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

Cyrus Omar and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15213, USA
{comar,aldrich}@cs.cmu.edu

Abstract. Researchers and domain experts often propose new language primitives as extensions to the semantics of an existing language. But today's statically-typed languages are monolithic: they do not expose language-internal mechanisms for implementing the static and dynamic semantics of new primitive types and their associated operators directly, so these experts must instead create new standalone languages. This causes problems for potential users because building applications from components written in many different languages can be both unsafe and unnatural. An internally-extensible language could address these issues, but designing a mechanism that is expressive while maintaining safety remains a challenge. Extensions must be modularly verified, their use in any combination must not weaken the metatheoretic properties of the language, nor can they interfere with one another. We introduce a mechanism called active type-checking and translation (AT&T) that aims to directly address these issues while remaining highly expressive. AT&T leverages type-level computation, typed compilation techniques and a form of type abstraction to enable library-based implementations of a variety of primitives over a flexible grammar in a safe manner.

Keywords: extensible languages; active libraries; typed compilation; type-level computation; type abstraction

1 Motivation

When designing and implementing a new abstraction, experts typically begin by attempting to define its new constructs in terms of existing language constructs. By leveraging the general-purpose abstraction mechanisms available in modern languages, such as inductive datatypes and object systems, this approach can be quite effective. For example, the Delite framework leverages Scala's powerful general-purpose mechanisms to enable a variety of interesting *embedded domain-specific languages* [**?**]. Unfortunately, there remain situations of interest where general-purpose abstractions fall short. For instance, it is difficult to adequately encode advanced type systems in terms of the simpler rules governing general-purpose abstractions (e.g. reasoning about units of measure requires built-in language support in F# [**?**]). Moreover, an encoding must not only be correct, but also sufficiently concise and natural. Regular expressions encoded

using inductive datatypes, for example, are often considered overly verbose, so most functional languages support them via strings, which is less safe. Finally, general-purpose abstractions are implemented in a uniform manner. Domain knowledge is thus not applied to eliminate overhead or perform optimizations, and implementations designed for typical application workloads may not be satisfactory in parts of a program where performance is a key criteria, particularly when targeting heterogeneous hardware platforms (e.g. programmable GPUs) and distributed computing resources.

sentence about custom error messages

Researchers or domain experts who run into situations like these, where more direct control over a language's semantics and implementation are needed, have little choice but to realize new abstractions by creating a new language of some form. They might develop a new standalone language from scratch, modify an implementation of an existing language, or use tools like compiler generators, DSL frameworks and language workbenches [6]. The increasing sophistication and ease-of-use of these tools have led to calls for a *language-oriented approach* to software development, where different components of an application are written in different specialized languages [14]. Indeed, a number of software ecosystems are now explicitly designed to support many different languages, both general-purpose and domain-specific, atop a common intermediate language. The Java virtual machine (JVM), the Common Language Infrastructure (CLI) and LLVM are prominent examples of such ecosystems.

Unfortunately, this leads to a critical problem at language boundaries: a library's external interface must only use constructs that can reasonably be expressed in *all possible client languages*. This restricts the design of languages by precluding constructs that rely on statically-checked invariants stronger than those supported by their underlying implementation in the common intermediate language. At best, constructs like these can be exposed by generating a wrapper where run-time checks have been inserted to guarantee these invariants. This compromises both verifiability and performance. Moreover, this approach exposes the internals of an implementation to clients, making the abstraction awkward to work with and causing code breakage when implementation details change. This defeats a primary purpose of high-level programming languages: hiding low-level details from clients of an abstraction. We diagram this fundamental *interoperability problem* in Figure 1(a).

F# example?

Internally-extensible programming languages promise to avoid these problems by providing researchers and domain experts with a mechanism for implementing the semantics of new primitive constructs directly within libraries. Because primitive constructs are granularly available, rather than packaged into monolithic collections, clients can simply import any necessary primitive constructs when using code that relies on them, and thus achieve full safety, ease-of-use and performance without requiring wrappers or glue code. Providers of components thus need only consider whether primitives that they use are appropriate for their domain, without also considering whether their code might be used in a context where these primitives are not available. We diagram this competing approach in Figure 1(b).

