Active Embeddings: 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 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 into the language, the "actively typed semantics" guarantees type safety. Going further, the semantics enforce an abstraction barrier 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 *actively embedded* by this mechanism as orthogonal "libraries".

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 is called $\mathcal{L}\{\rightarrow\}$, following the notational convention used by Harper [9] 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 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 must take a moment to reflect. In $\mathcal{L}\{\rightarrow \ \forall\}$, studied variously by Girard as System $\mathbf{F}$ [8] and Reynolds as the polymorphic lambda calculus [19], it is known that sums, products, and inductive and co-inductive types can all be weakly defined. This means that we can *translate* well-typed terms of a language like $\mathcal{L}\{\rightarrow \forall\ \texttt{nat} + \times\}$ to well-typed terms in $\mathcal{L}\{\rightarrow \forall\}$ in a manner that preserves their dynamic semantics. But Reynolds, in a remark that recalls the "Turing tarpit" of Perlis [17], reminds us that programming with the corresponding *embedding* may be unwise [19]:

> 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.

A stronger embedding requires constructing an isomorphism that preserves both static reasoning principles and dynamic behaviors, in a sense that we will make more precise as we go on. This is generally far more difficult to establish than a weak embedding, and fewer universality results can be applied. For example, the aforementioned embeddings into $\mathcal{L}\{\rightarrow \ \forall\}$ do not preserve type disequality and some other equational reasoning principles (we will see less subtle issues soon). Establishing an isomorphic embedding that is natural (as measured, to a first approximation, by the ratio of boilerplate code that must be manually generated in the embedding vs. in the fragment, and how likely one is to make a mistake when writing it) and that has a reasonable cost semantics is harder still.

Modern languages like ML do, of course, provide constructs, like datatypes and abstract types, that admit strong and often satisfying embeddings of many language fragments, occupying what is widely seen as a "sweet spot" in the design space. However, "general-purpose" and "all-purpose" remain quite distinct, and situations continue to arise in both research and practice where desirable fragments can still only be weakly defined in terms of general-purpose constructs, or an isomorphic embedding, while possible, is widely seen as too verbose or inefficient. For example:

1. General-purpose constructs continue to evolve to better handle various usage scenarios. There are many variants of product types: records, records with functional record update, labeled tuples (records that specify a field ordering) and record-like data structures with field delegation (of various sorts). These are either awkward to isomorphically embed or cannot be. Similarly, sum types also admit many variants (e.g. various forms of open sums and object systems, seen in a certain light). Even something as seemingly simple as a type-safe `sprintf` operator requires special support from languages 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 data parallel programming [2], concurrency [18], distributed programming [15], dataflow programming [12], authenticated data structures [14], database queries [16], units of measure [11] and many other domains.
3. Safe and natural foreign function interfaces (FFIs) require enforcing the type system of the foreign language within the calling language. For example, MLj extended Standard ML with constructs for safely and naturally interfacing with Java [1]. For any other language, including others on the JVM like Scala and the many languages that might be accessible via a 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 dialects 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 sufficiently diverse software ecosystem, dialect providers have little choice but to leave this task to clients (providing, perhaps, some informal guidelines or partial automation). Avoiding dialect composition is difficult, because interactions between components of an application written in different dialects can lead to precisely the problems of item 3.

These are not the sorts of problems usually faced by library providers. Well-designed languages preclude the possibility of "link-time" conflicts between 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 Drawing from this approach, we aim to introduce 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 general form consistent with this protocol, a n isomorphic embedding in the external language can be constructed by simply introducing the necessary constructors and defining directly the necessary type assignment and translation logic. We call such an embedding an *active embedding*.

The semantics imposes various checks to ensure type safety and decidability of type assignment and equality. Moreover, an important class of lemmas that play a key role in proofs that a library enables strong embeddings of a language fragment can be derived in a suitable "closed world" (that is, in the traditional manner where one reasons inductively over a *finite* collection of constructors). The semantics guarantees that they will be *conserved* in the "open world" by enforcing an abstraction barrier at extension boundaries. This avoids the most fundamental semantic problems that can arise when composing dialects and justifies the inclusion of the mechanism inside the language.

We will begin in Sec. 2 by introducing a minimal calculus with this kind of actively typed semantics, $@\lambda$. 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 simple fragments like the `nat` fragment of Gödel's $\mathbf{T}$ or ML-style n-ary products. We actively embed these into the external language 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 two sections comprise the fundamental contributions of this work.

While the core calculus addresses the issue of making a strong embedding possible, active embeddings into the core calculus are often 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: labeled sums and products, the latter with delegation and functional field update, and a variant of `sprintf`.

update this if something doesn't make it

While of surprising expressiveness given its minimality, $@\lambda$ itself is certainly not capable of admitting embeddings of the full variety of type system fragments enumerated earlier. Indeed, to guarantee safety and conservativity and because our intention in this paper is to cleanly expose the essence of this mechanism, our calculus makes perhaps too many assumptions about the type system being embedded. For example, it only admits fragments where the typing judgement looks essentially like the typing judgement of the $\rightarrow$ fragment, so new contexts cannot be introduced. We discuss this and some of the other reasons why a fragment may not be satisfyingly embeddable in Sec. 6. These limitations do not appear fundamental, so we also outline directions for future work. We conclude by considering related work in Sec. 7.

2. Core $@\lambda$

The syntax of Core $@\lambda$ is given in Fig. 1. An example of a program defining type and operator constructors that admit an active embedding of Gödel's $\mathbf{T}$ into $@\lambda$ is given in Fig. 2. We will discuss its semantics and how precisely that 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 as an abstract type, for example). We will provide more sophisticated examples where this is less feasible later on (and note that type abstraction itself ..... in Sec. 6).

if i don't have time/space, take this out

2.1 Overview

A *program*, ρ, consists of a series of static declarations (in the core, only constructor declarations) followed by an external term,

$$
\begin{array}{rclll}
\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 \textit{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 \leadsto \bar{\iota}]\!] \mid [\![\tau \leadsto \bar{\iota}]\!]^{\checkmark} \mid \mathsf{typeof}(\tau) \\
\text{rep types} & & \mid & \blacktriangleright(\bar{\sigma}) \\
\text{translational 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 (e.g. addition, comparison).

e. The syntax for external terms contains three forms: variables, λ terms, and a form for invoking user-defined operators, which we will discuss below. Compiling a program consists of first *kind checking* it then typechecking the external term and translating it to a term, ι, in the typed internal language. These correspond to the premises of the *central compilation judgement* $\rho \Longrightarrow \iota$:

$$\text{P-COMPILES}\quad \frac{\emptyset \vdash_{\Phi_0} \rho \qquad \vdash_{\Phi_0} \rho \Longrightarrow \iota}{\rho \Longrightarrow \iota}$$

This anchors our exposition; we will describe how it is derived (i.e. how to write a compiler for @λ) in the following sections.

The key judgement in the calculus is the *active typing judgement* (Fig. 7, which we describe starting in Sec. 2.3). It relates an external term, e, to a type, τ, called its *type assignment*, and an internal term, ι, called its *translation*, under *typing context* Γ and *constructor context* Φ:

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

The typing context Γ maps variables to types in essentially the conventional way ([9] contains the necessary background for this paper). The constructor context tracks user-defined type and operator constructors. There is no separate operational semantics for the external language. Instead, the dynamic behavior of an external term is determined by its translation to the internal language, which has a more conventional operational semantics. This form of semantics can be seen as lifting into the language specification the first stage of a type-directed compiler like the TIL compiler for Standard ML [22] and has some parallels to the Harper-Stone semantics for Standard ML, where external terms were also given meaning by elaboration from the EL to an IL [10].

