* : *type-spec* . *expression* | ▷ Abstraction |
| | \| | *expression expression* | ▷ Application |
| | \| | *identifier* | |
| | \| | *integer-constant* | |
| | \| | ( *expression* ) | |
| *type-spec* | ::= | *type-spec* -> *type-spec* | ▷ Function type |
| | \| | `int` | |
| | \| | ( *type-spec* ) | |
| *identifier* | ::= | a..z {a..z} | |
| *integer-constant* | ::= | 1..9 {0..9} | |

**Figure 2.** Syntax of the simply typed $\lambda$-calculus with integers.

## 4. Example: Simply Typed Lambda Calculus

As an introduction to the Typical language, we now present the *complete* type checker for the simply typed lambda calculus with integers [8]. For this example, we treat the $\lambda$-calculus not as a formal system, but rather as a programming language whose front-end we seek to implement. Figure 2 shows the $\lambda$-calculus' syntax, with *expression* and *type-spec* specifying the hierarchical syntax, and literals, *identifier*, and *integer-constant* representing tokens. The corresponding grammar is written for the *Rats!* parser generator [19] and largely follows the syntactic specification shown in the figure. It differs by encoding that applications are left-associative and that function types are right-associative. Furthermore, it declares, inline, the $\lambda$-calculus' AST, which has distinct nodes for abstractions, applications, identifiers, integer constants, function types, and integer types.

Figure 3 shows the $\lambda$-calculus' entire type checker written in Typical. The specification first defines the AST representation through the `expression` and `type_spec` variants (lines 1–9). Next, it defines the type representation through the `raw_type` variant (11–12). It then defines the $\lambda$-calculus' scoping rules and namespaces through the new `scope` and `namespace` constructs (14–18). Finally, it defines the type rules and error checking through the

```
 1 mltype expression =
 2   | Abstraction of expression * type_spec * expression
 3   | Application of expression * expression
 4   | Identifier of string
 5   | IntegerConstant of string ;
 6
 7 mltype type_spec =
 8   | FunctionType of type_spec * type_spec
 9   | IntegerType ;
10
11 mltype raw_type =
12   IntegerT | FunctionT of type * type ;
13
14 scope Abstraction _ as lambda ->
15   Scope(Anonymous("lambda"), [lambda]) ;
16
17 namespace default : type =
18   Identifier (id) -> SimpleName(id) ;
19
20 mlvalue analyze = function
21   | Abstraction (id, decl, body) ->
22       let param = analyze_type_spec decl in
23       let _     = define id param in
24       let res   = analyze body in
25         { type = FunctionT (param, res) }
26   | Application (lambda, expr) ->
27       let tl = analyze lambda
28       and tr = analyze expr in
29         begin match tl.type, tr with
30           | FunctionT (param, res), param -> res
31           | _ -> error "mistyped application"
32         end
33   | Identifier _ as id ->
34       lookup id
35   | IntegerConstant _ ->
36       { type = IntegerT } ;
37
38 mlvalue analyze_type_spec = function
39   | FunctionType (param, res) ->
40       FunctionT (analyze_type_spec param,
41                  analyze_type_spec res)
42   | IntegerType ->
43       IntegerT ;
```

**Figure 3.** Complete Typical type checker for the simply typed λ-calculus with integers.

`analyze` and `analyze_type_spec` functions (20–43). As a matter of preference, Typical's syntax is closer to Caml than Standard ML, with Typical also supporting Caml's declared record types, "or" patterns, and pattern guards. Type and value declarations use the "`ml`" prefix to distinguish these meta-language constructs from the corresponding source language concepts.

In more detail, the declaration of AST nodes (1–9) need not be written by hand. Rather, we have modified the *Rats!* parser generator to automatically generate this strongly typed view on the underlying dynamically typed AST to facilitate sharing between parser and type checker [20]. The AST declaration organizes `xtc`'s generic nodes into semantic categories, i.e., variants, while also abstracting away auxiliary information such as a node' source location, which is not directly relevant for semantic analysis. That information is still exposed to and used by Typical's runtime, for example, to report error locations.

The declaration of the λ-calculus' types (11–12) is very simple, defining one constructor for integer types and one for function types. The `raw_type` definition only captures each type's inherent properties, while separate `attribute` definitions declare types' annotations. Typical automatically synthesizes a record type combining all these declarations. In the case of the λ-calculus, there are no attributes and, as illustrated on lines 25 and 36, the type record only has a `type` field for the raw type. Type comparisons default to structural type equivalence, yielding the desired behavior on line 30, which relies on the repeated `param` variable to compare a function's parameter type with the argument type. At the same time, type checkers can override this default to express other forms of type equivalence.

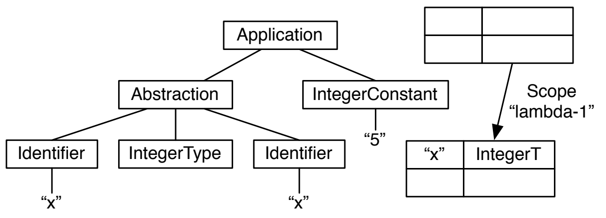

**Figure 4.** AST and symbol table for "`(\x:int.x) 5`".

The scope declaration (14–15) relies on a pattern match to map AST nodes to their scopes. For the λ-calculus, the declaration uses the built-in `Scope` and `Anonymous` constructors to specify that each abstraction node introduces an anonymous scope spanning the node and its children. Typical uses this declaration to (1) automatically create and enter the scope when first visiting an abstraction node, e.g., when entering the expression on lines 22–25, (2) exit the scope when returning from an abstraction node, e.g., when exiting the expression on lines 22–25, and (3) restore that scope when visiting the node again, which does not happen in the example.

