Active Typing: Conservatively Extending a Type System From Within

Cyrus Omar   Jonathan Aldrich
Carnegie Mellon University
{comar, aldrich}@cs.cmu.edu

Abstract

Researchers often need to extend an existing language with new type and operator constructors to realize a new abstraction in its strongest form. But this is not generally possible from within, so it is common to develop dialects and "toy" languages. Unfortunately, taking this approach limits the utility of these abstractions: they cannot be imported by clients of other dialects and languages, and building applications where different components rely on different type systems is both unsafe and unnatural.

We introduce @λ, a simply-typed lambda calculus with simply-kinded type-level computation where new indexed type and operator constructors can be declared from within. Type-level functions associated with operator constructors define their static and dynamic semantics, the latter by translation to a fixed typed internal language. By lifting compiler correctness techniques pioneered by the TIL compiler for Standard ML into the language, the "actively typed semantics" guarantees type safety. Going further, the semantics enforce a novel form of representation independence at extension boundaries that ensures that extensions are mutually conservative (i.e. they do not weaken or interfere with one another, so that they can be used together in any combination). We intend @λ as a minimal foundation for future work on safe language-integrated extension mechanisms for typed programming languages, but it is already quite expressive. We demonstrate by showing how a conventional concrete syntax can be introduced by type-directed dispatch to Core @λ, then discuss a number of typed language fragments that can be introduced as orthogonal "libraries" (and discuss the sorts of fragments that, as yet, cannot).

1. Introduction

Typed programming languages are often described in fragments, each consisting of a small number of indexed type constructors (often one) and associated (term-level) operator constructors. The simply typed lambda calculus (STLC), for example, consists of a single fragment containing a single type constructor, $\rightarrow$, indexed by a pair of types, and two operator constructors: λ, indexed by a type, and `ap`, which we may think of as being indexed trivially. A fragment is often identified by the type constructor it is organized around, so the STLC might also be called $\mathcal{L}\{\rightarrow\}$, following the notational convention used by Harper [6] and others.

Gödel's $\mathbf{T}$ consists of the $\rightarrow$ fragment and the `nat` fragment, defining natural numbers and a recursor that allows one to "fold" over a natural number. We might thus call it $\mathcal{L}\{\rightarrow \texttt{nat}\}$. This language is more powerful than the STLC because the STLC admits an embedding (trivially) into $\mathbf{T}$ but the reverse is not true. Buoyed by this fact, we might go on by adding fragments that define sums, products and various forms of inductive types (e.g. lists), or perhaps a general mechanism for defining inductive types. Each fragment clearly increases the expressiveness of our language.

If we consider the $\forall$ fragment, however, defining universal quantification over types (i.e. parametric polymorphism), we might take pause. In $\mathcal{L}\{\rightarrow\ \forall\}$, studied variously by Girard as System **F** [**?**] and Reynolds as the polymorphic lambda calculus [**?**], it is known that sums, products, and inductive and co-inductive types can all be weakly defined via Church-style encodings [**?**]. This fact is of substantial theoretical importance and is quite relevant if *translating* a language like $\mathcal{L}\{\rightarrow \forall\, \texttt{nat} + \times\}$ to $\mathcal{L}\{\rightarrow \forall\}$ is one's main interest. But Reynolds, in a remark that recalls the "Turing tarpit" of Perlis [8], cautioned against using the existence of a weak encoding as an argument for programming with the corresponding embedding directly [9]:

what to cite for this?

> To say that any reasonable function can be expressed by some program is not to say that it can be expressed by the most reasonable program. It is clear that the language requires a novel programming style. Moreover, it is likely that certain important functions cannot be expressed by their most efficient algorithms.

Modern "general-purpose" languages like ML, on the other hand, expose constructs, like datatypes and abstract types, that admit strong embeddings of many simpler fragments, occupying what is widely seen as a "sweet spot" in the design space. However, "general-purpose" and "all-purpose" are quite distinct, and situations continue to arise in both research and practice where introducing a new fragment would be desirable because it can still only be weakly defined in terms of existing fragments, or a strong embedding, while possible, is impractically verbose or inefficient:

1. General-purpose abstractions continue to evolve and admit variants. There are many variations on record types, for example: records with functional record update, specified field ordering (that is, labeled tuples) or prototypic delegation are either awkward to strongly embed or cannot be. Similarly, sum types also admit variants (e.g. various forms of open sums). Even something as seemingly simple as a type-safe `sprintf` operator requires extending a language like Ocaml [**?**].
2. Perhaps more interestingly, specialized type systems that enforce stronger invariants than general-purpose constructs are capable

of enforcing are often developed by researchers. One need take only a brief excursion through the literature to discover language extensions that support parallel programming [?], concurrent programming [?], distributed programming [?], dataflow programming [?], authenticated data structures [?], information flow security [? ?], database queries [?], aliased references [?], network protocols [?], units of measure [?] and many others.

citations

3. Foreign function interfaces (FFIs) that allow programmers to safely and naturally interact with libraries written in an existing language require enforcing the type system of the foreign language within the calling language. Safe FFIs generally require direct extensions to the language. For example, MLj extended Standard ML with constructs for safely and naturally interfacing with Java [?]. For any other language, including others on the JVM like Scala and the many languages that might be accessible via Java's native FFI, there is no way to guarantee that language-specific invariants are statically maintained, and the interface is certainly far from natural.

These sorts of innovations are generally disseminated as distinct *dialects* of an existing general-purpose language, constructed either as a fork, using tools like compiler generators, DSL frameworks or language workbenches, or directly within a particular compiler for the language, sometimes activated by a flag or pragma. This, we argue, is quite unsatisfying: a programmer can choose either a dialect supporting an innovative approach to parallel programming or one that builds in support for statically reasoning about units of measure, but there may not be an available dialect supporting both. Forming such a dialect is alarmingly non-trivial, even in the rare situation where a common framework has been used or the dialects are implemented within the same compiler, as these mechanisms do not guarantee that different combinations of individually sound language fragments remain sound when combined. Metatheoretic and compiler correctness results can only be derived for the dialect *resulting* from a language composition operation, so in a diverse fragment ecosystem, fragment providers have little choice but to leave this to clients. Avoiding language composition is difficult, as interoperability between components of an application written in different dialects faces precisely the problems described in item 3.