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

Figure 2. An embedding of Gödel's $\mathbf{T}$ in @λ, used to define $plus$ and two and calculate $plus\ two\ two$. Type constructors (line 1) and types (line 16) are discussed in Sec. 2.2. Schemas (line 1) and operator constructors (lines 2-14, used on lines 17-20) are discussed starting in Sec. 2.3. We use let to bind both external and type-level variables as needed for clarity of presentation. We will discuss further syntactic issues in Sec. 4.

In @λ, the internal language (IL) 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. The internal type system serves as a "floor": guarantees that must hold for *all* well-typed terms, independent of the constructor context (e.g. that out-of-bounds access to memory never occurs), must be maintained by the internal type system. User-defined constructors can (only) enforce invariants stronger than those the internal type system maintains. Performance is also ultimately limited by the internal language and downstream compilation stages that we do not here consider (safe compiler extension has been discussed in previous work, e.g. [23]).

2.2 Type Constructors and Types

In most languages, types are formed by applying one of a collection of *type constructors* to zero or more *indices*. In @λ, the situation is notionally similar. User-defined type constructors can be declared at the top of a program (or lifted to the top, in practice) using tycon. Each 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}. A type is introduced by applying a type constructor to an index of this kind, written $\textsc{Tycon}\langle\tau_{\text{idx}}\rangle$.

@λ supports, and makes extensive use of, simply-kinded type-level computation. Specifically, type-level terms, τ, themselves form a typed lambda calculus. The classifiers of type-level terms

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

$\boxed{\Delta \vdash_\Phi \rho}$ $\quad \Delta ::= \emptyset \mid \Delta, \mathbf{t} : \kappa \quad \Phi ::= \emptyset \mid \Phi, \textsc{Tycon}\{\kappa_{\text{idx}}; \tau_{\text{rep}}; \theta\}$

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

$$\textsc{K-E-Prog}\quad \frac{\Delta \vdash_\Phi e\ \texttt{wk}}{\Delta \vdash_\Phi e}$$

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

$$\textsc{K-Opcon}\quad \frac{\Delta \vdash_\Phi \tau_{\text{def}} : \kappa_{\text{idx}} \to \mathsf{list}[\mathsf{D}] \to (\mathsf{D} + 1)}{\Delta \vdash_\Phi \mathsf{opcon}\ \boldsymbol{op}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\tau_{\text{def}}\}} \qquad \textsc{K-Opcons}\quad \frac{\Delta \vdash_\Phi \theta_1 \qquad \Delta \vdash_\Phi \theta_2 \qquad \text{dom}(\theta_1) \cap \text{dom}(\theta_2) = \emptyset}{\Delta \vdash_\Phi \theta_1; \theta_2}$$

$\boxed{\Delta \vdash_\Phi e\ \texttt{wk}}$

$$\textsc{K-E-Var}\quad \frac{}{\Delta \vdash_\Phi x\ \texttt{wk}} \qquad \textsc{K-E-Lam}\quad \frac{\Delta \vdash_\Phi \tau : \mathsf{Ty} \qquad \Delta \vdash_\Phi e\ \texttt{wk}}{\Delta \vdash_\Phi \lambda x{:}\tau.e\ \texttt{wk}}$$

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

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

$$\textsc{T-Eq}\quad \frac{}{\mathsf{Ty}\ \mathsf{eq}} \qquad \textsc{I-Eq}\quad \frac{}{\mathsf{Int}\ \mathsf{eq}} \qquad \textsc{L-Eq}\quad \frac{}{\mathsf{Lbl}\ \mathsf{eq}} \qquad \textsc{List-Eq}\quad \frac{\kappa\ \mathsf{eq}}{\mathsf{list}[\kappa]\ \mathsf{eq}}$$

$$\textsc{U-Eq}\quad \frac{}{1\ \mathsf{eq}} \qquad \textsc{P-Eq}\quad \frac{\kappa_1\ \mathsf{eq} \qquad \kappa_2\ \mathsf{eq}}{\kappa_1 \times \kappa_2\ \mathsf{eq}} \qquad \textsc{S-Eq}\quad \frac{\kappa_1\ \mathsf{eq} \qquad \kappa_2\ \mathsf{eq}}{\kappa_1 + \kappa_2\ \mathsf{eq}}$$

Figure 3. Kinding for type and operator constructors and external terms, and equality kinds (see text). Type-level terms stored in the constructor context are not needed during kinding, indicated using a dash.

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

$$\textsc{K-Var}\quad \frac{}{\Delta, \mathbf{t} : \kappa \vdash_\Phi \mathbf{t} : \kappa} \qquad \textsc{K-Arrow-I}\quad \frac{\Delta, \mathbf{t} : \kappa_1 \vdash_\Phi \tau : \kappa_2}{\Delta \vdash_\Phi \lambda \mathbf{t}{:}\kappa_1.\tau : \kappa_1 \to \kappa_2} \qquad \textsc{K-Arrow-E}\quad \frac{\Delta \vdash_\Phi \tau_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash_\Phi \tau_2 : \kappa_1}{\Delta \vdash_\Phi \tau_1\ \tau_2 : \kappa_2}$$

(kinding for integers, labels, lists, products and sums also standard)

$$\textsc{K-Eq}\quad \frac{\kappa\ \mathsf{eq} \qquad \Delta \vdash_\Phi \tau_1 : \kappa \qquad \Delta \vdash_\Phi \tau_2 : \kappa \qquad \Delta \vdash_\Phi \tau_3 : \kappa' \qquad \Delta \vdash_\Phi \tau_4 : \kappa'}{\Delta \vdash_\Phi \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 : \kappa'} \qquad \textsc{K-Ty-I}\quad \frac{\textsc{Tycon}\{\kappa_{\text{idx}}; -; -\} \in \Phi \qquad \Delta \vdash_\Phi \tau_{\text{idx}} : \kappa_{\text{idx}}}{\Delta \vdash_\Phi \textsc{Tycon}\langle \tau_{\text{idx}} \rangle : \mathsf{Ty}}$$

$$\textsc{K-Ty-E}\quad \frac{\Delta \vdash_\Phi \tau : \mathsf{Ty} \qquad \textsc{Tycon}\{\kappa_{\text{idx}}; -; -\} \in \Phi \qquad \Delta, \mathbf{x} : \kappa_{\text{idx}} \vdash_\Phi \tau_1 : \kappa \qquad \Delta \vdash_\Phi \tau_2 : \kappa}{\Delta \vdash_\Phi \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Tycon}\langle \mathbf{x} \rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 : \kappa} \qquad \textsc{K-D-I}\quad \frac{\Delta \vdash_\Phi \tau : \mathsf{Ty} \qquad \Delta \vdash_\Phi \bar{\iota}\ \texttt{wk}}{\Delta \vdash_\Phi [\![\tau \rightsquigarrow \bar{\iota}]\!] : \mathsf{D}}$$

$$\textsc{K-D-I-Checked}\quad \frac{\Delta \vdash_\Phi \tau : \mathsf{Ty} \qquad \Delta \vdash_\Phi \bar{\iota}\ \texttt{wk}}{\Delta \vdash_\Phi [\![\tau \rightsquigarrow \bar{\iota}]\!]^{\checkmark} : \mathsf{D}} \qquad \textsc{K-D-E}\quad \frac{\Delta \vdash_\Phi \tau : \mathsf{D}}{\Delta \vdash_\Phi \mathsf{typeof}(\tau) : \mathsf{Ty}} \qquad \textsc{K-RepType-I}\quad \frac{\Delta \vdash_\Phi \bar{\sigma}\ \texttt{wk}}{\Delta \vdash_\Phi \blacktriangleright(\bar{\sigma}) : \mathsf{ITy}}$$

