Active Typechecking and Translation: A Safe Language-Internal Extension Mechanism

Abstract

Researchers and domain experts regularly propose new language primitives. Unfortunately, today's statically-typed languages are monolithic: they do not expose mechanisms for implementing new primitive types and their associated operators directly, so experts often turn to new standalone languages. Building applications from components written in many different languages, however, can be unsafe, inefficient and unnatural. An internally-extensible language could help address these issues, but designing a mechanism that is both expressive and safe remains a challenge. Extensions must be checked for internal consistency, their use in any combination must not weaken the metatheoretic properties of the language, nor can they be allowed to interfere with one another. We introduce a mechanism called active type-checking and translation (AT&T) that aims to directly address these issues. AT&T leverages type-level computation and typed compilation techniques to enable users to implement a wide variety of semantics over a uniform grammar in a checkably-safe manner from within libraries.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Extensible languages

Keywords extensible languages, domain-specific languages, type-level computation, typed compilation, active libraries

1. Introduction

Programming languages are typically designed around a monolithic collection of primitive types and operators. Consider, as a simple example, Godel's T [**?**], a typed lambda calculus with recursion on primitive natural numbers. Although a language designer may casually speak of "extending Godel's T with primitive product and sum types", modularly adding such new primitive types and their corresponding operators to this kind of language from within is impossible. That is, Godel's T is not *internally extensible*.

The only recourse language designers have in such situations is to attempt to encode new constructs in terms of existing constructs. Such encodings, collections of which are often called *embedded domain-specific languages (DSLs)* [6], must creatively combine the constructs available in the host "general-purpose" language. Unfortunately, such encodings are at times impossible, and even if possible, often impractical. In our example of adding products or sums to Godel's T, although Church encodings are possible [**?**], they require a reasonable level of creativity[1]. Moreover, they will not offer the same static safety guarantees as a primitive encoding, they are more verbose and they will incur performance overhead by their use of closures rather than a more direct representation. This is not only a problem for simple languages like Godel's T. Several Haskell-based embedded DSLs have also needed to make significant compromises at times [**?**]. examples? Scala?

Researchers or domain experts who cannot work around such limitations must develop new standalone languages. In our simple scenario, we may simply fork our implementation of Godel's T or even edit it directly (a pernicious technique for implementing a new language where the prior one is overwritten). In a more complex scenario, we may instead employ a tool like a compiler generator or DSL framework [6] that can generate a standalone implementation from declarative specifications of language constructs. Some of these tools allow you to package and reuse these specifications (with the important caveat that not all combinations of constructs are valid and free of conflicts, an important modularity issue that we will return to several times in this paper).

The increasing sophistication and ease-of-use of these tools have led many to suggest a *language-oriented approach* [14] to software development where different components of an application are written in different languages. Unfortunately, this leads to problems at language boundaries: a library's external interface must only use constructs that can reasonably be expressed in *all possible calling languages*. This can restrict domain-specific languages by, for example, precluding constructs that rely on statically-checked invariants stronger than those their underlying representation in a common target language normally supports. At best, constructs like these can be exposed by generating a wrapper where run-time checks have been inserted to guarantee necessary invariants. This compromises both verifiability and performance and requires the development of an interoperability layer for every DSL. Moreover, library clients must work with verbose and unnatural "glue code" when interfacing across languages, defeating the primary purpose of high-level programming languages: hiding the low-level details from the end-users of abstractions. We diagram this fundamental *compatibility problem* in Figure 1(a).

Internally-extensible programming languages promise to avoid these problems by providing researchers and domain experts with a language-supported mechanism for specifying new primitive types and operators. Using such extensions, library developers need only consider which abstractions are most appropriate for their domain, without also considering whether these constructs can be exposed using abstractions appropriate to the domains of client code. Clients can simply import any necessary constructs when using a library

[1] Anecdotally, Church encodings are among the more difficult-to-explain topics covered in our undergraduate programming languages course.

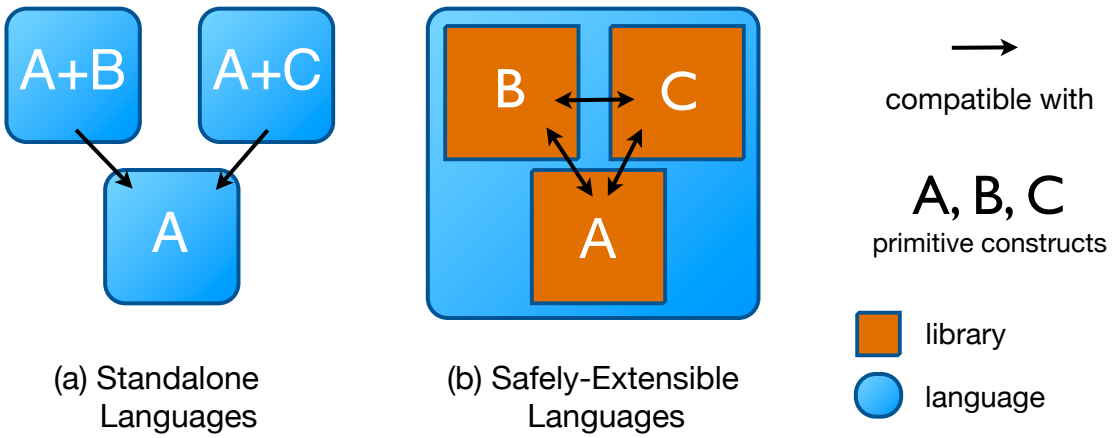

Figure 1. (a) With a language-oriented approach, novel constructs are packaged into separate languages. Users can only safely and naturally call into languages consisting of common constructs (often only the common target language, such as C or Java bytecode). (b) With a language-internal extensibility approach, there is one system providing a common internal language, where additional primitive constructs that strictly strengthen its static guarantees or perform specialized code generation are specified and distributed within libraries.

that relies on them, preserving safety and ease-of-use without the use of wrappers and glue code. We show this competing approach in Figure 1(b).

Unfortunately, a significant challenge remains before such a mechanism is possible: balancing expressiveness with concerns about maintaining safety properties in the presence of arbitrary combinations of user-defined language extensions. transition here Correctness properties of an extension itself should be modularly verifiable, so that its users can rely on it for verifying and compiling their own code. The mechanism must also ensure that desirable metatheoretic properties and global safety guarantees of the language cannot be weakened by extensions. And with multiple independently-developed extensions used at once, the mechanism must further guarantee that they cannot interfere with one another.

In this paper, we will describe a language-internal extensibility mechanism called *active typechecking and translation* (AT&T) that allows developers to specify new typechecking rules and translation logic from within libraries. The AT&T mechanism utilizes type-level computation of higher kind and integrates typed compilation techniques into the language to provide strong safety guarantees, while remaining straightforward and expressive.

