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 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 variety of interesting *embedded domain-specific languages* [**?**]. 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 governing general-purpose abstractions (e.g. reasoning about units of measure requires built-in language support in F# [**?**]). 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 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 not easily 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. Using variants of familiar abstractions backed by specialized implementation strategies are quite useful in these cases.

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 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 [15]. 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 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. ForMoreover, 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). As an example, F#'s type system prevents `null` values from occurring within data structures, but because it's type system is not available when calling into F# code from another language, like C#, run-time null checks must still be included in the implementation.

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 thus achieve full safety, ease-of-use and performance due to the absence of wrappers and glue code. Providers of components thus need

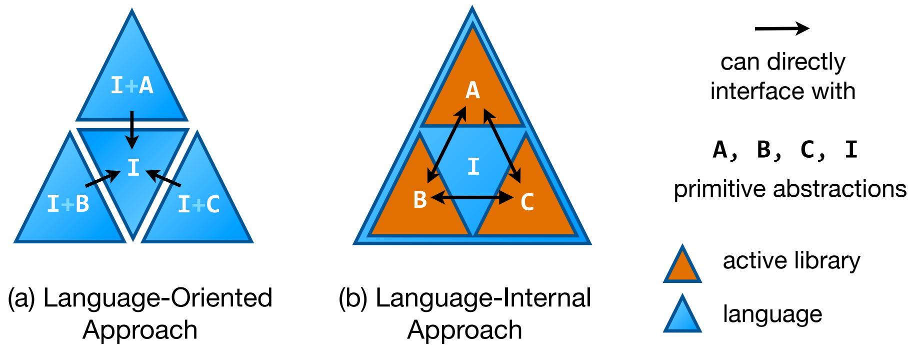
I+A
I
I+B
I+C
A
I
B
C
can directly interface with
A, B, C, I
primitive abstractions
active library
language
(a) Language-Oriented Approach
(b) Language-Internal Approach

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.

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 containing logic that is invoked at compile-time, as extension logic would be, have been called *active libraries* [14]. 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 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 simplify our presentation and analysis, but this syntax is 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 type families* – supports 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 (see Appendix A.1). 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. This issue is related to the widely-discussed *expression problem* (in a restricted sense – we do not consider adding new functions beyond typechecking and translation here, only adding logic to these) [?].

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++. We argue that this practice should be considered harmful.

A more appropriate and useful place for extensions like this is directly within libraries, alongside abstractions that do not require this level of control. 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, without allowing so much control that the reliability of the system suffers. 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

$$
\begin{array}{rcl}
\textbf{programs}\ \rho & ::= & \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 \\
\text{primitive ops}\ \theta & ::= & \boldsymbol{op}[\kappa_{\text{idx}}](\mathbf{i}, \mathbf{a}.\tau) \mid \theta;\theta \\
\\
\textbf{expressions}\ e & ::= & x \mid \lambda x{:}\tau.e \mid \textsc{Fam}.\boldsymbol{op}\langle\tau_{\text{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{h},\mathbf{t},\mathbf{r}.\tau_3) \\
\text{type-level data} & \mid & \bar{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 & \llbracket \tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}} \rrbracket \mid \mathsf{err} \mid \mathsf{case}\ \tau\ \mathsf{of}\ \llbracket \mathbf{x}\ \mathsf{as}\ \mathbf{t} \rrbracket \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
\text{reified 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{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{trans}(\llbracket \tau_1\ \mathsf{as}\ \tau_2 \rrbracket) \mid \vartriangle(\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 $\textsc{Fam}$ are family names (we assume that unique family names can be generated by some external mechanism) and $\boldsymbol{op}$ are operator names. *"str"* denotes string literals, $\bar{z}$ denotes integer literals and $\oplus$ stands for binary operations over integers.

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 language's static 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. 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 demonstrate that it is quite natural to unify the two in this work.

3 @$\boldsymbol{\lambda}$

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 when giving users this level of control over the semantics of a language and its implementation.

3.1 Overview

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 are type-level values of kind $\star$ (following System F_ω [?]) and classify expressions, e. The type-level language also includes other kinds of terms, such as type-level functions, lists (required by our mechanism) and integers, strings and products for the sake of our examples (see Sec. **??** for a discussion on other acceptable kinds of type-level data). It also includes constructs for developing extensions – denotations and reified internal terms and types – which we will discuss in the sections below.

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. Expressions can be either variables, lambdas, or applications of operators, 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. The grammars of internal terms, γ, and internal types, σ, also include special forms containing type-level terms; these are used for developing extensions and during compilation and will be erased before compilation ends, as we will explain below.

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 [?]. These will be implemented internally as integers (internal terms of internal type $\mathbb{Z}$). Figure 3 shows how the indexed type family $\textsc{Nat}$ is defined. This family is indexed by a unit value (of kind 1), so it contains only one type, $\textsc{Nat}\langle()\rangle$, 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 ($\boldsymbol{z}$, $\boldsymbol{s}$ and $\boldsymbol{rec}$) on lines 2-11 and use these to define a *plus* function on line 16 and compute 2+2.