$\boxed{\Delta \vdash_\Phi \bar{\iota}\ \texttt{wk}}$

$$\textsc{K-I-Var}\quad \frac{}{\Delta \vdash_\Phi x\ \texttt{wk}} \qquad \textsc{K-I-Fix}\quad \frac{\Delta \vdash_\Phi \bar{\sigma}\ \texttt{wk} \qquad \Delta \vdash_\Phi \bar{\iota}\ \texttt{wk}}{\Delta \vdash_\Phi \mathsf{fix}\ x{:}\bar{\sigma}\ \mathsf{is}\ \bar{\iota}\ \texttt{wk}} \qquad \textsc{K-I-Transof}\quad \frac{\Delta \vdash_\Phi \tau : \mathsf{D}}{\Delta \vdash_\Phi \mathsf{transof}(\tau)\ \texttt{wk}}$$

(omitted rules are trivially recursive, like K-I-FIX)

$\boxed{\Delta \vdash_\Phi \bar{\sigma}\ \texttt{wk}}$

$$\textsc{K-S-Unquote}\quad \frac{\Delta \vdash_\Phi \tau : \mathsf{ITy}}{\Delta \vdash_\Phi \blacktriangleleft(\tau)\ \texttt{wk}} \qquad \textsc{K-S-Repof}\quad \frac{\Delta \vdash_\Phi \tau : \mathsf{Ty}}{\Delta \vdash_\Phi \mathsf{repof}(\tau)\ \texttt{wk}}$$

(omitted forms are trivially recursive)

Figure 4. Kinding for type-level terms. The kinding context Δ maps type variables to kinds. Checked derivates (introduced by rule K-D-I-CHECKED, discussed in Sec. 2.5) should only be constructed by the semantics (this can be enforced by a parser, so we will assume it in the metatheory).

are called *kinds*, κ, to distinguish them from *types*. The kinding rules are specified in Fig. 3 and Fig. 4. Types are type-level values of kind Ty, introduced as just described. The kind Ty also has an elimination form, $\mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Tycon}\langle \mathbf{x} \rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2$, allowing the extraction of a type index by case analysis against a contextually-available type constructor. To a first approximation, one might think of type constructors as constructors of a built-in open datatype, Ty, at the type-level. Like open datatypes, there is no notion of exhaustiveness so the default case is required for totality.

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 products ($\kappa_1 \times \kappa_2$), binary sums ($\kappa_1 + \kappa_2$), 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 support comparison and play a distinguished role in the expanded syntax, as we will later discuss. Our first example, NAT, is indexed trivially, i.e. by unit kind, 1, so 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, LABELEDTUPLE has index kind $\mathsf{list}[\mathsf{Lbl} \times \mathsf{Ty}]$. The type constructor ARROW is included in the initial constructor context, Φ_0, shown in Fig. 6, and has index kind Ty × Ty.

Type constructors are not first-class; they do not themselves have arrow kind as in some kind systems (e.g. [25]; Ch. 22 of *PFPL* describes a related system [9]). 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 or uncurried version of a type constructor for convenience (indeed, such a wrapper could be generated automatically, though we do not do so).

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 equivalence at a type constructor's index kind must be decidable in this way. 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. A kind κ is an *equality kind* if $\kappa\ \mathsf{eq}$ can be derived (Fig. 3). Conditional branching on the basis of equality at an equality kind can be performed in the type-level language (e.g. in **ifeq**, Fig. 5, discussed below). Equivalence at arrow kind is not decidable by 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 type-level function taking a type as an argument could "smuggle in" a self reference as a type index, extracting it via case analysis (continuing our analogy to open datatypes, this is closely related to the positivity condition for inductive datatypes in total functional languages like Coq).

$$\begin{array}{rcl}
\textbf{failure} & := & \mathsf{inr}[\mathsf{D}](()) \\
\textbf{ok} & := & (\lambda\mathbf{d}{:}\mathsf{D}.\mathsf{inl}[1](\mathbf{d})) \\
\textbf{ifeq} & := & (\lambda\mathbf{t1}{:}\mathsf{Ty}.\lambda\mathbf{t2}{:}\mathsf{Ty}.\lambda\mathbf{d}{:}\mathsf{D}. \\
 & & \quad \mathsf{if}\ \mathbf{t1} \equiv_{\mathsf{Ty}} \mathbf{t2}\ \mathsf{then}\ \mathbf{ok}\ \mathbf{d}\ \mathsf{else}\ \textbf{failure} \\
\textbf{arity0} & := & (\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}].\lambda\mathbf{d}{:}(\mathsf{D}+1).\mathsf{fold}(\mathbf{a}; \mathbf{d}; \_, \_, \_.\textbf{failure})) \\
\textbf{arity1} & := & (\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}].\lambda\mathbf{k}{:}\mathsf{D} \to (\mathsf{D}+1). \\
 & & \quad \mathsf{fold}(\mathbf{a}; \textbf{tyerr}; \mathbf{h}, \mathbf{t}, \_.\textbf{arity0}\ \mathbf{t}\ (\mathbf{k}\ \mathbf{h}))) \\
\textbf{arity2} & := & (\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}].\lambda\mathbf{k}{:}\mathsf{D} \to \mathsf{D} \to (\mathsf{D}+1). \\
 & & \quad \mathsf{fold}(\mathbf{a}; \textbf{tyerr}; \mathbf{h}, \mathbf{t}, \_.\textbf{arity1}\ \mathbf{t}\ (\mathbf{k}\ \mathbf{h}))) \\
\textbf{arity3} & := & (\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}].\lambda\mathbf{k}{:}\mathsf{D} \to \mathsf{D} \to \mathsf{D} \to (\mathsf{D}+1). \\
 & & \quad \mathsf{fold}(\mathbf{a}; \textbf{tyerr}; \mathbf{h}, \mathbf{t}, \_.\textbf{arity2}\ \mathbf{t}\ (\mathbf{k}\ \mathbf{h})))
\end{array}$$

Figure 5. Useful (type-level) functions for writing operator definitions.

$$\begin{array}{rcl}
\Phi_0 & := & \emptyset, \textsc{Arrow}\{\mathsf{Ty} \times \mathsf{Ty}; \tau_0; \theta_0\} \\
\tau_0 & := & \lambda\mathbf{i}{:}\mathsf{Ty} \times \mathsf{Ty}.\blacktriangleright(\mathsf{repof}(\mathsf{fst}(\mathbf{i})) \rightharpoonup \mathsf{repof}(\mathsf{snd}(\mathbf{i}))) \\
\theta_0 & := & \mathsf{opcon}\ \boldsymbol{ap}\ \mathsf{of}\ 1\ \{\lambda\mathbf{i}{:}1.\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}].\textbf{arity2}\ \mathbf{a}\ \lambda\mathbf{d1}{:}\mathsf{D}.\lambda\mathbf{d2}{:}\mathsf{D}. \\
 & & \quad \mathsf{case}\ \mathsf{typeof}(\mathbf{d1})\ \mathsf{of}\ \textsc{Arrow}\langle\mathbf{x}\rangle \Rightarrow \\
 & & \qquad \textbf{ifeq}\ \mathsf{fst}(\mathbf{x})\ \mathsf{typeof}(\mathbf{d2}) \\
 & & \qquad\quad [\![\mathsf{snd}(\mathbf{x}) \rightsquigarrow \mathsf{transof}(\mathbf{d1})\ \mathsf{transof}(\mathbf{d2})]\!] \\
 & & \quad \mathsf{ow}\ \textbf{tyerr}\}
\end{array}$$

Figure 6. The initial constructor context, Φ_0, defines the ARROW type constructor. The introductory form, $\lambda x{:}\tau.e$, is built into the language but the elimination form goes through the operator constructor ARROW.***ap***.

