# Active Type-Checking and Translation

Cyrus Omar
School of Computer Science
Carnegie Mellon University
comar@cs.cmu.edu

## Abstract

We introduce a statically-typed language extensibility mechanism called active type-checking and translation (AT&T) that aims toward expressiveness, safety and composability. This mechanism allows users to equip type definitions with type-level functions that control the compilation process directly, at points that are relevant to that type's semantics.

***Categories and Subject Descriptors*** D.3.2 [*Programming Languages*]: Language Classifications—Extensible Languages; D.3.4 [*Programming Languages*]: Processors—Compilers; F.3.1 [*Logics & Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification Techniques

***Keywords*** extensible languages, type-level computation, typed compilation, specification languages

## 1. Extended Abstract

Programming languages have historically been specified and implemented monolithically. To introduce new primitive constructs, researchers or domain experts have developed a new language or a dialect of an existing language, with the help of tools like domain-specific language frameworks and compiler generators [1]. Unfortunately, taking a so-called *language-oriented approach* [3], where different languages are used for different components of an application, can lead to problems at language boundaries: a library's external interface must only rely on constructs that can be expressed in all possible calling languages. This means that specialized invariants cannot be checked statically, decreasing reliability and performance. It also often requires that developers generate verbose and unnatural "glue" code, defeating a primary purpose of specialized languages: hiding these low-level details from end-user developers.

Extensible programming languages promise to decrease the need for new standalone languages by providing more granular, language-based support for introducing new primitive constructs (that is, constructs that cannot be adequately expressed in terms of existing syntactic forms and primitive operations.) Developers would gain the freedom to choose those constructs that are most appropriate for their application domain and development discipline. Researchers would gain the ability to distribute new constructs for evaluation by a broader development community without requiring the approval of maintainers of mainstream languages, who are naturally risk-averse and uninterested in niche domains and still-emerging techniques.

A major example motivating our work is in the area of parallel programming abstractions. Many implemented abstractions require adding new type-checking and compilation logic to a language. Unfortunately, this has led to the development of new languages. An example for data parallel programming on accelerators and GPUs is the OpenCL language. Although OpenCL is largely based on C99, it is a distinct language and requires a separate set of tools. We would like a language that allows us to express the novel all of the novel aspects of OpenCL directly, so that it can be imported as easily as a library is today.

A significant challenge that faces language extensibility mechanisms is in maintaining the overall safety properties of the language and compilation process in the presence of arbitrary combinations of user extensions. The mechanism must ensure that basic metatheoretic and global safety guarantees of the language cannot be weakened, that extensions are safely composable, and that type checking and compilation remains decidable. The correctness of an extension itself should be modularly verifiable, so that its users can rely on it for verifying and compiling their own code. These are the issues that we seek to address in this work.

The approach we describe, *active type-checking and compilation* (AT&T), makes use of type-level computation in a novel way. To review, in languages supporting type-level computation, the syntactic class of types is not simply declarative. Instead, it forms a programming language itself (the *type-level language*). Types themselves are one kind of value in this language, but there can be many others.

To ensure the safety of type-level computations, *kinds* classify type-level terms, just as types classify expression-level terms. A growing number of implemented languages now feature more sophisticated type-level languages, including Haskell. Type-level computation occurs during compilation, rather than at run-time, because type-level terms that are used where types would normally be expected must be reduced to normal form before type-checking can proceed. In this work, we wish to allow extensions to strengthen the static semantics of our language. Naturally, extension specifications will also need to be evaluated during compilation and manipulate representations of types. This observation suggests that the type-level language may be able to serve directly as a specification language. In this work, we show that this is indeed the case.

By introducing some new constructs at the type-level, developers are able to specify the semantics of the primitive operators associated with newly-introduced families of primitive types using type-level functions. The compiler front-end invokes these functions to synthesize types for and assign meanings to expressions, by translation into a *typed internal language*. Unlike conventional metaprogramming systems, these *type-level specifications* do not directly manipulate or rewrite expressions. Instead, they examine and manipulate the types of these expressions. By using a sufficiently constraining kind system and incorporating techniques from typed compilation into the type-level language directly, the global safety properties of the language and compilation process can be guaranteed. In other words, users can only *increase* the safety of the language.

We focus on extending the static semantics of a language with a fixed, though flexible, grammar. Techniques for extensible parsing have been proposed in the past, but we do not discuss this further here. We also focus on extending implementations, rather than declarative specifications, of language constructs. Extracting a compiler from a declarative language specification (e.g. one written in Twelf) has not yet been shown practical, but we note that a future mechanism of this sort could target a language implementing AT&T.

The organization and key contributions of this work are:

- We develop a core calculus, $\lambda_{\text{A}}$, and give examples of language features that can be expressed with it. We judgmentally specify the front-end compilation process and state several lemmas that lead to useful safety theorems for the compiler and language as a whole. We show how AT&T requires that the language provide a solution to a type-level variant of Wadler's expression problem [2].
- We introduce the Ace programming language, which is based fundamentally on an elaboration of AT&T that supports a richer set of syntactic forms and a variant of type inference. It uses object-oriented inheritance to solve the type-level expression problem. A number of practical extensions have been written using Ace, including a complete implementation of the OpenCL type system (based on C99) as a library. Figure 1 shows how such an extension could be used. OpenCL's types are represented as objects in the type-level language (in Ace, this is Python). Here, `OpenCL.int.global_ptr` is the object representing the `__global int*` type. This object is an instance of a class extending `ace.Type`, and specifies methods for verifying and translating operations, such as indexing (`a[gid]`), into target language code. Compiling this code produces a function exactly equivalent to the OpenCL function in Figure 2.

```
import ace.OpenCL as OpenCL
T = OpenCL.int.global_ptr

@OpenCL.fn(T, T, T)
def sum(a, b, dest):
    gid = get_global_id(0)
    dest[gid] = a[gid] + b[gid]
```

**Figure 1.** An Ace function that uses the OpenCL extension.

```
typedef __global int* T;

__kernel void sum(T a, T b, T c) {
    size_t gid = get_global_id(0);
    dest[gid] = a[gid] + b[gid];
}
```

**Figure 2.** The corresponding OpenCL function.

- We briefly describe another point in the design space, a language design we call Birdie. Birdie lifts an extension of the Gallina language, used by the Coq proof assistant, into the type level (leading to a language with dependent kinds). This additional complexity allows for full proofs of correctness for type-level specifications, and can allow proofs soundness of functional specifications against conventional inductive specifications. The expression problem is solved using a constrained formulation of open data types, rather than using object-oriented inheritance. Birdie is ongoing work.

## Acknowledgments

The author is grateful to Jonathan Aldrich. This work was funded by the DOE Computational Science Graduate Fellowship under grant number DE-FG02-97ER25308.

## References

[1] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[2] P. Wadler. The expression problem. *java-genericity Mailing List*, 1998.

[3] M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.