3.3 Indexed Type Families and Types

The syntactic form family $\textsc{Fam}[\kappa_{\text{idx}}] \sim \mathbf{i}.\tau_{\text{rep}}\ \{\theta\}$ declares a new primitive type family named $\textsc{Fam}$ indexed by type-level values of kind κ_{idx} with representation schema $\mathbf{i}.\tau_{\text{rep}}$ and operators θ. This can be compared to adding a new constructor to the compiler-internal datatype `Type`, as suggested in Sec. 2. The index represents the data associated with the constructor. A base type like **nat** can be

$$\textsf{family}\ \textsc{Nat}[1] \sim \mathbf{i}.\blacktriangledown(\mathbb{Z})\ \{ \tag{1}$$
$$\quad \boldsymbol{z}[1](\mathbf{i}, \mathbf{a}.\mathbf{const}\ \mathbf{a}\ [\![\triangledown(0)\ \textsf{as}\ \textsc{Nat}\langle()\rangle]\!]); \tag{2}$$
$$\quad \boldsymbol{s}[1](\mathbf{i}, \mathbf{a}.\mathbf{pop\_final}\ \mathbf{a}\ \lambda\mathbf{x}{:}\textsf{ITm}.\lambda\mathbf{t}{:}\star. \tag{3}$$
$$\qquad \mathbf{check\_type}\ \mathbf{t}\ \textsc{Nat}\langle()\rangle\ [\![\triangledown(\vartriangle(\mathbf{x})+1)\ \textsf{as}\ \textsc{Nat}\langle()\rangle]\!]); \tag{4}$$
$$\quad \boldsymbol{rec}[1](\mathbf{i}, \mathbf{a}.\mathbf{pop}\ \mathbf{a}\ \lambda\mathbf{x1}{:}\textsf{ITm}.\lambda\mathbf{t1}{:}\star.\lambda\mathbf{a}{:}\textsf{list}[\textsf{Den}]. \tag{5}$$
$$\qquad \mathbf{pop}\ \mathbf{a}\ \lambda\mathbf{x2}{:}\textsf{ITm}.\lambda\mathbf{t2}{:}\star.\lambda\mathbf{a}{:}\textsf{list}[\textsf{Den}]. \tag{6}$$
$$\qquad \mathbf{pop\_final}\ \mathbf{a}\ \lambda\mathbf{x3}{:}\textsf{ITm}.\lambda\mathbf{t3}{:}\star. \tag{7}$$
$$\qquad \mathbf{check\_type}\ \mathbf{t1}\ \textsc{Nat}\langle()\rangle\ (\tag{8}$$
$$\qquad \mathbf{check\_type}\ \mathbf{t3}\ \textsc{Arrow}\langle(\textsc{Nat}\langle()\rangle, \textsc{Arrow}\langle(\mathbf{t2},\mathbf{t2})\rangle)\rangle \tag{9}$$
$$\qquad [\![\triangledown((\textsf{fix}\ f{:}\mathbb{Z}\to\textsf{rep}(\mathbf{t2})\ \textsf{is}\ \lambda x{:}\mathbb{Z}. \tag{10}$$
$$\qquad\quad \textsf{if}\ x \equiv_{\mathbb{Z}} 0\ \textsf{then}\ \ \vartriangle(\mathbf{x2})\ \textsf{else}\ \ \vartriangle(\mathbf{x3})\ (x-1)\ (f\ (x-1)))\ \ \vartriangle(\mathbf{x1}))\ \textsf{as}\ \mathbf{t2}]\!])) \tag{11}$$
$$\}; \tag{12}$$
$$\textsf{def}\ \mathbf{nat} : \star = \textsc{Nat}\langle()\rangle; \tag{13}$$
$$(\lambda \mathit{plus}{:}\textsc{Arrow}\langle(\mathbf{nat}, \textsc{Arrow}\langle(\mathbf{nat},\mathbf{nat})\rangle)\rangle.\lambda \mathit{two}{:}\mathbf{nat}. \tag{14}$$
$$\qquad \textsc{Arrow}.\boldsymbol{ap}\langle()\rangle(\textsc{Arrow}.\boldsymbol{ap}\langle()\rangle(\mathit{plus};\mathit{two});\mathit{two})) \tag{15}$$
$$\quad (\lambda x{:}\mathbf{nat}.\lambda y{:}\mathbf{nat}.\textsc{Nat}.\boldsymbol{rec}\langle()\rangle(x;y;\lambda p{:}\mathbf{nat}.\lambda r{:}\mathbf{nat}.\textsc{Nat}.\boldsymbol{s}\langle()\rangle(r))) \tag{16}$$
$$\quad \textsc{Nat}.\boldsymbol{s}\langle()\rangle(\textsc{Nat}.\boldsymbol{s}\langle()\rangle(\textsc{Nat}.\boldsymbol{z}\langle()\rangle())) \tag{17}$$

Fig. 3. Gödel's T in @λ, used to calculate 2+2. The statics we implement here are shown in Appendix A.1. Simple helper functions for working with argument lists (**const**, **pop**, **pop_final**) and types (**check_type**) are defined in Appendix A.2.

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[$\star$]. A type (that is, a type-level term of kind $\star$) is constructed by naming a family in scope and providing a type-level term of the appropriate kind as an index. For example, $\textsc{Ntuple}\langle\mathbf{nat} :: \mathbf{nat} :: []_\star\rangle$ might be the type of a pair of natural numbers. 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. Given a type, its family can be case analyzed to extract its index.

3.4 Representation Schemas

mention compiler correctness history here?

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 associated with it a single internal type. This is computed by substituting the type index for the bound variable $\mathbf{i}$ in the term τ_{rep}, called the *representation schema*, associated with the type family and evaluating to a value representing an internal type. Internal types, σ, are reified into type-level terms of kind ITy using the introductory form $\blacktriangledown(\sigma)$.

3.5 Indexed Operator Families and Denotations

Type families also have a collection of primitive operator families associated with them. An operator family named $\boldsymbol{op}$ is declared using the form $\boldsymbol{op}[\kappa_{\text{idx}}](\mathbf{i}, \mathbf{a}.\tau_{\text{op}})$. Like type families, operator families are indexed by values of some kind, κ_{idx}. Operators are not first-class type-level values in our calculus, so there are no restrictions on this kind related to equality. 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, $\boldsymbol{pr}$, 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.