These are not the sorts of problems usually faced by library providers. Well-designed languages preclude the possibility of conflict between two libraries and ensure that the semantics of one library cannot be weakened by another by strictly enforcing abstraction barriers. For example, a module declaring an abstract type in ML can rely on its representation invariants no matter which other modules are in use, so clients can assume that they will operate robustly in combination without needing to attend to burdensome "link-time" proof obligations.

Contributions We draw inspiration from this approach to design a mechanism that makes it possible to declare new type and operator constructors and define, *using type-level functions*, their static and dynamic semantics, the latter by translation to a fixed typed internal language. If a language fragment can be weakly defined in terms of the typed internal language and its statics are of a form that can be implemented by this protocol, it can be made strongly embeddable in the external language by introducing, in this way, the necessary constructors and static logic.

The semantics imposes checks on extensions to guarantee type safety and decidability of typing and type equality. Moreover, an important class of lemmas that play a key role in proofs that an extension enables strong embeddings of a language fragment can be derived assuming a "closed world" (that is, in the completely traditional manner where one constructs a complete language and reasons inductively). The semantics guarantees that they will be *conserved* in the "open world" by enforcing a form of representation independence at all extension boundaries. This avoids the biggest problems of composing language dialects and justifies the inclusion of the mechanism directly into the language.

We will begin in Sec. 2 by introducing a minimal calculus with this kind of *actively typed semantics*, @λ. The external language begins with only the $\rightarrow$ fragment and the typed internal language is a simple variant of PCF that permits only weak embeddings of even fragments like the `nat` fragment of Gödel's $\mathbf{T}$ or ML-style n-ary products. We implement these as extensions to explain the mechanism and motivate the constraints that the semantics imposes. We then examine the semantics in greater detail and state and sketch the key points in the proofs of the metatheory in Sec. 3. These are the fundamental contributions of this work.

While the core calculus addresses the issue of making a strong embedding possible, embeddings into the core calculus are quite clearly impractical syntactically. To begin to address the issue of syntax, we develop in Sec. 4 a flexible type-directed dispatch protocol for a more conventional *expanded syntax* that defers to the core calculus semantically. We then use the expanded syntax to discuss several more interesting examples in Sec. 5: . .

list these

While of surprising expressiveness, @λ itself is certainly not capable of expressing the full variety of type systems listed earlier. Indeed, to guarantee safety and conservativity, our calculus assumes perhaps too much about the type system being encoded. For example, it only admits fragments for which the typing judgement looks essentially like the typing judgement of the $\rightarrow$ fragment; new contexts cannot be introduced. We discuss this and some of the other reasons why a fragment may not be expressible in @λ in Sec. 6. Fortunately, these limitations do not appear fundamental to the approach, so we also outline in Sec. 6 directions for future work. We conclude with a discussion of related work in Sec. 7.

organization of rest of sections

2. Core @λ

The syntax of Core @λ is given in Fig. 1. An example of a program defining type and operator constructors that can be used to strongly embed Gödel's $\mathbf{T}$ in @λ is given in Fig. 2. We will discuss its semantics and how precisely the strong embedding, seen being used starting on line 16, works as we go on. Natural numbers can, of course, be strongly embedded in existing languages with a similar usage and asymptotic performance profile (up to function call overhead with an abstract type, for example). We will provide more sophisticated examples where this is less feasible later on.

2.1 Programs

A *program*, ρ, consists of a series of static declarations (in the core, only constructor declarations) followed by an external term, e. The syntax for external terms contains three term formers: variables, λ, and a form for all other operator invocations, which we will discuss below. Compiling a program consists of first *kind checking* its static declarations and type-level terms (Fig. 3, discussed below) then *type checking* the external term and, simultaneously, *translating* it to a term, ι, in the typed internal language (Fig. 4). The key judgement is the *active typing judgement*, relating an external term to a type and an internal term, called its *translation*, under *typing context* Γ and *constructor context* Φ:

$$\Gamma \vdash_\Phi e : \tau \Longrightarrow \iota$$

This judgement follows standard conventions for the typing judgement of a simply-typed lambda calculus in most respects. The typing context Γ maps variables to types and obeys standard structural properties. The key novelties here are 1) that the available type and operator constructors are not fixed by the language, but rather are tracked by the constructor context, Φ, and 2) that a translation is simultaneously derived. There is no separate