The namespace declaration (17–18) also relies on a pattern match, this time to map AST nodes to their namespaces and names for symbol table operations. For the λ-calculus, the declaration uses the built-in `SimpleName` constructor to specify that each identifier node maps to the default namespace and the node's string as the name. Typical uses this declaration for the symbol table accesses in lines 23 and 34; it also uses the type name `type` to strongly type these operations. Figure 4 illustrates the effect of the scope and namespace declarations by showing the AST and symbol table after processing "`(\x:int.x) 5`".

The `analyze` function (20–36) and `analyze_type_spec` function (38–43) implement the λ-calculus' type rules, with `analyze` processing `expression` variants and `analyze_type_spec` processing `type_spec` variants. Taken together, the two functions traverse the AST and map λ-calculus constructs to their types. They also enforce two constraints, namely that the parameter type of a function matches the argument type for an application (29–32) and, implicitly, that the identifier is defined on a symbol table lookup (34). The former specifies an explicit error message, while the latter relies on the default message provided by Typical. In both cases, the reported error location is the source location of the dynamically closest matching AST node, i.e., an application node in the former case and an identifier node in the latter case. On an error, both expressions evaluate to `bottom`. Comparable to Java's `null`, this value is automatically injected into every Typical type, but, unlike `null`, automatically threaded through a Typical program to avoid cascading error messages.

## 5. The Typical Language

The main goal for Typical is to make type checkers easier to implement and extend. In the process of developing our system, we have distilled the following design guidelines, which help Typical meet its goal:

1. *Be expressive*. Typical should incorporate constructs that directly capture common type checker idioms. At the same time,

it should include sufficient general purpose constructs to facilitate the implementation of uncommon operations.

2. *Be declarative*. Typical constructs should be declarative. Where possible, they should directly express an aspect of the source language's type system instead of providing a way to implement that aspect in code.
3. *Be prescriptive*. In general, Typical constructs should cleanly compose with each other. However, error detection and reporting should be integrated with other constructs and not require dedicated constructs.
4. *Be correct*. Typical constructs should help ensure that the implementation of a source language's type system is correct. At the least, Typical constructs must be statically typed. Preferably, they prevent incorrect implementation of (some) aspects of a type system altogether.

The first guideline directly reflects Typical's goal of simplifying type checker implementation; though it also acknowledges that Typical cannot possibly include constructs for *all* possible type systems. The second guideline is based on the observation that, when compared to code implementing behaviors, declarative constructs tend to be more concise, more flexible in their implementation, and more easily implemented correctly. For example, Typical's scope declarations precisely specify all of a source language's scoping rules in one location, do not depend on a particular symbol table implementation, and ensure that enter-scope and exit-scope operations are always well-matched.

The third guideline is based on the observation that real-world type checkers are not just boolean oracles that determine whether programs are well-typed. Rather, they need to provide developers with meaningful feedback on program errors. As a result, the implementation of type rules is often cluttered by the corresponding error management. Seminal [26] addresses this challenge by *decoupling* type checking from error reporting for ML. In contrast, we believe that *combining* error checking with other Typical constructs encourages a proper error discipline and thus provides a more general solution.

Finally, the fourth guideline is based on the observation that type checkers are critical for program safety. A type checker bug can easily subvert a source language's typing discipline and thus result in misbehavior or crashes of compiled programs. Consequently, program correctness is of particular importance for semantic analyzers. For the purposes of this paper, we focus on Typical as a strongly typed language for implementing real-world type checkers. In future work, we will formalize our language through a translation to ML and then leverage this foundation to automatically reason about the correctness of type checker implementations.

### 5.1 Core Language

With the design guidelines in place, we now turn to the actual design of our language. We initially explored building on a Java-like core, but settled on the functional core of ML for three reasons. First, variants provide a concise representation for syntax trees and types alike. Second, pattern matches enable a concise implementation of visitors, while also enabling the flexible decomposition of data structures. Third, ML is strongly typed and has a well-defined and well-understood semantics [29]. In other words, ML provides us with an expressive, declarative, and correct foundation; it also addresses concerns 1, 3, and 5 raised in Section 2.

Consistent with the organization of language specifications around their constructs, Typical models type checkers as mappings from AST nodes to their types. Since the AST is usually represented by more than one variant, this mapping is implemented by several functions. The main entry point is the `analyze` function, which takes the variant representing the AST's root its argument and returns the meta-language type `type`. The `analyze` function, in turn, invokes other functions, such as `analyze_type_spec` for the $\lambda$-calculus' type checker, to visit all AST nodes while also mapping each node to its type. To make typing information readily available to later compiler phases, Typical automatically annotates nodes with their types. It identifies relevant functions, i.e., those mapping nodes to types, by determining the transitive closure of variant types reachable from the AST's root type. Borrowing from Haskell's notation for type classes [32], an arbitrary variant representing an AST node can be written as "`Node 'a`".

### 5.2 Abstract Syntax Tree Nodes

A source language's abstract syntax tree is shared between the parser, type checker, and back-end. Consequently, the main challenge for representing ASTs in Typical is to reconcile two competing pressures: ASTs must be seamlessly shared with other compiler phases, while also integrating with our language's functional core. Typical meets this challenge through two techniques. First, the in-Typical representation of AST nodes abstracts away auxiliary information that is not directly relevant for semantic analysis. In particular, it hides a node's source location, and all error reporting is performed relative to AST nodes. Furthermore, it hides nodes' type annotations, which are automatically added during semantic analysis. Of course, that information is still accessible to and used by Typical's runtime. Second, while the AST declaration may be written by hand, it is usually created by the parser or AST generator. As already mentioned in Section 4, we have modified the *Rats!* parser generator to interface with Typical. The corresponding code analyzes a grammar's inline AST declarations, sorts `xtc`'s generic nodes into semantic categories, i.e., variants, and infers consistent types for each node's children [20]. A similar approach can be used for other parser generators, such as ANTLR [33] or SDF2 [38], that also rely on dynamically typed ASTs. At the same time, inferring a statically typed view is unnecessary for parser generators, such as Elkhound [28], that already model ASTs after ML's variant types.