To select an operator from a family by providing an index and apply it to n arguments, the grammar provides a uniform form: $\text{FAM}.\boldsymbol{op}\langle\tau_{\text{idx}}\rangle(e_1; \ldots; e_n)$, where $n \geq 0$. The typechecking and translation of an expression of this form is controlled by the implementation, τ_{op}. It 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 have. A *valid denotation* has the form $[\![\tau_{\text{iterm}} \ \mathsf{as} \ \tau_{\text{type}}]\!]$, where τ_{iterm} represents the *translation* of the expression to an internal term and τ_{type} is the type assigned to it. Internal terms are reified as type-level terms using the form $\triangledown(\gamma)$, and have kind ITm (similar to ITy). It is important to note that there are no elimination form for reified terms or types. If a type cannot be assigned to an operation according to the specification of the operator being implemented, then the *error denotation*, err, is returned instead of a valid denotation. In a more practical implementation, a specialized error message and other diagnostic information could be associated with err, but we omit such details for the sake of simplicity in our core calculus. 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 $\boldsymbol{z}$ checks that no arguments were provided using the simple helper function **const** (shown in Appendix A.2). If so, it returns a valid denotation by pairing the translation $\triangledown(0)$ with the type $\text{NAT}\langle()\rangle$, as expected[1]. If not, the helper function returns err. The successor operator, $\boldsymbol{s}$, does take an argument, so it pops a denotation off the argument list, making sure there are no more, and binds its translation and type to $\mathbf{x}$ and $\mathbf{t}$ 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 $\triangledown(\vartriangle(\mathbf{x}) + 1)$ with the type $\text{NAT}\langle()\rangle$ if so. The internal language includes a special form $\vartriangle(\tau)$ which is used to "un-reify" reified internal terms, of kind ITm (thus serving as the left-inverse of $\triangledown(\gamma)$). In this case, $\mathbf{x}$ is a type-level variable of kind ITm representing the translation of the argument to the successor operator. Because we have checked that it is a natural number, and natural numbers always translate to integers by the representation schema, we know that it is safe to

[1] In fact, there is no theoretical barrier to a different "zero" being used, since integers and natural numbers are both of countably infinite cardinality!

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

Fig. 4. Central Compilation Judgement of @λ.

add 1 to this internal term. The result remains an integer according to the type system of the internal language so it is safe to give the entire expression type Nat$\langle()\rangle$ without violating the schema. Indeed, the extension mechanism will check that these consistencies hold (though not the kind system; see Sec. **??**). 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).

3.6 Functions, Variables and Arrow Types

The recursor operator, ***rec***, proceeds similarly, extracting the translations and types of each of its three arguments. The interesting part comes when analyzing the third argument, the recursive case, 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 $\tau_1 \to \tau_2$, which in @λ corresponds to a built-in type family, Arrow, indexed by a pair of types, $\star \times \star$, that is always in scope. Its representation schema simply maps to the corresponding internal arrow type, $\mathbf{i}.\blacktriangledown(\mathsf{rep}(\mathsf{fst}(\mathbf{i})) \to \mathsf{rep}(\mathsf{snd}(\mathbf{i})))$. The special form $\mathsf{rep}(\tau)$ is used to refer, abstractly, to the representation of the two types it is indexed by (we also see this used on line 10). Although λ is built-in, application is an operator associated with the Arrow family, ***ap***, as is seen on line 15.

figure for arrow types

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, 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.

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

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

$$\textsc{def-kinding}\quad \frac{\Delta \vdash_\Sigma \tau : \kappa \qquad \Delta, \mathbf{t} : \kappa \vdash_\Sigma \rho\ \texttt{prog}}{\Delta \vdash_\Sigma \mathsf{def}\ \mathbf{t} : \kappa = \tau;\ \rho\ \texttt{prog}} \qquad \textsc{exp-kinding}\quad \frac{\Delta\ \emptyset \vdash_\Sigma e\ \texttt{expr}}{\Delta \vdash_\Sigma e\ \texttt{prog}}$$

$$\boxed{\Delta \vdash_\Sigma \theta : \Theta} \qquad \Theta ::= \boldsymbol{op}[\kappa_{\text{idx}}] \;\big|\; \Theta, \Theta$$

$$\textsc{op-kinding}\quad \frac{\Delta, \mathbf{i} : \kappa_{\text{i}}, \mathbf{a} : \mathsf{list[Den]} \vdash_\Sigma \tau : \mathsf{Den}}{\Delta \vdash_\Sigma \boldsymbol{op}[\kappa_{\text{idx}}](\mathbf{i}, \mathbf{a}.\tau) : \boldsymbol{op}[\kappa_{\text{idx}}]} \qquad \textsc{ops-kinding}\quad \frac{\Delta \vdash_\Sigma \theta_1 : \Theta_1 \qquad \Delta \vdash_\Sigma \theta_2 : \Theta_2 \qquad \text{dom}(\theta_1) \cap \text{dom}(\theta_2) = \emptyset}{\Delta \vdash_\Sigma \theta_1; \theta_2 : \Theta_1, \Theta_2}$$

$$\boxed{\Delta\ \Omega \vdash_\Sigma e\ \texttt{expr}} \qquad \Omega ::= \emptyset \;\big|\; \Omega, x$$

$$\textsc{e-var-kinding}\quad \frac{}{\Delta\ \Omega, x \vdash_\Sigma x\ \texttt{expr}} \qquad \textsc{e-lam-kinding}\quad \frac{\Delta \vdash_\Sigma \tau : \star \qquad \Delta\ \Omega, x \vdash_\Sigma e\ \texttt{expr}}{\Delta\ \Omega \vdash_\Sigma \lambda x{:}\tau.e\ \texttt{expr}}$$

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

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

3.8 Active Typechecking and Translation

The second phase of compilation involves typechecking and translating the program to produce an internal term, invoking the user-defined logic encoded in the introduced operator families when needed. The rules for active typechecking and translation 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 provide an evaluation semantics for type-level terms in Fig. 8.