$$
\begin{array}{rccl}
\textbf{programs} & \rho & ::= & \mathsf{tycon}\ \textsc{Tycon}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\mathsf{schema}\ \tau_{\text{rep}};\theta\};\ \rho \\
 & & \mid & e \\
 & \theta & ::= & \mathsf{opcon}\ \boldsymbol{op}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\tau_{\text{def}}\} \mid \theta;\theta \\
\\
\textbf{external terms} & e & ::= & x \mid \lambda x{:}\tau.e \mid \textsc{Tycon}.\boldsymbol{op}\langle\tau_{\text{idx}}\rangle(e_1;\ldots;e_n) \\
\\
\textbf{internal terms} & \iota & ::= & x \mid \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \iota \mid \lambda x{:}\sigma.\iota \mid \iota_1\ \iota_2 \\
\text{integers} & & \mid & \bar{\mathrm{z}} \mid \iota_1 \oplus \iota_2 \mid \mathsf{if}\ \iota_1 \equiv_{\mathsf{int}} \iota_2\ \mathsf{then}\ \iota_3\ \mathsf{else}\ \iota_4 \\
\text{products} & & \mid & () \mid (\iota_1,\iota_2) \mid \mathsf{fst}(\iota) \mid \mathsf{snd}(\iota) \\
\text{sums} & & \mid & \mathsf{inl}[\sigma_2](\iota_1) \mid \mathsf{inr}[\sigma_1](\iota_2) \\
 & & \mid & \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow e_1 \mid \mathsf{inr}(x) \Rightarrow e_2 \\
\textbf{internal types} & \sigma & ::= & \sigma_1 \rightharpoonup \sigma_2 \mid \mathsf{int} \mid 1 \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \\
\\
\textbf{type-level terms} & \tau & ::= & \mathbf{t} \mid \lambda\mathbf{t}{:}\kappa.\tau \mid \tau_1\ \tau_2 \mid \bar{\mathrm{z}} \mid \tau_1 \oplus \tau_2 \mid \mathit{label} \\
\text{lists} & & \mid & []_\kappa \mid \tau_1 :: \tau_2 \mid \mathsf{fold}(\tau_1;\tau_2;\mathbf{h},\mathbf{t},\mathbf{r}.\tau_3) \\
\text{products} & & \mid & () \mid (\tau_1,\tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau) \\
\text{sums} & & \mid & \mathsf{inl}[\kappa_2](\tau_1) \mid \mathsf{inr}[\kappa_1](\tau_2) \\
 & & \mid & \mathsf{case}\ \tau\ \mathsf{of}\ \mathsf{inl}(\mathbf{t}) \Rightarrow \tau_1 \mid \mathsf{inr}(\mathbf{t}) \Rightarrow \tau_2 \\
\text{types} & & \mid & \textsc{Tycon}\langle\tau\rangle \\
 & & \mid & \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Tycon}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
\text{equality} & & \mid & \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \\
\text{derivates} & & \mid & [\![\tau \longrightarrow \bar\iota]\!] \mid [\![\tau \longrightarrow \bar\iota]\!]^{\checkmark} \mid \mathsf{typeof}(\tau) \\
\text{rep types} & & \mid & \blacktriangleright(\bar\sigma) \\
\text{abstract IL} & \bar\iota & ::= & x \mid \mathsf{fix}\ x{:}\bar\sigma\ \mathsf{is}\ \bar\iota \mid \cdots \mid \mathsf{transof}(\tau) \\
 & \bar\sigma & ::= & \bar\sigma_1 \rightharpoonup \bar\sigma_2 \mid \cdots \mid \blacktriangleleft(\tau) \mid \mathsf{repof}(\tau) \\
\\
\textbf{kinds} & \kappa & ::= & \kappa_1 \to \kappa_2 \mid \mathsf{Int} \mid \mathsf{Lbl} \mid \mathsf{list}[\kappa] \mid 1 \mid \kappa_1 \times \kappa_2 \\
 & & \mid & \kappa_1 + \kappa_2 \mid \mathsf{Ty} \mid \mathsf{D} \mid \mathsf{ITy}
\end{array}
$$

Figure 1. Syntax of Core @λ. Here, x ranges over external and internal language variables, $\mathbf{t}$ ranges over type-level variables, TYCON ranges over type constructor names, ***op*** ranges over operator constructor names, $\bar{\mathrm{z}}$ ranges over integer literals, *label* ranges over label literals (see text) and $\oplus$ ranges over standard total binary operations over integers (e.g. addition).

operational semantics for the external language; the dynamic semantics are defined by this translation to the internal language (which might have a more conventional operational semantics). The two contexts and the external term can be thought of as input to the judgement, while the type and translation are output (we do not consider a bidirectional approach [?]or type inference in this work, though this is likely a promising avenue for future research).

cite Harper-Stone / denotational semantics

In our calculus, the internal language (IL), consisting of internal terms ι and internal types σ, provides partial functions (via the generic fixpoint operator of Plotkin's PCF [?]), simple product and sum types and a base type of integers (to make our example interesting and as a nod toward speed on contemporary machines). In practice, the internal language could be any typed language with a specification for which type safety and decidability of typechecking have been satisfyingly determined (e.g. Standard ML or CakeML [?], leaving translation to a simpler IL to later stages of the compilation process).

cite

2.2 Types and Type Constructors

@λ is an "actively typed" version of the simply-typed lambda calculus with simply-kinded type-level computation. Specifically, type-level terms, τ, themselves form a typed lambda calculus. The classifiers of terms at the type level are called *kinds*, κ to distinguish them from *types*, which are just one kind of type-level value. We will write the kind of types as Ty, though it is also written $\star$, T or Type in various similarly structured languages (see Sec. 7). The kinding rules for type-level terms are given in Fig. **??**.