AT&T is general with respect to many choices about the type-level language, the typed internal language and syntax. Choices along these dimensions can affect both expressiveness and ease-of-use. We will begin in Sec. 2 by introducing a minimal system called @λ (the "actively-typed lambda calculus") that distills the essence of the mechanism in a simply-typed, simply-kinded setting. This will allow us to fully and concisely formalize the language and compiler and give several key safety theorems. We will then continue in Sec. 3 by discussing variants of this mechanism based on other basic paradigms, considering dependently-typed functional languages and object-oriented languages, discussing trade-offs between expressivity and safety when doing so. We have developed a simple prototype called Ace and have used it to develop a number of full-scale language extensions as libraries. We will briefly discuss this language and these extensions in Sec. 4.

We note at the outset that AT&T focuses on extending the static semantics of languages with fixed, though flexible, syntax. Language-internal syntax extension mechanisms have been developed in the past (e.g. SugarJ [**?**]) but they have also suffered from safety problems because grammar composition is not always safe when done in an unconstrained manner. Constrained approaches that provide stronger safety guarantees have recently been outlined (e.g. Wyvern [**?**]) but we will leave integration of syntax extensions with semantic extensions as future work.

2. From Externally-Extensible Implementations to Internally-Extensible Languages

To understand the genesis of our internal extensibility mechanism, it is helpful to begin by considering why most implementations of programming languages are not even *externally extensible*. Let us consider again Godel's T. In a functional implementation, its types and operators will invariably be represented using closed datatypes. For example, a simple implementation in Standard ML may be based around these datatypes:

```
datatype Type = Nat | Arrow of Type * Type
datatype Exp = Var of var
             | Lam of var * Type * Exp
             | Ap of Exp * Exp
             | Z | S of Exp
             | Natrec of Exp * Exp * Exp
```

The logic governing the typechecking phase as well as the initial compilation phase (we call this phase *translation* to distinguish it from subsequent optimization phases) will proceed by exhaustive case analysis. In an object-oriented implementation of Godel's T, we could instead represent types and operators as instances of subclasses of abstract classes `Type` and `Exp`. If typechecking and translation then proceed by the ubiquitous *visitor pattern* [**?**], by dispatching against a fixed collection of known subclasses of `Exp`, the same basic issue is encountered: there is no modular way to add new primitive types or operators to the implementation. Indeed, this basic issue is the canonical example of the widely-discussed *expression problem* [**?**].

A number of language mechanisms have been proposed that allow new cases to be added to datatypes and functions in a modular way. In functional languages, these are known as *open datatypes* [**?**]. The language might allow you to add types and operators for products to an open variant of the above definitions like this:

```
newcase Prod of Type * Type extends Type
```

```
newcase Pair of Exp * Exp extends Exp
newcase PrL of Exp extends Exp
newcase PrR of Exp extends Exp
```

The corresponding logic for functionality like typechecking and translation can then be specified for only the new cases, e.g.:

```
typeof PrL(e) =
  case typeof e of
    Prod(t1, _) => t1
  | _ => raise TypeError("<message>")
```

If we allow users to define new modules containing definitions like these and link them into our compiler, we have succeeded in creating an externally-extensible language implementation, albeit one where safety is not guaranteed (we will return to this point shortly). We have not, however, created an extensible programming language, because other implementations of the language will not necessarily support the same mechanism. If our newly-introduced constructs are used at library interface boundaries, we introduce the same compatibility problems that developing a new standalone language can create. That is, **extending a language by extending a single implementation of it is equivalent to creating a new language**. Several prominent language ecosystems today are in a state where a prominent compiler has introduced extensions that many libraries have come to rely on, including the Glasgow Haskell Compiler and the GNU compilers for C and C++. We argue that this practice should be considered harmful.

A more appropriate and useful place for extensions like this is directly within libraries. To enable such use cases, the language must provide a mechanism that allows expert users to declare new type families, like `Prod`, their associated operators, like `Pair`, `PrL` and `PrR`, and their associated typechecking and translation logic. When a compiler encounters these declarations, it adds them to its internal representation of types and operators, as if they had been primitives of the language, so that when user-defined operators are used in expressions, the compiler can temporarily hand control over the typechecking and translation phases to them. Because the mechanism is language-internal, all compilers must support it to satisfy the language specification. Thus, library developers can use new primitive constructs at external interfaces more freely.

Statically-typed languages typically make a distinction between the expression language, where run-time logic is expressed, and the type-level language where, for example, datatypes and type aliases are statically declared. The description above suggests we may now need another layer in our language, an extension language, where users provide extension specifications. In fact, we will show that **the natural place for type system extensions is within the type-level language**. The intuition is that extensions will need to introduce and statically manipulate types and type families. Many languages already support notions of *type-level computation* where types are manipulated as values at compile-time (see Sec. **??** for examples of such languages). These are precisely the properties that such an extension language would have, so we unify the two.

3. @λ

Let us now make these ideas more concrete by developing a core calculus, and discuss how to address safety concerns that arise when giving users this level of control over the language.

3.1 Overview

Our calculus, called @λ for "actively-typed lambda calculus", is based on the simply-typed lambda calculus with simply-kinded type-level computation. That is, type-level terms themselves form a second-level simply-typed lambda calculus; we call the second level of types *kinds* for clarity. Types themselves, which classify

programs	ρ	::=	$\mathsf{family}\ \textsc{Fam}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\theta : \Theta \sim \mathbf{i}.\tau\};\ \rho$
		|	$\mathsf{def}\ \mathbf{t} : \kappa = \tau;\ \rho$
		|	e
expressions	e	::=	$x \mid \lambda x{:}\tau.e \mid \textsc{Fam}.\boldsymbol{op}\langle\tau_i\rangle(e_1;\ldots;e_n)$
type-level terms	τ	::=	$\mathbf{t} \mid \lambda\mathbf{t}{:}\kappa.\tau \mid \tau_1\ \tau_2 \mid \bar{\mathsf{z}} \mid \tau_1 \oplus \tau_2$
type-level data		|	$\textit{"str"} \mid () \mid (\tau_1,\tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau)$
		|	$[]_\kappa \mid \tau_1 :: \tau_2 \mid \mathsf{fold}(\tau_1;\tau_2;\mathbf{x},\mathbf{y}.\tau_3)$
equivalence		|	$\mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4$
types		|	$\textsc{Fam}\langle\tau\rangle$
		|	$\mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2$
denotations		|	$[\![\tau_1 \Longrightarrow \tau_2]\!] \mid \mathsf{err}$
		|	$\mathsf{case}\ \tau\ \mathsf{of}\ [\![\mathbf{x} \Longrightarrow \mathbf{y}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2$
quoted IL		|	$\triangledown(\hat{\gamma}) \mid \blacktriangledown(\hat{\sigma})$
kinds	κ	::=	$\star \mid \mathbb{Z} \mid \mathsf{Str} \mid 1 \mid \kappa_1 \times \kappa_2 \mid \mathsf{list}[\kappa]$
		|	$\kappa_1 \to \kappa_2 \mid \mathsf{Den} \mid \mathsf{IType} \mid \mathsf{ITerm}$
ops	θ	::=	$\cdot \mid \theta;\boldsymbol{op}(\mathbf{i}{:}\kappa_i,\mathbf{a}.\tau)$
ops signature	Θ	::=	$\cdot \mid \Theta;\boldsymbol{op}[\kappa_i]$
internal terms	$\mathcal{G}[g,s]$	::=	$x \mid \lambda x{:}s.g \mid g_1\ g_2 \mid \mathsf{fix}\ f{:}s\ \mathsf{is}\ g \mid \bar{\mathsf{z}}$
		|	$g_1 \oplus g_2 \mid (g_1,g_2) \mid \mathsf{fst}(g) \mid \mathsf{snd}(g)$
		|	$\mathsf{if}\ g_1 \equiv_{\mathbb{Z}} g_2\ \mathsf{then}\ g_3\ \mathsf{else}\ g_4$
abstracted	$\hat{\gamma}$	::=	$\mathcal{G}[\hat{\gamma},\hat{\sigma}] \mid \mathsf{val}([\![\tau_1 \Longrightarrow \tau_2]\!]) \mid \triangle(\tau)$
deabstracted	γ	::=	$\mathcal{G}[\gamma,\sigma]$
internal types	$\mathcal{S}[s]$	::=	$s_1 \to s_2 \mid \mathbb{Z} \mid s_1 \times s_2$
abstracted	$\hat{\sigma}$	::=	$\mathcal{S}[\hat{\sigma}] \mid \mathsf{rep}(\tau) \mid \blacktriangle(\tau)$
dequoted	$\bar{\sigma}$	::=	$\mathcal{S}[\bar{\sigma}] \mid \mathsf{rep}(\tau)$
deabstracted	σ	::=	$\mathcal{S}[\sigma]$