The key rule is ATT-EXP, which assigns a type, τ, and an *abstract translation*, γ_{abs}, to an expression, e, as specified by the judgement $\Gamma \vdash_\Phi e : \tau \Longrightarrow \gamma_{\text{abs}}$, then *deabstracts* the translation to produce the final translation. Deabstraction is specified in Fig. 10 by the judgement $\vdash_\Phi \gamma_{\text{abs}} \lightning \gamma$. The purpose of the abstract translation is to ensure that the implementation details of a type cannot be observed by operators associated with a different type, ensuring that invariants maintained by that type cannot be violated by operators in a different type family. For example, the 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 $\boldsymbol{badnum}[1](\mathbf{i}, \mathbf{a}.\mathbf{const}\ \mathbf{a}\ [\![\triangledown(-1)\ \mathsf{as}\ \textsc{Nat}\langle()\rangle]\!])$.

This is prevented by a mechanism quite similar to the mechanism by which abstract types in ML-style modules operate [**?**] – by keeping the representation schema hidden outside the type family it is associated with. As a result, it cannot be shown given the knowledge available in the type family containing $\boldsymbol{badnum}$ that the internal type associated with $\textsc{Nat}\langle()\rangle$ is $\mathbb{Z}$, and so -1 is not valid according to the rules we will describe below. The only way to produce a term of this type in an operator not associated with the type is to extract it from an input argument, where the invariants are guaranteed inductively to have been maintained. We will make this more concrete in Sec. 4.

Our representation schema abstraction mechanism is specified syntactically, so it relates to previous work on syntactic type abstraction [**?**]. While that work was focused on abstracting away the identity of a particular type outside of a "host", we focus on abstracting away the knowledge of how a primitive type family is implemented outside of the operators associated with it.

Variables translate directly to variables in the internal language, with the type determined by the typing context, Γ (rule ATT-VAR). Lambda terms translate to lambda terms in the internal language.

3.9 Abstract Representations

Note that there is no elimination form that could be used to determine the internal type associated with a type, but the special internal type $\mathsf{rep}(\tau)$, where τ is a type, can be used to refer to it abstractly

4 Safety of @$\boldsymbol{\lambda}$

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 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].)

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

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

$$\text{ATT-EXP}\quad \frac{\emptyset \vdash_{\Phi} e : \tau \Longrightarrow \gamma_{\mathrm{abs}} \qquad \vdash_{\Phi} \gamma_{\mathrm{abs}} \not\downarrow \gamma}{\vdash_{\Phi} e \Longrightarrow \gamma}$$

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

$$\text{ATT-VAR}\quad \frac{}{\Gamma, x : \tau \vdash_{\Phi} x : \tau \Longrightarrow x} \qquad \text{ATT-LAM}\quad \frac{\begin{array}{c}\tau_1 \Downarrow \tau_1' \qquad \vdash_{\Phi}^{\Xi_0} \tau_1' \rightsquigarrow \sigma_{\mathrm{abs}} \\ \Gamma, x : \tau_1' \vdash_{\Phi} e : \tau_2 \Longrightarrow \gamma_{\mathrm{abs}}\end{array}}{\Gamma \vdash_{\Phi} \lambda x{:}\tau_1.e : \textsc{Arrow}\langle(\tau_1', \tau_2)\rangle \Longrightarrow \lambda x{:}\sigma_{\mathrm{abs}}.\gamma_{\mathrm{abs}}}$$

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

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

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.

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 Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer Verlag, Denver, Colorado, USA, 2000. http://www.csg.is.titech.ac.jp/∼mich/openjava/papers/mich_2000lncs1826.pdf.
13. A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language algol 68. *Acta Informatica*, 5:1–236, 1975.
14. 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 Interoperable Scientific and Engineering Computing*, 1998.
15. M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

A Appendix

A.1 Statics of Gödel's T

$$\textsc{var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Arrow-I}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'} \qquad \textsc{Arrow-E}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

$$\textsc{Nat-I1}\ \frac{}{\Gamma \vdash \mathsf{z} : \mathtt{nat}} \qquad \textsc{Nat-I2}\ \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathsf{s}(e) : \mathtt{nat}} \qquad \textsc{Nat-E}\ \frac{\Gamma \vdash e_1 : \mathtt{nat} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \mathtt{nat} \to \tau \to \tau}{\Gamma \vdash \mathsf{natrec}(e_1; e_2; x, y.e_3) : \tau}$$

A.2 Helper Functions

describe these

$$\begin{aligned}
\mathbf{const} &:= (\lambda \mathbf{a}{:}\mathsf{list}[\mathsf{Den}].\lambda \mathbf{d}{:}\mathsf{Den}.\mathsf{fold}(\mathbf{a}; \mathbf{d}; \_, \_, \_.\mathsf{err})) \\
\mathbf{pop} &:= (\lambda \mathbf{a}{:}\mathsf{list}[\mathsf{Den}].\lambda \mathbf{f}{:}\mathsf{ITm} \to \star \to \mathsf{list}[\mathsf{Den}] \to \mathsf{Den}. \\
&\qquad \mathsf{fold}(\mathbf{a}; \mathsf{err}; \mathbf{d}, \mathbf{b}, \_.\mathsf{case}\ \mathbf{d}\ \mathsf{of}\ [\![\mathbf{x}\ \mathsf{as}\ \mathbf{t}]\!] \Rightarrow \mathbf{f}\ \mathbf{x}\ \mathbf{t}\ \mathbf{b}\ \mathsf{ow}\ \mathsf{err})) \\
\mathbf{pop\_final} &:= (\lambda \mathbf{a}{:}\mathsf{list}[\mathsf{Den}].\lambda \mathbf{f}{:}\mathsf{ITm} \to \star \to \mathsf{Den}. \\
&\qquad \mathsf{fold}(\mathbf{a}; \mathsf{err}; \mathbf{d}, \mathbf{b}, \_. \\
&\qquad \mathsf{fold}(\mathbf{b}; \mathsf{case}\ \mathbf{d}\ \mathsf{of}\ [\![\mathbf{x}\ \mathsf{as}\ \mathbf{t}]\!] \Rightarrow \mathbf{f}\ \mathbf{x}\ \mathbf{t}\ \mathsf{ow}\ \mathsf{err}; \_, \_, \_.\mathsf{err}))) \\
\mathbf{check\_type} &:= (\lambda \mathbf{t1}{:}\star.\lambda \mathbf{t2}{:}\star.\lambda \mathbf{d}{:}\mathsf{Den}.\mathsf{if}\ \mathbf{t1} \equiv_\star \mathbf{t2}\ \mathsf{then}\ \mathbf{d}\ \mathsf{else}\ \mathsf{err})
\end{aligned}$$