refer

$$\mathsf{tycon}\ \textsc{Nat}\ \mathsf{of}\ 1\ \{\mathsf{schema}\ \lambda\mathbf{idx}{:}1.\blacktriangleright(\mathsf{int}); \tag{1}$$
$$\quad \mathsf{opcon}\ \boldsymbol{z}\ \mathsf{of}\ 1\ \{\lambda\mathbf{i}{:}1.\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}].\mathbf{arity0\ a} \tag{2}$$
$$\qquad [\![\textsc{Nat}\langle()\rangle \longrightarrow 0]\!]\}; \tag{3}$$
$$\quad \mathsf{opcon}\ \boldsymbol{s}\ \mathsf{of}\ 1\ \{\lambda\mathbf{i}{:}1.\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}].\mathbf{arity1\ a}\ \lambda\mathbf{d}{:}\mathsf{D}. \tag{4}$$
$$\qquad \mathbf{ifeq}\ \mathsf{typeof}(\mathbf{d})\ \textsc{Nat}\langle()\rangle \tag{5}$$
$$\qquad\quad [\![\textsc{Nat}\langle()\rangle \longrightarrow \mathsf{transof}(\mathbf{d}) + 1]\!]\}; \tag{6}$$
$$\quad \mathsf{opcon}\ \boldsymbol{rec}\ \mathsf{of}\ 1\ \{\lambda\mathbf{i}{:}1.\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}].\mathbf{arity3\ a}\ \lambda\mathbf{d1}{:}\mathsf{D}.\lambda\mathbf{d2}{:}\mathsf{D}.\lambda\mathbf{d3}{:}\mathsf{D}. \tag{7}$$
$$\qquad \mathbf{ifeq}\ \mathsf{typeof}(\mathbf{d1})\ \textsc{Nat}\langle()\rangle \tag{8}$$
$$\qquad \mathsf{let}\ \mathbf{t2} = \mathsf{typeof}(\mathbf{d2})\ \mathsf{in} \tag{9}$$
$$\qquad \mathbf{ifeq}\ \mathsf{typeof}(\mathbf{d3})\ \textsc{Arrow}\langle(\textsc{Nat}\langle()\rangle, \textsc{Arrow}\langle(\mathbf{t2},\mathbf{t2})\rangle)\rangle \tag{10}$$
$$\qquad\quad [\![\mathbf{t2} \longrightarrow (\mathsf{fix}\ f{:}\mathsf{int} \rightharpoonup \mathsf{repof}(\mathbf{t2})\ \mathsf{is}\ \lambda x{:}\mathsf{int}. \tag{11}$$
$$\qquad\qquad\quad \mathsf{if}\ x \equiv_{\mathsf{int}} 0\ \mathsf{then}\ \mathsf{transof}(\mathbf{d2})\ \mathsf{else} \tag{12}$$
$$\qquad\qquad\qquad \mathsf{transof}(\mathbf{d3})\ (x-1)\ (f\ (x-1)) \tag{13}$$
$$\qquad\qquad)\ \mathsf{transof}(\mathbf{d1})]\!]\} \tag{14}$$
$$\}; \tag{15}$$
$$\mathsf{let}\ \mathbf{nat} = \textsc{Nat}\langle()\rangle\ \mathsf{in} \tag{16}$$
$$\mathsf{let}\ \mathit{two} = \textsc{Nat}.\boldsymbol{s}\langle()\rangle(\textsc{Nat}.\boldsymbol{s}\langle()\rangle(\textsc{Nat}.\boldsymbol{z}\langle()\rangle()))\ \mathsf{in} \tag{17}$$
$$\mathsf{let}\ \mathit{plus} = \lambda x{:}\mathbf{nat}.\lambda y{:}\mathbf{nat}. \tag{18}$$
$$\quad \textsc{Nat}.\boldsymbol{rec}\langle()\rangle(x;y;\lambda p{:}\mathbf{nat}.\lambda r{:}\mathbf{nat}.\textsc{Nat}.\boldsymbol{s}\langle()\rangle(r))\ \mathsf{in} \tag{19}$$
$$\textsc{Arrow}.\boldsymbol{ap}\langle()\rangle(\mathit{plus}; \textsc{Arrow}.\boldsymbol{ap}\langle()\rangle(\mathit{two};\mathit{two})) \tag{20}$$

Figure 2. Gödel's **T** in @λ, used to calculate 2+2. The simple helper functions **arity**n and **ifeq** have return kind D+1 (see text). Their definitions can be found in the appendix. We use let to bind both external and type-level variables for clarity of presentation; these can be eliminated manually. We will discuss the issue of a more natural syntax in Sec. 4.

In most languages, types are formed by applying one of a fixed collection of *type constructors* to zero or more *indices*. In @λ, user-defined type constructors can be declared at the top-level of a program (or lifted there, in a practical implementation) using tycon (Fig. 3). Each type constructor in the program must have a unique name, written e.g. NAT.[1] A type constructor must also declare an *index kind*, κ_{idx}. Introducing a type is thus type constructor application to an index, written $\textsc{Tycon}\langle\tau_{\text{idx}}\rangle$. The kind Ty also has an elimination form: a type can be case analyzed against an available type constructor to extract its index. We are, to a first approximation, treating type constructors as constructors of a built-in open datatype at the type-level. Like open datatypes, there is no longer a notion of exhaustiveness and the default case is required for case analysis.

To permit the implementation of interesting type systems, the type-level language includes several kinds other than Ty. We lift several standard functional fragments to the type level: unit (1), binary sums ($\kappa_1 + \kappa_2$), binary products ($\kappa_1 \times \kappa_2$), lists ($\mathsf{list}[\kappa]$) and integers (int). We also include labels (Lbl), written in a slanted font, e.g. *myLabel*, which are string-like values that only admit comparisons, here only for equality, and play a distinguished role in the expanded syntax, as we will later discuss (strings themselves could also be included, but we omit them for concision). Our first example, NAT, is indexed trivially, i.e. by unit kind, 1, because there is only one natural number type, $\textsc{Nat}\langle()\rangle$, but we will show examples of type constructors that are indexed in more interesting ways in later portions of this work. For example, TUPLE has index kind list[Ty] and LABELEDTUPLE has index kind list[Lbl $\times$ Ty]. The type constructor ARROW is included in the initial constructor context and has index kind Ty $\times$ Ty.

[1] We assume naming conflicts can be avoided by some extrinsic mechanism.

$$\boxed{\Delta \vdash_\Phi \rho}$$

K-TYCON
$$\frac{\textsc{Tycon} \notin \mathsf{dom}(\Phi) \qquad \kappa_{\text{idx}}\ \mathsf{eq} \qquad \Delta \vdash_\Phi \tau_{\text{rep}} : \kappa_{\text{idx}} \to \mathsf{ITy} \qquad \Delta \vdash_{\Phi,\textsc{Tycon}\{\kappa_{\text{idx}};\tau_{\text{rep}};\theta\}} \theta \qquad \Delta \vdash_{\Phi,\textsc{Tycon}\{\kappa_{\text{idx}};\tau_{\text{rep}};\theta\}} \rho}{\Delta \vdash_\Phi \mathsf{tycon}\ \textsc{Tycon}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\mathsf{schema}\ \tau_{\text{rep}};\theta\};\ \rho}$$

