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

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.

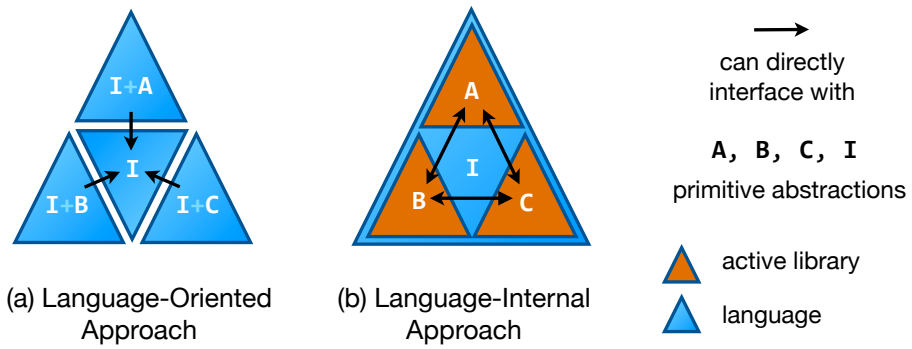
1. Motivation

When designing and implementing a new abstraction, experts typically begin by attempting to define new constructs in terms of existing language constructs. This approach is often effective because modern general-purpose abstraction mechanisms, like inductive datatypes and object systems, are highly expressive. For example, the Delite framework leverages Scala's powerful general-purpose mechanisms to enable a number of useful *embedded domain-specific languages* [1]. Unfortunately, there remain some 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 that govern general-purpose abstractions (e.g. reasoning about units of measure requires built-in language support in F# [9]). Even if a full encoding is possible, it may not be useful if it is overly verbose or unnatural, or if the error messages are overly abstract. For example, regular expressions encoded using inductive datatypes are quite verbose, so most functional languages support them via strings, which is less safe. Finally, general-purpose abstractions are implemented in a uniform manner, meaning domain-specific heuristics cannot be 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, such as when targeting heterogeneous hardware platforms (e.g. programmable GPUs) and distributed computing resources.

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 today but to realize new abstractions by creating a new language in some way. 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 [7]. 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 designed explicitly 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 discourages languages from including 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).

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. As a result, clients can granularly import any necessary primitive constructs when using code that relies on them, and thereby achieve full safety, ease-of-use and performance. 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 otherwise appropriate. Libraries



(a) Language-Oriented Approach

(b) Language-Internal Approach

Figure 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.

containing logic that is invoked at compile-time, as extension logic would be, have been called *active libraries* [13]. We adopt this terminology and diagram this competing approach in Figure 1(b).

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 6.

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 and operator families 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 non-interference of extensions, as we outline in Section 4. In Section 5, we suggest 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.

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 (see Appendix). A compiler for this language written using a functional language will invariably represent the primitive type families and operators using closed inductive 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 typechecking and translation to a suitable intermediate language (for subsequent optimization and compilation by some back-end) will proceed 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 fixed collection of known subclasses of `Exp`.

In either case, we encounter the same basic issue: there is no way to modularly add new primitive type families and operators and implement their associated typechecking and translation logic.

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 and we have written our compiler in a language supporting open inductive datatypes, it might be possible to add new cases 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 could then be implemented for only these new cases. For example, the `typeof` function that assigns a type to an expression could be extended like so:

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

If we allowed users to define new modules containing definitions like these and link them into our compiler, we will have

succeeded in creating an externally-extensible compiler, albeit one where safety is not guaranteed (we will return to this point shortly). We have not, however, created an extensible programming language, for two reasons. First, compiler extensions are distributed and activated separately from libraries, so dependencies become more difficult to manage. Second, other compilers for the same language will not necessarily support the same extensions. 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 compiler for it is morally equivalent to creating a new language. Several prominent language ecosystems today are in a state where a prominent compiler has introduced or enabled 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++.

A more appropriate and useful place for extensions like this is directly within libraries, alongside abstractions that can be adequately implemented in terms of existing primitive abstractions. To enable this, the language must allow for the introduction new primitive type families, like `Prod`, operators, like `Pair`, `PrL` and `PrR`, and associated typechecking and translation logic. When encountering these new operators in expressions, the compiler must effectively hand control over typechecking and translation to the appropriate user-defined logic. Because this mechanism is language-internal, all compilers must support it to satisfy the language specification.

Statically-typed languages typically make a distinction between *expressions*, which describe run-time computations, and type-level constructs like types, type aliases 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 most natural place for type system extensions is within the type-level language**. The intuition is that extensions to a statically-typed language's semantics will need to manipulate types as values at compile-time. Many languages already allow users to write type-level functions for various reasons, effectively supporting this notion of types as values at compile-time (see Sec. 6 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 it is quite natural to unify the two, as we will show in this work.

3. @λ

In this section, we will develop a core calculus, called @λ for the "actively-typed lambda calculus", by way of a semantics and a simple example, and discuss how it addresses the safety concerns that arise.

3.1 Overview

The grammar of @λ is shown in Figure 2. A program, ρ, consists 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 primitive type family declared using family, which can contain implementations of one or more operator families, θ. Expressions, e, can be either variables, lambdas, or applications of operators.

The language is structured as a simply-typed lambda calculus with simply-kinded type-level computation. Kinds, κ, classify type-level terms, τ. Types are type-level values of kind $\star$ (following System F_ω) and classify expressions. The type-level language also includes other kinds of terms: type-level functions, lists (required by our mechanism), integers, strings and products for the sake of our examples (see Sec. 4 for a discussion on other acceptable kinds of type-level data) and constructs for developing extensions – denotations and reified internal terms and types – which we will discuss in the sections below.