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

VAR-KIND
$$\frac{}{\Delta, \mathbf{t} : \kappa \vdash_\Sigma \mathbf{t} : \kappa}$$

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

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)

TL-EQUALITY
$$\frac{\begin{array}{c}\kappa\ \mathsf{eq} \qquad \Delta \vdash_\Sigma \tau_1 : \kappa \qquad \Delta \vdash_\Sigma \tau_2 : \kappa \\ \Delta \vdash_\Sigma \tau_3 : \kappa' \qquad \Delta \vdash_\Sigma \tau_4 : \kappa'\end{array}}{\Delta \vdash_\Sigma \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 : \kappa'}$$

TYPE-INTRO
$$\frac{\begin{array}{c}\textsc{Fam}[\kappa_{\text{idx}}, \Theta] \in \Sigma \\ \Delta \vdash_\Sigma \tau_{\text{idx}} : \kappa_{\text{idx}}\end{array}}{\Delta \vdash_\Sigma \textsc{Fam}\langle \tau_{\text{idx}} \rangle : \star}$$

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

DEN-INTRO-VALID
$$\frac{\begin{array}{c}\Delta \vdash_\Sigma \tau_1 : \mathsf{ITm} \\ \Delta \vdash_\Sigma \tau_2 : \star\end{array}}{\Delta \vdash_\Sigma \llbracket \tau_1\ \mathsf{as}\ \tau_2 \rrbracket : \mathsf{Den}}$$

DEN-INTRO-ERR
$$\frac{}{\Delta \vdash_\Sigma \mathsf{err} : \mathsf{Den}}$$

DEN-ELIM
$$\frac{\begin{array}{c}\Delta \vdash_\Sigma \tau : \mathsf{Den} \\ \Delta, \mathbf{x} : \mathsf{ITm}, \mathbf{t} : \star \vdash_\Sigma \tau_1 : \kappa \qquad \Delta \vdash_\Sigma \tau_2 : \kappa\end{array}}{\Delta \vdash_\Sigma \mathsf{case}\ \tau\ \mathsf{of}\ \llbracket \mathbf{x}\ \mathsf{as}\ \mathbf{t} \rrbracket \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 : \kappa}$$

ITERM-INTRO
$$\frac{\Delta\ \emptyset \vdash_\Sigma \gamma\ \mathtt{iterm}}{\Delta \vdash_\Sigma \triangledown(\gamma) : \mathsf{ITm}}$$

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

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

T-EQ
$$\frac{}{\star\ \mathsf{eq}}$$

Z-EQ
$$\frac{}{\mathbb{Z}\ \mathsf{eq}}$$

S-EQ
$$\frac{}{\mathsf{Str}\ \mathsf{eq}}$$

U-EQ
$$\frac{}{1\ \mathsf{eq}}$$

P-EQ
$$\frac{\kappa_1\ \mathsf{eq} \qquad \kappa_2\ \mathsf{eq}}{\kappa_1 \times \kappa_2\ \mathsf{eq}}$$

L-EQ
$$\frac{\kappa\ \mathsf{eq}}{\mathsf{list}[\kappa]\ \mathsf{eq}}$$

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

I-VAR-KINDING
$$\frac{}{\Delta\ \Omega, x \vdash_\Sigma x\ \mathtt{iterm}}$$

I-LAM-KINDING
$$\frac{\begin{array}{c}\Delta \vdash_\Sigma \sigma\ \mathtt{itype} \\ \Delta\ \Omega, x \vdash_\Sigma \gamma\ \mathtt{iterm}\end{array}}{\Delta\ \Omega \vdash_\Sigma \lambda x{:}\sigma.\gamma\ \mathtt{iterm}}$$

I-FIX-KINDING
$$\frac{\begin{array}{c}\Delta \vdash_\Sigma \sigma\ \mathtt{itype} \\ \Delta\ \Omega, x \vdash_\Sigma \gamma\ \mathtt{iterm}\end{array}}{\Delta\ \Omega \vdash_\Sigma \mathsf{fix}\ f{:}\sigma\ \mathsf{is}\ \gamma\ \mathtt{iterm}}$$

(omitted forms have trivially recursive rules)

ITERM-DEREIFY-KINDING
$$\frac{\Delta \vdash_\Sigma \tau : \mathsf{ITm}}{\Delta\ \Omega \vdash_\Sigma \vartriangle(\tau)\ \mathtt{iterm}}$$

ABS-TRANS-KINDING
$$\frac{\Delta \vdash_\Sigma \tau_{\text{iterm}} : \mathsf{ITm} \qquad \Delta \vdash_\Sigma \tau_{\text{type}} : \star}{\Delta\ \Omega \vdash_\Sigma \mathsf{trans}(\llbracket \tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}} \rrbracket)\ \mathtt{iterm}}$$

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

I-INT-KINDING
$$\frac{}{\Delta \vdash_\Sigma \mathbb{Z}\ \mathtt{itype}}$$

I-PROD-KINDING
$$\frac{\begin{array}{c}\Delta \vdash_\Sigma \sigma_1\ \mathtt{itype} \\ \Delta \vdash_\Sigma \sigma_2\ \mathtt{itype}\end{array}}{\Delta \vdash_\Sigma \sigma_1 \times \sigma_2\ \mathtt{itype}}$$