K-E-PROG
$$\frac{\Delta \vdash_\Phi e\ \mathtt{ok}}{\Delta \vdash_\Phi e}$$

$$\boxed{\Delta \vdash_\Phi \theta}$$

K-OPCON
$$\frac{\Delta \vdash_\Phi \tau_{\text{def}} : \kappa_{\text{idx}} \to \mathsf{list}[\mathsf{D}] \to (\mathsf{D}+1)}{\Delta \vdash_\Phi \mathsf{opcon}\ \boldsymbol{op}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\tau_{\text{def}}\}}$$

K-OPCONS
$$\frac{\Delta \vdash_\Phi \theta_1 \qquad \Delta \vdash_\Phi \theta_2 \qquad \mathrm{dom}(\theta_1) \cap \mathrm{dom}(\theta_2) = \emptyset}{\Delta \vdash_\Phi \theta_1;\theta_2}$$

$$\boxed{\Delta \vdash_\Phi e\ \mathtt{ok}}$$

K-E-VAR
$$\frac{}{\Delta \vdash_\Phi x\ \mathtt{ok}}$$

K-E-LAM
$$\frac{\Delta \vdash_\Phi \tau : \mathsf{Ty} \qquad \Delta \vdash_\Phi e\ \mathtt{ok}}{\Delta \vdash_\Phi \lambda x{:}\tau.e\ \mathtt{ok}}$$

K-E-OP
$$\frac{\textsc{Tycon}\{-;-;\theta\} \in \Phi \qquad \mathsf{opcon}\ \boldsymbol{op}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{-\} \in \theta \qquad \Delta \vdash_\Phi \tau_{\text{idx}} : \kappa_{\text{idx}} \qquad \Delta \vdash_\Phi e_1\ \mathtt{ok} \quad \cdots \quad \Delta \vdash_\Phi e_n\ \mathtt{ok}}{\Delta \vdash_\Phi \textsc{Tycon}.\boldsymbol{op}\langle\tau_{\text{idx}}\rangle(e_1;\ldots;e_n)\ \mathtt{ok}}$$

Figure 3. Kinding for programs. The kinding context Δ maps type variables to kinds (we do not specify let binding of type variables at the level of programs or external terms in the core, but this would be a simple addition). Type-level terms stored in the constructor context are not needed during kinding, indicated using a dash. Kinding for type-level terms is specified in Figure **??**.

$$\boxed{\vdash_\Phi \rho \Longrightarrow \gamma} \qquad \Phi ::= \emptyset \mid \Phi, \textsc{Tycon}\{\kappa_{\text{idx}};\tau_{\text{rep}};\theta\}$$

ATT-TYCON
$$\frac{\vdash_{\Phi,\textsc{Tycon}\{\kappa_{\text{idx}};\tau_{\text{rep}};\theta\}} \rho \Longrightarrow \gamma}{\vdash_\Phi \mathsf{tycon}\ \textsc{Tycon}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\mathsf{schema}\ \tau_{\text{rep}};\theta\};\ \rho \Longrightarrow \gamma}$$

ATT-E-PROG
$$\frac{\emptyset \vdash_\Phi e : \tau \Longrightarrow \iota}{\vdash_\Phi e \Longrightarrow \gamma}$$

$$\boxed{\Gamma \vdash_\Phi e : \tau \Longrightarrow \iota} \qquad \Gamma ::= \emptyset \mid \Gamma, x : \tau$$

ATT-E
$$\frac{\Gamma \vdash_\Phi e : \tau \longrightarrow \bar\iota \qquad \vdash_\Phi \bar\iota \lightning \iota}{\Gamma \vdash_\Phi e : \tau \Longrightarrow \iota}$$

$$\boxed{\Gamma \vdash_\Phi e : \tau \longrightarrow \bar\iota}$$

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

ATT-LAM
$$\frac{\tau_1 \Downarrow \tau_1' \qquad \Gamma, x : \tau_1' \vdash_\Phi e : \tau_2 \longrightarrow \bar\iota}{\Gamma \vdash_\Phi \lambda x{:}\tau_1.e : \textsc{Arrow}\langle(\tau_1', \tau_2)\rangle \longrightarrow \lambda x{:}\mathsf{repof}(\tau_1').\bar\iota}$$

ATT-OP
$$\frac{\begin{array}{c}\textsc{Tycon}\{-;-;\theta\} \in \Phi \qquad \mathsf{opcon}\ \boldsymbol{op}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\tau_{\text{def}}\} \in \theta \\ \Gamma \vdash_\Phi e_1 : \tau_1 \longrightarrow \bar\iota_1 \quad \cdots \quad \Gamma \vdash_\Phi e_n : \tau_n \longrightarrow \bar\iota_n \\ \tau_{\text{def}}\ \tau_{\text{idx}}\ (\llbracket\tau_1 \longrightarrow \bar\iota_1\rrbracket^{\checkmark} :: \cdots :: \llbracket\tau_n \longrightarrow \bar\iota_n\rrbracket^{\checkmark} :: []_{\mathsf{D}}) \Downarrow \mathsf{inl}[1](\llbracket\tau \longrightarrow \bar\iota\rrbracket) \\ \vdash_\Phi^{\textsc{Arrow},\textsc{Tycon}} \mathsf{repof}(\tau) \leadsto \bar\sigma \\ \vdash_\Phi^{\textsc{Arrow},\textsc{Tycon}} \Gamma \leadsto \Psi \qquad \Psi \vdash_\Phi^{\textsc{Arrow},\textsc{Tycon}} \bar\iota \sim \bar\sigma\end{array}}{\Gamma \vdash_\Phi \textsc{Tycon}.\boldsymbol{op}\langle\tau_{\text{idx}}\rangle(e_1;\ldots;e_n) : \tau \Longrightarrow \bar\iota}$$

Figure 4. Active typechecking and translation. Normalization judgement for type-level terms is specified in Fig. **??**. Local concretization ($\leadsto$) and abstract internal typechecking ($\sim$) judgements are specified in Fig. **??**. Full concretization ($\lightning$) is specified in Fig. **??**.