programs	ρ	::=	$\mathsf{family}\ \textsc{Fam}[\kappa_{\text{idx}}] \sim \mathbf{i}.\tau_{\text{rep}}\ \{\theta\};\ \rho \mid \mathsf{def}\ \mathbf{t} : \kappa = \tau;\ \rho \mid e$
primitive ops	θ	::=	$\boldsymbol{op}[\kappa_{\text{idx}}](\mathbf{i}, \mathbf{a}.\tau) \mid \theta; \theta$
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	::=	$\mathbf{t} \mid \lambda \mathbf{t}{:}\kappa.\tau \mid \tau_1\ \tau_2 \mid []_\kappa \mid \tau_1 :: \tau_2 \mid \mathsf{fold}(\tau_1; \tau_2; \mathbf{h}, \mathbf{t}, \mathbf{r}.\tau_3)$
type-level data			$\bar{\mathsf{z}} \mid \tau_1 \oplus \tau_2 \mid$ *"str"* $\mid () \mid (\tau_1, \tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau)$
structural equality			$\mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4$
types			$\textsc{Fam}\langle\tau\rangle \mid \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2$
denotations			$[\![\tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}}]\!] \mid \mathsf{err} \mid \mathsf{case}\ \tau\ \mathsf{of}\ [\![\mathbf{x}\ \mathsf{as}\ \mathbf{t}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2$
reified IL			$\triangledown(\gamma) \mid \blacktriangledown(\sigma)$
kinds	κ	::=	$\kappa_1 \to \kappa_2 \mid \mathsf{list}[\kappa] \mid \mathbb{Z} \mid \mathsf{Str} \mid 1 \mid \kappa_1 \times \kappa_2 \mid \star \mid \mathsf{Den} \mid \mathsf{IT}$
internal terms	γ	::=	$x \mid \lambda x{:}\sigma.\gamma \mid \gamma_1\ \gamma_2 \mid \mathsf{fix}\ f{:}\sigma\ \mathsf{is}\ \gamma \mid (\gamma_1, \gamma_2) \mid \mathsf{fst}(\gamma) \mid \mathsf{snd}$
			$\bar{\mathsf{z}} \mid \gamma_1 \oplus \gamma_2 \mid \mathsf{if}\ \gamma_1 \equiv_{\mathbb{Z}} \gamma_2\ \mathsf{then}\ \gamma_3\ \mathsf{else}\ \gamma_4$
			$\mathsf{trans}([\![\tau_1\ \mathsf{as}\ \tau_2]\!]) \mid \triangle(\tau)$
internal types	σ	::=	$\sigma_1 \to \sigma_2 \mid \mathbb{Z} \mid \sigma_1 \times \sigma_2 \mid \mathsf{rep}(\tau) \mid \blacktriangle(\tau)$

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

All expressions are 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. The grammar of internal terms, γ, and internal types, σ, also includes special forms containing type-level terms; these are used for developing extensions and during compilation and will be erased before compilation ends.

3.2 Example: Gödel's T as an Active Type Family

To make our explanation of each of the constructs in the calculus concrete, we will work through an example showing how to introduce primitive natural numbers with bounded recursion in the style of Gödel's T [8]. These will be implemented internally as integers (that is, internal terms of internal type $\mathbb{Z}$). Figure 3 shows how to define the indexed type family NAT. This family contains only one type, written $\textsc{Nat}\langle()\rangle$, which we alias on line 13 by defining the type-level variable **nat**. We define the typechecking and translation logic for the operators associated with this family (***z***, ***s*** and ***rec***) on lines 2-11 and use these to define a $plus$ function on line 16 and compute 2+2. We will refer back to this figure as we describe each construct below.

3.3 Indexed Type Families and Types

The syntactic form $\mathsf{family}\ \textsc{Fam}[\kappa_{\text{idx}}] \sim \mathbf{i}.\tau_{\text{rep}}\ \{\theta\}$ declares a new primitive type family named FAM indexed by type-level values of kind κ_{idx} with representation schema $\mathbf{i}.\tau_{\text{rep}}$ and operators θ. The purpose of the representation schema and of associating of operators directly with types will be explained below.

Declaring a type family in this way is a language-internal analog to adding a new constructor to the compiler-internal datatype `Type`, as suggested in Sec. 2. The index represents the data associated with this constructor. A type (that is, a type-level term of kind $\star$) is constructed by naming a family in scope and providing a type-

```
family NAT[1] ∼ i.▼(ℤ) {                                                    (1)
  z[1](i, a.empty a ⟦▽(0) as NAT⟨()⟩⟧);                                      (2)
  s[1](i, a.pop_final a λx:ITm.λt:⋆.                                         (3)
    check_type t NAT⟨()⟩ ⟦▽(△(x) + 1) as NAT⟨()⟩⟧);                          (4)
  rec[1](i, a.pop a λx1:ITm.λt1:⋆.λa:list[Den].                              (5)
    pop a λx2:ITm.λt2:⋆.λa:list[Den].                                        (6)
    pop_final a λx3:ITm.λt3:⋆.                                               (7)
    check_type t1 NAT⟨()⟩ (                                                  (8)
    check_type t3 ARROW⟨(NAT⟨()⟩, ARROW⟨(t2, t2)⟩)⟩                          (9)
    ⟦▽((fix f:ℤ → rep(t2) is λx:ℤ.                                           (10)
      if x ≡ℤ 0 then △(x2) else △(x3) (x − 1) (f (x − 1))) △(x1)) as t2⟧)}
                                                                             (11)
};                                                                           (12)
def nat : ⋆ = NAT⟨()⟩;                                                       (13)
(λplus:ARROW⟨(nat, ARROW⟨(nat, nat)⟩)⟩.λtwo:nat.                             (14)
    ARROW.ap⟨()⟩(ARROW.ap⟨()⟩(plus; two); two))                              (15)
  (λx:nat.λy:nat.NAT.rec⟨()⟩(x; y; λp:nat.λr:nat.NAT.s⟨()⟩(r)))
                                                                             (16)
  NAT.s⟨()⟩(NAT.s⟨()⟩(NAT.z⟨()⟩()))                                          (17)
```

Figure 3. Gödel's T in @λ, used to calculate 2+2. Helper functions for working with lists (**empty**, **pop**, **pop_final**) and types (**check_type**), described below, are given in the appendix. We will refer to this program as ρ_{nat} in the text.

level term of the appropriate kind as an index. A base type like **nat** can be thought of as being the only type in the family NAT trivially indexed by the unit value, of kind 1, while families like NTUPLE might be indexed by a list of types, having kind list[⋆]. For example, NTUPLE⟨**nat** :: **nat** :: []⋆⟩ might be the type of a pair of natural numbers. Given a type, its family can be case analyzed to extract the value of its index. It is important that type equality be decidable, so only kinds for which equivalence coincides with syntactic equality can be used as type family indices. The main consequence of this restriction is that indices cannot contain type-level functions.

3.4 Representations and Representation Schemas

As we will discuss further below, it is important that all expressions classified by a type compile to consistently-typed internal terms. For this reason, we require that every type have a single internal type associated with it, called its *representation*. This is computed by substituting the type index for the bound variable **i** in the term τ_{rep}, called the *representation schema* of the type family, and evaluating to a value representing an internal type. Internal types, σ, are reified as type-level terms of kind ITy using the introductory form $\blacktriangledown(\sigma)$.

3.5 Indexed Operator Families and Denotations

Type families are also equiped with a collection of primitive operator families. An operator family named ***op*** is declared using the form $\textbf{\textit{op}}[\kappa_{\text{idx}}](\mathbf{i}, \mathbf{a}.\tau_{\text{op}})$. Like type families, operator families are indexed by values of some kind, κ_{idx}, but because operators are not first-class type-level values in our calculus, there are no equality restrictions. In the example in Fig. 3, all the operators are trivially indexed by the kind 1, so each family only contains one operator. However, a type family like NTUPLE would be equipped with a family of projection operators, ***pr***, indexed by a position (e.g. an integer in our calculus). A family implementing record types or object types might have a similar operator indexed by a type-level string representing the field being accessed (see Sec. 5).