mention active libraries

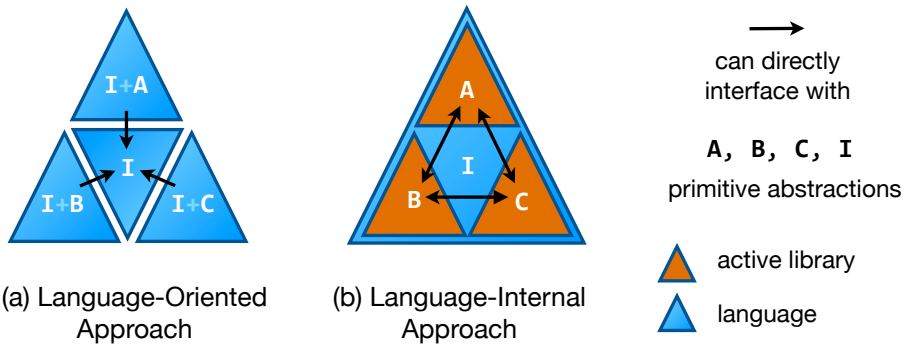

Fig. 1. (a) With the language-oriented approach, different primitive abstractions are packaged into separate languages that extend and target a common intermediate language (e.g. JVM bytecode). Users can only interface with libraries written in another language via the constructs in the common language, causing *interoperability problems*. (b) With the language-internal approach, the semantics of new abstractions (i.e. the logic that governs typechecking and translation to a fixed internal language, here labeled `I`) can be implemented directly within so-called *active libraries*. Clients can import and use these abstractions directly whenever needed.

For a language-internal extension mechanism to be feasible, however, it must achieve expressiveness while also ensuring that extensions cannot compromise the safety properties of the language and its tools, nor interfere with one another. That is, extensions cannot simply be permitted to add arbitrary logic to the type system or compiler, because this would make it possible to break type safety, decidability or adequacy theorems that are critical to the operation of the language, the compiler or other extensions. We review some previous attempts at language extensibility, and highlight how they do not adequately achieve both safety and expressiveness, in Section 7.

In this paper, we introduce a language-internal extensibility mechanism called *active typechecking and translation* (AT&T) that allows developers to introduce and implement the logic governing new primitive type families and operators from within libraries. We argue that this can be accomplished by enriching the type-level language, rather than introducing a separate metalanguage into the system. To make this proposal concrete, we begin by introducing a simple core calculus, called @λ (for the "actively-typed lambda calculus"), in Section 3. This calculus uses type-level computation of higher kind, along with techniques borrowed from the typed compilation literature and a form of type abstraction that ensures that the implementation details of an extension are not externally visible to guarantee the safety of the language, the decidability of typechecking and compilation and composability of extensions.

In Section 5, we show that despite these constraints, this mechanism is expressive enough to admit, within libraries, a number of general-purpose and domain specific abstractions that normally require built-in language support. Our core calculus uses a uniform abstract syntax for primitive operators to sim-

plify our presentation and analysis, but this syntax would be too verbose to be practical. Thus, we begin this section by showing how a key design choice made in the calculus – to associate operators with type families, forming what we call *active types* – enables a novel type-directed desugaring mechanism that permits the use of conventional concrete syntax for language extensions.

Our choice of a simply-typed, simply-kinded calculus where expressions are given meaning by translation to a simply-typed internal language appears to occupy a "sweet spot" in the design space, and relates closely to how simply-typed functional languages like ML and Haskell are specified and implemented today. In Section 6, we briefly discuss other points in the design space of actively-typed languages and describe the sorts of abstractions that the mechanism as we have introduced it is not capable of expressing, suggesting several directions for future research. We conclude with a discussion of related work in Section 7.

2 From Extensible Compilers to Extensible Languages

To understand the genesis of our internal extension mechanism, it is helpful to begin by considering why most implementations of programming languages cannot even be externally extended. Let us consider, as a simple example, an implementation of Gödel's T, a typed lambda calculus with recursion on primitive natural numbers [**?**]. A compiler for this language written using a functional language will invariably represent the primitive type families and operators of the object language using closed inductive datatypes. For example, a simple implementation in Standard ML may be based around these datatypes:

put statics in Appendix?

```
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 typechecking and translation to an intermediate language (which suitable for optimization by later phases of compilation) will be implemented by exhaustive case analysis over the constructors of `Exp`.

In an object-oriented implementation of Godel's T, we might instead encode types and operators as subclasses of abstract classes `Type` and `Exp`. Typechecking and translation will proceed by the ubiquitous *visitor pattern* [**?**] by dispatching against a set of known subclasses of `Exp`.

In either case, we encounter the same basic issue: there is no way to modularly extend the representation of primitive type families and operators and implement their associated typechecking and translation logic. This issue is related to the widely-discussed *expression problem* (we do not consider the case of adding new functions entirely here) [**?**].

A number of language mechanisms have been proposed that allow new cases to be added to datatypes and the functions that operate over them in a modular manner. In functional languages, we might use *open datatypes* [**?**]. For example,

if we wish to extend Gödel's T with product types, a language supporting open datatypes might allow you to add types and operators for products to the above definitions, if they have been declared open, like this:

```
newcase Prod of Type * Type extends Type
newcase Pair of Exp * Exp extends Exp     (* Intro *)
newcase PrL of Exp extends Exp            (* Elim Left *)
newcase PrR of Exp extends Exp            (* Elim Right *)
```

The logic for functionality like typechecking and translation might then be specified for only the new cases. For example, a function `typeof` that assigns a type to an expression could be extended to support the `PrL` operator like so:

```
typeof PrL(e) =
  case typeof e of
    Prod(t1, _) => t1
  | _ => raise TypeError("<appropriate error 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 exposed at a library's interface boundary, clients using different compilers face the same problems with interoperability that those using different languages face. 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 or supported the introduction of extensions that many libraries have come to rely on, including the Glasgow Haskell Compiler, SML/NJ 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 this, the language must include an appropriately-constrained mechanism to introduce new type families, like `Prod`, and operators, like `Pair`, `PrL` and `PrR`, and implement the associated typechecking and translation logic. When encountering these new operators in expressions, the compiler must effectively hand control over typechecking and translation to user-defined logic. Because this mechanism is language-internal, all compilers must support it to satisfy the language specification, and extensions are packaged directly within libraries.

Statically-typed languages typically make a distinction between *expressions*, which describe run-time computations, and static constructs like types and datatype declarations. The design described above suggests we may now need to add another layer to our language, an extension language, where extensions can be declared and implemented. In fact, we will show that **the natural place for type system extensions is within the type-level language**. The intuition is that extensions to a language's static semantics will need to manipulate types as values at compile-time. Many languages already allow users to write functions

$$
\begin{array}{rrcl}
\textbf{programs} & \rho & ::= & \mathsf{family}\ \textsc{Fam}[\kappa_{\mathrm{idx}}] \sim \mathbf{i}.\tau\ \{\theta\};\ \rho \mid \mathsf{def}\ \mathbf{t} : \kappa = \tau;\ \rho \mid e \\
\text{primitive ops} & \theta & ::= & \cdot \mid \theta; \boldsymbol{op}(\mathbf{i}{:}\kappa_{\mathrm{i}}, \mathbf{a}.\tau) \\
\\
\textbf{expressions} & e & ::= & x \mid \lambda x{:}\tau.e \mid \textsc{Fam}.\boldsymbol{op}\langle\tau_{\mathrm{i}}\rangle(e_1; \ldots; e_n) \\
\\
\textbf{type-level terms} & \tau & ::= & \mathbf{t} \mid \lambda \mathbf{t}{:}\kappa.\tau \mid \tau_1\ \tau_2 \mid []_\kappa \mid \tau_1 :: \tau_2 \mid \mathsf{fold}(\tau_1; \tau_2; \mathbf{x}, \mathbf{y}.\tau_3) \\
\text{type-level data} & & & \mid \bar{\mathrm{z}} \mid \tau_1 \oplus \tau_2 \mid \textit{“str”} \mid () \mid (\tau_1, \tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau) \\
\text{structural equality} & & & \mid \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \\
\text{types} & & & \mid \textsc{Fam}\langle\tau\rangle \mid \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
\text{denotations} & & & \mid [\![\tau_1 \Longrightarrow \tau_2]\!] \mid \mathsf{err} \mid \mathsf{case}\ \tau\ \mathsf{of}\ [\![\mathbf{x} \Longrightarrow \mathbf{y}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
\text{quoted IL} & & & \mid \triangledown(\gamma) \mid \blacktriangledown(\sigma) \\
\\
\textbf{kinds} & \kappa & ::= & \kappa_1 \to \kappa_2 \mid \mathsf{list}[\kappa] \mid \mathbb{Z} \mid \mathsf{Str} \mid 1 \mid \kappa_1 \times \kappa_2 \mid \star \mid \mathsf{Den} \mid \mathsf{ITy} \mid \mathsf{ITm} \\
\\
\textbf{internal terms} & \gamma & ::= & x \mid \lambda x{:}\sigma.\gamma \mid \gamma_1\ \gamma_2 \mid \mathsf{fix}\ f{:}\sigma\ \mathsf{is}\ \gamma \mid (\gamma_1, \gamma_2) \mid \mathsf{fst}(\gamma) \mid \mathsf{snd}(\gamma) \\
& & & \mid \bar{\mathrm{z}} \mid \gamma_1 \oplus \gamma_2 \mid \mathsf{if}\ \gamma_1 \equiv_{\mathbb{Z}} \gamma_2\ \mathsf{then}\ \gamma_3\ \mathsf{else}\ \gamma_4 \\
& & & \mid \mathsf{val}([\![\tau_1 \Longrightarrow \tau_2]\!]) \mid \triangle(\tau) \\
\textbf{internal types} & \sigma & ::= & \sigma_1 \to \sigma_2 \mid \mathbb{Z} \mid \sigma_1 \times \sigma_2 \mid \mathsf{rep}(\tau) \mid \blacktriangle(\tau)
\end{array}
$$

Fig. 2. Syntax of @λ. Variables x are used in expressions and internal terms and are distinct from type-level variables, $\mathbf{t}$. Names Fam are family names (we assume that unique family names can be generated by some external mechanism) and ***op*** are operator names. *“str”* denotes string literals, $\bar{\mathrm{z}}$ denotes integer literals and $\oplus$ stands for binary operations over integers.

over types, effectively manipulating them as values at compile-time (see Sec. 7 for examples). The type-level language is often constrained by its own type system (where the types of type-level values are called *kinds* for clarity) that prevents type-level functions from causing problems during compilation. This is precisely the structure that a distinct extension layer would have, and so we will show that it is quite naturally to unify the two in this work.