Two type-level terms of kind Ty are equivalent if they apply the same constructor, identified by name, to equivalent indices. Going further, we ensure that deciding type equivalence requires only checking for syntactic equality after normalization by imposing the restriction that type constructors can only be indexed by kinds for which equivalence can be decided in this way. All combinations of kinds introduced thus far have this property and this is sufficient for many interesting examples. Our treatment of equivalence in the type-level language is thus quite similar to the treatment of term-level equality using "equality types" in a language like Standard ML. An *equality kind* is defined by the judgement . We discuss introducing a richer notion of equivalence of types that fragment providers can extend as future work in Sec. 6.

eq type discussion

Type constructors are not first-class; they do not themselves have arrow kind as in some kind systems (e.g. [?]; Ch. 22 of *PFPL* describes a related system [6]). The type-level language does, however, include total functions of conventional arrow kind, written $\kappa_1 \to \kappa_2$. Type constructor application can be wrapped in a type-level function to emulate a first-class type constructor (indeed, such a wrapper could be generated automatically, though we avoid this for simplicity in our semantics). Equivalence at arrow kind does not meet our criteria, so type-level functions cannot appear within type indices. This also prevents general recursion from arising at the type level. Without this restriction, a function of type $\mathsf{Ty} \to \kappa$ could "smuggle" in a self-reference as a type index and extract it via case analysis (an issue related to the positivity condition for inductive datatypes in total functional languages).

do this

cite

Every type constructor also defines a *representation schema* using schema, a type-level function that we will discuss in Sec. **??**after introducing operator constructors.

ref

2.3 Operator Constructors

For reasons that we will discuss, our calculus associates every operator constructor[2] with a type constructor.

User-defined operator constructors are declared using opcon (Fig. 3). The *fully-qualified name* of every operator constructor, e.g. $\textsc{Nat}.\boldsymbol{z}$, must be unique. Operator constructors are indexed by type-level values, like type constructors, and so also declare an index kind κ_{idx}. In our first example, all the operator constructors are indexed trivially (by index kind 1), but later examples will use more interesting indices. For example, the projection operator for labeled tuples, $\textsc{LabeledTuple}.\boldsymbol{prj}$, has index kind Lbl. Note that neither operator constructors nor operators are first-class type-level values and we do not need to impose any restrictions on their index kinds.

defne operator

In the external language, an operator is selected and invoked by applying an operator constructor to a type-level index and $n \geq 0$ *arguments*, written $\textsc{Tycon}.\boldsymbol{op}\langle\tau_{\text{idx}}\rangle(e_1;\ldots;e_n)$. For example, on line 18 of Fig. 2, we see the operator constructors $\textsc{Nat}.\boldsymbol{z}$ and $\textsc{Nat}.\boldsymbol{s}$ being invoked to compute *two* (the syntactic overhead will be relieved later). To derive the active typing judgement for this form, the semantics invokes the operator constructor's *definition*: a type-level function that must examine the provided operator index and the recursively determined *derivates* of the arguments to determine a derivate for the operation as a whole, or indicate an error. A derivate is a type-level representation of the result of deriving the

[2] It may be helpful to emphasize that *term formers* are distinct from type constructors and operator constructors. There are term formers at all levels in the calculus. For example, type constructor application is a type-level term former. An abstract syntax at the type level like Harper's [6] might write it $\mathtt{tcapp}[\textsc{Tycon}](\tau)$, emphasizing the distinction.

$$\boxed{\vdash^{\Xi}_{\Phi} \bar{\sigma} \rightsquigarrow \bar{\sigma}'} \quad \Xi ::= \emptyset \mid \Xi, \textsc{Tycon}$$

$$\textsc{local-conc-show-rep}\ \frac{\textsc{Tycon} \in \Xi \qquad \textsc{Tycon}\{-; \tau_{\text{rep}}; -\} \in \Phi \qquad \tau_{\text{rep}}\ \tau_{\text{idx}} \Downarrow \blacktriangleright(\bar{\sigma}) \qquad \vdash^{\Xi}_{\Phi} \bar{\sigma} \rightsquigarrow \bar{\sigma}'}{\vdash^{\Xi}_{\Phi} \mathsf{repof}(\textsc{Tycon}\langle\tau_{\text{idx}}\rangle) \rightsquigarrow \bar{\sigma}'}$$

$$\textsc{local-conc-hide-rep}\ \frac{\textsc{Tycon} \notin \Xi}{\vdash^{\Xi}_{\Phi} \mathsf{repof}(\textsc{Tycon}\langle\tau_{\text{idx}}\rangle) \rightsquigarrow \mathsf{repof}(\textsc{Tycon}\langle\tau_{\text{idx}}\rangle)}$$

$$\textsc{local-conc-unquote}\ \frac{\vdash^{\Xi}_{\Phi} \bar{\sigma} \rightsquigarrow \bar{\sigma}'}{\vdash^{\Xi}_{\Phi} \blacktriangleleft(\blacktriangleright(\bar{\sigma})) \rightsquigarrow \bar{\sigma}'} \qquad \textsc{local-conc-parr}\ \frac{\vdash^{\Xi}_{\Phi} \bar{\sigma}_1 \rightsquigarrow \bar{\sigma}_1' \qquad \vdash^{\Xi}_{\Phi} \bar{\sigma}_2 \rightsquigarrow \bar{\sigma}_2'}{\vdash^{\Xi}_{\Phi} \bar{\sigma}_1 \rightharpoonup \bar{\sigma}_2 \rightsquigarrow \bar{\sigma}_1' \rightharpoonup \bar{\sigma}_2'}$$

(remaining rules follow recursively, omitted)

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

$$\textsc{abs-empty}\ \frac{}{\vdash^{\Xi}_{\Phi} \emptyset \rightsquigarrow \emptyset} \qquad \textsc{abs-ctx}\ \frac{\vdash^{\Xi}_{\Phi} \Gamma \rightsquigarrow \Psi \qquad \vdash^{\Xi}_{\Phi} \mathsf{repof}(\tau) \rightsquigarrow \bar{\sigma}}{\vdash^{\Xi}_{\Phi} \Gamma, x : \tau \rightsquigarrow \Psi, x \sim \bar{\sigma}}$$