To apply an operator, the grammar provides a uniform form of expression: $\text{FAM}.\textbf{\textit{op}}\langle\tau_{\text{idx}}\rangle(e_1; \ldots; e_n)$, where $n \geq 0$. The type-checking and translation of an expression of this form is controlled by the term τ_{op} in the operator's declaration. This term must evaluate to a *denotation*, which is a type-level value of kind Den, when given the operator index, τ_{idx}, and a list constructed from the denotations recursively assigned to each argument, e_1 through e_n.

There are two forms of denotations that an expression can be assigned. A *valid denotation* has the form $[\![\tau_{\text{iterm}} \text{ as } \tau_{\text{type}}]\!]$, where τ_{iterm} is the *translation* of the expression to an internal term and τ_{type} is the type it has been assigned. Internal terms are represented as type-level terms using the form $\triangledown(\gamma)$, and have kind ITm (similar to ITy). If a type error is detected by an operator, the *error denotation*, err, is returned instead of a valid denotation. In a practical implementation, a specialized error message and other diagnostic information would be provided when returning err, but we omit such details for simplicity. Terms of kind Den can be case analyzed to determine if they are valid or errors, and if valid, to extract the translation and type.

In the example in Fig. 3, the operator ***z*** checks that no arguments were passed in using the simple helper function **empty**. If so, it returns a valid denotation by pairing the translation $\triangledown(0)$ with the type $\text{NAT}\langle()\rangle$, as expected[1]. If an argument was provided, the helper function returns err. The successor operator, ***s***, takes one argument, so it pops a denotation off the argument list, making sure there are no more, and binds its translation and type to **x** and **t** respectively, all using the helper function **pop_final**. It then checks that the argument's type is also $\text{NAT}\langle()\rangle$, returning a denotation pairing the translation $\triangledown(\triangle(\mathbf{x}) + 1)$ with the type $\text{NAT}\langle()\rangle$ if so. The form $\triangle(\tau)$ is used to "un-reify" reified internal terms, of kind ITm (thus serving as the left-inverse of $\triangledown(\gamma)$). In this case, **x** is a type-level variable of kind ITm representing the translation of the argument to the successor operator, so we simply need to add one to it. Because we have checked that the denotation's type was $\text{NAT}\langle()\rangle$, and the representation schema will guarantee that expressions of this type always translate to integers, we know that it is safe to do so. If any of these steps fail, the various helper functions we use simply return err (in practice, it would be prudent to equip each failure condition with a different error message). Compilation will also fail if we accidentally violate the representation schema, as we will see in Sec. 3.8.

3.6 Functions, Variables and Arrow Types

The recursor operator, ***rec***, proceeds similarly, extracting the translations and types of each of its three arguments. Of note, however, is how it handles the third argument, which binds two variables (the predecessor and the result of recursing on it). In @λ, the built-in λ operator serves as the sole mechanism for introducing bound variables. In most calculi, lambda terms have types of the form $\tau_1 \to \tau_2$. In @λ, → corresponds to the built-in type family, ARROW. This family is indexed by a pair of types, $\star \times \star$, and is always in scope. Its representation schema simply maps to the corresponding internal arrow type, $\mathbf{i}.\blacktriangledown(\text{rep}(\text{fst}(\mathbf{i})) \to \text{rep}(\text{snd}(\mathbf{i})))$. The special form $\text{rep}(\tau)$ is used to refer, abstractly, to the representation of the type τ (we also see this used on line 10). Although the λ operator is built-in, because it needs to bind variables, application is just an operator associated with the ARROW family, ***ap***, as is seen on line 15. The details are straightforward and given in the appendix.

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

Figure 4. Central Compilation Judgement of $@\lambda$.

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

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

DEF-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
$$\frac{\Delta\ \emptyset \vdash_\Sigma e\ \texttt{expr}}{\Delta \vdash_\Sigma e\ \texttt{prog}}$$

$\boxed{\Delta \vdash_\Sigma \theta : \Theta} \quad \Theta ::= \boldsymbol{op}[\kappa_{\text{idx}}] \mid \Theta, \Theta$

OP-KINDING
$$\frac{\Delta, \mathbf{i} : \kappa_{\text{i}}, \mathbf{a} : \mathsf{list}[\mathsf{Den}] \vdash_\Sigma \tau : \mathsf{Den}}{\Delta \vdash_\Sigma \boldsymbol{op}[\kappa_{\text{idx}}](\mathbf{i}, \mathbf{a}.\tau) : \boldsymbol{op}[\kappa_{\text{idx}}]}$$

OPS-KINDING
$$\frac{\Delta \vdash_\Sigma \theta_1 : \Theta_1 \qquad \Delta \vdash_\Sigma \theta_2 : \Theta_2 \qquad \text{dom}(\theta_1) \cap \text{dom}(\theta_2) = \emptyset}{\Delta \vdash_\Sigma \theta_1; \theta_2 : \Theta_1, \Theta_2}$$

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

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

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

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

Figure 5. Kinding for programs. Variable contexts Δ and Ω obey standard structural properties. Kinding rules for type-level terms are given in Figure 7.

3.7 Compilation

The *central compilation judgement*, shown in Figure 4, captures the two phases of compilation: kind checking and active typechecking and translation. The first phase ensures that all type-level terms in the program are well-kinded and that all expressions and internal terms are closed. The kinding rules for programs are given in Figure 5, and they rely on the kinding rules for type-level terms given in Figure 7. These rules use contexts Σ and Θ to track family and operator signatures, but beyond that, the rules are largely consistent with those of a simply-typed lambda calculus shifted into the type-level, with the addition of the handful of special forms constrained as described in the previous sections. The reader is encouraged to verify that the example in Fig. 3 is well-kinded. The second phase of compilation involves invoking the logic implemented by user-defined operator families to typecheck and translate the program. The rules for this phase are given in Fig. 6. The context Φ tracks the operator definitions and representation schemas associated with families in scope. Because the logic inside operators must be invoked during this phase, we need an evaluation semantics for type-level terms, given in Fig. 8. This is again a largely unsurprising collection of rules with the exception of DENCASE-EVAL-VALID, which we will explain below.

[1] Actually, there is no theoretical barrier to a different "zero" being used!