Figure 2. Syntax of @λ. Variables x are used in expressions and internal terms and $\mathbf{t}$ in type-level terms. Names FAM are family names (we assume that unique family names can be generated by some extrinsic mechanism) and ***op*** for operator names. *"str"* denotes string literals, $\bar{\mathsf{z}}$ denotes integer literals and $\oplus$ stands for binary operations over integers. The metavariables related to the internal language are written using generators $\mathcal{G}$ and $\mathcal{S}$ to avoid duplicating the syntax of common terms.

expressions, are type-level terms of kind $\star$ (following conventional notation [**?**]), but there are also other kinds of type-level terms, including type-level lambda abstraction and other type-level data structures (we include type-level strings, nullary and binary products and primitive lists for the sake of our examples; see Sec. **??** for a discussion of what other kinds of data would be acceptable here). Type-level data and types can be compared for structural equality (i.e. a subset of kinds are comparable, discussed below).

Our calculus then introduces type-level mechanisms for introducing new type families and associated primitive operators. Operators specify their run-time semantics by giving a translation to a typed *internal language*, here based on an expanded variant of Plotkin's PCF [**?**] for the sake of example (see Sec. **??**). The abstract syntax for expressions, types, kinds and the typed internal language (consisting of *internal types* and *internal terms*) is shown in Figure 2. We will now discuss each of these components of the calculus with some examples.

We build in lambdas as the only binding structure in the language to simplify the core calculus. All other operators are associated with a type. For example, nat / tuples / records.

Operator families.

Records:

$$\frac{\overbrace{\emptyset \vdash_{\Sigma_0} \rho\ \texttt{prog}}^{\text{Kind Checking}} \qquad \overbrace{\vdash_{\Phi_0} \rho \Longrightarrow \gamma}^{\text{Active Typechecking and Translation}}}{\rho \longrightarrow \gamma}$$

Figure 3. Central Compilation Judgement of @λ.

$$\begin{array}{rcl} \Sigma_0 & := & \textsc{Arrow}[\star \times \star, \textbf{\textit{ap}}[1]] \\ \Phi_0 & := & \textsc{Arrow}[\theta_0, \mathbf{i}.\blacktriangledown(\mathsf{rep}(\mathsf{fst}(\mathbf{i})) \to \mathsf{rep}(\mathsf{snd}(\mathbf{i})))] \\ \theta_0 & := & \textbf{\textit{ap}}(\mathbf{i}{:}1, \mathbf{a}.x) \end{array}$$

Figure 4. The ARROW type family is included by default as defined above. The definition of XXX, written in terms of fold, has been alided for concision.

$\boxed{\Delta \vdash_\Sigma \rho\ \texttt{prog}}$ $\quad \Delta ::= \emptyset \mid \Delta, \mathbf{t} : \kappa \quad \Sigma ::= \Sigma_0 \mid \Sigma, \textsc{Fam}[\kappa_{\text{idx}}, \Theta]$

FAMILY DECL KINDING
$$\frac{\textsc{Fam} \notin \Sigma \qquad \kappa_{\text{idx}}\ \mathsf{eq} \qquad \Delta \vdash_{\Sigma, \textsc{Fam}[\kappa_{\text{idx}}, \Theta]} \theta : \Theta \qquad \Delta, \mathbf{i} : \kappa_{\text{idx}} \vdash_{\Sigma, \textsc{Fam}[\kappa_{\text{idx}}, \Theta]} \tau : \mathsf{IType} \qquad \Delta \vdash_{\Sigma, \textsc{Fam}[\kappa_{\text{idx}}, \Theta]} \rho\ \texttt{prog}}{\Delta \vdash_\Sigma \mathsf{family}\ \textsc{Fam}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\theta : \Theta \sim \mathbf{i}.\tau\};\ \rho\ \texttt{prog}}$$

TYPE-LEVEL BIND KINDING
$$\frac{\Delta \vdash_\Sigma \tau : \kappa \qquad \Delta, \mathbf{t} : \kappa \vdash_\Sigma \rho\ \texttt{prog}}{\Delta \vdash_\Sigma \mathsf{def}\ \mathbf{t} : \kappa = \tau;\ \rho\ \texttt{prog}}$$

EXP KINDING
$$\frac{\Delta\ \emptyset \vdash_\Sigma e\ \texttt{expr}}{\Delta \vdash_\Sigma e\ \texttt{prog}}$$

$\boxed{\Delta \vdash_\Sigma \theta : \Theta}$

NO OPS
$$\frac{}{\Delta \vdash_\Sigma \cdot : \cdot}$$

OPS
$$\frac{\Delta \vdash_\Sigma \theta : \Theta \qquad \textbf{\textit{op}} \notin \theta \qquad \Delta, \mathbf{i} : \kappa_{\text{i}}, \mathbf{a} : \mathsf{list}[\mathsf{Den}] \vdash_\Sigma \tau : \mathsf{Den}}{\Delta \vdash_\Sigma \theta; \textbf{\textit{op}}(\mathbf{i}{:}\kappa_{\text{i}}, \mathbf{a}.\tau) : \Theta; \textbf{\textit{op}}[\kappa_{\text{i}}]}$$

$\boxed{\Delta\ \Omega \vdash_\Sigma e\ \texttt{expr}}$ $\quad \Omega ::= \emptyset \mid \Omega, x$

E-VAR-KIND
$$\frac{}{\Delta\ \Omega, x \vdash_\Sigma x\ \texttt{expr}}$$

E-LAM-KIND
$$\frac{\Delta \vdash_\Sigma \tau : \star \qquad \Delta\ \Omega, x \vdash_\Sigma e\ \texttt{expr}}{\Delta\ \Omega \vdash_\Sigma \lambda x{:}\tau.e\ \texttt{expr}}$$

E-OP-KIND
$$\frac{\textsc{Fam}[\kappa_{\text{idx}}, \Theta] \in \Sigma \qquad \textbf{\textit{op}}[\kappa_{\text{i}}] \in \Theta \qquad \Delta \vdash_\Sigma \tau_{\text{i}} : \kappa_{\text{i}} \qquad \Delta\ \Omega \vdash_\Sigma e_1\ \texttt{expr} \qquad \cdots \qquad \Delta\ \Omega \vdash_\Sigma e_n\ \texttt{expr}}{\Delta\ \Omega \vdash_\Sigma \textsc{Fam}.\textbf{\textit{op}}\langle\tau_{\text{i}}\rangle(e_1; \ldots; e_n)\ \texttt{expr}}$$

Figure 5. Kinding for programs. Variable contexts Δ and Ω obey standard structural properties. Kinding for type-level terms in Figure **??**.