I-ARROW-KINDING
$$\frac{\begin{array}{c}\Delta \vdash_\Sigma \sigma_1\ \mathtt{itype} \\ \Delta \vdash_\Sigma \sigma_2\ \mathtt{itype}\end{array}}{\Delta \vdash_\Sigma \sigma_1 \to \sigma_2\ \mathtt{itype}}$$

ITYPE-DEREIFY-KINDING
$$\frac{\Delta \vdash_\Sigma \tau : \mathsf{ITy}}{\Delta \vdash_\Sigma \blacktriangle(\tau)\ \mathtt{itype}}$$

ABS-REP-KINDING
$$\frac{\Delta \vdash_\Sigma \tau : \star}{\Delta \vdash_\Sigma \mathsf{rep}(\tau)\ \mathtt{itype}}$$

Fig. 7. Kinding for type-level terms

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

$$\textsc{TL-Lam-Eval}\quad \frac{}{\lambda\mathbf{t}{:}\kappa.\tau \Downarrow \lambda\mathbf{t}{:}\kappa.\tau} \qquad \textsc{TL-Ap-Eval}\quad \frac{\tau_1 \Downarrow \lambda\mathbf{t}{:}\kappa.\tau \qquad \tau_2 \Downarrow \tau_2' \qquad [\tau_2'/\mathbf{t}]\tau \Downarrow \tau'}{\tau_1\ \tau_2 \Downarrow \tau'}$$

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

$$\textsc{TL-Eq-Eval-Equal}\quad \frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_1' \qquad \tau_3 \Downarrow \tau_3'}{\mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \Downarrow \tau_3'} \qquad \textsc{TL-Eq-Eval-Inequal}\quad \frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_2' \qquad \tau_1' \neq \tau_2' \qquad \tau_4 \Downarrow \tau_4'}{\mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \Downarrow \tau_4'}$$

$$\textsc{Type-Eval}\quad \frac{\tau_{\text{idx}} \Downarrow {\tau_{\text{idx}}}'}{\textsc{Fam}\langle\tau_{\text{idx}}\rangle \Downarrow \textsc{Fam}\langle{\tau_{\text{idx}}}'\rangle} \qquad \textsc{Famcase-Eval-Match}\quad \frac{\tau \Downarrow \textsc{Fam}\langle\tau_{\text{idx}}\rangle \qquad [\tau_{\text{idx}}/\mathbf{x}]\tau_1 \Downarrow \tau_1'}{\mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_1'}$$

$$\textsc{Famcase-Eval-Fail}\quad \frac{\tau \Downarrow \textsc{Fam'}\langle\tau_{\text{idx}}\rangle \qquad \textsc{Fam} \neq \textsc{Fam'} \qquad \tau_2 \Downarrow \tau_2'}{\mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Fam}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_2'} \qquad \textsc{Den-Valid-Eval}\quad \frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_2'}{\llbracket \tau_1\ \mathsf{as}\ \tau_2 \rrbracket \Downarrow \llbracket \tau_1'\ \mathsf{as}\ \tau_2' \rrbracket}$$

$$\textsc{Den-Err-Eval}\quad \frac{}{\mathsf{err} \Downarrow \mathsf{err}} \qquad \textsc{Dencase Eval Valid}\quad \frac{\tau \Downarrow \llbracket \tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}} \rrbracket \qquad [\triangledown(\mathsf{trans}(\llbracket \tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}} \rrbracket))/\mathbf{x}, \tau_{\text{type}}/\mathbf{t}]\tau_1 \Downarrow \tau_1'}{\mathsf{case}\ \tau\ \mathsf{of}\ \llbracket \mathbf{x}\ \mathsf{as}\ \mathbf{t} \rrbracket \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_1'}$$

$$\textsc{Dencase-Eval-Err}\quad \frac{\tau \Downarrow \mathsf{err} \qquad \tau_2 \Downarrow \tau_2'}{\mathsf{case}\ \tau_{\text{den}}\ \mathsf{of}\ \llbracket \mathbf{y}\ \mathsf{as}\ \mathbf{x} \rrbracket \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \Downarrow \tau_2'} \qquad \textsc{Iterm-Reify}\quad \frac{\gamma \curlyvee \gamma'}{\triangledown(\gamma) \Downarrow \triangledown(\gamma')} \qquad \textsc{Itype-Reify}\quad \frac{\sigma \curlyvee \sigma'}{\blacktriangledown(\sigma) \Downarrow \blacktriangledown(\sigma')}$$

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

$$\textsc{I-Var-Eval}\quad \frac{}{x \curlyvee x} \qquad \textsc{I-Lam-Eval}\quad \frac{\sigma \curlyvee \sigma' \qquad \gamma \curlyvee \gamma'}{\lambda x{:}\sigma.\gamma \curlyvee \lambda x{:}\sigma'.\gamma'} \qquad \textsc{I-Fix-Eval}\quad \frac{\sigma \curlyvee \sigma' \qquad \gamma \curlyvee \gamma'}{\mathsf{fix}\ f{:}\sigma\ \mathsf{is}\ \gamma \curlyvee \mathsf{fix}\ f{:}\sigma'\ \mathsf{is}\ \gamma'}$$

(omitted forms have trivially recursive rules)

$$\textsc{Iterm Unquote Eval}\quad \frac{\tau \Downarrow \tau'}{\triangle(\tau) \curlyvee \triangle(\tau')} \qquad \textsc{Val From Den Eval}\quad \frac{\tau_{\text{iterm}} \Downarrow {\tau_{\text{iterm}}}' \qquad \tau_{\text{type}} \Downarrow {\tau_{\text{type}}}'}{\mathsf{trans}(\llbracket \tau_{\text{iterm}}\ \mathsf{as}\ \tau_{\text{type}} \rrbracket) \curlyvee \mathsf{trans}(\llbracket {\tau_{\text{iterm}}}'\ \mathsf{as}\ {\tau_{\text{type}}}' \rrbracket)}$$