3 @λ

We will now develop a core calculus, called @λ for the “actively-typed lambda calculus”, and discuss how to address safety concerns that arise when giving users this level of control over the semantics of a language and its implementation.

3.1 Overview

remove Θ from family

The grammar of @λ is shown in Figure 2. The language is a simply-typed lambda calculus with simply-kinded type-level computation. Kinds, κ, classify type-level terms, τ. Types classify expressions, e, and are type-level values of kind $\star$ (following System F_ω [**?**]). The type-level language also includes other kinds of terms,

```
family NAT[1] ~ i.▼(ℤ) {                                                    (1)
  z(i:1, a.no_args a λ_:1.[[NAT⟨()⟩ ⟹ ▽(0)]]);                              (2)
  s(i:1, a.last_arg a λt1:⋆.λv1:ITm.                                        (3)
    check_type t1 NAT⟨()⟩ [[NAT⟨()⟩ ⟹ ▽(△(v1) + 1)]]);                     (4)
  rec(i:1, a.next_arg a λt1:⋆.λv1:ITm.λa:list[Den].                         (5)
    next_arg a λt2:⋆.λv2:ITm.λa:list[Den].                                  (6)
    last_arg a λt3:⋆.λv3:ITm.                                               (7)
    check_type t1 NAT⟨()⟩ (                                                 (8)
    check_type t3 ARROW⟨(NAT⟨()⟩, ARROW⟨(t2, t2)⟩)⟩ (                       (9)
    c))                                                                     (10)
}                                                                           (11)
```

Fig. 3. Example: primitive natural numbers and n-ary ($n > 1$) products in @λ.

such as type-level functions and other type-level data structures. We include lists (required by our mechanism) as well as type-level integers, strings and products for the sake of our examples (see Sec. **??** for design constraints in general).

At the top level, programs, ρ, consist of a series of declarations followed by an expression. Declarations can be either bindings of type-level terms to type-level variables using def or a declaration of a new primitive type family using family (Sec. 3.4). Expressions can be either variables, lambdas, or applications of operators (Sec. 3.5), and are ultimately given meaning by translation to a typed internal language. This language has been chosen, for simplicity, to be a variant of Plotkin's PCF with primitive integers and products, but in practice would include other constructs consistent with its role as a high-level intermediate language. Terms of this language are denoted γ and types are denoted σ. Both of these productions also include additional forms containing type-level terms, τ; these are used during compilation and will be explained below.

3.2 Example: Natural Numbers

To make our discussion of each of these components of the calculus concrete, we will begin with an example showing how to introduce primitive natural numbers, implemented as integers internally, as a user-defined extension.

3.3 Central Compilation Judgement

The *central compilation judgement*, shown in Figure 4, captures the two phases of compilation: kind checking and active typechecking and translation. The latter phase requires normalizing type-level terms to type-level values, so the kind checking phase preceding it will ensure that this process not "get stuck", as we will elaborate upon when discussing safety and decidability in Sec. 4.

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

Fig. 4. Central Compilation Judgement of @λ.

3.4 Indexed Type Families and Types

Declaring a new primitive indexed type family by using family can be compared to adding a new constructor to the compiler's `Type` datatype, as suggested in Section 2. The kind of type-level data that the family is indexed by is specified by κ_{idx}. For example, a base type like **nat** can be thought of as being the only type in a family, NAT, trivially indexed by the unit value, of kind $\mathbf{1}$, while families like PAIR might be indexed by a pair of types, of kind $\star \times \star$. A type (that is, a type-level term of kind $\star$) is created by naming a family that has been previously been introduced and providing an index of the appropriate kind. For example, $\textsc{Prod}\langle(\mathbf{nat}, \mathbf{nat})\rangle$ is the type of a pair of natural numbers in a context where the families PROD and NAT have been introduced as described above and the type-level variable **nat** has been bound to the type $\textsc{Nat}\langle()\rangle$.

3.5 Indexed Operator Families and Active Types

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.

3.6 Active Typechecking and Translation

3.7 Abstract Representations

4 Safety of @λ

5 Examples

6 Design Considerations

7 Related Work

7.1 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

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

FAMILY DECL KINDING

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

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

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

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

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

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

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

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].)

7.2 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.