$\boxed{\vdash_\Phi \rho \Longrightarrow \gamma}$ $\quad \Phi ::= \Phi_0 \mid \Phi, \textsc{Fam}[\theta, \mathbf{i}.\tau]$

ATT-FAM
$$\frac{\vdash_{\Phi, \textsc{Fam}[\theta, \mathbf{i}.\tau]} \rho \Longrightarrow \gamma}{\vdash_\Phi \mathsf{family}\ \textsc{Fam}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\theta : \Theta \sim \mathbf{i}.\tau\};\ \rho \Longrightarrow \gamma}$$

ATT-DEF
$$\frac{\tau \Downarrow \tau' \qquad \vdash_\Phi [\tau'/\mathbf{t}]\rho \Longrightarrow \gamma}{\vdash_\Phi \mathsf{def}\ \mathbf{t} : \kappa = \tau;\ \rho \Longrightarrow \gamma}$$

ATT-EXP
$$\frac{\emptyset \vdash_\Phi e : \tau \Longrightarrow \hat{\gamma} \qquad \vdash_\Phi \hat{\gamma} \rightsquigarrow \gamma}{\vdash_\Phi e \Longrightarrow \gamma}$$

$\boxed{\Gamma \vdash_\Phi e : \tau \Longrightarrow \hat{\gamma}}$ $\quad \Gamma ::= \emptyset \mid \Gamma, x : \tau$

ATT-VAR
$$\frac{}{\Gamma, x : \tau \vdash_\Phi x : \tau \Longrightarrow x}$$

ATT-LAM
$$\frac{\tau_1 \Downarrow \textsc{Fam}\langle\tau_{\text{idx}}\rangle \qquad \Gamma, x : \textsc{Fam}\langle\tau_{\text{idx}}\rangle \vdash_\Phi e : \tau_2 \Longrightarrow \hat{\gamma} \qquad \vdash_\Phi^{\textsc{Arrow}} \textsc{Fam}\langle\tau_{\text{idx}}\rangle \sim \bar{\sigma}}{\Gamma \vdash_\Phi \lambda x{:}\tau_1.e : \textsc{Arrow}\langle(\textsc{Fam}\langle\tau_{\text{idx}}\rangle, \tau_2)\rangle \Longrightarrow \lambda x{:}\bar{\sigma}.\hat{\gamma}}$$

ATT-OP
$$\frac{\begin{array}{c}\textsc{Fam}[\theta, \mathbf{i}.\tau] \in \Phi \qquad \textbf{\textit{op}}(\mathbf{i}{:}\kappa_{\text{i}}, \mathbf{a}.\tau_{\text{op}}) \in \theta \qquad \tau_{\text{i}} \Downarrow \tau_{\text{i}}' \\ \Gamma \vdash_\Phi e_1 : \tau_1 \Longrightarrow \hat{\gamma}_1 \quad \cdots \quad \Gamma \vdash_\Phi e_n : \tau_n \Longrightarrow \hat{\gamma}_n \\ \left\{ \begin{array}{c} \left[\begin{array}{c} \tau_{\text{i}}'/\mathbf{i} \\ [\![\tau_1 \Longrightarrow \triangledown(\hat{\gamma}_1)]\!] :: \cdots :: [\![\tau_n \Longrightarrow \triangledown(\hat{\gamma}_n)]\!] :: []_{\mathsf{Den}}/\mathbf{a} \end{array} \right] \tau_{\text{op}} \\ \Downarrow \\ [\![\textsc{Fam}'\langle\tau_{\text{idx}}\rangle \Longrightarrow \triangledown(\hat{\gamma})]\!] \end{array} \right\} \\ \vdash_\Phi^{\textsc{Fam}} \textsc{Fam}'\langle\tau_{\text{idx}}\rangle \sim \bar{\sigma} \qquad \vdash_\Phi^{\textsc{Fam}} \Gamma \sim \Psi \qquad \Psi \vdash_\Phi^{\textsc{Fam}} \hat{\gamma} \sim \bar{\sigma}\end{array}}{\Gamma \vdash_\Phi \textsc{Fam}.\textbf{\textit{op}}\langle\tau_{\text{i}}\rangle(e_1; \ldots; e_n) : \textsc{Fam}'\langle\tau_{\text{idx}}\rangle \Longrightarrow \hat{\gamma}}$$

Figure 6. Active typechecking and translation

3.2 Statics

4. Safety of @λ

4.1 Dependent Kinds

5. Active Typechecking and Translation in Ace

6. Related Work

6.1 Type-Level Computation

System XX with simple case analysis provides the basis of type-level computation in Haskell (where type-level functions are called type families [1]). Ur uses type-level records and names to support typesafe metaprogramming, with applications to web programming [3]. Ωmega adds algebraic data types at the type-level, using these to increase the expressive power of algebraic data types at the expression level [10]. Dependently-typed languages blur the traditional phase separation between types and expressions, so type-level computation is often implicitly used (though not always in its most general form, e.g. Deputy [2], ATS [4].)

6.2 Run-Time Indirection

Operator overloading [13] and *metaobject dispatch* [7] are run-time protocols that translate operator invocations into function calls. The function is typically selected according to the type or value of one or more operands. These protocols share the notion of *inversion of control* with type-level specification. However, type-level specification is a *compile-time* protocol focused on enabling specialized verification and implementation strategies, rather than simply enabling run-time indirection.

6.3 Term Rewriting Systems

Many languages and tools allow developers to rewrite expressions according to custom rules. These can broadly be classified as *term rewriting systems*. Macro systems, such as those characteristic of the LISP family of languages [8], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [9]), and external rewrite systems also exist for many languages. These facilities differ from type-level specification in one or more of the following ways:

1. In type-level specification, the type of a value is determined separately from its representation; in fact, the same representation may be generated by multiple types.
2. We draw a distinction between the metalanguage, used to specify types and compile-time logic, the source grammar, used to describe run-time behavior, and the internal language, used to implement this behavior. Term rewriting systems generally do not draw this distinction. By doing so, each component language can be structured and constrained as appropriate for its distinct role, as we show.