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

$$\textsc{I-Int-Eval}\quad \frac{}{\mathbb{Z} \curlyvee \mathbb{Z}} \qquad \textsc{I-Prod-Eval}\quad \frac{\sigma_1 \curlyvee \sigma_1' \qquad \sigma_2 \curlyvee \sigma_2'}{\sigma_1 \times \sigma_2 \curlyvee \sigma_1' \times \sigma_2'} \qquad \textsc{I-Arrow-Eval}\quad \frac{\sigma_1 \curlyvee \sigma_1' \qquad \sigma_2 \curlyvee \sigma_2'}{\sigma_1 \to \sigma_2 \curlyvee \sigma_1' \to \sigma_2'}$$

$$\textsc{Itype-Eval}\quad \frac{\tau \Downarrow \tau'}{\blacktriangle(\tau) \curlyvee \blacktriangle(\tau')} \qquad \textsc{Abs-Rep-Eval}\quad \frac{\tau \Downarrow \tau'}{\mathsf{rep}(\tau) \curlyvee \mathsf{rep}(\tau')}$$

Fig. 8. Evaluation semantics for type-level terms

$\boxed{\vdash^{\textsc{Fam}}_{\Phi} \sigma \rightsquigarrow \sigma'}$ $\quad \Phi ::= \Phi_0 \mid \Phi, \textsc{Fam}[\theta, \mathbf{i}.\tau]$

$$\textsc{abs int}\ \frac{}{\vdash^{\textsc{Fam}}_{\Phi} \mathbb{Z} \rightsquigarrow \mathbb{Z}}$$

$$\textsc{abs arrow}\ \frac{\vdash^{\textsc{Fam}}_{\Phi} \sigma_1 \rightsquigarrow \sigma_1' \qquad \vdash^{\textsc{Fam}}_{\Phi} \sigma_2 \rightsquigarrow \sigma_2'}{\vdash^{\textsc{Fam}}_{\Phi} \sigma_1 \rightarrow \sigma_2 \rightsquigarrow \sigma_1' \rightarrow \sigma_2'}$$

$$\textsc{abs prod}\ \frac{\vdash^{\textsc{Fam}}_{\Phi} \sigma_1 \rightsquigarrow \sigma_1' \qquad \vdash^{\textsc{Fam}}_{\Phi} \sigma_2 \rightsquigarrow \sigma_2'}{\vdash^{\textsc{Fam}}_{\Phi} \sigma_1 \times \sigma_2 \rightsquigarrow \sigma_1' \times \sigma_2'}$$

$$\textsc{abs+cancel unquote}\ \frac{\vdash^{\textsc{Fam}}_{\Phi} \sigma \rightsquigarrow \sigma'}{\vdash^{\textsc{Fam}}_{\Phi} \blacktriangle(\blacktriangledown(\sigma)) \rightsquigarrow \sigma'}$$

$$\textsc{abs rep from type visible}\ \frac{\textsc{Fam}[\theta, \mathbf{i}.\tau] \in \Phi \qquad [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\sigma) \qquad \vdash^{\textsc{Fam}}_{\Phi} \sigma \rightsquigarrow \sigma}{\vdash^{\textsc{Fam}}_{\Phi} \mathsf{rep}(\textsc{Fam}\langle\tau_{\text{idx}}\rangle) \rightsquigarrow \sigma}$$

$$\textsc{abs rep from type hidden}\ \frac{\textsc{Fam} \neq \textsc{Fam'}}{\vdash^{\textsc{Fam}}_{\Phi} \mathsf{rep}(\textsc{Fam'}\langle\tau_{\text{idx}}\rangle) \rightsquigarrow \mathsf{rep}(\textsc{Fam'}\langle\tau_{\text{idx}}\rangle)}$$

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

$$\textsc{abs rep from type}\ \frac{\textsc{Fam'}[\theta, \mathbf{i}.\tau] \in \Phi \qquad [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\sigma) \qquad \vdash^{\textsc{Fam}}_{\Phi} \sigma \rightsquigarrow \sigma}{\vdash^{\textsc{Fam}}_{\Phi} \textsc{Fam'}\langle\tau_{\text{idx}}\rangle \sim \sigma}$$

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

$$\textsc{abs empty}\ \frac{}{\vdash^{\textsc{Fam}}_{\Phi} \emptyset \rightsquigarrow \emptyset}$$

$$\textsc{abs ctx}\ \frac{\vdash^{\textsc{Fam}}_{\Phi} \Gamma \rightsquigarrow \Psi \qquad \vdash^{\textsc{Fam}}_{\Phi} \tau \rightsquigarrow \sigma}{\vdash^{\textsc{Fam}}_{\Phi} \Gamma, x : \tau \rightsquigarrow \Psi, x : \sigma}$$

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

$$\textsc{abs i-var}\ \frac{}{\Psi, x : \sigma \vdash^{\textsc{Fam}}_{\Phi} x \sim \sigma}$$

$$\textsc{abs i-lam}\ \frac{\vdash^{\textsc{Fam}}_{\Phi} \sigma_1 \rightsquigarrow \sigma_1 \qquad \Psi, x : \sigma_1 \vdash^{\textsc{Fam}}_{\Phi} \gamma \sim \sigma_2}{\Psi \vdash^{\textsc{Fam}}_{\Phi} \lambda x{:}\sigma_1.\gamma \sim \sigma_1 \rightarrow \sigma_2}$$

$$\textsc{abs i-ap}\ \frac{\Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma_1 \sim \sigma_1 \rightarrow \sigma_2 \qquad \Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma_2 \sim \sigma_1}{\Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma_1\ \gamma_2 \sim \sigma_2}$$

$$\textsc{abs i-fix}\ \frac{\vdash^{\textsc{Fam}}_{\Phi} \sigma \rightsquigarrow \sigma \qquad \Psi, x : \sigma \vdash^{\textsc{Fam}}_{\Phi} \gamma \sim \sigma}{\Psi \vdash^{\textsc{Fam}}_{\Phi} \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \gamma \sim \sigma}$$

