Active Type-Checking and Translation

Cyrus Omar Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
{comar, jonathan.aldrich}@cs.cmu.edu

Abstract

Researchers and domain experts propose novel language constructs regularly. Unfortunately, today's statically-typed languages are *monolithic*: they do not give users the ability to specify the semantics of new types and primitive operations directly and safely. This has led to a proliferation of distinct languages built around a few situationally useful constructs. However, using multiple languages in an application remains difficult. An alternative to this language-oriented approach is to work with an *extensible* programming language. Designing an extensibility mechanism that is both expressive and safe has been a challenge – extensions must not weaken the global safety properties of the language, compromise the integrity of the compilation process, or interfere with each other.

This paper introduces a mechanism called *active type-checking and translation* (AT&T) that aims to address these issues. By using type-level computation in a novel way, AT&T can support a wide range of user-defined semantics over a fixed grammar safely and modularly. In a language supporting AT&T, all types are represented uniformly at the type level. Each type is equipped with type-level functions that are invoked specifically during the type-checking and translation of operations relevant to that type. We discuss three points in the design space: (1) a judgmental lambda calculus-based formulation designed to distill essential concepts and admit formal safety theorems, (2) a practically-realized language called Ace that we use to demonstrate the expressive power of AT&T in practice, and (3) a proposed language called Birdie that lifts Coq's rich system of dependent types into the type-level to allow extension developers to fully prove the correctness of an extension statically.

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. Introduction

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 [**?**]. Unfortunately, taking a so-called *language-oriented approach* [**?**], 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.

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. The simplest example of a language featuring type-level computation is Girard's System F_ω [**?**]. In F_ω, types have kind $\star$ and type-level functions have kinds like $\star \rightarrow \star$. A growing number of implemented languages now feature more sophisticated type-level languages (see Section 5). We emphasize that 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 paper, we show that this is indeed the case. By introducing some new constructs at the type-level, developers can specify the semantics of operators associated with newly-introduced families of primitive types with 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 (e.g. [**?**]), and we conjecture that these can be made compatible with the mechanism described in this paper with some simple modifications, but we do not discuss this further here. We also focus on *functional*, rather than declarative, specifications of language constructs. Extracting a compiler from a declarative language specification (e.g. in Twelf [**?**]) has not yet been shown practical, but we note that a future mechanism of this sort could safely target a language implementing the mechanism we discuss here.

The organization and key contributions of this paper are:

- In Section 2, we develop a core calculus, λ_{A}, and give simple examples of language features that can be expressed with it. We formalize the compiler front-end 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 **??**.
- In Section 3, we briefly 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.
- In Section 4, 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.
- In Section 5 we compare AT&T to previous work on extensible languages and compilers, metaprogramming systems and formal specification languages and conclude in Section 6 with a discussion of future work.

2. Type-Level Specifications in λ_{A}

2.1 Example: Natural Numbers in λ_{A}

We begin with a simple calculus with no primitive notion of natural numbers, nor any more general notion of an inductive data type. We can, however, concretely specify both the static and dynamic semantics of natural numbers, including the natural recursor of Gödel's T [**?**], using type-level specifications. Let us begin in Figure 1 with the type **nat** and its constructors, **z** and **s**.