$\boxed{\Delta \vdash_\Sigma \tau : \kappa}$

$$\text{VAR KIND}\ \frac{}{\Delta, \mathbf{t} : \kappa \vdash_\Sigma \mathbf{t} : \kappa} \qquad \text{K-ARROW INTRO}\ \frac{\Delta, \mathbf{t} : \kappa_1 \vdash_\Sigma \tau : \kappa_2}{\Delta \vdash_\Sigma \lambda \mathbf{t}{:}\kappa_1.\tau : \kappa_1 \to \kappa_2}$$

$$\text{K-ARROW ELIM}\ \frac{\Delta \vdash_\Sigma \tau_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash_\Sigma \tau_2 : \kappa_1}{\Delta \vdash_\Sigma \tau_1\ \tau_2 : \kappa_2}$$

(standard statics for integers, strings, products and lists omitted)

$$\text{EQ KIND}\ \frac{\kappa\ \mathsf{eq} \qquad \Delta \vdash_\Sigma \tau_1 : \kappa \qquad \Delta \vdash_\Sigma \tau_2 : \kappa \qquad \Delta \vdash_\Sigma \tau_3 : \kappa' \qquad \Delta \vdash_\Sigma \tau_4 : \kappa'}{\Delta \vdash_\Sigma \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 : \kappa'} \qquad \text{TYPE INTRO}\ \frac{\textsc{Fam}[\kappa_{\text{idx}}, \Theta] \in \Sigma \qquad \Delta \vdash_\Sigma \tau_{\text{idx}} : \kappa_{\text{idx}}}{\Delta \vdash_\Sigma \textsc{Fam}\langle \tau_{\text{idx}} \rangle : \star}$$

$$\text{TYPE ELIM}\ \frac{\textsc{Fam}[\kappa_{\text{idx}}, \Theta] \in \Sigma \qquad \Delta \vdash_\Sigma \tau : \star \qquad \Delta, \mathbf{x} : \kappa_{\text{idx}} \vdash_\Sigma \tau_0 : \kappa \qquad \Delta \vdash_\Sigma \tau_1 : \kappa}{\Delta \vdash_\Sigma \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle \mathbf{x} \rangle \Rightarrow \tau_0\ \mathsf{ow}\ \tau_1 : \kappa}$$

$$\text{DEN INTRO VALID}\ \frac{\Delta \vdash_\Sigma \tau_\star : \star \qquad \Delta \vdash_\Sigma \tau_{\text{val}} : \mathsf{ITerm}}{\Delta \vdash_\Sigma [\![\tau_\star \Longrightarrow \tau_{\text{val}}]\!] : \mathsf{Den}} \qquad \text{DEN INTRO ERR}\ \frac{}{\Delta \vdash_\Sigma \mathsf{err} : \mathsf{Den}}$$

$$\text{DEN ELIM}\ \frac{\Delta \vdash_\Sigma \tau_{\text{den}} : \mathsf{Den} \qquad \Delta, \mathbf{x} : \mathsf{ITerm}, \mathbf{y} : \star \vdash_\Sigma \tau_1 : \kappa \qquad \Delta \vdash_\Sigma \tau_2 : \kappa}{\Delta \vdash_\Sigma \mathsf{case}\ \tau_{\text{den}}\ \mathsf{of}\ [\![\mathbf{x} \Longrightarrow \mathbf{y}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 : \kappa}$$

$$\text{ITERM INTRO}\ \frac{\Delta \emptyset \vdash_\Sigma \hat{\gamma}\ \mathtt{iterm}}{\Delta \vdash_\Sigma \triangledown(\hat{\gamma}) : \mathsf{ITerm}} \qquad \text{ITYPE INTRO}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}\ \mathtt{itype}}{\Delta \vdash_\Sigma \blacktriangledown(\hat{\sigma}) : \mathsf{IType}}$$

$\boxed{\kappa\ \mathsf{eq}}$

$$\text{T-EQ}\ \frac{}{\star\ \mathsf{eq}} \qquad \text{Z-EQ}\ \frac{}{\mathbb{Z}\ \mathsf{eq}} \qquad \text{STR-EQ}\ \frac{}{\mathsf{Str}\ \mathsf{eq}} \qquad \text{U-EQ}\ \frac{}{1\ \mathsf{eq}} \qquad \text{PROD-EQ}\ \frac{\kappa_1\ \mathsf{eq} \qquad \kappa_2\ \mathsf{eq}}{\kappa_1 \times \kappa_2\ \mathsf{eq}} \qquad \text{LIST-EQ}\ \frac{\kappa\ \mathsf{eq}}{\mathsf{list}[\kappa]\ \mathsf{eq}}$$

$\boxed{\Delta\ \Omega \vdash_\Sigma \hat{\gamma}\ \mathtt{iterm}}$

$$\text{I-VAR KIND}\ \frac{}{\Delta\ \Omega, x \vdash_\Sigma x\ \mathtt{iterm}} \qquad \text{I-LAM KIND}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}\ \mathtt{itype} \qquad \Delta\ \Omega, x \vdash_\Sigma \hat{\gamma}\ \mathtt{iterm}}{\Delta\ \Omega \vdash_\Sigma \lambda x{:}\hat{\sigma}.\hat{\gamma}\ \mathtt{iterm}}$$

$$\text{I-FIX KIND}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}\ \mathtt{itype} \qquad \Delta\ \Omega, x \vdash_\Sigma \hat{\gamma}\ \mathtt{iterm}}{\Delta\ \Omega \vdash_\Sigma \mathsf{fix}\ f{:}\hat{\sigma}\ \mathsf{is}\ \hat{\gamma}\ \mathtt{iterm}}$$

(omitted forms have trivially recursive rules)

$$\text{ITERM UNQUOTE KIND}\ \frac{\Delta \vdash_\Sigma \tau : \mathsf{ITerm}}{\Delta\ \Omega \vdash_\Sigma \triangle(\tau)\ \mathtt{iterm}} \qquad \text{VAL FROM DEN KIND}\ \frac{\Delta \vdash_\Sigma \tau_\star : \star \qquad \Delta \vdash_\Sigma \tau_{\text{val}} : \mathsf{ITerm}}{\Delta\ \Omega \vdash_\Sigma \mathsf{val}([\![\tau_\star \Longrightarrow \tau_{\text{val}}]\!])\ \mathtt{iterm}}$$

$\boxed{\Delta \vdash_\Sigma \hat{\sigma}\ \mathtt{itype}}$

$$\text{I-INT KIND}\ \frac{}{\Delta \vdash_\Sigma \mathbb{Z}\ \mathtt{itype}} \qquad \text{I-PROD KIND}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}_1\ \mathtt{itype} \qquad \Delta \vdash_\Sigma \hat{\sigma}_2\ \mathtt{itype}}{\Delta \vdash_\Sigma \hat{\sigma}_1 \times \hat{\sigma}_2\ \mathtt{itype}}$$

$$\text{I-ARROW KIND}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}_1\ \mathtt{itype} \qquad \Delta \vdash_\Sigma \hat{\sigma}_2\ \mathtt{itype}}{\Delta \vdash_\Sigma \hat{\sigma}_1 \to \hat{\sigma}_2\ \mathtt{itype}}$$

$$\text{ITYPE UNQUOTE KIND}\ \frac{\Delta \vdash_\Sigma \tau : \mathsf{IType}}{\Delta \vdash_\Sigma \blacktriangle(\tau)\ \mathtt{itype}} \qquad \text{REP FROM TYPE KIND}\ \frac{\Delta \vdash_\Sigma \tau : \star}{\Delta \vdash_\Sigma \mathsf{rep}(\tau)\ \mathtt{itype}}$$