Every type constructor also defines a *schema*, a type-level function that associates with every type an internal *representation type*. We will return to this in Sec. 2.5 after introducing operators.

2.3 Operators and Operator Constructors

User-defined operator constructors are declared using opcon. For reasons that we will discuss, our calculus associates every operator constructor with a type constructor. The *fully-qualified name* of every operator constructor, e.g. NAT.***z***, must be unique. Operator constructors, like type constructors, declare an index kind, κ_{idx}. In our first example, all the operator constructors are indexed trivially (by index kind 1), but other examples use more interesting indices. For example, LABELEDTUPLE.***prj*** is the operator constructor used to access a field of a labeled tuple, so it has index kind Lbl. An operator itself is, notionally, selected by indexing an operator constructor, e.g. NAT.$\boldsymbol{s}\langle()\rangle$, but technically neither operator constructors nor operators are first-class at any level (additional machinery would be needed, e.g. an Op kind, but this is not fundamental to our calculus). Instead, in the external language, an operator constructor is applied by simultaneously providing an index and $n \geq 0$ *arguments*, written $\textsc{Tycon}.\boldsymbol{op}\langle\tau_{\text{idx}}\rangle(e_1; \ldots; e_n)$[2]. For example, on line 18 of Fig. 2, we see the operator constructors NAT.***z*** and NAT.***s*** being applied to compute two. In SML, the projection operator #3 can be applied to an n-tuple, e, iff $n \geq 3$. Note that it thus cannot be a function with a standard arrow type. Notionally, # is an operator constructor and 3 is its index. In an active embedding of n-tuples into @λ, this would be written $\textsc{Tuple}.\boldsymbol{prj}\langle 3\rangle(e)$ (we will nearly recover ML's syntax later).

2.4 Operator Definitions and Derivates

The expressive and metatheoretic power of the calculus arises from how the rules for the active typing judgement handle this operator constructor application form. Rather than fixing the specification of a finite collection of operator constructors and tasking the *compiler* with deciding a typing derivation on its basis, the specification instead tasks the *operator constructor's definition*: a type-level function that must decide the type assignment and the translation on the basis of the index and arguments, or decide that this is not possible. As indicated by rule K-OPCON in Fig. 3, an operator constructor with index kind κ_{idx} must have a definition of kind

$$\kappa_{\text{idx}} \to \mathsf{list}[\mathsf{D}] \to (\mathsf{D}+1)$$

As this suggests, the active typing judgement will provide the operator index and a list of recursively determined *derivates*, which have kind D, for the arguments and ask the definition to return a value of "option kind", $\mathsf{D}+1$, where the trivial case indicates that a derivate cannot be constructed due to an invalid index, an incorrect number of arguments or an argument of an invalid type.[3]

A *derivate* is introduced in operator definitions by the form $[\![\tau \rightsquigarrow \bar{\iota}]\!]$, where τ is a type assignment and has kind Ty and $\bar{\iota}$ is a *translational internal term*. The syntax for translational internal terms mirrors that of internal terms, ι, adding an additional form, $\mathsf{transof}(\tau_{\mathrm{D}})$, that permits one translation to be composed using the translations of other derivates (e.g. the derivate of an argument). Fig. 4 shows the kinding rules for derivates and translational internal terms (we will discuss checked derivates shortly; they are only used internally by the semantics). The type assignment can be extracted from a derivate using $\mathsf{typeof}(\tau_{\mathrm{D}})$.

Let us return to Fig. 2 to build intuition. The definition of NAT.***z*** in Fig. 2 is quite simple: it returns the derivate $[\![\textsc{Nat}\langle()\rangle \rightsquigarrow 0]\!]$ if no arguments were provided, and indicates an error otherwise. This is done by calling a simple helper function, **arity0**, seen in Fig. 5. The definition of NAT.***s*** is only slightly more complex, because it requires inspecting a single argument. The helper function **arity1** checks the arity of the argument list. If it is correct, it passes the single derivate to the shown continuation, returning the error case otherwise (which we give a more descriptive name, **failure**). The continuation we provided checks that the type of the argument is $\textsc{Nat}\langle()\rangle$ using a similar helper function, **ifeq**. If so, it constructs the derivate $[\![\textsc{Nat}\langle()\rangle \rightsquigarrow \mathsf{transof}(\mathbf{d}) + 1]\!]$ and **ifeq** wraps it with **ok**. We will return to the recursor shortly.

These two operator definitions make the following rules admissible in a constructor context, let us call it $\Phi_{\mathbf{T}}$, containing the initial constructor context (Fig. 6) and the constructors of Fig. 2 and any typing context well-kinded under that constructor context (the rules governing context kinding are in the appendix):

$$\textsc{GT-Z}\quad \frac{}{\Gamma \vdash_{\Phi_{\mathbf{T}}} \textsc{Nat}.\boldsymbol{z}\langle()\rangle() : \textsc{Nat}\langle()\rangle \Longrightarrow 0}$$

$$\textsc{GT-S}\quad \frac{\Gamma \vdash_{\Phi_{\mathbf{T}}} e : \textsc{Nat}\langle()\rangle \Longrightarrow \iota}{\Gamma \vdash_{\Phi_{\mathbf{T}}} \textsc{Nat}.\boldsymbol{s}\langle()\rangle(e) : \textsc{Nat}\langle()\rangle \Longrightarrow \iota + 1}$$

We will consider the complexities shortly, but it requires little more than basic intuition to see that we can strongly embed the types and introductory forms of Gödel's $\mathbf{T}$ in @λ quite directly (adding essentially only a barrage of trivial indices which will be removed syntactically in the expanded syntax). If we keep

[2] It may be helpful to distinguish between type/operator constructors and *term formers*, which are the metatheoretic entities that ultimately define the syntax. There are term formers at all levels in the calculus. For example, operator constructor application and λ are external term formers, and type constructor application is a type-level term former. An abstract syntax following Harper's conventions highlights this distinction [9]: the term formers `lam`, `tcapp` and `ocapp`, define, respectively the forms $\mathtt{lam}[\tau](x.e)$, $\mathtt{tcapp}[\textsc{Tycon}](\tau)$ and $\mathtt{ocapp}[\textsc{Tycon}, \boldsymbol{op}, \tau_{\text{idx}}](e_1; \ldots; e_n)$.

[3] In practice, we would require operator constructor providers to report information about the precise location of the error (e.g. which argument was of incorrect type) and provide an appropriate error message and other metadata, but we omit this in the semantics.