$$
\begin{array}{rcll}
\textbf{expressions} & e & ::= & \mathrm{x} \mid \lambda \mathrm{x}{:}\psi.e \mid e_1\ e_2 \mid \psi_{\text{type}}.\mathbf{op}\langle \psi_1, \ldots, \psi_n \rangle(e_1, \ldots, e_n) \\
 & & & \\
\textbf{type-level specifications} & \psi & ::= & \mathbf{t} \mid \lambda[\mathbf{t}_1{:}\kappa_1,\ \ldots\ ,\mathbf{t}_n{:}\kappa_n].\psi \mid \psi(\psi_1, \ldots, \psi_n) \mid \mathsf{if}\ \psi_0 \equiv \psi_1\ \mathsf{then}\ \psi_2\ \mathsf{else}\ \psi_3 \\
\text{standard terms} & & \mid & () \mid (\psi_1, \psi_2) \mid \mathsf{fst}\ \psi \mid \mathsf{snd}\ \psi \\
\text{types} & & \mid & \mathsf{type}\ \mathbf{t}[\psi_{\text{idx}} : \kappa_{\text{idx}}]\ ::\ \delta_{\text{rep}}\ \{\theta\} \mid \mathsf{idx}\ \psi_{\text{type}} \mid \mathsf{idxcase}\ \psi_{\text{type}}\ \mathsf{of}\ \kappa\ \mathsf{then}\ \psi_0\ \mathsf{else}\ \psi_1 \\
\text{denotations} & & \mid & [\![\mu \sim \psi_{\text{type}}]\!] \mid \mathsf{typeof}\ \psi_{\text{den}} \mid \mathsf{err} \mid \mathsf{dencase}\ \psi_{\text{den}}\ \mathsf{of}\ \kappa\ \mathsf{then}\ \psi_0\ \mathsf{else}\ \psi_1 \\
\text{programs} & & \mid & \mathsf{program}(e) \\
\text{operators} & & \mid & \mathsf{const}(\psi) \mid \mathsf{op}(\psi) \\
\text{operator specifications} & \theta & ::= & \mathbf{op} = \psi \mid \theta_1; \theta_2 \\
\textbf{kinds} & \kappa & ::= & [\kappa_1, \ldots, \kappa_n] \to \kappa \mid \texttt{Unit} \mid \kappa_1 \times \kappa_2 \mid \texttt{Type}[\kappa] \mid \texttt{Type} \mid \texttt{Den}[\kappa] \mid \texttt{Den} \mid \texttt{Program} \mid \texttt{Op}_n \\
 & & & \\
\textbf{internal terms} & \mu & ::= & \mathsf{u} \mid \lambda \mathsf{u} :: \psi.\mu \mid \mathsf{fix\ f} :: \psi.\mu \mid \mu_1\ \mu_2 \mid \mathsf{if}\ \mu_0 \equiv \mu_1\ \mathsf{then}\ \mu_2\ \mathsf{else}\ \mu_3 \\
 & & \mid & n \mid \mu_1 + \mu_2 \mid (\mu_1, \mu_2) \mid \mathsf{fst}\ \mu \mid \mathsf{snd}\ \mu \mid \mathsf{valof}(\psi_{\text{den}}) \\
\textbf{internal types} & \delta & ::= & \delta_1 \to \delta_2 \mid \texttt{int} \mid \delta_1 \times \delta_2 \mid \mathsf{repof}(\psi_{\text{den}})
\end{array}
$$

Figure 1. Syntax of λ_{A}. $\mathbb{Z}$ denotes integer literals and LABEL denotes any label.

$$
\frac{\overbrace{\cdot \vdash \psi_{\text{prog}} : \texttt{Program}}^{\text{kind checking}} \qquad \overbrace{\psi_{\text{prog}} \Downarrow_{\text{ok}} \mathsf{program}(e)}^{\text{specification normalization}} \qquad \overbrace{\cdot \vdash e \longrightarrow [\![\mu \sim \mathsf{type}\ \mathbf{t}[\psi_{\text{idx}} : \kappa_{\text{idx}}]\ ::\ \delta_{\text{rep}}\ \{\theta\}]\!]}^{\text{verification \& translation}}}{\psi_{\text{prog}} \Longrightarrow [\![\mu \sim \mathsf{type}\ \mathbf{t}[\psi_{\text{idx}} : \kappa_{\text{idx}}]\ ::\ \delta_{\text{rep}}\ \{\theta\}]\!]}
$$

Figure 2. Central compilation judgement of λ_{A}.