Figure 7. Kinding for type-level terms

$\boxed{\tau \Downarrow \tau'}$

$$\text{TL-LAM EVAL}\ \frac{}{\lambda \mathbf{t}{:}\kappa.\tau \Downarrow \lambda \mathbf{t}{:}\kappa.\tau} \qquad \text{TL-AP EVAL}\ \frac{\tau_1 \Downarrow \lambda \mathbf{t}{:}\kappa.\tau \qquad \tau_2 \Downarrow \tau_2' \qquad [\tau_2'/\mathbf{t}]\tau \Downarrow \tau'}{\tau_1\ \tau_2 \Downarrow \tau'}$$

(standard evaluation rules for integers, strings, products and lists omitted)

$$\text{TL-EQ EVAL EQ}\ \frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_1' \qquad \tau_3 \Downarrow \tau_3'}{\mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \Downarrow \tau_3'}$$

$$\text{TL-EQ EVAL NEQ}\ \frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_2' \qquad \tau_1' \neq \tau_2' \qquad \tau_4 \Downarrow \tau_4'}{\mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \Downarrow \tau_4'}$$

$$\text{TYPE EVAL}\ \frac{\tau_{\text{idx}} \Downarrow \tau_{\text{idx}}'}{\textsc{Fam}\langle \tau_{\text{idx}} \rangle \Downarrow \textsc{Fam}\langle \tau_{\text{idx}}' \rangle} \qquad \text{FAMCASE EVAL MATCH}\ \frac{\tau \Downarrow \textsc{Fam}\langle \tau_{\text{idx}} \rangle \qquad [\tau_{\text{idx}}/\mathbf{x}]\tau_1 \Downarrow \tau_1'}{\mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle \mathbf{x} \rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_1'}$$

$$\text{FAMCASE EVAL FAIL}\ \frac{\tau \Downarrow \textsc{Fam'}\langle \tau_{\text{idx}} \rangle \qquad \textsc{Fam} \neq \textsc{Fam'} \qquad \tau_2 \Downarrow \tau_2'}{\mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle \mathbf{x} \rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_2'}$$

$$\text{DEN EVAL}\ \frac{\tau_\star \Downarrow \tau_\star' \qquad \tau_{\text{val}} \Downarrow \tau_{\text{val}}'}{[\![\tau_\star \Longrightarrow \tau_{\text{val}}]\!] \Downarrow [\![\tau_\star' \Longrightarrow \tau_{\text{val}}']\!]} \qquad \text{ERR EVAL}\ \frac{}{\mathsf{err} \Downarrow \mathsf{err}}$$

$$\text{DENCASE EVAL VALID}\ \frac{\tau_{\text{den}} \Downarrow [\![\tau_\star \Longrightarrow \tau_{\text{val}}]\!] \qquad [\tau_\star/\mathbf{x}, \triangledown(\mathsf{val}([\![\tau_\star \Longrightarrow \tau_{\text{val}}]\!]))/\mathbf{y}]\tau_1 \Downarrow \tau_1'}{\mathsf{case}\ \tau_{\text{den}}\ \mathsf{of}\ [\![\mathbf{x} \Longrightarrow \mathbf{y}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_1'}$$

$$\text{DENCASE EVAL ERR}\ \frac{\tau_{\text{den}} \Downarrow \mathsf{err} \qquad \tau_2 \Downarrow \tau_2'}{\mathsf{case}\ \tau_{\text{den}}\ \mathsf{of}\ [\![\mathbf{x} \Longrightarrow \mathbf{y}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_2'}$$

$$\text{ITERM QUOTE EVAL}\ \frac{\hat{\gamma} \curlyvee \hat{\gamma}'}{\triangledown(\hat{\gamma}) \Downarrow \triangledown(\hat{\gamma}')} \qquad \text{ITYPE QUOTE EVAL}\ \frac{\hat{\sigma} \curlyvee \hat{\sigma}'}{\blacktriangledown(\hat{\sigma}) \Downarrow \blacktriangledown(\hat{\sigma}')}$$

$\boxed{\hat{\gamma} \curlyvee \hat{\gamma}'}$

$$\text{I-VAR EVAL}\ \frac{}{x \curlyvee x} \qquad \text{I-LAM EVAL}\ \frac{\hat{\sigma} \curlyvee \hat{\sigma}' \qquad \hat{\gamma} \curlyvee \hat{\gamma}'}{\lambda x{:}\hat{\sigma}.\hat{\gamma} \curlyvee \lambda x{:}\hat{\sigma}'.\hat{\gamma}'} \qquad \text{I-FIX EVAL}\ \frac{\hat{\sigma} \curlyvee \hat{\sigma}' \qquad \hat{\gamma} \curlyvee \hat{\gamma}'}{\mathsf{fix}\ f{:}\hat{\sigma}\ \mathsf{is}\ \hat{\gamma} \curlyvee \mathsf{fix}\ f{:}\hat{\sigma}'\ \mathsf{is}\ \hat{\gamma}'}$$

(omitted forms have trivially recursive rules)

$$\text{ITERM UNQUOTE EVAL}\ \frac{\tau \Downarrow \tau'}{\triangle(\tau) \curlyvee \triangle(\tau')} \qquad \text{VAL FROM DEN EVAL}\ \frac{\tau_\star \Downarrow \tau_\star' \qquad \tau_{\text{val}} \Downarrow \tau_{\text{val}}'}{\mathsf{val}([\![\tau_\star \Longrightarrow \tau_{\text{val}}]\!]) \curlyvee \mathsf{val}([\![\tau_\star' \Longrightarrow \tau_{\text{val}}']\!])}$$

$\boxed{\sigma \curlyvee \sigma'}$

$$\text{I-INT EVAL}\ \frac{}{\mathbb{Z} \curlyvee \mathbb{Z}} \qquad \text{I-PROD EVAL}\ \frac{\hat{\sigma}_1 \curlyvee \hat{\sigma}_1' \qquad \hat{\sigma}_2 \curlyvee \hat{\sigma}_2'}{\hat{\sigma}_1 \times \hat{\sigma}_2 \curlyvee \hat{\sigma}_1' \times \hat{\sigma}_2'} \qquad \text{I-ARROW EVAL}\ \frac{\hat{\sigma}_1 \curlyvee \hat{\sigma}_1' \qquad \hat{\sigma}_2 \curlyvee \hat{\sigma}_2'}{\hat{\sigma}_1 \to \hat{\sigma}_2 \curlyvee \hat{\sigma}_1' \to \hat{\sigma}_2'}$$

$$\text{ITYPE UNQUOTE EVAL}\ \frac{\tau \Downarrow \tau'}{\blacktriangle(\tau) \curlyvee \blacktriangle(\tau')} \qquad \text{REP FROM TYPE EVAL}\ \frac{\tau \Downarrow \tau'}{\mathsf{rep}(\tau) \curlyvee \mathsf{rep}(\tau')}$$