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

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

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
$$\frac{\emptyset \vdash_\Phi e : \tau \Longrightarrow \gamma_{\text{abs}} \qquad \vdash_\Phi \gamma_{\text{abs}} \not\downarrow \gamma}{\vdash_\Phi e \Longrightarrow \gamma}$$

$\boxed{\Gamma \vdash_\Phi e : \tau \Longrightarrow \gamma_{\text{abs}}} \quad \Gamma ::= \emptyset \mid \Gamma, x : \tau$

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

ATT-LAM
$$\frac{\tau_1 \Downarrow \tau_1' \qquad \Gamma, x : \tau_1' \vdash_\Phi e : \tau_2 \Longrightarrow \gamma_{\text{abs}}}{\Gamma \vdash_\Phi \lambda x{:}\tau_1.e : \textsc{Arrow}\langle(\tau_1', \tau_2)\rangle \Longrightarrow \lambda x{:}\mathsf{rep}(\tau_1').\gamma_{\text{abs}}}$$

ATT-OP
$$\frac{\begin{array}{c}\textsc{Fam}[\theta, \mathbf{i}.\tau_{\text{rep}}] \in \Phi \qquad \boldsymbol{op}[\kappa_{\text{idx}}](\mathbf{i}, \mathbf{a}.\tau_{\text{op}}) \in \theta \qquad \tau_{\text{idx}} \Downarrow {\tau_{\text{idx}}}' \\ \Gamma \vdash_\Phi e_1 : \tau_1 \Longrightarrow \gamma_1 \quad \cdots \quad \Gamma \vdash_\Phi e_n : \tau_n \Longrightarrow \gamma_n \\ \left[\begin{array}{c}{\tau_{\text{idx}}}'/\mathbf{i} \\ \llbracket\nabla(\gamma_1)\ \mathsf{as}\ \tau_1\rrbracket :: \cdots :: \llbracket\nabla(\gamma_n)\ \mathsf{as}\ \tau_n\rrbracket :: []_{\mathsf{Den}}/\mathbf{a}\end{array}\right] \tau_{\text{op}} \Downarrow \llbracket\nabla(\gamma_{\text{abs}})\ \mathsf{as}\ \tau\rrbracket \\ \vdash_\Phi^{\Xi_0, \textsc{Fam}} \mathsf{rep}(\tau) \rightsquigarrow \sigma_{\text{abs}} \\ \vdash_\Phi^{\Xi_0, \textsc{Fam}} \Gamma \rightsquigarrow \Psi \qquad \Psi \vdash_\Phi^{\Xi_0, \textsc{Fam}} \gamma_{\text{abs}} \sim \sigma_{\text{abs}}\end{array}}{\Gamma \vdash_\Phi \textsc{Fam}.\boldsymbol{op}\langle\tau_{\text{idx}}\rangle(e_1; \ldots; e_n) : \tau \Longrightarrow \gamma_{\text{abs}}}$$

Figure 6. Active typechecking and translation of programs. Evaluation semantics for type-level terms are given in Fig. 8.

3.8 Abstract Representations and Abstract Internal Types

To typecheck and translate an expression, e, the rule ATT-EXP first assigns a type, τ, and an *abstract translation*, γ_{abs}, to it, as determined by the judgement $\Gamma \vdash_\Phi e : \tau \Longrightarrow \gamma_{\text{abs}}$. It then *deabstracts* this abstract translation to complete the compilation process, as determined by the judgement $\vdash_\Phi \gamma_{\text{abs}} \not\downarrow \gamma$. The purpose of the abstract translation phase is to ensure that the implementation details of type families are not exposed to other families, so that invariants that they rely on are preserved when families are composed. For example, our implementation of natural numbers as integers in Fig. 3 maintains the invariant that the translation is non-negative. If the knowledge that natural numbers are implemented as integers was externally visible, a different extension could introduce an operator like ***badnat***$[1](\mathbf{i}, \mathbf{a}.\mathbf{const}\ \mathbf{a}\ \llbracket\nabla(-1)\ \mathsf{as}\ \textsc{Nat}\langle()\rangle\rrbracket)$, breaking this invariant (and thus the guarantee that the recursor always terminates!)

We preclude such operations by a mechanism similar to the abstract type mechanism supported by ML-style module systems [8]. By keeping a type family's representation schema private to the operators associated with it, they maintain full control over representation invariants. So, because it cannot be shown given the knowledge available in the type family containing ***badnum*** that the internal type associated with $\textsc{Nat}\langle()\rangle$ is $\mathbb{Z}$, the translation -1 will not be permitted according to the rules we will give below. The only way to produce a term of type $\textsc{Nat}\langle()\rangle$ as the result of applying an operator not associated with the family NAT is if that operator extracts the term from an argument (e.g. when projecting it out of an n-tuple), in which case the necessary invariants are inductively maintained (see Sec. 4). Translations extracted from denotations via case analysis are tracked during this phase as *abstract internal terms*, $\mathsf{trans}(\llbracket\tau_1\ \mathsf{as}\ \tau_2\rrbracket)$ (see rule DENCASE-EVAL-VALID).

Let us first review how expressions are assigned types and abstract translations. Variables (ATT-VAR) translate directly to vari-

ables in the internal language, with the type determined by the typing context, Γ. Lambda terms (ATT-LAM) also translate to lambda terms in the internal language and are given the appropriate ARROW type by extending the typing context and proceeding recursively as is usual. The internal type of the argument, however, is left as $\mathsf{rep}(\tau_1')$, which is called the *abstract representation* of τ_1'. The actual representation of this type is only available from within operators in its family.

Now we will consider the important operator application rule (ATT-OP). This rule operates by extracting the appropriate operator definition, evaluating the operator index to a value, ${\tau_{\text{idx}}}'$, and recursively assigning a type and abstract translation to each argument. From these, it constructs a list of denotations and passes this list, along with the fully evaluated operator index, to the operator implementation, τ_{op}. If this results in a valid denotation, compilation can proceed, but if an err is produced, compilation will stop because no other rule will apply (in practice, we would display a type error at this point).

Before a valid denotation produced in this way can be used, however, we check that it is *representationally consistent*, meaning that the abstract translation, γ_{abs}, is of an abstract internal type consistent with the abstraction representation of the type it is paired with, τ. The premise $\vdash_{\Phi}^{\Xi_0,\text{FAM}} \mathsf{rep}(\tau) \rightsquigarrow \sigma_{\text{abs}}$ determines the abstract internal type and the premise $\Psi \vdash_{\Phi}^{\Xi_0,\text{FAM}} \gamma_{\text{abs}} \sim \sigma_{\text{abs}}$ performs the internal type checking. To do so, the variable context, Γ, which associates variables with types, must be converted to a context associating those variables with their corresponding abstract internal types, Ψ. The relevant judgements are defined in Fig. 9. The schema context Ξ is used to track which representation schemas are visible. In our calculus, $\Xi_0 = \text{ARROW}$ is always visible alongside the schema of the family associated with the operator being considered. This could be exanded to support module-scoped visibility (we do not include modules in our calculus for simplicity), mutually-defined type families or type families that intentionally expose their representation schemas publicly (as ARROW does).