$\boxed{\vdash_\Phi \rho \Longrightarrow \iota}$

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

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

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

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

Figure 7. Active typing judgement for programs and external terms. The deabstraction ($\lightning$) judgement is in Fig. 10.

the constructor context fixed, we should be able to prove the embedding correct inductively. We will define this embedding later.

refer to it now

Figure 7 shows how Φ_{T} would be built up and how these conclusions could be derived. The active typing judgement simply defers to the *checked active typing judgement* (Fig. 8) to first produce a type assignment and a translational internal term, called the *abstract translation*. It then *deabstracts* the abstract translation to produce a type assignment and translation (Fig. 10).

The rule ATT-OP shows how operator constructor definitions are invoked. First, type assignments and translational internal terms are recursively derived. Then, the operator constructor definition, τ_{def} is extracted from the constructor context and applied to the index, τ_{idx}, and a list of *checked derivates* constructed from the derivations produced in the first step. If this produces a derivate, τ_{D}, via the left case of the sum, then it is itself, finally, *checked* to produce a type assignment, τ, and translational internal term, $\bar{\iota}$.

The normalization rules for type-level terms are shown in Fig. 12 and are rather uninteresting. We will discuss their metatheoretic properties in Sec. 3.

Derivate checking, shown in Fig. 9, is described in the next two subsections. Note that derivate checks are still static checks; they do not change the translation. Derivate checking is sufficient to guarantee type safety and preclude interfere issues between extensions, as we will describe, because taking a path through an operator constructor definition that produces a problematic derivate will fail. If providers do not want to burden clients with this variety of error, which would require them to debug the constructor definition itself, we will discuss how constructor correctness could be verified mechanically by strengthening the kind system in Sec. 6.

2.5 Representational Consistency Implies Type Safety

In our example, $\textsc{Nat}.\boldsymbol{z}$ produces a translation of internal type int. For the translation produced by $\textsc{Nat}.\boldsymbol{s}$ to be well-typed requires that $\mathsf{transof}(\mathbf{d1})$ be of internal type int. Because we have checked that $\mathsf{typeof}(\mathbf{d1})$ is $\textsc{Nat}\langle()\rangle$ and the only other introductory form is $\textsc{Nat}.\boldsymbol{z}$, this is not an issue. Well-typed external terms of type $\textsc{Nat}\langle()\rangle$ will always translate to well-typed internal terms of internal type int as given.

This would not be true if, for example, we added an operator constructor $\textsc{Nat}.\boldsymbol{z2}$ that produced the derivate

$$[\![\textsc{Nat}\langle()\rangle \rightsquigarrow (0, ())]\!]$$

In this case, there would be two different internal types, int and $\mathsf{int} \times 1$, associated with a single external type, $\textsc{Nat}\langle()\rangle$. This makes it impossible to reason *compositionally* about 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.

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

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

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

ATT-OP
$$\frac{\begin{array}{c}\Gamma \vdash_\Phi e_1 : \tau_1 \rightsquigarrow \bar{\iota}_1 \qquad \cdots \qquad \Gamma \vdash_\Phi e_n : \tau_n \rightsquigarrow \bar{\iota}_n \\ \textsc{Tycon}\{-;-;\theta\} \in \Phi \qquad \mathsf{opcon}\ \boldsymbol{op}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{\tau_{\text{def}}\} \in \theta \\ \tau_{\text{def}}\ \tau_{\text{idx}}\ ([\![\tau_1 \rightsquigarrow \bar{\iota}_1]\!]^{\checkmark} :: \cdots :: [\![\tau_n \rightsquigarrow \bar{\iota}_n]\!]^{\checkmark} :: []_{\text{D}}) \Downarrow \mathsf{inl}[1](\tau_{\text{D}}) \\ \Gamma \vdash_\Phi^{\textsc{Arrow},\textsc{Tycon}} \tau_{\text{D}} \checkmark \tau \rightsquigarrow \bar{\iota}\end{array}}{\Gamma \vdash_\Phi \textsc{Tycon}.\boldsymbol{op}\langle\tau_{\text{idx}}\rangle(e_1;\ldots;e_n) : \tau \rightsquigarrow \bar{\iota}}$$

Figure 8. The checked active typing judgement. The normalization judgement for type-level terms ($\Downarrow$) is specified in Fig. 12. Derivate checking is specified in Fig. 9. Note that checked derivates can *only* be constructed by the ATT-OP rule (we enforce this judgementally in Sec. 3).

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. This violates type safety: there are now well-typed external terms, according to the active typing judgement, for which evaluating the corresponding translation would "go wrong".

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 that the statics maintains 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 λ and operators like $\textsc{Nat}.\boldsymbol{s}$ are defined compositionally. It is closely related to the concept of *type-preserving compilation* developed by Morrisett et al. for the TIL compiler for SML [22].

For the semantics to ensure that representational consistency is maintained by all operator definitions, we require, as briefly mentioned in Sec. 2.2, that each type constructor must declare a *representation schema* after the keyword schema. This must be a type-level function of kind $\kappa_{\text{idx}} \rightarrow \mathsf{ITy}$, where κ_{idx} is the index kind of the type constructor. When the semantics (i.e. the compiler) needs to determine the representation type of the type $\textsc{Tycon}\langle\tau_{\text{idx}}\rangle$ it simply applies the representation schema of $\textsc{Tycon}$ to τ_{idx}.

The kind ITy has introductory form $\blacktriangleright(\bar{\sigma})$ and no elimination form. It is not an equality kind. There are two forms of *translational internal type*, $\bar{\sigma}$, that do not correspond to forms in σ that, as with translational internal terms, allow compositional formation:

1. $\blacktriangleleft(\tau)$ "unquotes" the translational internal type τ, so that internal types can be formed programmatically[4]

   example

2. $\mathsf{repof}(\tau)$ refers to the representation type of type τ. This type may have come from an argument derivate or extracted from a type or operator index.

These additional forms are not needed by the representation schema of $\textsc{Nat}$ because it is trivially indexed. In Fig. 11, we show an example of the type constructor $\textsc{Tuple}$, implementing the semantics of n-tuples by translation to nested binary products.

[4] An analagous form could be added to translational internal terms, but the same effect can be achieved by working directly with derivates, albeit with some inconvenience when "temporary intermediates" are needed.

add this example from appendix of ESOP

Here, the representation schema requires referring to the representations of the tuple's constituent types, given in the type index.

Translational internal terms also contain translational internal types. We see this in the definition of the recursor on natural numbers, NAT.***rec***. After checking the arity, extracting the derivates of its three arguments, and checking their types (using ARROW to bind variables in the third position), it produces a derivate that implements the necessarily recursive dynamic semantics using a fixpoint computation in the internal language. This derivate's type assignment is the arbitrary type, **t2**. We cannot know what the representation type of **t2** is (even in a closed constructor context, because the schema is not directly callable), so we refer to it abstractly using repof(**t2**). Luckily, the translation is well-typed *independent of the representation type*. In other words, a proof of representational consistency of the derivate produced by the definition of NAT.***rec*** is parametric (in a metamathematical sense here) over the representation type of **t2**. In fact, we will demand this for all types constructed by a type constructor other than the one an operator constructor is associated with. This leads us into the concept of conservativity.

2.6 Representational Independence Implies Conservativity

In isolation (let us call this language $@\lambda[\Phi_{\mathbf{T}}]$) our definition of natural numbers can be shown to be a strong encoding of the semantics of Gödel's $\mathbf{T}$. That is, there is a mapping of terms and types into $@\lambda[\Phi_{\mathbf{T}}]$ and an inverse mapping. Indeed, these are the entirely obvious one, given how directly we constructed $@\lambda[\Phi_{\mathbf{T}}]$.