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

$$\text{ATT-FAM}\ \frac{\vdash_{\Phi, \textsc{Fam}[\theta,\mathbf{i}.\tau]} \rho \Longrightarrow \gamma}{\vdash_{\Phi} \mathsf{family}\ \textsc{Fam}[\kappa_{\mathrm{idx}}] \sim \mathbf{i}.\tau\ \{\theta\};\ \rho \Longrightarrow \gamma} \qquad \text{ATT-DEF}\ \frac{\tau \Downarrow \tau' \qquad \vdash_{\Phi} [\tau'/\mathbf{t}]\rho \Longrightarrow \gamma}{\vdash_{\Phi} \mathsf{def}\ \mathbf{t} : \kappa = \tau;\ \rho \Longrightarrow \gamma}$$

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

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

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

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

Fig. 6. Active typechecking and translation

7.3 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.

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).

7.4 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.

7.5 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.

7.6 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

8 Discussion

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 Reflec-*

tion 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.

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

$$\textsc{var kind}\ \frac{}{\Delta, \mathbf{t} : \kappa \vdash_\Sigma \mathbf{t} : \kappa} \qquad \textsc{k-arrow intro}\ \frac{\Delta, \mathbf{t} : \kappa_1 \vdash_\Sigma \tau : \kappa_2}{\Delta \vdash_\Sigma \lambda \mathbf{t}{:}\kappa_1.\tau : \kappa_1 \to \kappa_2} \qquad \textsc{k-arrow elim}\ \frac{\Delta \vdash_\Sigma \tau_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash_\Sigma \tau_2 : \kappa_1}{\Delta \vdash_\Sigma \tau_1\ \tau_2 : \kappa_2}$$

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

$$\textsc{eq kind}\ \frac{\kappa\ \mathsf{eq} \qquad \Delta \vdash_\Sigma \tau_1 : \kappa \qquad \Delta \vdash_\Sigma \tau_2 : \kappa \qquad \Delta \vdash_\Sigma \tau_3 : \kappa' \qquad \Delta \vdash_\Sigma \tau_4 : \kappa'}{\Delta \vdash_\Sigma \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 : \kappa'} \qquad \textsc{type intro}\ \frac{\textsc{Fam}[\kappa_{\mathrm{idx}}, \Theta] \in \Sigma \qquad \Delta \vdash_\Sigma \tau_{\mathrm{idx}} : \kappa_{\mathrm{idx}}}{\Delta \vdash_\Sigma \textsc{Fam}\langle \tau_{\mathrm{idx}} \rangle : \star}$$

$$\textsc{type elim}\ \frac{\textsc{Fam}[\kappa_{\mathrm{idx}}, \Theta] \in \Sigma \qquad \Delta \vdash_\Sigma \tau : \star \qquad \Delta, \mathbf{x} : \kappa_{\mathrm{idx}} \vdash_\Sigma \tau_0 : \kappa \qquad \Delta \vdash_\Sigma \tau_1 : \kappa}{\Delta \vdash_\Sigma \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle \mathbf{x} \rangle \Rightarrow \tau_0\ \mathsf{ow}\ \tau_1 : \kappa} \qquad \textsc{den intro valid}\ \frac{\Delta \vdash_\Sigma \tau_\star : \star \qquad \Delta \vdash_\Sigma \tau_{\mathrm{val}} : \mathsf{ITm}}{\Delta \vdash_\Sigma [\![\tau_\star \Longrightarrow \tau_{\mathrm{val}}]\!] : \mathsf{Den}}$$

$$\textsc{den intro err}\ \frac{}{\Delta \vdash_\Sigma \mathsf{err} : \mathsf{Den}} \qquad \textsc{den elim}\ \frac{\Delta \vdash_\Sigma \tau_{\mathrm{den}} : \mathsf{Den} \qquad \Delta, \mathbf{x} : \mathsf{ITm}, \mathbf{y} : \star \vdash_\Sigma \tau_1 : \kappa \qquad \Delta \vdash_\Sigma \tau_2 : \kappa}{\Delta \vdash_\Sigma \mathsf{case}\ \tau_{\mathrm{den}}\ \mathsf{of}\ [\![\mathbf{x} \Longrightarrow \mathbf{y}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 : \kappa}$$

$$\textsc{iterm intro}\ \frac{\Delta\ \emptyset \vdash_\Sigma \hat{\gamma}\ \mathtt{iterm}}{\Delta \vdash_\Sigma \triangledown(\hat{\gamma}) : \mathsf{ITm}} \qquad \textsc{itype intro}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}\ \mathtt{itype}}{\Delta \vdash_\Sigma \blacktriangledown(\hat{\sigma}) : \mathsf{ITy}}$$