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

$$\textsc{abs-i-var}\ \frac{}{\Psi, x \sim \bar{\sigma} \vdash^{\Xi}_{\Phi} x \sim \bar{\sigma}} \qquad \textsc{abs-i-fix}\ \frac{\vdash^{\Xi}_{\Phi} \bar{\sigma} \rightsquigarrow \bar{\sigma}' \qquad \Psi, x \sim \bar{\sigma}' \vdash^{\Xi}_{\Phi} \bar{\iota} \sim \bar{\sigma}'}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{fix}\ x{:}\bar{\sigma}\ \mathsf{is}\ \gamma_{\text{abs}} \sim \bar{\sigma}'}$$

(remaining rules follow from a standard statics similarly, omitted)

$$\textsc{show-trans}\ \frac{\textsc{Tycon} \in \Xi \qquad \Psi \vdash^{\Xi}_{\Phi} \bar{\iota} \sim \bar{\sigma}}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{transof}(\llbracket \textsc{Tycon}\langle\tau_{\text{idx}}\rangle \longrightarrow \bar{\iota} \rrbracket) \sim \bar{\sigma}}$$

$$\textsc{hide-trans}\ \frac{\textsc{Tycon} \notin \Xi}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{transof}(\llbracket \textsc{Tycon}\langle\tau_{\text{idx}}\rangle \longrightarrow \bar{\iota} \rrbracket^{\checkmark}) \sim \mathsf{repof}(\textsc{Tycon}\langle\tau_{\text{idx}}\rangle)}$$

Figure 5. Abstracted internal typing

active typing judgement[3]: an internal term, called the translation, paired with a type. Derivates have kind D and introductory form $\llbracket \tau_{\text{ty}} \longrightarrow \tau_{\text{trans}} \rrbracket$. The elimination form $\mathsf{let}\ \llbracket \mathbf{t} \longrightarrow \mathbf{x} \rrbracket = \tau\ \mathsf{in}\ \tau'$ extracts the translation and type from the derivate τ, binding it to $\mathbf{x}$ and $\mathbf{t}$ respectively in τ'. Because constructing a derivate is not always possible (e.g. when there is a type error in the client's code, or an invalid index was provided), the return kind of the function is an "option kind", $\mathsf{D} + 1$, where the trivial case indicates a statically-detected error in the client's program. In practice, it would instead require providers to report information about the precise location of the error and an appropriate error message.

A translation, as we have said, is an internal term. To construct an internal term using a type-level function, as we must do to construct a derivate, we must expose a type-level representation of the internal language. A *quoted internal term* is a type-level value of kind ITm with introductory form $\triangleright(\bar{\iota})$ and a *quoted internal type* is a type-level value of kind ITy with introductory form $\blacktriangleright(\bar{\sigma})$. Neither kind has an elimination form. Instead, the syntax for the quoted internal language includes complementary *unquote forms* $\triangleleft(\tau)$ and $\blacktriangleleft(\tau)$ that permit the interpolation of another quoted term or type, respectively, into the one being formed. Interpolation is capture-avoiding, as our semantics will clarify. We have now described all the kinds in our language.

[3] Technically, the abstract active typing judgement, which is similar in form and we will introduce shortly.

The definition of $\textsc{Nat}.\boldsymbol{z}$ is quite simple: it returns the derivate $\llbracket \textsc{Nat}\langle()\rangle \longrightarrow \triangleright(0) \rrbracket$ if no arguments were provided, and indicates an error (an *arity error*, though we do not distinguish this in our calculus) otherwise. This is done by calling a simple helper function, **is_empty** : $\mathsf{list}[\mathsf{D}] \to \mathsf{D} \to (\mathsf{D}+1)$, that selects the appropriate case of the sum given the derivate that should be produced if the list is indeed empty. The definition of $\textsc{Nat}.\boldsymbol{s}$ is only slightly more complex, because it requires inspecting a single argument. The helper function **pop_final** : $\mathsf{list}[\mathsf{D}] \to (\mathsf{Ty} \to \mathsf{ITm} \to (\mathsf{D}+1)) \to (\mathsf{D}+1)$ "pops" a derivate from the head of the list and, if no other arguments remain, passes its translation and type to the "continuation", returning the error case otherwise. The continuation checks if the argument type is equal to $\textsc{Nat}\langle()\rangle$ using **check_type** : $\mathsf{Ty} \to \mathsf{Ty} \to \mathsf{D} \to (\mathsf{D}+1)$, which operates similarly. If these arity and type checks succeed, the resulting derivate is composed by adding one to the translation of the argument, passed into the continuation as $\mathbf{x}$ in our example, and pairing it with the natural number type: $\llbracket \textsc{Nat}\langle()\rangle \longrightarrow \triangleright(\triangleleft(\mathbf{x}) + 1) \rrbracket$. We will return to the definition of $\textsc{Nat}.\boldsymbol{rec}$ after in the next subsection.

By writing the typechecking and translation logic in this way, as a total function, we are taking a rather "implementation-focused view" rather than attempting to extract such a function from a declarative specification. That is, we leave to the provider (or to a program generator or kind-specific language that transforms such a specification into an operator definition) the problem of finding a deterministic algorithm that adequately implements their intended semantics, allowing us to prove decidability of typechecking for the language as a whole by essentially just citing the termination theorem for the type-level language. We also avoid difficulties with error reporting in practice. Rob Simmons' undergraduate thesis has a good discussion of these issues [**?**].

Because the input to an operator definition is the recursively determined derivate of the argument in the same context as the operator appears in, our mechanism does not presently permit the definition of operator constructors that bind variables themselves or require a different form of typing judgement (e.g. additional forms of contexts). This is also why the λ operator constructor needs to be built in. It is the only operator in our language that can be used to bind variables. The type constructor $\textsc{Arrow}$ can be defined from within the language (and is included in the "prelude" constructor context), but only defines the operator constructor, $\boldsymbol{ap}$. These two operators translate to their corresponding forms in the internal language directly, as we will discuss below. We leave adding the ability to declare and manipulate new contexts as future work.