To clarify this fundamental mechanism let us examine how the expression $\text{NAT}.\boldsymbol{s}\langle()\rangle(\text{NAT}.\boldsymbol{z}\langle()\rangle())$ will be processed during this phase. The inner application of $\boldsymbol{z}$ will produce the abstract translation 0 paired with the type $\text{NAT}\langle()\rangle$. Because the representation schema for NAT is available, we have that $\vdash_{\Phi}^{\Xi_0,\text{NAT}} \mathsf{rep}(\text{NAT}\langle()\rangle) \rightsquigarrow \mathbb{Z}$ by SHOW-REP, as needed. In contrast, if ***badnat*** were used inside, the representation schema of NAT would not be available, so we can only apply HIDE-REP, which does not give us enough information to give -1 a type.

When the compiler next considers the outer application of $\boldsymbol{s}$, it will pass the denotation $[\![\triangledown(0)\ \mathsf{as}\ \text{NAT}\langle()\rangle]\!]$ into its definition. There, it binds the translation to the variable **x** (within the **pop_final** helper function). However, **x** is not simply $\triangledown(0)$. Instead, its provenance is tracked by using the form $\mathsf{trans}([\![\triangledown(0)\ \mathsf{as}\ \text{NAT}\langle()\rangle]\!])$. When attempting to check that $\triangle\ (\mathbf{x}) + 1$ is valid, the SHOW-TRANS rule will reveal that it is an integer because, again, the appropriate representation schema is available. If the schema were not available, the most that could be derived is that $\Psi \vdash_{\Phi}^{\Xi} \triangle(\mathbf{x}) \sim \mathsf{rep}(\text{NAT}\langle()\rangle)$. The fact that natural numbers are represented using integers is not derivable (though it is true), so the addition operation would fail to type. This fact would, however, be sufficient for implementing families, like NTUPLE, that do not require knowledge about a type's representation. We encourage the reader to derive these judgements for the logic in the ***rec*** operator to strengthen their understanding of this fundamental mechanism.

We cannot leave abstract representations and abstract internal terms in the result of compilation, so a final deabstraction phase erases these, replacing them with their underlying internal types and terms in all contexts. Just as with abstract types in modules or existential types, there is no run-time overhead to this mechanism – programs run at full speed. We give the deabstraction rules in the appendix due to their simplicity.

4. Safety of @λ

In giving users of a language direct influence on the typechecking and translation of expressions, it is essential to consider safety properties. It must not be possible for well-typed programs to fail to finish compiling or go wrong at run-time because of a buggy extension. It is also considered desirable for typechecking to be decidable. And for extensions to be reliable, they must not be allowed to interfere with one another under any circumstance. By carefully designing our extension mechanism, we believe we have achieved each of these goals.

4.1 Representational Consistency and Type Safety

We do not need to give a semantics to internal terms and internal types that do not survive deabstraction. The remaining terms and types form a variant of PCF with well-understood primitives (here, integers and binary products) known to be type safe [8]. If compilation can be shown to always result in a well-typed term of this language, then type safety follows by composing these two facts. Fortunately, this fact is a corollary of our representational consistency mechanism, described in Sec. 3.8. The proof is by straightforward (though deeply nested!) rule induction.

[Representational Consistency] If $\vdash_{\Sigma} e\ \mathtt{prog}$ and $\vdash_{\Phi} e : \tau \Longrightarrow \gamma_{\text{abs}}$ and $\vdash_{\Phi} \gamma_{\text{abs}} \not\downarrow \gamma$ and $\vdash_{\Phi}^{\Xi_0} \mathsf{rep}(\tau) \rightsquigarrow \sigma_{\text{abs}}$ and $\vdash_{\Phi} \sigma_{\text{abs}} \not\downarrow \sigma$ then $\vdash_{\Phi}^{\Xi_0} \gamma \sim \sigma$.

4.2 Decidability

To show that compilation is decidable, we need to show that both kind checking and active typechecking and compilation are decidable. This hinges on showing that the type-level language is type-safe and that evaluation of type-level terms always terminates.

[Kind Safety and Termination] If $\vdash_{\Sigma} \tau : \kappa$ then $\tau \Downarrow \tau'$ such that $\vdash_{\Sigma_0} \tau' : \kappa$ and $\tau' \Downarrow \tau'$.

The proof is straightforward because the type-level language is based on a conventional simply-typed lambda calculus with only bounded recursion over lists, so standard techniques (e.g. logical relations) can be used directly. The only new constructs are the types, denotations and reified internal language forms, but because none of these can contain type-level functions by the statics, there is no risk of introducing self-reference by their inclusion. If the type-level language included constructs like inductive datatypes (without a strict positivity condition) or general recursion, this theorem would be weakened.

4.3 Non-Interference

Our abstraction mechanism guarantees that the representation invariants collectively maintained by the operators associated with a type family cannot be violated by operators in other type families, by ensuring that introductory forms cannot be defined outside the family. One way to state this is as follows:

[Non-Interference] For any property $P(\gamma)$, if $\vdash_{\Sigma} e\ \mathtt{prog}$ and $\vdash_{\Phi} e : \tau \Longrightarrow \gamma_{\text{abs}}$ and $\vdash_{\Phi} \gamma_{\text{abs}} \not\downarrow \gamma$ implies $P(\gamma)$, then for any Σ' and Φ' disjoint from Σ and Φ, we have that if $\vdash_{\Sigma,\Sigma'} e'\ \mathtt{prog}$ and $\vdash_{\Phi,\Phi'} e' : \tau \Longrightarrow \gamma'_{\text{abs}}$ and $\vdash_{\Phi} \gamma'_{\text{abs}} \not\downarrow \gamma'$ then $P(\gamma')$. The proof relies on the fact that the show/hide rules are mutually exclusive and so there are two cases to consider for each form of expression, and weakening properties of the family contexts. This powerful theorem allows us to compose type families arbitrarily without needing to handle cases in our implementation that are ruled out based on a local analysis. The details will be provided in the appendix.

5. Use Cases

Indexed type families and indexed operator families are a powerful tool for programming language semantics. In *Practical Foundations for Programming Languages*, for example, nearly every chapter defines a new collection of indexed types and operators and gives their semantics in isolation from the constructs in the other chapters [8]. For example, n-ary sums and products are families indexed by a list of types. Labeled variants of these simply pair the labels with the types as static strings. Even nested pattern matching, such as that found in modern functional languages, can be understood as an operator family indexed by a series of patterns, which can be represented as type-level data. In the appendix, we show how to implement sums and products in @λ.

The book also shows constructs as varied as dynamic types as well as a number of constructs for parallelism and concurrency, such as futures, promises and actors. With a sufficiently capable internal language (exposing basic concurrency primitives, for example), each of these could be fully implemented as libraries using our mechanism. Few languages include primitive support for several such abstractions, even though they are often useful together. In a preliminary implementation of this calculus, Ace, which is beyond the scope of this paper, we have implemented the entirety of the OpenCL programming language as a library, along with primitives supporting partitioned global address spaces.