### 5.3 Source Language Types

The representation of a source language's types is central to its semantic analysis. It needs to capture (1) the types themselves, (2) their attributes, and (3) their equivalence and subtyping relationships. Furthermore, attributes may have fixed semantics, such as qualifiers for C or visibility modifiers for Java. They may also be user-defined, capturing a particular application domain's restricted semantics. Finally, not all languages require attributes; Typical, just like ML, has none. Our language meets these needs by representing a source language's types through a variant—thus facilitating easy case analysis through pattern matches—and by relying on separate attribute declarations—thus allowing for easy extensibility. It also supports overridable `=` and `<:` operators for expressing equivalence and subtyping relationships.

A source language's type representation is expressed through the meta-language type `raw_type`. For example, the definition for C's types includes the following constructor definitions:

```
mltype raw_type =
  VoidT | IntT | PointerT of type | ... ;
```

C's storage class and qualifiers are declared separately, using Typical's `attribute` and `eqattribute` constructs. For example, the following type and attribute represent C's storage class; the `typedef` specifier is treated as a storage class, since it introduces a type alias and thus has *no* storage:

```
mltype storage_class =
  Auto | Extern | Register | Static | Typedef ;
attribute storage : storage_class ;
```

Typical automatically collects the raw type and its attributes into an ML record. In the case of C, it reads:

```
mltype type =
  { type: raw_type; storage: storage_class; ... } ;
```

Typical uses a record to represent attributed types, because ML's record patterns and "`{ ... with ... }`" construct for creating a record relative to another record need not specify all fields. As a result, Typical's attribute declarations combine with ML's support for record types to allow for the seamless introduction of new attributes. Of course, for language without attributes, the compiler optimizes the type record away again.

To express type equality and subtyping relationships, Typical relies on the polymorphic = and `<:` operators. By default, both operators test for structural equality; though they do ignore attributes introduced by `attribute` as opposed to `eqattribute`. Typical's new `equality` construct is used to override ='s behavior for variant types; its (non-exhaustive) pattern specifies which constructor arguments to compare through either a variable or wildcard pattern. For example, consider the following definition of C's structure type:

```
mltype raw_type = ...
  | StructT of int * string * type list ;
```

The integer argument is a nonce that distinguishes between structures with the same name and members but declared in different scopes, which are distinct types in C. The corresponding equality declaration reads:

```
equality raw_type = ... | StructT (n, _, _) ;
```

Taking a cue from Haskell's type classes [32], the = operator can be further customized through a function definition that explicitly specifies the non-polymorphic signature. The above declaration is equivalent to the following function definition:

```
mlvalue (=) : raw_type -> raw_type -> bool =
  fun t1 t2 -> match t1, t2 with ...
    | StructT (n1,_,_), StructT (n2,_,_) -> n1 = n2
    | _ -> false ;
```

The `<:` operator can be similarly customized; though we have not yet made use of this feature.

## 5.4 Name Management

Name management is another issue central to type checker implementation. It generally requires an environment or symbol table abstraction, which maps names to their types while also capturing a source language's scopes and namespaces. Furthermore, when using visitors, it requires that the current scope be coordinated with the AST traversal. Of course, a source language's scopes limit the static visibility of identifiers. Furthermore, namespaces assign different meaning to the same name in the same scope, with the type checker disambiguating names based on enclosing construct. For example, C's functions contain the visibility of locally declared variables, while namespaces support the use of the same name as a variable and a `goto` label.

Typical meets the requirements for name management by providing (1) a symbol table that is implicitly available to all programs, (2) the `scope` construct to declare which AST nodes correspond to which scopes, and (3) the `namespace` construct to declare individual namespaces, the types of their values, and which AST nodes correspond to which names. Both the `scope` and `namespace` constructs concisely capture aspects of a type checker. Consequently, they eliminate the need for boiler-plate code sprinkled throughout the type checker, which addresses concern 2 raised in Section 2. Both also are implemented by "weaving" the corresponding code into the type checker, rewriting the right-hand sides of pattern matches on nodes for the `scope` construct and rewriting symbol table operations for the `namespace` construct.

Having said that, we first need to settle on an appropriate data structure for the implicit symbol table. A persistent, i.e., functional, data structure would fit well with Typical's functional core and could be supported by rewriting functions during translation to pass an explicit symbol table argument. However, correctly managing that extra argument is difficult for type checkers that make several passes over the AST. For example, C's `goto` statements may forward reference labels. Correctly checking them requires two passes: the first to collect all defined labels and the second to verify that `goto` labels are, in fact, defined. To cleanly support multiple passes over a program's AST, Typical relies on an imperative, block-structured symbol table. In fact, the symbol table provides the only mutable state available to Typical programs.

The `scope` construct illustrated on lines 14–15 in Figure 3 (non-exhaustively) maps AST nodes to scopes; each right-hand side must evaluate to a built-in `Scope` value:

```
mltype scope_kind =
  | Named of name
  | Anonymous of string
  | Temporary of string ;
mltype scope =
  | Scope of scope_kind * (Node '?) list ;
```

The `'?` serves as a wildcard, indicating that any variant representing a node may appear in the list. Each scope spans the specified list of nodes, is automatically created when first visiting a node on the list, and automatically restored when visiting any of the nodes. Furthermore, a scope can be one of three kinds. First, a *named* scope is introduced by a named function or class. Second, an *anonymous* scope is introduced by a block or compound statement. Third, a *temporary* scope, unlike the other two kinds, is deleted after the visitor has left the scope's AST nodes. It is useful for typing C or C++ function prototypes, which may use parameter names different from the corresponding definitions. Named scopes assume the specified name, while anonymous and temporary scopes receive a fresh name based on the specified string.