Typing is preserved by these mappings, and we can show by a bisimulation argument that the translation produced by $@\lambda[\Phi_{\mathbf{T}}]$ implements the dynamic semantics of Gödel's $\mathbf{T}$. We must leave full details to an appendix, but the techniques are fundamentally standard when reasoning about compiler correctness, here lifted into the language. A necessary lemma in the proofs, however, is that the value of every translation of an external term of type $\textsc{Nat}\langle()\rangle$ 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 $\mathbf{T}$. A very similar lemma is needed to show that the translation of every external term of type $\textsc{Nat}\langle()\rangle$ is equivalent to the translation that would be produced by some combination of applications of NAT.***z*** and NAT.***s*** (corresponding to the canonical forms lemma of $\mathbf{T}$).

Fortunately, the definitions we have provided thus far admit this lemma. Indeed, the proof is quite straightforward. However, it is also quite precarious because it relies on exhaustive induction over the available operators. As soon as we load another operator or type constructor into the language, these theorems may no longer be *conserved*. We already saw an example where a representationally inconsistent operator caused problems, but the semantics render uses of it ill-typed using the representation schema (in a manner we will make more precise shortly). But there are operator constructors that are representationally consistent, but still cause problems because they violate internal invariants. For example,

$$\mathsf{opcon}\ \boldsymbol{n1}\ \mathsf{of}\ 1\ \{[\![\textsc{Nat}\langle()\rangle \rightsquigarrow -1]\!]\}$$

Were this operator constructor introduced inside NAT, there is no way for our semantics to distinguish between an incorrect implementation of NAT and a correct implementation of some other type where non-termination of the recursor was the intended behavior (without a separate specification).

But in any *other* type constructor, this clearly appears to violate the tacit, thus far, abstraction barrier between type constructors. In particular, if we extended $@\lambda[\Phi_{\mathbf{T}}]$ with such a type constructor, let us call it EVIL, then our crucial lemma relating to non-negativity would not be conserved, and thus, our carefully constructed strong encoding would no longer be a strong encoding. In particular, our canonical forms lemma now has a new case, and the recursor behaves badly. *If we assume an abstraction barrier between type constructors*[5] *must exist, we do not need need a specification to see that* EVIL.***n1*** *is violating it*. We can prevent problems by using a technique closely related to the one used to enforce abstraction barriers between modules – holding the representation type abstract. If EVIL.***n1*** cannot know that the representation type of $\textsc{Nat}\langle()\rangle$ is int, then its definition is incorrect independent of the specification of NAT.

$$\boxed{\Gamma \vdash^{\Xi}_{\Phi} \tau_{\mathrm{D}} \checkmark \tau \rightsquigarrow \bar{\iota}} \quad \Xi ::= \emptyset \mid \Xi, \textsc{Tycon}$$

D-CHECK

$$\frac{\vdash^{\Xi}_{\Phi} \mathsf{repof}(\tau) \looparrowright \hat{\sigma} \qquad \emptyset \vdash^{\Xi}_{\Phi} \bar{\iota} \sim \hat{\sigma}}{\Gamma \vdash^{\Xi}_{\Phi} [\![\tau \rightsquigarrow \bar{\iota}]\!] \checkmark \tau \rightsquigarrow \bar{\iota}}$$

D-CHECKED

$$\frac{}{\Gamma \vdash^{\Xi}_{\Phi} [\![\tau \rightsquigarrow \bar{\iota}]\!]^{\checkmark} \checkmark \tau \rightsquigarrow \bar{\iota}}$$

$$\boxed{\vdash^{\Xi}_{\Phi} \bar{\sigma} \looparrowright \hat{\sigma}} \quad \hat{\sigma} ::= \hat{\sigma} \rightharpoonup \hat{\sigma}' \mid \cdots \mid \mathsf{absrepof}(\tau)$$

ABS-TRITY-PARR

$$\frac{\vdash^{\Xi}_{\Phi} \bar{\sigma}_1 \looparrowright \hat{\sigma}_1 \qquad \vdash^{\Xi}_{\Phi} \bar{\sigma}_2 \looparrowright \hat{\sigma}_2}{\vdash^{\Xi}_{\Phi} \bar{\sigma}_1 \rightharpoonup \bar{\sigma}_2 \looparrowright \hat{\sigma}_1 \rightharpoonup \hat{\sigma}_2}$$

ABS-TRITY-CANCEL

$$\frac{\vdash^{\Xi}_{\Phi} \bar{\sigma} \looparrowright \hat{\sigma}}{\vdash^{\Xi}_{\Phi} \blacktriangleleft(\blacktriangleright(\bar{\sigma})) \looparrowright \hat{\sigma}}$$

ABS-TRITY-CONCRETE

$$\frac{\begin{array}{c}\textsc{Tycon} \in \Xi \qquad \textsc{Tycon}\{-; \tau_{\mathrm{rep}}; -\} \in \Phi \\ \tau_{\mathrm{rep}}\ \tau_{\mathrm{idx}} \Downarrow \blacktriangleright(\bar{\sigma}) \qquad \vdash^{\Xi}_{\Phi} \bar{\sigma} \looparrowright \hat{\sigma}\end{array}}{\vdash^{\Xi}_{\Phi} \mathsf{repof}(\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle) \looparrowright \hat{\sigma}}$$

ABS-TRITY-ABSTRACT

$$\frac{\textsc{Tycon} \notin \Xi}{\vdash^{\Xi}_{\Phi} \mathsf{repof}(\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle) \looparrowright \mathsf{absrepof}(\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle)}$$

(omitted rules follow recursively like ABS-IT-PARR)

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

ABS-I-VAR

$$\frac{}{\Psi, x \sim \bar{\sigma} \vdash^{\Xi}_{\Phi} x \sim \bar{\sigma}}$$

ABS-I-FIX

$$\frac{\vdash^{\Xi}_{\Phi} \bar{\sigma} \looparrowright \hat{\sigma} \qquad \Psi, x \sim \hat{\sigma} \vdash^{\Xi}_{\Phi} \bar{\iota} \sim \hat{\sigma}}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{fix}\ x{:}\bar{\sigma}\ \mathsf{is}\ \bar{\iota} \sim \bar{\sigma}'}$$

(omitted rules follow from a standard statics similarly)

ABS-TRANS-LOCAL-UNCHECKED

$$\frac{\textsc{Tycon} \in \Xi \qquad \vdash^{\Xi}_{\Phi} \mathsf{repof}(\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle) \looparrowright \hat{\sigma} \qquad \Psi \vdash^{\Xi}_{\Phi} \bar{\iota} \sim \hat{\sigma}}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{transof}([\![\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle \rightsquigarrow \bar{\iota}]\!]) \sim \hat{\sigma}}$$

ABS-TRANS-LOCAL-CHECKED

$$\frac{\textsc{Tycon} \in \Xi \qquad \vdash^{\Xi}_{\Phi} \mathsf{repof}(\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle) \looparrowright \hat{\sigma}}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{transof}([\![\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle \rightsquigarrow \bar{\iota}]\!]^{\checkmark}) \sim \hat{\sigma}}$$

ABS-TRANS-FOREIGN-CHECKED

$$\frac{\textsc{Tycon} \notin \Xi}{\Psi \vdash^{\Xi}_{\Phi} \mathsf{transof}([\![\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle \rightsquigarrow \bar{\iota}]\!]^{\checkmark}) \sim \mathsf{absrepof}(\textsc{Tycon}\langle\tau_{\mathrm{idx}}\rangle)}$$

Figure 9. Abstracted internal typing involves typechecking the internal language while holding the representation of all types other than those in the current fragment, listed in the fragment context, Ξ, abstract. They key rule is ABS-TRANS-FOREIGN-CHECKED, permitting the translation of a checked derivate that has a type assignment outside the fragment to be used against that type's abstract representation type.