Object systems too could be implemented using this mechanism, with indices serving to capture the inheritance data and the signatures. Operator families for reading and writing fields and/or sending messages would be parameterized by strings naming them. The operator definition would simply search the signatures going up the inheritance hierarchy, and implement objects in a conventional way (e.g. using a v-table together with a pointer to a structure containing field data). A variety of object systems could coexist within the same language. Another related example would be to implement a safe and efficient interoperability layer with an existing OO language by capturing its type system as an extension, to be used only at language boundaries.

Finally, a variety of domain-specific type systems, capturing complex rule systems like the system of scientific units of measure (a family indexed by the measure being used and the type which the measure is being applied to), regular expressions [**?**] (a family indexed by the number of captured groups) or XML [**?**] (indexed by a document schema) would be possible. Recent work examined a specialized type system capturing the semantics of the widely-used jQuery javascript library [**?**]. Today, these all require *ad hoc* language-external solutions, or the development of new languages.

6. Related Work

Our representation schema abstraction mechanism relates closely to abstract and existential types [8**?**]. Our calculus enforces the abstraction barriers in a purely syntactic manner, as in previous work on syntactic type abstraction [**?**]. While this work is all focused on abstracting away the identity of a particular type outside of the "principal" it is associated with (e.g. a module), we focus on abstracting away the knowledge of how a primitive type family is implemented outside of a limited scope.

The representational consistency mechanism brings into the language work on typed compilation, especially work done on Standard ML in the TILT project [**?**]. Indeed, the specification of Standard ML is structured around a typed internal language and a judgement that assigns a type and an internal term to each expression [**?**]. Representational consistency is related to the notion of type-directed copmilation in this work.

Type-level computation is supported in some form by a growing number of languages. For example, Haskell supports a simple form of it [2]). Ur uses type-level records and names to support type-safe metaprogramming, with applications to web programming [4]. Ωmega adds algebraic data types at the type-level, using these to increase the expressive power of algebraic data types at the expression level [12]. 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 [3], ATS [5]). We show how to integrate language extensions into the type-level language, drawing on ideas about [13].

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 [10], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [11]), and external rewrite systems also exist for many languages. These facilities differ from AT&T in that they involve direct manipulation of terms, while AT&T involves extending typechecking and translation logic directly. We also draw a distinction between the type-level, used to specify types and compile-time logic, the expression grammar, used to describe run-time behavior, and the internal language, used to implement this behavior. By doing so, each component can be structured and constrained as appropriate for its distinct role, as we have shown.

Previous work on extensible languages has suffered from problems with either expressiveness or safety. For example, a number of projects, such as SugarJ [6], allow for user-defined desugarings (and indeed, our system would clearly benefit from integration with such a mechanism), but this does not allow the semantics to be fundamentally extended nor for implementation details to be hidden. Recent variants of this work has investigated introducing new typing rules, but only if they are admissible by the base type system [**?**]. Our work allows for entirely new logic to be added to the system, requiring only that the implementation of this logic respect the internal type system. A number of other extensible languages (e.g. Xroma [**?**]) and compilers do allow new static semantics to be introduced, but in an unconstrained manner based on global pattern matching and rewriting, leading to significant problems with interference and safety. Our work aims to provide a sound and safe underpinning, based around well-understood concepts of type and operator families, to language extensibility.

In the future, we will investigate mechanisms to enable type systems with specialized binding and scoping rules, as well as integration of dependent kinds to support mechanized verification of key properties like representational consistency and adequacy against a declarative specification at kind-checking time. We will also investigate mechanisms that enable a more natural syntax (e.g. Wyvern's type-directed syntax [**?**]).

References

[1] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, 2011.

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

[8] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[9] A. Kennedy. Types for units-of-measure: Theory and practice. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2009.

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

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

[12] 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.

[13] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proc. 1998 SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

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

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

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

$$\textsc{k-arrow-elim}\ \frac{\Delta \vdash_\Sigma \tau_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash_\Sigma \tau_2 : \kappa_1}{\Delta \vdash_\Sigma \tau_1\ \tau_2 : \kappa_2}$$

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

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

$$\textsc{type-elim}\ \frac{\textsc{Fam}[\kappa_{\text{idx}}, \Theta] \in \Sigma \qquad \Delta \vdash_\Sigma \tau : \star \qquad \Delta, \mathbf{x} : \kappa_{\text{idx}} \vdash_\Sigma \tau_0 : \kappa \qquad \Delta \vdash_\Sigma \tau_1 : \kappa}{\Delta \vdash_\Sigma \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle \mathbf{x} \rangle \Rightarrow \tau_0\ \mathsf{ow}\ \tau_1 : \kappa} \qquad \textsc{den-intro-valid}\ \frac{\Delta \vdash_\Sigma \tau_1 : \mathsf{ITm} \qquad \Delta \vdash_\Sigma \tau_2 : \star}{\Delta \vdash_\Sigma [\![\tau_1\ \mathsf{as}\ \tau_2]\!] : \mathsf{Den}} \qquad \textsc{den-intro-err}\ \frac{}{\Delta \vdash_\Sigma \mathsf{err} : \mathsf{Den}}$$

$$\textsc{den-elim}\ \frac{\Delta \vdash_\Sigma \tau : \mathsf{Den} \qquad \Delta, \mathbf{x} : \mathsf{ITm}, \mathbf{t} : \star \vdash_\Sigma \tau_1 : \kappa \qquad \Delta \vdash_\Sigma \tau_2 : \kappa}{\Delta \vdash_\Sigma \mathsf{case}\ \tau\ \mathsf{of}\ [\![\mathbf{x}\ \mathsf{as}\ \mathbf{t}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 : \kappa} \qquad \textsc{iterm-intro}\ \frac{\Delta\ \emptyset \vdash_\Sigma \gamma\ \mathtt{iterm}}{\Delta \vdash_\Sigma \triangledown(\gamma) : \mathsf{ITm}}$$

$$\textsc{itype intro}\ \frac{\Delta \vdash_\Sigma \sigma\ \mathtt{itype}}{\Delta \vdash_\Sigma \blacktriangledown(\sigma) : \mathsf{ITy}}$$

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

$$\textsc{t-eq}\ \frac{}{\star\ \mathsf{eq}} \qquad \textsc{z-eq}\ \frac{}{\mathbb{Z}\ \mathsf{eq}} \qquad \textsc{s-eq}\ \frac{}{\mathsf{Str}\ \mathsf{eq}} \qquad \textsc{u-eq}\ \frac{}{1\ \mathsf{eq}} \qquad \textsc{p-eq}\ \frac{\kappa_1\ \mathsf{eq} \qquad \kappa_2\ \mathsf{eq}}{\kappa_1 \times \kappa_2\ \mathsf{eq}}$$

$$\textsc{l-eq}\ \frac{\kappa\ \mathsf{eq}}{\mathsf{list}[\kappa]\ \mathsf{eq}}$$