To automatically synchronize scopes and visitors, Typical places one restriction on pattern matches on AST nodes in general. For the current scope to be well-defined, each expression in a Typical program must be dynamically "dominated" by a unique node. To ensure this condition, pattern matches on nodes may be arbitrarily complex, but may only use variable or alias patterns for arguments of a single constructor. That node then dominates the right-hand side and thus determines the current scope. With this restriction in place, Typical's runtime can trace the dynamic traversal path from an AST's root to the closest matching node as a stack. When it first pushes a node onto the stack, it evaluates the `scope` construct's pattern match against the node, annotates the node with the result, and, if necessary, updates the current scope. If the node already has a scope annotation, it simply sets that scope, thus always synchronizing scopes and visitors.

The `namespace` construct illustrated on lines 17–18 in Figure 3 introduces a new namespace while also specifying the type of its values and the (non-exhaustive) pattern match mapping AST nodes to names. Since the symbol table is mutable, all values stored in a given namespace must observe ML's value restriction [15, 39] and cannot be polymorphic. Otherwise, the type is unrestricted, e.g., need not be the meta-language type `type`. Using the following built-in type, names can either be simple, omitting the scope, or qualified, explicitly specifying a scope:

```
mltype name =
  | SimpleName of string
  | QualifiedName of string list ;
```

Typical provides the obvious symbol table operations, including define, is_defined, and lookup. To denote the namespace and name, these operations include a namespace tag and node; though, as illustrated for define on line 23 and for lookup on line 34 in Figure 3, the tag may be omitted for the default namespace. Typical relies on the namespace declaration's type to statically type symbol table operations. It relies on the declaration's pattern to dynamically map the node to a symbol table name and then performs the actual operation.

## 5.5 Error Management

After discussing Typical's facilities for expressing a source language's abstract syntax tree, types, and name management, we now turn to error management. The main challenge is how to detect and report failures, while also avoiding cascading error messages. Developers prefer to be notified of as many error conditions as possible rather than receiving just one notification per compiler invocation—as long as those error messages reflect actual program errors. Consequently, a type checker needs to continue processing the AST even after encountering an error, ignore error conditions caused by other errors, and report error conditions unrelated to other errors. Typical addresses this challenge through a so-called "no-information" monad and through constructs that integrate error detection and reporting, notably require to express boolean constraints and guard to express arbitrary constraints.

Many type checkers prevent cascading error messages by explicitly threading an error type, such as xtc's ErrorT, through their code. This is tedious to write and clutters the implementation. In contrast, Typical provides a system-wide no information monad, which automatically threads a bottom value through the type checker. In particular, all types are automatically injected into the monad and include bottom as a value. Built-in constructs and primitives generally return bottom if any of their arguments are bottom. Furthermore, all pattern matches return bottom due to an implicit clause mapping bottom to itself; though this default can be overridden by explicitly matching bottom. At the same time, type constructors, such as those for tuples, lists, and variants, treat bottom just like any other value. It allows for marking, say, a type attribute as having no information without forcing the entire type record to bottom. Finally, the is_bottom primitive detects bottom values, since the = and != operators yield bottom when either operand is bottom.

By representing errors as bottom and injecting (almost) all operations into the corresponding monad, Typical programs can avoid cascading error messages without notational overhead. To actually detect and report error conditions, they use require and guard. The require construct enforces one or more boolean constraints on an expression. For example, the $\lambda$-calculus' type checker could be improved by replacing the implicit type comparison on line 30 of Figure 3 with an explicit requirement:

```
require param = arg error "mistyped argument" in res
```

If the constraints, here "param = arg", evaluate to true, require evaluates the expression, here "res", and returns the corresponding value. If the constraints evaluate to false, require reports an error, here "mistyped argument", and returns bottom. Unless specified explicitly, the reported source location defaults to the location of the closest matching AST node. Finally, if the constraints evaluate to bottom, require silently returns bottom—thus avoiding cascading error messages.

The guard construct enforces arbitrary constraints by protecting against bottom values. For example, consider this snippet from Typical's own type checker, which unifies the types of a cons expression:

```
guard unify th tt error "type mismatch for cons"
```

```
reduce lst to singleton "type" with
  | { Long, Double, Complex } -> LongDoubleComplexT
  | { Double, Complex } -> DoubleComplexT
  | { Float, Complex } -> FloatComplexT
  | { Long, Double } -> LongDoubleT
  | { Double } -> DoubleT
  | { Float } -> FloatT
  | ... (* Similarly for integer types. *)
  | { Void } -> VoidT
  | { TypedefName _ as node } ->
      analyze_derived_type_spec node ref_is_decl
  | ... (* Similarly for enum, struct, and union specifiers. *)
```

**Figure 5.** C's constraints on type specifiers in Typical. The ref_is_decl flag indicates whether a reference to a structure or union type also is a declaration.

If none of the expression's free variables, here unify, th, and tt, are bottom, guard evaluates the expression. If the result is bottom, guard reports the specified error; otherwise, it just returns the expression's value. However, if either of the expression's free variables is bottom, guard silently returns bottom as well—again to avoid cascading error messages. By delegating error checking and reporting to guard, invocations of the unify function can be chained to unify more than two types but without resulting in duplicate error messages.

In addition to these primary error management constructs, Typical also integrates error checking and reporting with symbol table operations. In particular, define verifies that a name has *not* been defined, and lookup verifies that a name has *indeed* been defined. (redefine avoids the former check.) Furthermore, as illustrated on line 31 in Figure 3, Typical includes stand-alone error and warning constructs.

Overall, Typical's integration of the no-information monad with its error management constructs removes most of the clutter of explicitly threading error values through a type checker. As a result, the no-information monad cleanly addresses concern 4 raised in Section 2. Compared to Ramsey's extended error monad [34], it is also simpler, since it does not require exceptions, and more flexible, since it can also thread other conditions through a type checker. For example, our C type checker relies on the no-information monad to thread compile-time constant values through its code, without explicitly checking that an expression's operands are indeed compile-time constant.