2.4 Representational Consistency Implies Type Safety

In our example, natural numbers are represented internally as integers. Were this not the case – if, for example, we added an operator constructor $\textsc{Nat}.\boldsymbol{z2}$ that produced the derivate $\llbracket \textsc{Nat}\langle()\rangle \longrightarrow \triangleright(\mathsf{inl}[\mathsf{int}](())) \rrbracket$, then there would be two different internal types, int and $1 + \mathsf{int}$, associated with a single external type, $\textsc{Nat}\langle()\rangle$. This makes it impossible to operate compositionally on the translation of an external term of type $\textsc{Nat}\langle()\rangle$, so our implementation of $\textsc{Nat}.\boldsymbol{s}$ would produce ill-typed translations in some cases but not others. Similarly, we wouldn't be able to write functions over all natural numbers because there would not be a well-typed translation to give to such a function.

To reason compositionally about the semantics of well-typed external terms when they are given meaning by translation to a typed internal language, we must have the following property: for every type, τ, there must exist an internal type, σ, called its *representation type*, such that the translation of every external term of type τ has internal type σ. This principle of *representational consistency* arises essentially as a strengthening of the inductive hypothesis necessary to prove that all well-typed external terms translate to well-typed internal terms, precisely because operators like λ and NAT.s are defined compositionally. It is closely related to the concept of *type-preserving compilation* developed by Morrisett et al. for the TIL compiler for Standard ML [**?**].

cite

It is easy to show by induction that, under a "closed-world assumption" where the only available operators are NAT.z and NAT.s, the representation type of NAT$\langle()\rangle$ is int. If we can maintain this under an "open-world assumption" (requiring that, for example, applications of operator constructors like NAT.$z2$, above, are not well-typed), and we target a type safe internal language, then we will achieve type safety: well-typed external terms cannot go wrong, because they always translate to well-typed internal terms, which cannot go wrong. For the semantics to ensure that representational consistency is maintained by all operator definitions, we require that each type constructor must declare a *representation schema* with the keyword schema. This must be a type-level function of kind $\kappa_{\text{idx}} \rightarrow$ ITy, where κ_{idx} is the index kind of the type constructor. When the compiler needs to determine the representation type of the type TYCON$\langle\tau_{\text{idx}}\rangle$ it simply applies the representation schema of TYCON to τ_{idx}.

As described above, the kind ITy has introductory form $\blacktriangleright(\bar{\sigma})$ and no elimination form. There are two forms in $\bar{\sigma}$ that do not correspond to forms in σ that allow quoted internal types to be formed compositionally:

1. $\blacktriangleleft(\tau)$, already described, unquotes (or "interpolates") the quoted internal type τ
2. repof(τ) refers to the representation type of type τ (we will see in the next subsection why this needs to be in the syntax for $\bar{\sigma}$ and not directly in the type-level language)

These additional forms are not needed by the representation schema of NAT because it is trivially indexed. In Fig. **??**, we show an example of the type constructor TUPLE, implementing the semantics of n-tuples by translation to nested binary products. Here, the representation schema requires referring to the representations of the tuple's constituent types, given in the type index.

add this example from appendix of ESOP

Operator constructor definitions might also need to refer to the representation of a type. We see this in the definition of the recursor on natural numbers, NAT.***rec***. After checking the arity and extracting the types and translations of its three arguments, it produces a derivate that implements the necessarily recursive dynamic semantics using a fixpoint computation in the internal language. This fixpoint computation produces a result of some arbitrary type, **t2**. We cannot know what the representation type of **t2** is, we refer to it abstractly using repof(**t2**). The translation is well-typed no matter what the representation type is. In other words, a proof of representational consistency of the derivate produced by the definition of NAT.***rec*** is parametric (in the metamathematics) over the representation type of **t2**. This leads us into the concept of conservativity.

2.5 Representational Independence Implies Conservativity

In isolation, our definition of natural numbers can be shown to be a strong encoding of the semantics of Gödel's **T**. That is, there is a bijection between the terms and types of the two languages and the typing judgements preserve this mapping. Moreover, we can show by a bisimulation argument that the translation produced by @λ implements the dynamic semantics of Gödel's **T**. We will provide more details on this later. A necessary lemma in the proof, however, is that the value of every translation of an external term of type NAT$\langle()\rangle$ is a non-negative integer. This is needed to show that the fixpoint computation in the definition of NAT.***rec*** is terminating, as the recursor always does in **T**. A very similar lemma is needed to show that the translation of every external term of type NAT$\langle()\rangle$ is equivalent to the translation that would be produced by some combination of applications of NAT.z and NAT.s (the analog to a canonical forms lemma in our setting). Fortunately, the definitions we have provided thus far admit such lemmas.

However, these theorems are all quite precarious because they rely on exhaustive induction over the available operators. As soon as we load another library into the program, these theorems may no longer be *conserved*. For example, the following operator declaration in some other type that we have loaded would topple our house of cards:

TBC

3. Semantics and Metatheory

4. Expanded Syntax

5. Examples

6. Limitations

Indeed, it is not a stretch to imagine a reasonably faithful implementation of our calculus starting with a pure, total subset of the term language of ML to construct the type-level language. We leave the development of such an implementation (and of bootstrapping type-level and internal languages that are themselves user-defined or extensible) as areas for future work to focus sharply on the theoretical foundations in this work.

integrate this

7. Related Work

Our representation schema abstraction mechanism relates closely to abstract and existential types [6**?**]. 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 [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 [11]. 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]). We show how to integrate language extensions into the type-level language, drawing on ideas about [12].

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 [7], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [10]), 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 [5], 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] 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] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, 2011.

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

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

[8] A. J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, pages 7–13, Sept. 1982.

[9] J. C. Reynolds. An introduction to polymorphic lambda calculus. In *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, 1994.

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

[11] 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.

[12] 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.