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

$$\textsc{i-var-kinding}\ \frac{}{\Delta\ \Omega, x \vdash_\Sigma x\ \mathtt{iterm}} \qquad \textsc{i-lam-kinding}\ \frac{\Delta \vdash_\Sigma \sigma\ \mathtt{itype} \qquad \Delta\ \Omega, x \vdash_\Sigma \gamma\ \mathtt{iterm}}{\Delta\ \Omega \vdash_\Sigma \lambda x{:}\sigma.\gamma\ \mathtt{iterm}}$$

$$\textsc{i-fix-kinding}\ \frac{\Delta \vdash_\Sigma \sigma\ \mathtt{itype} \qquad \Delta\ \Omega, x \vdash_\Sigma \gamma\ \mathtt{iterm}}{\Delta\ \Omega \vdash_\Sigma \mathsf{fix}\ f{:}\sigma\ \mathsf{is}\ \gamma\ \mathtt{iterm}}$$

(omitted forms have trivially recursive rules)

$$\textsc{iterm-dereify-kinding}\ \frac{\Delta \vdash_\Sigma \tau : \mathsf{ITm}}{\Delta\ \Omega \vdash_\Sigma \triangle(\tau)\ \mathtt{iterm}} \qquad \textsc{abs-trans-kinding}\ \frac{\Delta \vdash_\Sigma \tau_{\text{iterm}} : \mathsf{ITm} \qquad \Delta \vdash_\Sigma \tau_{\text{type}} : \star}{\Delta\ \Omega \vdash_\Sigma \mathsf{trans}([\![\tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}}]\!])\ \mathtt{iterm}}$$

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

$$\textsc{i-int-kinding}\ \frac{}{\Delta \vdash_\Sigma \mathbb{Z}\ \mathtt{itype}} \qquad \textsc{i-prod-kinding}\ \frac{\Delta \vdash_\Sigma \sigma_1\ \mathtt{itype} \qquad \Delta \vdash_\Sigma \sigma_2\ \mathtt{itype}}{\Delta \vdash_\Sigma \sigma_1 \times \sigma_2\ \mathtt{itype}}$$

$$\textsc{i-arrow-kinding}\ \frac{\Delta \vdash_\Sigma \sigma_1\ \mathtt{itype} \qquad \Delta \vdash_\Sigma \sigma_2\ \mathtt{itype}}{\Delta \vdash_\Sigma \sigma_1 \to \sigma_2\ \mathtt{itype}} \qquad \textsc{itype-dereify-kinding}\ \frac{\Delta \vdash_\Sigma \tau : \mathsf{ITy}}{\Delta \vdash_\Sigma \blacktriangle(\tau)\ \mathtt{itype}}$$

$$\textsc{abs-rep-kinding}\ \frac{\Delta \vdash_\Sigma \tau : \star}{\Delta \vdash_\Sigma \mathsf{rep}(\tau)\ \mathtt{itype}}$$

Figure 7. Kinding for type-level terms

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

TL-LAM-EVAL
$$\frac{}{\lambda \mathbf{t}{:}\kappa.\tau \Downarrow \lambda \mathbf{t}{:}\kappa.\tau}$$

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)

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

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

TYPE-EVAL
$$\frac{\tau_{\text{idx}} \Downarrow \tau_{\text{idx}}'}{\textsc{Fam}\langle \tau_{\text{idx}} \rangle \Downarrow \textsc{Fam}\langle \tau_{\text{idx}}' \rangle}$$

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

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

DEN-VALID-EVAL
$$\frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_2'}{[\![\tau_1\ \mathsf{as}\ \tau_2]\!] \Downarrow [\![\tau_1'\ \mathsf{as}\ \tau_2']\!]}$$

DEN-ERR-EVAL
$$\frac{}{\mathsf{err} \Downarrow \mathsf{err}}$$

DENCASE-EVAL-VALID
$$\frac{\tau \Downarrow [\![\tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}}]\!] \qquad [\nabla(\mathsf{trans}([\![\tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}}]\!]))/\mathbf{x}, \tau_{\text{type}}/\mathbf{t}]\tau_1 \Downarrow \tau_1'}{\mathsf{case}\ \tau\ \mathsf{of}\ [\![\mathbf{x}\ \mathsf{as}\ \mathbf{t}]\!] \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_1'}$$

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

ITERM-REIFY
$$\frac{\gamma \curlyvee \gamma'}{\nabla(\gamma) \Downarrow \nabla(\gamma')}$$

ITYPE-REIFY
$$\frac{\sigma \curlyvee \sigma'}{\blacktriangledown(\sigma) \Downarrow \blacktriangledown(\sigma')}$$

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

I-VAR-EVAL
$$\frac{}{x \curlyvee x}$$

I-LAM-EVAL
$$\frac{\sigma \curlyvee \sigma' \qquad \gamma \curlyvee \gamma'}{\lambda x{:}\sigma.\gamma \curlyvee \lambda x{:}\sigma'.\gamma'}$$

I-FIX-EVAL
$$\frac{\sigma \curlyvee \sigma' \qquad \gamma \curlyvee \gamma'}{\mathsf{fix}\ f{:}\sigma\ \mathsf{is}\ \gamma \curlyvee \mathsf{fix}\ f{:}\sigma'\ \mathsf{is}\ \gamma'}$$

(omitted forms have trivially recursive rules)

ITERM UNQUOTE EVAL
$$\frac{\tau \Downarrow \tau'}{\triangle(\tau) \curlyvee \triangle(\tau')}$$

VAL FROM DEN EVAL
$$\frac{\tau_{\text{iterm}} \Downarrow \tau_{\text{iterm}}' \qquad \tau_{\text{type}} \Downarrow \tau_{\text{type}}'}{\mathsf{trans}([\![\tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}}]\!]) \curlyvee \mathsf{trans}([\![\tau_{\text{iterm}}'\ \mathsf{as}\ \tau_{\text{type}}']\!])}$$

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

I-INT-EVAL
$$\frac{}{\mathbb{Z} \curlyvee \mathbb{Z}}$$

I-PROD-EVAL
$$\frac{\sigma_1 \curlyvee \sigma_1' \qquad \sigma_2 \curlyvee \sigma_2'}{\sigma_1 \times \sigma_2 \curlyvee \sigma_1' \times \sigma_2'}$$

I-ARROW-EVAL
$$\frac{\sigma_1 \curlyvee \sigma_1' \qquad \sigma_2 \curlyvee \sigma_2'}{\sigma_1 \to \sigma_2 \curlyvee \sigma_1' \to \sigma_2'}$$

ITYPE-EVAL
$$\frac{\tau \Downarrow \tau'}{\blacktriangle(\tau) \curlyvee \blacktriangle(\tau')}$$