## 5.6 List Reductions

C-like languages rely on lists of so-called specifiers or modifiers to express the properties of declarations, including types and their attributes. In fact, both C# and Java support the expression of arbitrary, user-defined attributes. In processing such lists, a type checker needs to (1) map AST nodes to the corresponding internal representation, (2) enforce semantic constraints on the number and combination of specifiers, and (3) provide comprehensive error reporting. Lists of specifiers are not only a common source language idiom, the state machine implementation necessary to enforce their semantic constraints can also be relatively complicated, easily leading to type checker bugs. Therefore, Typical directly addresses the three needs through the reduce construct.

The reduce construct is illustrated in Figure 5 on the example of C's type specifiers. It specifies the list to be reduced, here "lst", the constraints on the result, here "singleton", a string describing the result, here "type", and a (non-exhaustive) pattern match. Essentially, reduce repeatedly applies the pattern match to the list's elements until no more clauses trigger. It uses an extension of ML's pattern syntax, the set pattern "`{...}`", to indicate that a pattern's elements may appear in any order. Alternatively, a list pattern indicates that the elements must appear in the specified order. In either case, list elements that do not match any pattern elements are ignored, even if they appear between matching pattern elements. For each triggered clause, `reduce` evaluates the right-hand side, collecting the result.

While mapping the pattern match over the list, `reduce` also enforces the constraints. The `singleton` constraint in the example indicates that the pattern match may be triggered at most once, while `set` and `list` constraints allow for multiple triggers, with duplicates being ignored for `set` constraints. A further `optional` constraint specifies that the pattern match need not match any elements. Finally, like `require` and `guard`, `reduce` integrates error checking and reporting, deriving any message from the string describing the result. In the example, if the list is "`[Double, Double]`", `reduce` reports a "duplicate type" and returns `bottom`.

The design of the `reduce` construct reflects our analysis of how to type check C specifiers or Java modifiers and supports a generalization of the corresponding requirements. In particular, a list of C specifiers must include a single type specifier such as `int`, reducing the list to a "`singleton`". It may optionally include one storage class specifier such as `register`, reducing to an "`optional singleton`". It may also include one or more qualifiers such as `const`, which may be duplicated and thus reduce to an "`optional set`". As demonstrated here, the `reduce` construct cleanly captures C's rather convoluted specifier semantics, while also hiding the corresponding state machine implementation. As an added benefit, the pattern match in Figure 5 directly mirrors the listing in § 6.7.2 of the C99 standard [23]. As a result, we believe that `reduce` represents an appropriate combination of expressivity, prescriptiveness, and correctness.

### 5.7 Summary

Typical addresses the five concerns raised in Section 2 as follows. (1) To statically type AST nodes and visitors, it implements nodes as variant types and visitors as pattern matches. (2) To reduce the need for boiler-plate code, it automatically annotates nodes with their types and synchronizes the current symbol table scope with the AST traversal. (3) To simplify the decomposition of types and their properties, it also relies on pattern matches. (4) To avoid manually threading error values through programs, it provides a no-information monad that is also integrated with error detection and reporting. (5) To reduce the complexity of the type representation, it represents types through a variant and separately declared attributes. Additionally, it relies on list reductions to simplify specifier processing for C-like languages. The overall result is a language designed to be expressive, declarative, prescriptive, and correct.

## 6. Module System and Extensibility

To enable the reuse of type checker specifications, Typical provides a module system specifically tailored to the domain of semantic analysis; though our compiler does not yet implement it. Similar to other module systems, our module system relies on module declarations to group related Typical definitions, on import declarations to explicitly track dependencies, and on parameter declarations to delay the specification of actual dependencies, thus facilitating flexible composition. Parameters represent module names and, on instantiation, are replaced throughout a module with the actual arguments. Type, constructor, field, and value names are all relative to the defining module and may be written in qualified form (i.e., by using dots between name components). If a name is not qualified, it must be defined in either the same module or a single imported module.

```
mlvalue analyze_expression <<
  MultiplicativeExpression (n1, "*", n2) ->
    match t1, t2 with
      | { nonzero = true }, { nonzero = true } ->
          { result with nonzero = true }
      | _ -> result ;
mlvalue analyze_expression <<
  MultiplicativeExpression (n1, "/", n2) ->
    match t1, t2 with
      | { nonzero = true }, { nonzero = true } ->
          { result with nonzero = true }
      | _, { nonzero = true } ->
          { result with nonzero = false }
      | _ -> error "Potential division by zero" ;
```

**Figure 6.** Two example rules for adding a `nonzero` qualifier to C.

While helpful, this basic design is not sufficiently expressive to facilitate the implementation of language extensions. For example, consider adding structure inheritance to C to reduce the need for unsafe casts. Implementing this new language feature requires updating the grammar and AST. It also requires changing the type checker by updating the AST node for structures, the representation of structure types, and the analysis code for structure definitions and type compatibility. In contrast, adding a `nonzero` qualifier to C does not require modifying the representation of nodes or types, but it does require adding a new type attribute and supplemental analysis code.

To provide fine-grained extensibility for type checkers, we take a cue from extensible syntax, which supports the rewriting of a grammar's productions by adding, overriding, and removing individual alternatives [6, 19]. Analogously, Typical does not distinguish between a module's external signature and its internal implementation and treats type, equality, scope, namespace, and function definitions as collections of rules that can be directly modified. In particular, Typical's module modifications support four rewriting operators:

- `+=` to add new rules;
- `<<` to add supplemental rules to functions;
- `:=` to override existing rules;
- `-=` to remove existing rules.

The rule to be modified is either a constructor declaration for types or a pattern matching clause for all other definitions. It is identified, respectively, by either the constructor name or by the original rule's pattern; though any subpatterns not necessary for uniquely identifying a clause may be omitted. Additionally, when modifying functions, several patterns may be combined with the `->` operator to identify a clause in an arbitrarily nested match expression.

Figure 6 illustrates module modifications by showing two rewrite rules for adding a `nonzero` qualifier to C; the qualifier enables the static detection of division by zero errors [7]. The two rules depend on the declaration of a new type attribute:

```
eqattribute nonzero : bool ;
```

The module implementing the `nonzero` qualifier simply includes this declaration.