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

$$\textsc{t-eq}\ \frac{}{\star\ \mathsf{eq}} \qquad \textsc{z-eq}\ \frac{}{\mathbb{Z}\ \mathsf{eq}} \qquad \textsc{str-eq}\ \frac{}{\mathsf{Str}\ \mathsf{eq}} \qquad \textsc{u-eq}\ \frac{}{1\ \mathsf{eq}} \qquad \textsc{prod-eq}\ \frac{\kappa_1\ \mathsf{eq} \qquad \kappa_2\ \mathsf{eq}}{\kappa_1 \times \kappa_2\ \mathsf{eq}} \qquad \textsc{list-eq}\ \frac{\kappa\ \mathsf{eq}}{\mathsf{list}[\kappa]\ \mathsf{eq}}$$

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

$$\textsc{i-var kind}\ \frac{}{\Delta\ \Omega, x \vdash_\Sigma x\ \mathtt{iterm}} \qquad \textsc{i-lam kind}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}\ \mathtt{itype} \qquad \Delta\ \Omega, x \vdash_\Sigma \hat{\gamma}\ \mathtt{iterm}}{\Delta\ \Omega \vdash_\Sigma \lambda x{:}\hat{\sigma}.\hat{\gamma}\ \mathtt{iterm}} \qquad \textsc{i-fix kind}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}\ \mathtt{itype} \qquad \Delta\ \Omega, x \vdash_\Sigma \hat{\gamma}\ \mathtt{iterm}}{\Delta\ \Omega \vdash_\Sigma \mathsf{fix}\ f{:}\hat{\sigma}\ \mathsf{is}\ \hat{\gamma}\ \mathtt{iterm}}$$

(omitted forms have trivially recursive rules)

$$\textsc{iterm unquote kind}\ \frac{\Delta \vdash_\Sigma \tau : \mathsf{ITm}}{\Delta\ \Omega \vdash_\Sigma \triangle(\tau)\ \mathtt{iterm}} \qquad \textsc{val from den kind}\ \frac{\Delta \vdash_\Sigma \tau_\star : \star \qquad \Delta \vdash_\Sigma \tau_{\mathrm{val}} : \mathsf{ITm}}{\Delta\ \Omega \vdash_\Sigma \mathsf{val}([\![\tau_\star \Longrightarrow \tau_{\mathrm{val}}]\!])\ \mathtt{iterm}}$$

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

$$\textsc{i-int kind}\ \frac{}{\Delta \vdash_\Sigma \mathbb{Z}\ \mathtt{itype}} \qquad \textsc{i-prod kind}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}_1\ \mathtt{itype} \qquad \Delta \vdash_\Sigma \hat{\sigma}_2\ \mathtt{itype}}{\Delta \vdash_\Sigma \hat{\sigma}_1 \times \hat{\sigma}_2\ \mathtt{itype}} \qquad \textsc{i-arrow kind}\ \frac{\Delta \vdash_\Sigma \hat{\sigma}_1\ \mathtt{itype} \qquad \Delta \vdash_\Sigma \hat{\sigma}_2\ \mathtt{itype}}{\Delta \vdash_\Sigma \hat{\sigma}_1 \to \hat{\sigma}_2\ \mathtt{itype}}$$

$$\textsc{itype unquote kind}\ \frac{\Delta \vdash_\Sigma \tau : \mathsf{ITy}}{\Delta \vdash_\Sigma \blacktriangle(\tau)\ \mathtt{itype}} \qquad \textsc{rep from type kind}\ \frac{\Delta \vdash_\Sigma \tau : \star}{\Delta \vdash_\Sigma \mathsf{rep}(\tau)\ \mathtt{itype}}$$