$$\boxed{\Delta \vdash \psi : \kappa} \qquad \Delta ::= \cdot \mid \Delta, \mathbf{t} : \kappa$$

$$\frac{\mathbf{t} : \kappa \in \Delta}{\Delta \vdash \mathbf{t} : \kappa}\,(\text{var}^{\psi}) \qquad \frac{}{\Delta \vdash () : \mathtt{Unit}}\,(\text{Unit}) \qquad \frac{\Delta \vdash \psi_1 : \kappa_1 \qquad \Delta \vdash \psi_2 : \kappa_2}{\Delta \vdash (\psi_1, \psi_2) : \kappa_1 \times \kappa_2}\,(\times_I) \qquad \frac{\Delta \vdash \psi : \kappa_1 \times \kappa_2}{\Delta \vdash \mathsf{fst}\ \psi : \kappa_1}\,(\times_E^1) \qquad \frac{\Delta \vdash \psi : \kappa_1 \times \kappa_2}{\Delta \vdash \mathsf{snd}\ \psi : \kappa_2}\,(\times_E^2)$$

$$\frac{\Delta, \mathbf{t}_1{:}\kappa_1,\ \ldots\ , \mathbf{t}_n{:}\kappa_n \vdash \psi : \kappa}{\Delta \vdash \lambda[\mathbf{t}_1{:}\kappa_1,\ \ldots\ , \mathbf{t}_n{:}\kappa_n].\psi : [\kappa_1, \ldots, \kappa_n] \to \kappa}\,(\lambda_I^{\psi}) \qquad \frac{\Delta \vdash \psi : [\kappa_1, \ldots, \kappa_n] \to \kappa \qquad \Delta \vdash \psi_1 : \kappa_1 \qquad \cdots \qquad \Delta \vdash \psi_n : \kappa_n}{\Delta \vdash \psi(\psi_1, \ldots, \psi_n) : \kappa}\,(\lambda_E^{\psi})$$

$$\frac{\Delta \vdash \psi_0 : \kappa_1 \qquad \Delta \vdash \psi_1 : \kappa_1 \qquad \Delta \vdash \psi_2 : \kappa_2 \qquad \Delta \vdash \psi_3 : \kappa_2}{\Delta \vdash \mathsf{if}\ \psi_0 \equiv \psi_1\ \mathsf{then}\ \psi_2\ \mathsf{else}\ \psi_3 : \kappa_2}\,(\text{cond}^{\psi})$$

$$\frac{\mathtt{Type}\ \text{strictly positive in}\ \kappa_{\text{idx}} \qquad \Delta \vdash \psi_{\text{idx}} : \kappa_{\text{idx}} \qquad \Delta \vdash \delta_{\text{rep}} \qquad \Delta, \mathbf{t} : \mathtt{Type}[\kappa_{\text{idx}}] \vdash \theta}{\Delta \vdash \mathsf{type}\ \mathbf{t}[\psi_{\text{idx}} : \kappa_{\text{idx}}]\ ::\ \delta_{\text{rep}}\ \{\theta\} : \mathtt{Type}[\kappa_{\text{idx}}]}\,(\text{Type}_I) \qquad \frac{\Delta \vdash \psi : \mathtt{Type}[\kappa]}{\Delta \vdash \psi : \mathtt{Type}}\,(\text{Type-}\subseteq)$$

$$\frac{\Delta \vdash \psi : \mathtt{Type}[\kappa]}{\Delta \vdash \mathsf{idx}\ \psi : \kappa}\,(\text{idx}) \qquad \frac{\Delta \vdash \psi_{\text{type}} : \mathtt{Type} \qquad \Delta \vdash \psi_0 : [\mathtt{Type}[\kappa_{\text{idx}}]] \to \kappa \qquad \Delta \vdash \psi_1 : \kappa}{\Delta \vdash \mathsf{idxcase}\ \psi_{\text{type}}\ \mathsf{of}\ \kappa_{\text{idx}}\ \mathsf{then}\ \psi_0\ \mathsf{else}\ \psi_1 : \kappa}\,(\text{idxcase})$$