To combine a supplemental rule with an existing rule, Typical hoists the existing rule's outermost bindings while also creating a new binding for that rule's result. In particular, consider the right-hand side of an existing rule:

`let` *bindings* `in` *expr*

Typical combines this expression with the supplemental rule's right-hand side $expr_{sup}$ as following:

```
let bindings in let result = expr in expr_sup
```

Consequently, the code in Figure 6 can access the original rule's bindings for `t1` and `t2` on lines 3 and 9, while also accessing the original rule's result on lines 5, 6, 11, and 13 under the well-known variable name `result`.

Adding new rules (instead of supplemental ones) to a pattern match raises a different issue: the new clause must be added at an appropriate position, since pattern matches are ordered. To automatically determine that position, Typical leverages ML's irredundancy analysis. Conceptually, module resolution tries out every position in a pattern match until it finds a position that does not cause an irredundancy conflict. This search starts from the top to give priority to newly added clauses. In practice, we expect the Typical compiler to try only few positions, since most pattern match clauses match distinct variant constructors.

Comparable to C++ templates, the design of Typical's module system trades off flexibility and extensibility against modular type checking. In particular, since module modifications specify how to rewrite other modules, they are necessarily incomplete, and the Typical compiler can fully type check a module only after all modifications have been applied. We believe this trade-off to be acceptable for two reasons. First, Typical programs implement only type checkers and are relatively small, covering a few thousand lines of code at most and thus lessening the need for modular type checking. Second, the Typical compiler can still perform consistency checks before applying module modifications, ensuring, for example, that all modifications apply to existing definitions. It can also incrementally type check a program, as long as a module's parameters have been resolved and all modifications have been applied. As we gain more experience with Typical, we expect to revisit this trade-off. Finally, note that Typical programs as a whole are always strongly typed.

## 7. Implementation

Our Typical compiler is implemented as a source-to-source translator from Typical to Java. It is written in Java, with exception of the type checker, which is written in Typical—thus demonstrating our system's seamless integration with other, imperative compiler code. While the compiler supports the complete language, it does not yet implement the module system.

The mapping from Typical to Java represents variant types as collections of classes, with one superclass for each variant and one subclass for each constructor. Each subclass provides fields and accessors for the constructor's arguments. It also includes methods to easily test for a particular constructor in translated pattern matches. For example, the class implementing the `PointerT` constructor has the following two methods:

```
public boolean isPointerT() { return true; }
public Tag tag() { return Tag.PointerT; }
```

Both methods are also declared by the superclass, with the implementation of `isPointerT()` returning `false` and `tag()` being abstract. The tag is a type-safe Java `enum` over each variant's constructors. Where possible, it is used to optimize the implementation of pattern matches through `switch` statements.

Next, the mapping represents record types as classes with the corresponding fields and accessors. For type checkers without attributes, the compiler optimizes the type record away again, replacing all uses with the raw type. Our mapping implements lists through a utility class, Typical strings as Java strings, and boolean and numerical types through the corresponding boxed Java classes. Typical's `int` maps to `BigInteger` to enable unlimited precision arithmetic when tracking compile-time constant values. Finally, the mapping represents `bottom` as `null`.

A Typical program's top-level values are mapped to (immutable) fields of the type checker's main class. Closures, i.e., functions and `let` expressions, are translated to anonymous inner classes that implement interfaces representing their "uncurryed" signatures. As an optimization, the compiler folds directly nested let expressions whose bindings do not overlap into a single closure. The `reduce` construct is translated to the corresponding state machine, which tracks matches on individual element patterns. Finally, the implementation of the `scope` and `namespace` constructs largely follows the description in Section 5.4, with the compiler first creating ML functions based on the declarations' pattern matches and then translating the resulting functions. It also injects calls to these functions into pattern matches on AST nodes and symbol table operations.

While largely straight-forward, our mapping has one shortcoming: it preserves tail-recursive function invocations and relies on the Java compiler or virtual machine to perform tail call elimination. Since most Java implementations do not perform this optimization, we (somewhat) mitigate this limitation by implementing Typical's list library in Java, using iterative algorithms where possible.

## 8. Evaluation

In evaluating Typical, we consider three criteria: its expressiveness, conciseness, and performance. To this end, we have implemented the semantic analyzer for Typical in Typical itself. The type checker supports higher-order functions, parametric polymorphism, parameterized data types, and Hindley-Milner type inference. It also verifies that pattern matches are exhaustive and irredundant, i.e., cover all cases without duplicate clauses. Finally, it automatically detects mutually recursive functions, thus eliminating the need for explicit "`let rec`" declarations. We have also implemented a semantic analyzer for C99 [23] with common gcc extensions. C99, in turn, is a significant extension of Kernighan and Ritchie C [24]. The C type checker passes all regression tests for the gcc 4.1 front-end and also processes the entire Linux 2.6 kernel.

Taken together, the two type checkers demonstrate that our language is sufficiently expressive to capture the static semantics of substantially different programming languages, equally supporting a conventional monomorphic type system and an inference-based polymorphic type system. However, we are not quite satisfied with Typical's support for type unification. In particular, the current implementation uses an explicit hashtable to map type variables to types when performing unification. We believe that its performance is close to optimal, since experiments with a fast union/find implementation have not resulted in performance improvements while also increasing memory utilization. At the same time, we would like to hide the hashtable behind a more declarative interface and will explore better language support for unification as future work.

To evaluate conciseness and performance, we compare three type checkers for C:

- Our own type checker, which is written in Typical;
- xtc's type checker, which is written in Java;
- A third, derived, type checker, which is written in O'Caml.

The Typical and xtc versions use the same parser, AST representation, and symbol table implementation. Since the two versions process the same inputs and use the same libraries, any difference in performance reflects the difference between handwritten Java and Typical code. We derived the O'Caml version from the Typical version, using CIL's parser and complete AST representation for C [30]. Since Typical is based on ML, any difference in conciseness reflects Typical's improvements over the base language.