Fig. 7. Kinding for type-level terms

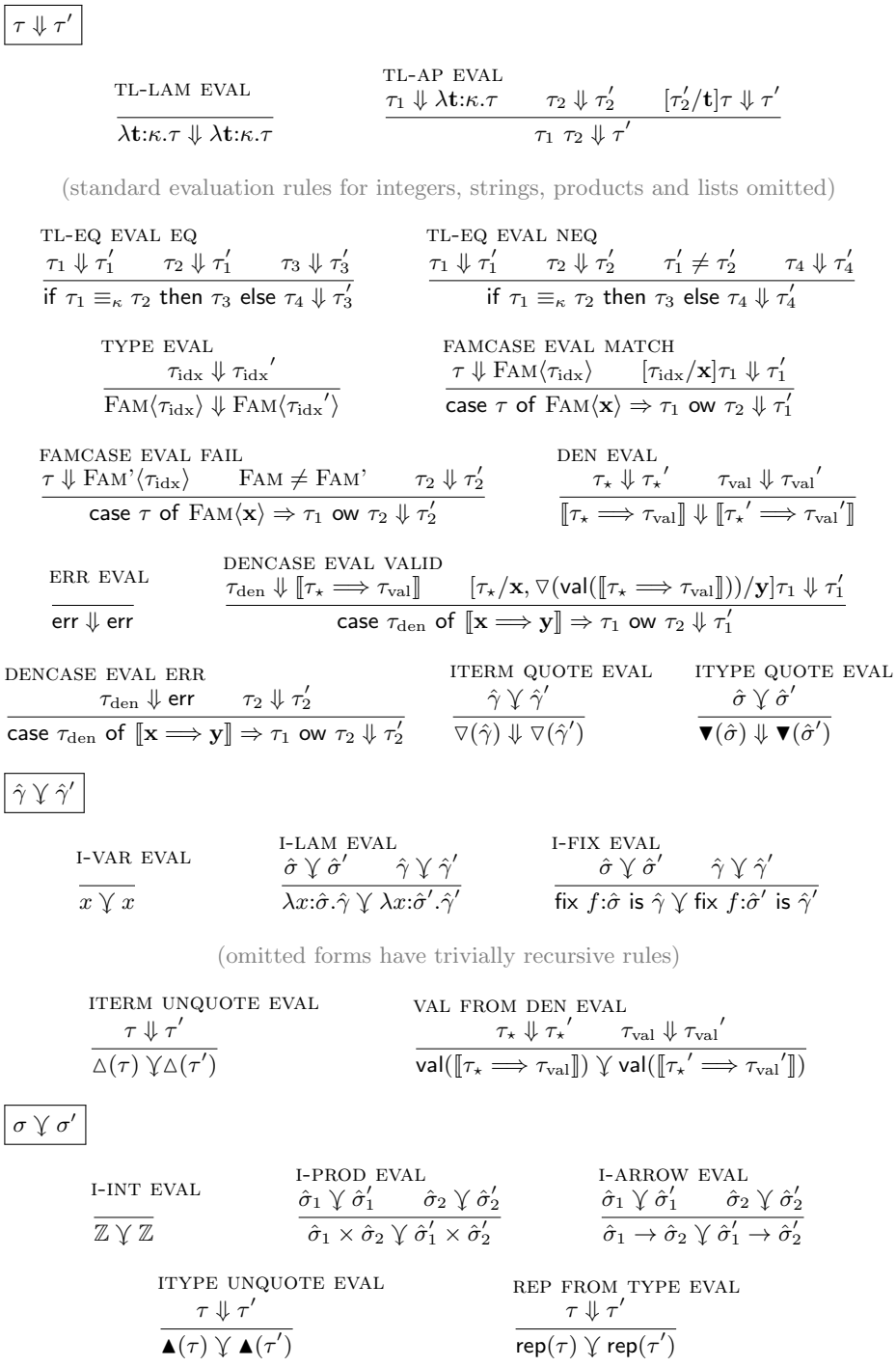

Fig. 8. Evaluation semantics for type-level terms

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

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

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

$$\text{ABS INT}\quad \frac{}{\vdash^{\text{FAM}}_{\Phi} \mathbb{Z} \sim \mathbb{Z}} \qquad \text{ABS ARROW}\quad \frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \sim \bar{\sigma}_1 \qquad \vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_2 \sim \bar{\sigma}_2}{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \to \hat{\sigma}_2 \sim \bar{\sigma}_1 \to \bar{\sigma}_2} \qquad \text{ABS PROD}\quad \frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \sim \bar{\sigma}_1 \qquad \vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_2 \sim \bar{\sigma}_2}{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma}_1 \times \hat{\sigma}_2 \sim \bar{\sigma}_1 \times \bar{\sigma}_2}$$

$$\text{ABS+CANCEL UNQUOTE}\quad \frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma}}{\vdash^{\text{FAM}}_{\Phi} \blacktriangle(\blacktriangledown(\hat{\sigma})) \sim \bar{\sigma}} \qquad \text{ABS REP FROM TYPE VISIBLE}\quad \frac{\text{FAM}[\theta, \mathbf{i}.\tau] \in \Phi \qquad [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\hat{\sigma}) \qquad \vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma}}{\vdash^{\text{FAM}}_{\Phi} \mathsf{rep}(\text{FAM}\langle\tau_{\text{idx}}\rangle) \sim \bar{\sigma}}$$

$$\text{ABS REP FROM TYPE HIDDEN}\quad \frac{\text{FAM} \neq \text{FAM'}}{\vdash^{\text{FAM}}_{\Phi} \mathsf{rep}(\text{FAM'}\langle\tau_{\text{idx}}\rangle) \sim \mathsf{rep}(\text{FAM'}\langle\tau_{\text{idx}}\rangle)}$$