$$\frac{\Delta, \cdot \vdash \mu \qquad \Delta \vdash \psi_{\text{type}} : \mathtt{Type}[\kappa]}{\Delta \vdash \llbracket \mu \sim \psi_{\text{type}} \rrbracket : \mathtt{Den}[\kappa]}\,(\text{Den}_I) \qquad \frac{\Delta, \cdot \vdash \mu \qquad \Delta \vdash \psi_{\text{type}} : \mathtt{Type}}{\Delta \vdash \llbracket \mu \sim \psi_{\text{type}} \rrbracket : \mathtt{Den}}\,(\text{Den}_I\text{-}\subseteq) \qquad \frac{\Delta \vdash \psi : \mathtt{Den}[\kappa]}{\Delta \vdash \psi : \mathtt{Den}}\,(\text{Den-}\subseteq) \qquad \frac{}{\Delta \vdash \mathsf{err} : \mathtt{Den}}\,(\text{err})$$

$$\frac{\Delta \vdash \psi : \mathtt{Den}[\kappa]}{\Delta \vdash \mathsf{typeof}\ \psi : \mathtt{Type}[\kappa]} \qquad \frac{\Delta \vdash \psi_{\text{den}} : \mathtt{Den} \qquad \Delta \vdash \psi_0 : [\mathtt{Den}[\kappa_{\text{idx}}]] \to \kappa \qquad \Delta \vdash \psi_1 : \kappa}{\Delta \vdash \mathsf{dencase}\ \psi_{\text{den}}\ \mathsf{of}\ \kappa_{\text{idx}}\ \mathsf{then}\ \psi_0\ \mathsf{else}\ \psi_1 : \kappa}\,(\text{dencase})$$

$$\frac{\Delta, \cdot \vdash e}{\Delta \vdash \mathsf{program}(e) : \mathtt{Program}}\,(\text{program}) \qquad \frac{\Delta \vdash \psi : \mathtt{Den}}{\Delta \vdash \mathsf{const}(\psi) : \mathtt{Op}_0}\,(\text{const}) \qquad \frac{\Delta \vdash \psi : [\mathtt{Den}, \overset{n}{\cdots}, \mathtt{Den}] \to \mathtt{Den}}{\Delta \vdash \mathsf{op}(\psi) : \mathtt{Op}_n}\,(\text{op})$$

$$\boxed{\Delta \vdash \theta}$$

$$\frac{\Delta \vdash \psi : \mathtt{Op}_n}{\Delta \vdash \mathbf{op} = \psi}\,(\text{op-def}) \qquad \frac{\Delta \vdash \psi : [\kappa_1, \ldots, \kappa_m] \to \mathtt{Op}_n}{\Delta \vdash \mathbf{op} = \psi}\,(\text{op-fam-def}) \qquad \frac{\Delta \vdash \theta_1 \qquad \Delta \vdash \theta_2}{\Delta \vdash \theta_1; \theta_2}\,(\text{op-seq})$$

$$\boxed{\Delta, \overline{\Sigma} \vdash \mu} \qquad \overline{\Sigma} ::= \cdot \mid \overline{\Sigma}, \mathrm{u}$$

$$\frac{\mathrm{u} \in \overline{\Sigma}}{\Delta, \overline{\Sigma} \vdash \mathrm{u}}\,(\text{var-form}^{\mu}) \qquad \frac{\Delta \vdash \delta \qquad \Delta, \overline{\Sigma}, u \vdash \mu}{\Delta, \overline{\Sigma} \vdash \lambda \mathrm{u} :: \delta.\mu}\,(\lambda\text{-form}^{\mu}) \qquad \frac{\Delta \vdash \delta \qquad \Delta, \overline{\Sigma}, \mathrm{f} \vdash \mu}{\Delta, \overline{\Sigma} \vdash \mathsf{fix}\ \mathrm{f} :: \delta.\mu}\,(\text{fix-form}^{\mu}) \qquad \frac{\Delta, \overline{\Sigma} \vdash \mu_1 \qquad \Delta, \overline{\Sigma} \vdash \mu_2}{\Delta, \overline{\Sigma} \vdash \mu_1\ \mu_2}\,(\text{app-form}^{\mu})$$