Table 1 shows the code size breakdown for the C type checkers, including the size of the Typical-generated Java code. It measures code size in lines of code (LoC) for the Typical and O'Caml

| Functionality | Typical (LoC) | Compiled (NCSS) | Java (NCSS) | O'Caml (LoC) |
|---|---|---|---|---|
| AST declaration | 250 | [960] | [960] | 200 |
| Type declaration | 50 | 810 | 1,810 | 50 |
| Type operations | 350 | 1,660 | 920 | 400 |
| Symbol table | [0] | [310] | [310] | [100] |
| Scopes, namespaces | 70 | 310 | 0 | 0 |
| Specifiers | 170 | 400 | 670 | 210 |
| Compound init. | 140 | 430 | 320 | 170 |
| Rest of AST analysis | 1,300 | 4,900 | 2,660 | 1,970 |
| Total size | 2,330 | 8,510 | 6,380 | 3,000 |

**Table 1.** Code size breakdown for the C type checkers. Bracketed numbers reflect library code not counting towards the totals; Typical's runtime comprises another 1,680 NCSS.

| Metric | Typical | Java | O'Caml |
|---|---|---|---|
| Latency (s) | 1275.8 | 1080.9 (0.85×) | 169.6 (0.13×) |
| Mem. pressure (MB) | 5512.7 | 2116.9 (0.38×) | 6068.9 (1.10×) |

**Table 2.** Performance of type checking the Linux kernel.

versions, while using non-commenting source statements (NCSS) for Java. NCSS are roughly equivalent to counting semicolons and open braces; they are a more conservative metric that avoids skewing results in the presence of many small methods and their documentation.

The results demonstrate that Typical enables the concise specification of real-world type checkers. Compared to the Java version, the Typical version is 2.7 times shorter, with the variant for the raw type, pattern matching in type operations, and list reductions for specifier processing having the biggest impact. Compared to the O'Caml version, the Typical version still is 1.3 times shorter, or almost 700 lines of code. Compared to both, the Typical version also is more robust. It is statically typed, unlike the Java version, and avoids boiler-plate code for synchronizing the symbol table and for threading error as well as compile-time constant values through the type checker, unlike the Java and O'Caml versions. At the same time, the results demonstrate the effectiveness of xtc's type representation. Even though the class hierarchy and type operations themselves require significant code, they do facilitate a relatively concise coding style, with the main analyzer being only 2 times larger than the Typical version and 1.4 times larger than the O'Caml version.

Table 2 compares the performance of the C type checkers for the entire Linux 2.6 kernel. Latency measures the time for type checking each file, excluding parsing. Memory pressure measures the difference in available memory before and after type checking each file. We forced GC before type checking, and there was no GC during type checking. Reported numbers are the sums of the averages of the last three out of four runs for each file. All measurements were performed with Sun's 1.5.0_11 JVM and O'Caml 3.10 generated native code on an off-the-shelf PC running Linux.

The results show that the performance of the Typical-generated version is competitive with that of the handwritten one. The latency of the Typical version is only 1.18 times larger than that of the Java version; though its memory pressure is 2.60 times larger. Profiling of the Typical version shows that a significant fraction of the execution time is spent in translated list reductions and pattern matches on nodes. We will thus focus future tuning efforts on the memory overhead of closures and the latency of list reductions and pattern matches. By comparison, the O'Caml version runs significantly faster, by a factor of 7.52 when compared to the Typical version. It demonstrates the benefits of native code, as opposed to a virtual machine, and of aggressive optimizations, as opposed to a relatively straight-forward mapping of ML to Java. Though just like the Typical version, it has higher memory pressure than the Java version.

## 9. Related Work

Several efforts explore how to add user-defined type constraints to C and Java. In particular, CQual [14] and JQual [18] extend the type systems of C and Java respectively with support for user-defined qualifiers based on subtyping relationships. Next, Clarity [7] provides a more expressive language, whose rules can be automatically validated with an automatic theorem prover. Finally, JavaCOP [1] further increases expressivity, supporting a rich, rule-based language for implementing user-defined type constraints. CQual, JQual, Clarity, and JavaCOP focus on making a *particular* language's type checker extensible, while our work focuses on making type checkers *in general* easier to implement and extend. As a result, the four systems feature a tighter integration with a particular language's compiler, while our system is more general.

Meta-programming systems take a different approach and leverage a single formalism for specifying all aspects of processing another language, including parsing, type checking, interpretation, or translation [21]. For instance, PSG [2] relies on denotational semantics, while CENTAUR [3] relies on natural semantics. Both ASF+SDF [36] and Stratego/XT [5] rely on the rewriting of algebraic expressions. Finally, Eli [17], JastAdd [13], Ruler [10], Silver [37], the synthesizer generator [35], and xoc [9] build on attribute grammars, with Ruler focusing on type checkers and xoc focusing on extensions to C. Compared to our work, meta-programming systems are more ambitious but also suffer from a lack of orthogonality: they work best when entire compilers are implemented with them. Furthermore, they require that compiler writers map familiar concepts, such as scopes and namespaces, onto the underlying formalism, such as attributes or rewrite rules. In contrast, Typical is designed to directly capture such concepts and to seamlessly integrate with other compiler code.

Nominally, the TCG system [16] comes closest to our work by exploring how to structure a type checker generator. To this end, TCG introduces its own language and formalism, which has well-defined operational semantics. Treating type derivation as a proof, TCG then performs a symbolic, backward proof search to determine whether a program is well-typed. While formally rigorous and expressive, TCG suffers from two limitations. First, to keep the proof search tractable, TCG needs explicit annotations, thus requiring that compiler writers understand not only the formalism but also the proof search system. Second, TCG has limited support for error management. While it does include support for visualizing the proof search, it lacks facilities for reporting user-friendly type errors and for recovering from cascading error conditions, both of which are critical for real-world compilers.