Figure 8. Evaluation semantics for type-level terms

$\boxed{\vdash^{\text{FAM}}_{\Phi} \tau \sim \bar{\sigma}}$ $\quad \Phi ::= \Phi_0 \mid \Phi, \text{FAM}[\theta, \mathbf{i}.\tau]$

ABS REP FROM TYPE

$$\frac{\text{FAM'}[\theta, \mathbf{i}.\tau] \in \Phi \qquad [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\hat{\sigma}) \qquad \vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma}}{\vdash^{\text{FAM}}_{\Phi} \text{FAM'}\langle\tau_{\text{idx}}\rangle \sim \bar{\sigma}}$$

$\boxed{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma}}$

ABS INT

$$\frac{}{\vdash^{\text{FAM}}_{\Phi} \mathbb{Z} \sim \mathbb{Z}}$$

ABS ARROW

$$\frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \sim \bar{\sigma}_1 \qquad \vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_2 \sim \bar{\sigma}_2}{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \to \hat{\sigma}_2 \sim \bar{\sigma}_1 \to \bar{\sigma}_2}$$

ABS PROD

$$\frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \sim \bar{\sigma}_1 \qquad \vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_2 \sim \bar{\sigma}_2}{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \times \hat{\sigma}_2 \sim \bar{\sigma}_1 \times \bar{\sigma}_2}$$

ABS+CANCEL UNQUOTE

$$\frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma}}{\vdash^{\text{FAM}}_{\Phi} \blacktriangle(\blacktriangledown(\hat{\sigma})) \sim \bar{\sigma}}$$

ABS REP FROM TYPE VISIBLE

$$\frac{\text{FAM}[\theta, \mathbf{i}.\tau] \in \Phi \qquad [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\hat{\sigma}) \qquad \vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma}}{\vdash^{\text{FAM}}_{\Phi} \mathsf{rep}(\text{FAM}\langle\tau_{\text{idx}}\rangle) \sim \bar{\sigma}}$$

ABS REP FROM TYPE HIDDEN

$$\frac{\text{FAM} \neq \text{FAM'}}{\vdash^{\text{FAM}}_{\Phi} \mathsf{rep}(\text{FAM'}\langle\tau_{\text{idx}}\rangle) \sim \mathsf{rep}(\text{FAM'}\langle\tau_{\text{idx}}\rangle)}$$

$\boxed{\vdash^{\text{FAM}}_{\Phi} \Gamma \sim \Psi}$ $\quad \Psi ::= \emptyset \mid \Psi, x : \bar{\sigma}$

ABS EMPTY

$$\frac{}{\vdash^{\text{FAM}}_{\Phi} \emptyset \sim \emptyset}$$

ABS CTX

$$\frac{\vdash^{\text{FAM}}_{\Phi} \Gamma \sim \Psi \qquad \vdash^{\text{FAM}}_{\Phi} \tau \sim \bar{\sigma}}{\vdash^{\text{FAM}}_{\Phi} \Gamma, x : \tau \sim \Psi, x : \bar{\sigma}}$$

$\boxed{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma} \sim \bar{\sigma}}$

ABS I-VAR

$$\frac{}{\Psi, x : \bar{\sigma} \vdash^{\text{FAM}}_{\Phi} x \sim \bar{\sigma}}$$

ABS I-LAM

$$\frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \sim \bar{\sigma}_1 \qquad \Psi, x : \bar{\sigma}_1 \vdash^{\text{FAM}}_{\Phi} \gamma \sim \bar{\sigma}_2}{\Psi \vdash^{\text{FAM}}_{\Phi} \lambda x{:}\hat{\sigma}_1.\gamma \sim \bar{\sigma}_1 \to \bar{\sigma}_2}$$

ABS I-AP

$$\frac{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_1 \sim \bar{\sigma}_1 \to \bar{\sigma}_2 \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_2 \sim \bar{\sigma}_1}{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_1\ \hat{\gamma}_2 \sim \bar{\sigma}_2}$$

ABS I-FIX

$$\frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma} \qquad \Psi, x : \bar{\sigma} \vdash^{\text{FAM}}_{\Phi} \hat{\gamma} \sim \bar{\sigma}}{\Psi \vdash^{\text{FAM}}_{\Phi} \mathsf{fix}\ x{:}\hat{\sigma}\ \mathsf{is}\ \hat{\gamma} \sim \bar{\sigma}}$$

(standard statics for integers and products omitted)

ABS IF EQ

$$\frac{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_1 \sim \mathbb{Z} \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_2 \sim \mathbb{Z} \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_3 \sim \bar{\sigma} \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_4 \sim \bar{\sigma}}{\Psi \vdash^{\text{FAM}}_{\Phi} \mathsf{if}\ \hat{\gamma}_1 \equiv_{\mathbb{Z}} \hat{\gamma}_2\ \mathsf{then}\ \hat{\gamma}_3\ \mathsf{else}\ \hat{\gamma}_4 \sim \bar{\sigma}}$$

ABS ITERM UNQUOTE

$$\frac{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma} \sim \bar{\sigma}}{\Psi \vdash^{\text{FAM}}_{\Phi} \triangle(\triangledown(\hat{\gamma})) \sim \bar{\sigma}}$$

ABS VAL FROM DEN VISIBLE

$$\frac{\vdash^{\text{FAM}}_{\Phi} \text{FAM}\langle\tau_{\text{idx}}\rangle \sim \bar{\sigma} \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma} \sim \bar{\sigma}}{\Psi \vdash^{\text{FAM}}_{\Phi} \mathsf{val}([\![\text{FAM}\langle\tau_{\text{idx}}\rangle \Longrightarrow \triangledown(\hat{\gamma})]\!]) \sim \bar{\sigma}}$$

ABS VAL FROM DEN HIDDEN

$$\frac{\text{FAM} \neq \text{FAM'} \qquad \vdash^{\text{FAM}}_{\Phi} \text{FAM'}\langle\tau_{\text{idx}}\rangle \sim \bar{\sigma} \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma} \sim \bar{\sigma}}{\Psi \vdash^{\text{FAM}}_{\Phi} \mathsf{val}([\![\text{FAM'}\langle\tau_{\text{idx}}\rangle \Longrightarrow \triangledown(\hat{\gamma})]\!]) \sim \mathsf{rep}(\text{FAM'}\langle\tau_{\text{idx}}\rangle)}$$

Figure 9. Abstracted internal typing

$\boxed{\vdash_{\Phi} \bar{\sigma} \rightsquigarrow \sigma}$

DEABS INT

$$\frac{}{\vdash_{\Phi} \mathbb{Z} \rightsquigarrow \mathbb{Z}}$$

DEABS ARROW

$$\frac{\vdash_{\Phi} \bar{\sigma}_1 \rightsquigarrow \sigma_1 \qquad \vdash_{\Phi} \bar{\sigma}_2 \rightsquigarrow \sigma_2}{\vdash_{\Phi} \bar{\sigma}_1 \to \bar{\sigma}_2 \rightsquigarrow \sigma_1 \to \sigma_2}$$