$$\frac{\Delta, \overline{\Sigma} \vdash \mu_1 \qquad \Delta, \overline{\Sigma} \vdash \mu_2 \qquad \Delta, \overline{\Sigma} \vdash \mu_3 \qquad \Delta, \overline{\Sigma} \vdash \mu_4}{\Delta, \overline{\Sigma} \vdash \mathsf{if}\ \mu_1 \equiv \mu_2\ \mathsf{then}\ \mu_3\ \mathsf{else}\ \mu_4}\,(\text{if-form}^{\mu}) \qquad \frac{}{\Delta, \overline{\Sigma} \vdash n}\,(\text{int-form}^{\mu}) \qquad \frac{\Delta, \overline{\Sigma} \vdash \mu_1 \qquad \Delta, \overline{\Sigma} \vdash \mu_2}{\Delta, \overline{\Sigma} \vdash \mu_1 + \mu_2}\,(+\text{-form}^{\mu})$$

$$\frac{\Delta, \overline{\Sigma} \vdash \mu_1 \qquad \Delta, \overline{\Sigma} \vdash \mu_2}{\Delta, \overline{\Sigma} \vdash (\mu_1, \mu_2)}\,(\times\text{-form}^{\mu}) \qquad \frac{\Delta, \overline{\Sigma} \vdash \mu}{\Delta, \overline{\Sigma} \vdash \mathsf{fst}\ \mu}\,(\text{fst-form}^{\mu}) \qquad \frac{\Delta, \overline{\Sigma} \vdash \mu}{\Delta, \overline{\Sigma} \vdash \mathsf{snd}\ \mu}\,(\text{snd-form}^{\mu}) \qquad \frac{\Delta \vdash \psi_{\text{den}} : \mathtt{Den}}{\Delta, \overline{\Sigma} \vdash \mathsf{valof}(\psi_{\text{den}})}\,(\text{valof})$$

$$\boxed{\Delta \vdash \delta}$$

$$\frac{\Delta \vdash \delta_1 \qquad \Delta \vdash \delta_2}{\Delta \vdash \delta_1 \to \delta_2}\,(\to\text{-form}^{\delta}) \qquad \frac{}{\Delta \vdash \mathtt{int}}\,(\text{int-form}^{\delta}) \qquad \frac{\Delta \vdash \delta_1 \qquad \Delta \vdash \delta_2}{\Delta \vdash \delta_1 \times \delta_2}\,(\times\text{-form}^{\delta}) \qquad \frac{\Delta \vdash \psi_{\text{den}} : \mathtt{Den}}{\Delta \vdash \mathsf{repof}(\psi_{\text{den}})}\,(\text{repof})$$

$$\boxed{\Delta, \overline{\Gamma} \vdash e} \qquad \overline{\Gamma} ::= \cdot \mid \overline{\Gamma}, \mathrm{x}$$

$$\frac{\mathrm{x} \in \overline{\Gamma}}{\Delta, \overline{\Gamma} \vdash \mathrm{x}}\,(\text{var-form}^{e}) \qquad \frac{\Delta \vdash \psi : \mathtt{Type} \qquad \Delta, \overline{\Gamma}, \mathrm{x} \vdash e}{\Delta, \overline{\Gamma} \vdash \lambda \mathrm{x}{:}\psi.e}\,(\lambda\text{-form}^{e}) \qquad \frac{\Delta, \overline{\Gamma} \vdash e_1 \qquad \Delta, \overline{\Gamma} \vdash e_2}{\Delta, \overline{\Gamma} \vdash e_1\ e_2}\,(\text{app-form}^{e})$$

$$\frac{\Delta \vdash \psi_{\text{type}} : \mathtt{Type} \qquad \Delta \vdash \psi_1 : \kappa_1 \qquad \cdots \qquad \Delta \vdash \psi_m : \kappa_m \qquad \Delta, \overline{\Gamma} \vdash e_1 \qquad \cdots \qquad \Delta, \overline{\Gamma} \vdash e_n}{\Delta, \overline{\Gamma} \vdash \psi_{\text{type}}.\mathbf{op}\langle \psi_1, \ldots, \psi_m \rangle (e_1, \ldots, e_n)}\,(\text{op-app-form}^{e})$$

Figure 3. Kind checking rules for λ_{A}. All contexts have standard substructural properties (not shown).