ABS-REP-EVAL
$$\frac{\tau \Downarrow \tau'}{\mathsf{rep}(\tau) \curlyvee \mathsf{rep}(\tau')}$$

Figure 8. Evaluation semantics for type-level terms

$\boxed{\vdash^{\Xi}_{\Phi} \tau \rightsquigarrow \sigma}$

GET-REP
$$\frac{\textsc{Fam}[\theta, \mathbf{i}.\tau] \in \Phi \qquad [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\sigma) \qquad \vdash^{\Xi}_{\Phi} \sigma \rightsquigarrow \sigma_{\text{conc}}}{\vdash^{\Xi}_{\Phi} \textsc{Fam}\langle \tau_{\text{idx}} \rangle \rightsquigarrow \sigma_{\text{conc}}}$$

$\boxed{\vdash^{\Xi}_{\Phi} \sigma \rightsquigarrow \sigma'}$ $\quad \Phi ::= \Phi_0 \mid \Phi, \textsc{Fam}[\theta, \mathbf{i}.\tau_{\text{rep}}] \quad \Xi ::= \Xi_0 \mid \Xi, \textsc{Fam} \quad \Xi_0 := \textsc{Arrow}$

ABS-INT
$$\frac{}{\vdash^{\Xi}_{\Phi} \mathbb{Z} \rightsquigarrow \mathbb{Z}}$$

ABS-ARROW
$$\frac{\vdash^{\Xi}_{\Phi} \sigma_1 \rightsquigarrow \sigma_1' \qquad \vdash^{\Xi}_{\Phi} \sigma_2 \rightsquigarrow \sigma_2'}{\vdash^{\Xi}_{\Phi} \sigma_1 \to \sigma_2 \rightsquigarrow \sigma_1' \to \sigma_2'}$$

ABS-PROD
$$\frac{\vdash^{\Xi}_{\Phi} \sigma_1 \rightsquigarrow \sigma_1' \qquad \vdash^{\Xi}_{\Phi} \sigma_2 \rightsquigarrow \sigma_2'}{\vdash^{\Xi}_{\Phi} \sigma_1 \times \sigma_2 \rightsquigarrow \sigma_1' \times \sigma_2'}$$

ITYPE-INVERSE
$$\frac{\vdash^{\Xi}_{\Phi} \sigma \rightsquigarrow \sigma_{\text{abs}}}{\vdash^{\Xi}_{\Phi} \blacktriangle(\blacktriangledown(\sigma)) \rightsquigarrow \sigma_{\text{abs}}}$$

SHOW-REP
$$\frac{\textsc{Fam} \in \Xi \qquad \vdash^{\Xi}_{\Phi} \textsc{Fam}\langle \tau_{\text{idx}} \rangle \rightsquigarrow \sigma_{\text{conc}}}{\vdash^{\Xi}_{\Phi} \mathsf{rep}(\textsc{Fam}\langle \tau_{\text{idx}} \rangle) \rightsquigarrow \sigma_{\text{conc}}}$$

HIDE-REP
$$\frac{\textsc{Fam} \notin \Xi}{\vdash^{\Xi}_{\Phi} \mathsf{rep}(\textsc{Fam}\langle \tau_{\text{idx}} \rangle) \rightsquigarrow \mathsf{rep}(\textsc{Fam}\langle \tau_{\text{idx}} \rangle)}$$

$\boxed{\vdash^{\Xi}_{\Phi} \Gamma \rightsquigarrow \Psi}$ $\quad \Psi ::= \emptyset \mid \Psi, x : \sigma$

ABS-EMPTY
$$\frac{}{\vdash^{\Xi}_{\Phi} \emptyset \rightsquigarrow \emptyset}$$

ABS-CTX
$$\frac{\vdash^{\Xi}_{\Phi} \Gamma \rightsquigarrow \Psi \qquad \vdash^{\Xi}_{\Phi} \mathsf{rep}(\tau) \rightsquigarrow \sigma}{\vdash^{\Xi}_{\Phi} \Gamma, x : \tau \rightsquigarrow \Psi, x : \sigma}$$

$\boxed{\Psi \vdash^{\Xi}_{\Phi} \gamma \sim \sigma}$

ABS-I-VAR
$$\frac{}{\Psi, x : \sigma \vdash^{\Xi}_{\Phi} x \sim \sigma}$$

ABS-I-LAM
$$\frac{\vdash^{\Xi}_{\Phi} \sigma_1 \rightsquigarrow \sigma_1' \qquad \Psi, x : \sigma_1' \vdash^{\Xi}_{\Phi} \gamma \sim \sigma_2}{\Psi \vdash^{\Xi}_{\Phi} \lambda x{:}\sigma_1.\gamma \sim \sigma_1' \to \sigma_2}$$

ABS-I-AP
$$\frac{\Psi \vdash^{\Xi}_{\Phi} \gamma_1 \sim \sigma_1 \to \sigma_2 \qquad \Psi \vdash^{\Xi}_{\Phi} \gamma_2 \sim \sigma_1}{\Psi \vdash^{\Xi}_{\Phi} \gamma_1\ \gamma_2 \sim \sigma_2}$$

ABS-I-FIX
$$\frac{\vdash^{\Xi}_{\Phi} \sigma \rightsquigarrow \sigma' \qquad \Psi, x : \sigma' \vdash^{\Xi}_{\Phi} \gamma \sim \sigma'}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \gamma_{\text{abs}} \sim \sigma'}$$

(standard statics for integers and products omitted)

ABS-ITERM-INVERSE
$$\frac{\Psi \vdash^{\Xi}_{\Phi} \gamma \sim \sigma}{\Psi \vdash^{\Xi}_{\Phi} \triangle(\nabla(\gamma)) \sim \sigma}$$

SHOW-TRANS
$$\frac{\textsc{Fam} \in \Xi \qquad \vdash^{\Xi}_{\Phi} \textsc{Fam}\langle \tau_{\text{idx}} \rangle \rightsquigarrow \sigma \qquad \Psi \vdash^{\Xi}_{\Phi} \gamma \sim \sigma}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{trans}([\![\nabla(\gamma)\ \mathsf{as}\ \textsc{Fam}\langle \tau_{\text{idx}} \rangle]\!]) \sim \sigma}$$

HIDE-TRANS
$$\frac{\textsc{Fam} \notin \Xi \qquad \vdash^{\Xi}_{\Phi} \textsc{Fam}\langle \tau_{\text{idx}} \rangle \rightsquigarrow \sigma \qquad \Psi \vdash^{\Xi}_{\Phi} \gamma \sim \sigma}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{trans}([\![\nabla(\gamma)\ \mathsf{as}\ \textsc{Fam}\langle \tau_{\text{idx}} \rangle]\!]) \sim \mathsf{rep}(\textsc{Fam}\langle \tau_{\text{idx}} \rangle)}$$

Figure 9. Abstracted internal typing