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

$$\textsc{deabs-it-repof}\quad \frac{\textsc{Tycon}\{-;\tau_{\text{rep}};-\}\in\Phi \qquad \tau_{\text{rep}}\ \tau_{\text{idx}} \Downarrow \blacktriangleright(\bar\sigma) \qquad \vdash_\Phi \bar\sigma \lightning \sigma}{\vdash_\Phi \mathsf{repof}(\textsc{Tycon}\langle\tau_{\text{idx}}\rangle) \lightning \sigma}$$

$$\textsc{deabs-it-cancel}\quad \frac{\vdash_\Phi \bar\sigma \lightning \sigma}{\vdash_\Phi \blacktriangleleft(\blacktriangleright(\bar\sigma)) \lightning \sigma} \qquad \textsc{deabs-it-parr}\quad \frac{\vdash_\Phi \bar\sigma_1 \lightning \sigma_1 \qquad \vdash_\Phi \bar\sigma_2 \lightning \sigma_2}{\vdash_\Phi \bar\sigma_1 \rightharpoonup \bar\sigma_2 \lightning \sigma_1 \rightharpoonup \sigma_2}$$

(remaining forms follow recursively like DEABS-IT-PARR, omitted)

$$\boxed{\vdash_\Phi \bar\iota \lightning \iota}$$

$$\textsc{deabs-i-var}\quad \frac{}{\vdash_\Phi x \lightning x} \qquad \textsc{deabs-i-fix}\quad \frac{\vdash_\Phi \bar\sigma \lightning \sigma \qquad \vdash_\Phi \bar\iota \lightning \iota}{\vdash_\Phi \mathsf{fix}\ x{:}\bar\sigma\ \mathsf{is}\ \bar\iota \lightning \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \iota}$$

(remaining forms follow recursively like DEABS-I-FIX, omitted)

$$\textsc{deabs-trans-unchecked}\quad \frac{\vdash_\Phi \bar\iota \lightning \iota}{\vdash_\Phi \mathsf{transof}(\llbracket \tau \leadsto \bar\iota \rrbracket) \lightning \iota} \qquad \textsc{deabs-trans-checked}\quad \frac{\vdash_\Phi \bar\iota \lightning \iota}{\vdash_\Phi \mathsf{transof}(\llbracket \tau \leadsto \bar\iota \rrbracket^{\checkmark}) \lightning \iota}$$

Figure 10. Deabstraction simply recursively eliminates syntactic abstraction barriers and composition-related constructs, and is the final step.

3. Metatheory

3.1 Active Typing

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 and proving that the underlying semantics are otherwise well-founded.

4. Expanded Syntax

5. Examples

6. Limitations and Future Work

integrate this

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.

Because the input to an operator definition is a list of the recursively determined derivates of the argument in the same typing context as it 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 why the λ form needs to be built in. It is the only form in the external language that can be used to bind variables. The type constructor ARROW can, however, be defined from within the language and is included in the initial constructor context (it could be written out and included as a "prelude" as well). As seen in Fig. 6, it only defines the operator constructor ***ap***.

By writing the type assignment and translation logic in this way, as a type-level function, we are taking a rather "implementation-focused view" rather than attempting to extract a compiler from a declarative specification of a type system and an operational semantics. That is, @λ leaves ultimately to the provider the problem of finding a decidable type assignment algorithm and resolve non-determinism that may arise. Similarly, we do not attempt to turn an operational semantics into a compiler automatically. We also avoid difficulties with useful error reporting in practice. Simmons' discusses the difficulties of these issues [?]. Composing two dialect specifications directly is more fraught with metatheoretic issues than analyzing two functional programs used together. By looking at the problem as a functional programmer, we can apply intuitions about abstraction barriers more directly. A solution that permits implementations to be extracted from fragment specifications could easily be layered atop this mechanism and inherit these guarantees. Alternatively, extension correctness could be mechanized by encoding these semantics in a proof assistant like Coq and then using its extraction mechanisms to produce a compiler.

The contexts and external term are "input" to the judgement, while the type assignment and translation are "output" (we briefly discuss potential variants in Sec. 6).