(standard statics for integers and products omitted)

$$\textsc{abs if eq}\ \frac{\begin{array}{cc}\Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma_1 \sim \mathbb{Z} & \Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma_2 \sim \mathbb{Z} \\ \Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma_3 \sim \sigma & \Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma_4 \sim \sigma\end{array}}{\Psi \vdash^{\textsc{Fam}}_{\Phi} \mathsf{if}\ \gamma_1 \equiv_{\mathbb{Z}} \gamma_2\ \mathsf{then}\ \gamma_3\ \mathsf{else}\ \gamma_4 \sim \sigma}$$

$$\textsc{abs iterm unquote}\ \frac{\Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma \sim \sigma}{\Psi \vdash^{\textsc{Fam}}_{\Phi} \vartriangle(\triangledown(\gamma)) \sim \sigma}$$

$$\textsc{abs val from den visible}\ \frac{\vdash^{\textsc{Fam}}_{\Phi} \textsc{Fam}\langle\tau_{\text{idx}}\rangle \sim \sigma \qquad \Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma \sim \sigma}{\Psi \vdash^{\textsc{Fam}}_{\Phi} \mathsf{trans}(\llbracket \triangledown(\gamma)\ \mathsf{as}\ \textsc{Fam}\langle\tau_{\text{idx}}\rangle \rrbracket) \sim \sigma}$$

$$\textsc{abs val from den hidden}\ \frac{\textsc{Fam} \neq \textsc{Fam'} \qquad \vdash^{\textsc{Fam}}_{\Phi} \textsc{Fam'}\langle\tau_{\text{idx}}\rangle \sim \sigma \qquad \Psi \vdash^{\textsc{Fam}}_{\Phi} \gamma \sim \sigma}{\Psi \vdash^{\textsc{Fam}}_{\Phi} \mathsf{trans}(\llbracket \triangledown(\gamma)\ \mathsf{as}\ \textsc{Fam'}\langle\tau_{\text{idx}}\rangle \rrbracket) \sim \mathsf{rep}(\textsc{Fam'}\langle\tau_{\text{idx}}\rangle)}$$

Fig. 9. Abstracted internal typing

$$\boxed{\vdash_{\Phi} \sigma \lightning \sigma}$$

$$\textsc{deabs int}\ \frac{}{\vdash_{\Phi} \mathbb{Z} \lightning \mathbb{Z}} \qquad \textsc{deabs arrow}\ \frac{\begin{array}{c}\vdash_{\Phi} \sigma_1 \lightning \sigma_1 \\ \vdash_{\Phi} \sigma_2 \lightning \sigma_2\end{array}}{\vdash_{\Phi} \sigma_1 \rightarrow \sigma_2 \lightning \sigma_1 \rightarrow \sigma_2} \qquad \textsc{deabs prod}\ \frac{\begin{array}{c}\vdash_{\Phi} \sigma_1 \lightning \sigma_1 \\ \vdash_{\Phi} \sigma_2 \lightning \sigma_2\end{array}}{\vdash_{\Phi} \sigma_1 \times \sigma_2 \lightning \sigma_1 \times \sigma_2}$$

$$\textsc{deabs rep from type hidden}\ \frac{\begin{array}{cc}\textsc{Fam}[\theta, \mathbf{i}.\tau] \in \Phi & [\tau_{\text{idx}}/\mathbf{i}]\tau \Downarrow \blacktriangledown(\sigma) \\ \vdash_{\Phi}^{\textsc{Fam}} \sigma \rightsquigarrow \sigma & \vdash_{\Phi} \sigma \lightning \sigma\end{array}}{\vdash_{\Phi} \mathsf{rep}(\textsc{Fam}\langle\tau_{\text{idx}}\rangle) \lightning \sigma}$$

$$\boxed{\vdash_{\Phi} \gamma \lightning \gamma}$$

$$\textsc{deabs i-var}\ \frac{}{\vdash_{\Phi} x \lightning x} \qquad \textsc{deabs i-lam}\ \frac{\begin{array}{c}\vdash_{\Phi}^{\cdot} \sigma \rightsquigarrow \sigma \\ \vdash_{\Phi} \sigma \lightning \sigma \\ \vdash_{\Phi} \gamma \lightning \gamma\end{array}}{\vdash_{\Phi} \lambda x{:}\sigma.\gamma \lightning \lambda x{:}\sigma.\gamma} \qquad \textsc{deabs i-fix}\ \frac{\begin{array}{c}\vdash_{\Phi}^{\cdot} \sigma \rightsquigarrow \sigma \\ \vdash_{\Phi} \sigma \lightning \sigma \\ \vdash_{\Phi} \gamma \lightning \gamma\end{array}}{\vdash_{\Phi} \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \gamma \lightning \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \gamma}$$

(omitted forms have trivially recursive rules)

$$\textsc{deabs unquote}\ \frac{\vdash_{\Phi} \gamma \lightning \gamma}{\vdash_{\Phi} \vartriangle(\triangledown(\gamma)) \lightning \gamma} \qquad \textsc{deabs val from den hidden}\ \frac{\vdash_{\Phi} \gamma \lightning \gamma}{\vdash_{\Phi} \mathsf{trans}([\![\triangledown(\gamma)\ \mathsf{as}\ \textsc{Fam}\langle\tau_{\text{idx}}\rangle]\!]) \lightning \gamma}$$

Fig. 10. Deabstraction rules