To achieve fine-grained extensibility, Typical's module system supports the modification of variant declarations and pattern matches through rule rewriting. The idea of treating declarations, i.e., productions, in a compiler-compiler's specification as modifiable clauses, i.e., alternatives, was first introduced by Cardelli et al. [6] and then made practical in *Rats!* [19]. In contrast, attribute grammars provide a simpler extension model: just add a new attribute. This works exceedingly well when extending a language with new constructs and the corresponding attribute grammar productions. But it requires additional machinery when modifying existing attribute definitions [11]. Aspect-oriented programming [25] and J&'s nested inheritance [31] are viable alternatives, with J& also providing static type safety. At the same time, they are more general and more complex.

Unlike the previous systems, which are largely targeted at real-world languages and tools, the TinkerType system [27] is explicitly designed for exploring (versions of) the typed lambda calculus. Its goal is to facilitate the compact and modular description of formal systems, including type systems. To meet this goal, TinkerType models formal systems as collections of clauses, such as a type system's inference rules, and features, such as a type system's types. It then lets developers combine different clauses with different features and automatically ensures that a given combination is consistent. TinkerType's distinction between clauses and features is reminiscent of Typical's distinction between case analysis, i.e., pattern matching, and case definitions, i.e., variant type definitions, but, overall, the two systems have fundamentally different goals.

## 10. Conclusions

We have presented Typical, a practical type checker generator for real-world languages. Our type checker specification language is designed to be expressive, declarative, prescriptive, and correct. To this end, it builds on the functional core of ML and extends it with declarative constructs where appropriate. Constructs new to Typical include those capturing a language's type attributes, type equivalence, scopes, namespaces, and typing constraints. Additionally, type checker specifications are organized into modules, which support fine-grained extensibility through rule rewriting. Experiences with type checkers for Typical and C demonstrate that our language concisely captures the type rules of these substantially different languages and that Typical-generated type checkers exhibit reasonable performance. Typical has been released as open source.

## Acknowledgments

We thank Benjamin Goldberg for serving as a sounding board while designing Typical. We also thank Martin Hirzel and Allan Gottlieb for their feedback on earlier versions of this paper. This material is based upon work supported by the National Science Foundation under Grants No. CNS-0448349 and CNS-0615129.

## References

[1] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proc. 2006 OOPSLA*, pp. 57–74, Oct. 2006.

[2] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM TOPLAS*, 8(4):547–576, Oct. 1986.

[3] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proc. 3rd SESPSDE*, pp. 14–24, Nov. 1989.

[4] G. Bracha. Pluggable type systems. In *Proc. of the OOPSLA '04 Workshop on Revival of Dynamic Languages*, Oct. 2004.

[5] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2005.

[6] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Tech. Report 121, DEC SRC, Feb. 1994.

[7] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proc. 2005 PLDI*, pp. 85–95, June 2005.

[8] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(3):114–115, Sept. 1940.

[9] R. Cox, T. Bergan, A. Clements, F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS*, Mar. 2008.

[10] A. Dijkstra and S. D. Swierstra. Ruler: Programming type rules. In *Proc. 8th FLOPS*, vol. 3945 of *LNCS*, pp. 30–46, Apr. 2006.

[11] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Proc. 18th ECCOP*, vol. 3086 of *LNCS*, pp. 147–171, June 2004.

[12] T. Ekman and G. Hedin. Modular name analysis for Java using JastAdd. In *Proc. GTTSE*, vol. 4143 of *LNCS*, pp. 422–436, July 2005.

[13] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proc. 2007 OOPSLA*, pp. 1–18, Oct. 2007.

[14] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proc. 1999 PLDI*, pp. 192–203, May 1999.

[15] J. Garrigue. Relaxing the value restriction. In *Proc. 7th FLOPS*, vol. 2998 of *LNCS*, pp. 196–213, Apr. 2004.

[16] H. Gast. *A Generator for Type Checkers*. PhD thesis, University of Tübingen, Aug. 2005.

[17] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *CACM*, 35(2):121–130, Feb. 1992.

[18] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *Proc. 2007 OOPSLA*, pp. 321–336, Oct. 2007.

[19] R. Grimm. Better extensibility through modular syntax. In *Proc. 2006 PLDI*, pp. 38–51, June 2006.

[20] R. Grimm. Declarative syntax tree engineering (or, one grammar to rule them all). Tech. Report TR2007-905, NYU, Nov. 2007.

[21] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, 35(3):39–48, Mar. 2000.

[22] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Proc. 2007 OOPSLA*, pp. 19–38, Oct. 2007.

[23] ISO. Information Technology—Programming Languages—C. ISO/IEC Standard 9899:TC2, May 2005.

[24] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Feb. 1978.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *CACM*, 44(10):59–65, Oct. 2001.

[26] B. Lerner, D. Grossman, and C. Chambers. SEMINAL: Searching for ML type-error messages. In *Proc. of the 2006 Workshop on ML*, pp. 63–73, Sept. 2006.

[27] M. Y. Levin and B. C. Pierce. TinkerType: A language for playing with formal systems. *JFP*, 13(2):295–316, Mar. 2003.

[28] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proc. 13th CC*, vol. 2985 of *LNCS*, pp. 73–88, Mar. 2004.

[29] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, May 1997.

[30] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. 11th CC*, vol. 2304 of *LNCS*, pp. 213–228, Apr. 2002.

[31] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *Proc. 2006 OOPSLA*, pp. 21–36, Oct. 2006.

[32] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proc. 7th FPCA*, pp. 135–146, June 1995.

[33] T. Parr. *The Definitive ANTLR Reference*. The Pragmatic Programmers, May 2007.

[34] N. Ramsey. Eliminating spurious error messages using exceptions, polymorphism, and higher-order functions. *Computer Journal*, 42(5):360–372, 1999.

[35] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer, 1989.

[36] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, July 2002.

[37] E. van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In *Proc. 21st ECCOP*, pp. 575–599, July 2007.

[38] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sept. 1997.

[39] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, Dec. 1995.