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

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

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

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

$$\text{ABS I-AP}\quad \frac{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_1 \sim \bar{\sigma}_1 \to \bar{\sigma}_2 \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_2 \sim \bar{\sigma}_1}{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_1\ \hat{\gamma}_2 \sim \bar{\sigma}_2} \qquad \text{ABS I-FIX}\quad \frac{\vdash^{\text{FAM}}_{\Phi} \hat{\sigma} \sim \bar{\sigma} \qquad \Psi, x : \bar{\sigma} \vdash^{\text{FAM}}_{\Phi} \hat{\gamma} \sim \bar{\sigma}}{\Psi \vdash^{\text{FAM}}_{\Phi} \mathsf{fix}\ x{:}\hat{\sigma}\ \mathsf{is}\ \hat{\gamma} \sim \bar{\sigma}}$$

(standard statics for integers and products omitted)

$$\text{ABS IF EQ}\quad \frac{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_1 \sim \mathbb{Z} \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_2 \sim \mathbb{Z} \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_3 \sim \bar{\sigma} \qquad \Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma}_4 \sim \bar{\sigma}}{\Psi \vdash^{\text{FAM}}_{\Phi} \mathsf{if}\ \hat{\gamma}_1 \equiv_{\mathbb{Z}} \hat{\gamma}_2\ \mathsf{then}\ \hat{\gamma}_3\ \mathsf{else}\ \hat{\gamma}_4 \sim \bar{\sigma}} \qquad \text{ABS ITERM UNQUOTE}\quad \frac{\Psi \vdash^{\text{FAM}}_{\Phi} \hat{\gamma} \sim \bar{\sigma}}{\Psi \vdash^{\text{FAM}}_{\Phi} \triangle(\triangledown(\hat{\gamma})) \sim \bar{\sigma}}$$

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

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

Fig. 9. Abstracted internal typing

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

$$\textsc{deabs int}\ \frac{}{\vdash_\Phi \mathbb{Z} \rightsquigarrow \mathbb{Z}} \qquad \textsc{deabs arrow}\ \frac{\vdash_\Phi \bar{\sigma}_1 \rightsquigarrow \sigma_1 \quad \vdash_\Phi \bar{\sigma}_2 \rightsquigarrow \sigma_2}{\vdash_\Phi \bar{\sigma}_1 \to \bar{\sigma}_2 \rightsquigarrow \sigma_1 \to \sigma_2} \qquad \textsc{deabs prod}\ \frac{\vdash_\Phi \bar{\sigma}_1 \rightsquigarrow \sigma_1 \quad \vdash_\Phi \bar{\sigma}_2 \rightsquigarrow \sigma_2}{\vdash_\Phi \bar{\sigma}_1 \times \bar{\sigma}_2 \rightsquigarrow \sigma_1 \times \sigma_2}$$

$$\textsc{deabs rep from type hidden}\ \frac{\textsc{Fam}[\theta, \mathbf{i}.\tau] \in \Phi \qquad [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\hat{\sigma}) \qquad \vdash_\Phi^{\textsc{Fam}} \hat{\sigma} \sim \bar{\sigma} \qquad \vdash_\Phi \bar{\sigma} \rightsquigarrow \sigma}{\vdash_\Phi \mathsf{rep}(\textsc{Fam}\langle\tau_{\text{idx}}\rangle) \rightsquigarrow \sigma}$$

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

$$\textsc{deabs i-var}\ \frac{}{\vdash_\Phi x \rightsquigarrow x} \qquad \textsc{deabs i-lam}\ \frac{\vdash_{\bar{\Phi}} \hat{\sigma} \sim \bar{\sigma} \qquad \vdash_\Phi \bar{\sigma} \rightsquigarrow \sigma \qquad \vdash_\Phi \hat{\gamma} \rightsquigarrow \gamma}{\vdash_\Phi \lambda x{:}\hat{\sigma}.\hat{\gamma} \rightsquigarrow \lambda x{:}\sigma.\gamma}$$

$$\textsc{deabs i-fix}\ \frac{\vdash_{\bar{\Phi}} \hat{\sigma} \sim \bar{\sigma} \qquad \vdash_\Phi \bar{\sigma} \rightsquigarrow \sigma \qquad \vdash_\Phi \hat{\gamma} \rightsquigarrow \gamma}{\vdash_\Phi \mathsf{fix}\ x{:}\hat{\sigma}\ \mathsf{is}\ \hat{\gamma} \rightsquigarrow \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \gamma}$$

(omitted forms have trivially recursive rules)

$$\textsc{deabs unquote}\ \frac{\vdash_\Phi \hat{\gamma} \rightsquigarrow \gamma}{\vdash_\Phi \triangle(\nabla(\hat{\gamma})) \rightsquigarrow \gamma} \qquad \textsc{deabs val from den hidden}\ \frac{\vdash_\Phi \hat{\gamma} \rightsquigarrow \gamma}{\vdash_\Phi \mathsf{val}([\![\textsc{Fam}\langle\tau_{\text{idx}}\rangle \Longrightarrow \nabla(\hat{\gamma})]\!]) \rightsquigarrow \gamma}$$

Fig. 10. Deabstraction rules