DEABS PROD

$$\frac{\vdash_{\Phi} \bar{\sigma}_1 \rightsquigarrow \sigma_1 \qquad \vdash_{\Phi} \bar{\sigma}_2 \rightsquigarrow \sigma_2}{\vdash_{\Phi} \bar{\sigma}_1 \times \bar{\sigma}_2 \rightsquigarrow \sigma_1 \times \sigma_2}$$

DEABS REP FROM TYPE HIDDEN

$$\frac{\text{FAM}[\theta, \mathbf{i}.\tau] \in \Phi \qquad [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\hat{\sigma}) \qquad \vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma} \qquad \vdash_{\Phi} \bar{\sigma} \rightsquigarrow \sigma}{\vdash_{\Phi} \mathsf{rep}(\text{FAM}\langle\tau_{\text{idx}}\rangle) \rightsquigarrow \sigma}$$

$\boxed{\vdash_{\Phi} \hat{\gamma} \rightsquigarrow \gamma}$

DEABS I-VAR

$$\frac{}{\vdash_{\Phi} x \rightsquigarrow x}$$

DEABS I-LAM

$$\frac{\vdash^{\sim}_{\Phi} \hat{\sigma} \sim \bar{\sigma} \qquad \vdash_{\Phi} \bar{\sigma} \rightsquigarrow \sigma \qquad \vdash_{\Phi} \hat{\gamma} \rightsquigarrow \gamma}{\vdash_{\Phi} \lambda x{:}\hat{\sigma}.\hat{\gamma} \rightsquigarrow \lambda x{:}\sigma.\gamma}$$

DEABS I-FIX

$$\frac{\vdash^{\sim}_{\Phi} \hat{\sigma} \sim \bar{\sigma} \qquad \vdash_{\Phi} \bar{\sigma} \rightsquigarrow \sigma \qquad \vdash_{\Phi} \hat{\gamma} \rightsquigarrow \gamma}{\vdash_{\Phi} \mathsf{fix}\ x{:}\hat{\sigma}\ \mathsf{is}\ \hat{\gamma} \rightsquigarrow \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \gamma}$$

(omitted forms have trivially recursive rules)

DEABS UNQUOTE

$$\frac{\vdash_{\Phi} \hat{\gamma} \rightsquigarrow \gamma}{\vdash_{\Phi} \triangle(\triangledown(\hat{\gamma})) \rightsquigarrow \gamma}$$

DEABS VAL FROM DEN HIDDEN

$$\frac{\vdash_{\Phi} \hat{\gamma} \rightsquigarrow \gamma}{\vdash_{\Phi} \mathsf{val}([\![\text{FAM}\langle\tau_{\text{idx}}\rangle \Longrightarrow \triangledown(\hat{\gamma})]\!]) \rightsquigarrow \gamma}$$

Figure 10. Deabstraction rules

3. Many common macro systems and metaprogramming facilities operate at run-time. Compilers for some forms of LISP employ aggressive compile-time specialization techniques to attempt to minimize this overhead. Static and staged term-rewriting systems also exist (e.g. OpenJava[12], Template Haskell[11], MetaML [9] and others).

6.4 Language Frameworks

When the mechanisms available in an existing language prove insufficient, researchers and domain experts must design a new language. A number of tools have been developed to assist with this task, including compiler generators, language workbenches and domain-specific language frameworks (cf [6]).

A major barrier to adoption is the fact that interoperability is intrinsically problematic. Even languages which target a common platform, such as the Java Virtual Machine, can only interact using its limited set of primitives. Specialized typing rules are not checked at language boundaries, performance often suffers, and the syntax can be unnatural, particularly for languages which differ significantly from the platform's native language (e.g. Java).

Instead of focusing on defining standalone languages, type-level specification gives greater responsibility in a granular manner to libraries. In this way, a range of constructs can coexist within the same program and, assuming that it can be shown by some method that various constructs are safely composable, be mixed and matched. The main limitation is that the protocol requires defining a fixed source grammar, whereas a specialized language has considerable flexibility in that regard. Nevertheless, as Ace shows, a simple grammar can be used quite flexibly.

6.5 Extensible Compilers

An alternative methodology is to implement language features granularly as compiler extensions. As discussed in Section 1, existing designs suffer from the same problems related to composability, modularity, safety and security as extensible languages, while also adding the issue of language fragmentation.

Type-level specification can in fact be implemented within a compiler, rather than provided as a core language feature. This would resolve some of the issues, as described in this paper. However, by leveraging type-level computation to integrate the protocol directly into the language, we benefit from common module systems and other shared infrastructure. We also avoid the fragmentation issue.

6.6 Specification Languages

Several *specification languages* (or *logical frameworks*) based on these theoretical formulations exist, including the OBJ family of languages (e.g. CafeOBJ [5]). They provide support for verifying a program against a language specification, and can automatically execute these programs as well in some cases. The language itself specifies which verification and execution strategies are used.

Type-determined compilation takes a more concrete approach to the problem, focusing on combining *implementations* of different logics, rather than simply their specifications. In other words, it focuses on combining *type checkers* and *implementation strategies* rather than more abstract representations of a language's type system and dynamic semantics. In Section 4, we outlined a preliminary approach based on proof assistant available for the type-level language to unify these approaches, and we hope to continue this line of research in future work.

CANT GUARANTEE THAT SPECIFICATIONS ARE ACTUALLY DECIDABLE

7. Discussion

AT&T focuses on functional extensions of a language's type-checker and translator, rather than declarative extensions. Techniques for automatically generating functional implementations from declarative specifications have been developed in the past and we believe that these can also target our mechanism, but we will leave this also as future work.

References

[1] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. *ACM SIGPLAN Notices*, 40(1):1–13, Jan. 2005.

[2] C. Chen and H. Xi. Combining programming with theorem proving. In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 66–77. ACM, 2005.

[3] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In B. G. Zorn and A. Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 122–133. ACM, 2010.

[4] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In R. D. Nicola, editor, *Proceedings of the 16th European Symposium on Programming, ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2007.

[5] R. Diaconescu and K. Futatsugi. Logical foundations of CafeOBJ. *Theoretical Computer Science*, 2001. This volume.

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

[7] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[8] J. McCarthy. History of lisp. In *History of programming languages I*, pages 173–185. ACM, 1978.

[9] T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–??, 1999.

[10] T. Sheard and N. Linger. Programming in omega. In Z. Horváth, R. Plasmeijer, A. Soós, and V. Zsók, editors, *CEFP*, volume 5161 of *Lecture Notes in Computer Science*, pages 158–227. Springer, 2007.

[11] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[12] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for java. In *Reflection and Software Engineering, Papers from OORaSE 1999, 1st OOPSLA Workshop on Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer Verlag, Denver, Colorado, USA, 2000. http://www.csg.is.titech.ac.jp/∼mich/openjava/papers/mich_2000lncs1826.pdf.

[13] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language algol 68. *Acta Informatica*, 5:1–236, 1975.

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