remove if not

$$\mathsf{tycon}\ \textsc{Tuple}\ \mathsf{of}\ \mathsf{list}[\mathsf{Ty}]\ \{\mathsf{schema}\ (\lambda\mathbf{i}{:}\mathsf{list}[\mathsf{Ty}]. \tag{21}$$
$$\quad \mathsf{fold}(\mathbf{i}; \blacktriangleright(1); \mathbf{s}, \mathbf{j}, \mathbf{r}.\mathsf{fold}(\mathbf{j}; \blacktriangleright(\mathsf{repof}(\mathbf{s})); \_, \_, \_. \tag{22}$$
$$\quad\quad \blacktriangleright(\mathsf{repof}(\mathbf{s}) \times \blacktriangleleft(\mathbf{r}))))); \tag{23}$$
$$\mathsf{opcon}\ \boldsymbol{\mathit{new}}\ \mathsf{of}\ 1\ \{\lambda\_{:}1.\lambda\mathbf{a}{:}\mathsf{list}[\mathsf{D}]. \tag{24}$$
$$\quad \mathsf{fold}(\mathbf{a}; \llbracket \textsc{Tuple}\langle[]_{\mathsf{Ty}}\rangle \leadsto () \rrbracket; \mathbf{d}, \mathbf{b}, \mathbf{r}. \tag{25}$$
$$\quad\quad \mathsf{case}\ \mathsf{typeof}(\mathbf{r})\ \mathsf{of}\ \textsc{Ntuple}\langle\mathbf{i}\rangle \Rightarrow \tag{26}$$
$$\quad\quad\quad \mathsf{fold}(\mathbf{b}; \llbracket \textsc{Ntuple}\langle\mathsf{typeof}(\mathbf{d}) :: \mathbf{i}\rangle \leadsto \mathsf{transof}(\mathbf{d}) \rrbracket \tag{27}$$
$$\quad\quad\quad ; \_, \_, \_. \llbracket \textsc{Ntuple}\langle\mathsf{typeof}(\mathbf{d}) :: \mathbf{i}\rangle \leadsto \tag{28}$$
$$\quad\quad\quad\quad (\mathsf{transof}(\mathbf{d}), \mathsf{transof}(\mathbf{r})) \rrbracket)\ \mathsf{ow\ error})\}; \tag{29}$$
$$\mathsf{opcon}\ \boldsymbol{\mathit{prj}}\ \mathsf{of\ Int}\ \{\lambda\mathbf{i} : \mathsf{Int}.\lambda\mathbf{a} : \mathsf{list}[\mathsf{D}].\mathbf{arity1\ a}\ \lambda\mathbf{d}{:}\mathsf{D}. \tag{30}$$
$$\quad \mathsf{case}\ \mathsf{typeof}(\mathbf{d})\ \mathsf{of}\ \textsc{Ntuple}\langle\mathbf{nl}\rangle \Rightarrow \tag{31}$$
$$\quad \mathsf{fold}(\mathbf{nl}; \mathsf{error}; \mathbf{t1}, \mathbf{j}, \_. \tag{32}$$
$$\quad \mathsf{fold}(\mathbf{j}; \mathsf{if}\ \mathbf{i} \equiv_{\mathsf{int}} 1\ \mathsf{then}\ \llbracket \mathbf{t1} \leadsto \mathsf{transof}(d) \rrbracket\ \mathsf{else\ error} \tag{33}$$
$$\quad\quad ; \_, \_, \_. \tag{34}$$
$$\quad\quad (\lambda\mathbf{p}{:}\mathsf{D} \times \mathsf{int.if}\ \mathsf{snd}(\mathbf{p}) \equiv_{\mathsf{int}} \mathbf{i}\ \mathsf{then}\ \mathsf{fst}(\mathbf{p})\ \mathsf{else\ error}) \tag{35}$$
$$\quad\quad (\mathbf{foldl\ nl}\ (\llbracket \mathbf{nt} \leadsto \mathbf{x} \rrbracket, 0)\ \lambda\mathbf{r}{:}\mathsf{D} \times \mathsf{int}.\lambda\mathbf{t}{:}\mathsf{Ty}.\lambda\mathbf{ts}{:}\mathsf{list}[\mathsf{Ty}]. \tag{36}$$
$$\quad\quad\quad \mathsf{if}\ \mathbf{i} \equiv_{\mathsf{int}} \mathsf{snd}(\mathbf{r})\ \mathsf{then}\ \mathbf{r}\ \mathsf{else\ case}\ \mathsf{fst}(\mathbf{r})\ \mathsf{of}\ \llbracket \_ \leadsto \mathbf{rx} \rrbracket \Rightarrow \tag{37}$$
$$\quad\quad\quad\quad \mathsf{if}\ \mathbf{i} \equiv_{\mathsf{int}} \mathsf{snd}(\mathbf{r}) + 1\ \mathsf{then} \tag{38}$$
$$\quad\quad\quad\quad\quad \mathsf{if}\ \mathbf{ts} \equiv_{\mathsf{list[Ty]}} []_{\mathsf{Ty}}\ \mathsf{then}\ (\llbracket \mathbf{t} \leadsto \mathbf{rx} \rrbracket, \mathbf{i}) \tag{39}$$
$$\quad\quad\quad\quad\quad \mathsf{else}\ (\llbracket \mathbf{t} \leadsto \rhd(\mathsf{fst}(\lhd(\mathbf{rx}))) \rrbracket, \mathbf{i}) \tag{40}$$
$$\quad\quad\quad\quad \mathsf{else}\ (\llbracket \mathbf{t} \leadsto \rhd(\mathsf{snd}(\lhd(\mathbf{rx}))) \rrbracket, \mathsf{snd}(\mathbf{r}) + 1)\ \mathsf{ow\ error})))\mathsf{error\ ow}\ \} \tag{41}$$
$$\} \tag{42}$$

Figure 11. ABC

7. Related Work

Our representation schema abstraction mechanism relates closely to abstract and existential types [9?]. 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

[5] A fragment consisting of finitely many type constructors can be easily turned into just one by adding a sum type to the index and case analyzing against it whenever a distinction is necessary.

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 [3]). Ur uses type-level records and names to support typesafe metaprogramming, with applications to web programming [5]. Ωmega adds algebraic data types at the type-level, using these to increase the expressive power of algebraic data types at the expression level [21]. 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 [4], ATS [6]). We show how to integrate language extensions into the type-level language, drawing on ideas about [24].

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 [13], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [20]), 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 [7], 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 [**?**]).

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

N-T-LAM

$$\frac{}{\lambda\mathbf{t}{:}\kappa.\tau \Downarrow \lambda\mathbf{t}{:}\kappa.\tau}$$

N-T-AP

$$\frac{\tau_1 \Downarrow \lambda\mathbf{t}{:}\kappa.\tau \qquad \tau_2 \Downarrow \tau_2' \qquad [\tau_2'/\mathbf{t}]\tau \Downarrow \tau'}{\tau_1\ \tau_2 \Downarrow \tau'}$$

(normalization for integers, labels, lists, products and sums also standard)

N-T-EQ-EQUAL

$$\frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_1' \qquad \tau_3 \Downarrow \tau_3'}{\mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \Downarrow \tau_3'}$$

N-T-EQ-DISEQUAL

$$\frac{\begin{array}{cc}\tau_1 \Downarrow \tau_1' & \tau_2 \Downarrow \tau_2' \\ \tau_1' \neq \tau_2' & \tau_4 \Downarrow \tau_4'\end{array}}{\mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \Downarrow \tau_4'}$$

NT-TY

$$\frac{\tau_{\text{idx}} \Downarrow {\tau_{\text{idx}}}'}{\textsc{Tycon}\langle\tau_{\text{idx}}\rangle \Downarrow \textsc{Tycon}\langle{\tau_{\text{idx}}}'\rangle}$$

NT-TYCASE-MATCH

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

NT-TYCASE-FAIL

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

NT-D

$$\frac{\tau \Downarrow \tau' \qquad \bar{\iota} \curlyvee \bar{\iota}'}{[\![\tau \rightsquigarrow \bar{\iota}]\!] \Downarrow [\![\tau' \rightsquigarrow \bar{\iota}']\!]}$$

NT-D-CHECKED

$$\frac{\tau \Downarrow \tau' \qquad \bar{\iota} \curlyvee \bar{\iota}'}{[\![\tau \rightsquigarrow \bar{\iota}]\!]^{\checkmark} \Downarrow [\![\tau' \rightsquigarrow \bar{\iota}']\!]^{\checkmark}}$$

NT-TYPEOF

$$\frac{\tau \Downarrow [\![\tau' \rightsquigarrow \bar{\iota}]\!]}{\mathsf{typeof}(\tau) \Downarrow \tau'}$$

NT-TYPEOF-CHECKED

$$\frac{\tau \Downarrow [\![\tau' \rightsquigarrow \bar{\iota}]\!]^{\checkmark}}{\mathsf{typeof}(\tau) \Downarrow \tau'}$$

NT-REPTYPE

$$\frac{\bar{\sigma} \curlyvee \bar{\sigma}'}{\blacktriangleright(\bar{\sigma}) \Downarrow \blacktriangleright(\bar{\sigma}')}$$

$$\boxed{\bar{\iota} \curlyvee \bar{\iota}'}$$

NT-I-VAR

$$\frac{}{x \curlyvee x}$$

NT-I-FIX

$$\frac{\bar{\sigma} \curlyvee \bar{\sigma}' \qquad \bar{\iota} \curlyvee \bar{\iota}'}{\mathsf{fix}\ x{:}\bar{\sigma}\ \mathsf{is}\ \bar{\iota} \curlyvee \mathsf{fix}\ x{:}\bar{\sigma}'\ \mathsf{is}\ \bar{\iota}'}$$

NT-I-TRANSOF

$$\frac{\tau \Downarrow \tau'}{\mathsf{transof}(\tau) \curlyvee \mathsf{transof}(\tau')}$$

(omitted forms have similarly recursive rules)

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

NT-I-PARR

$$\frac{\bar{\sigma}_1 \curlyvee \bar{\sigma}_1' \qquad \bar{\sigma}_2 \curlyvee \bar{\sigma}_2'}{\bar{\sigma}_1 \rightharpoonup \bar{\sigma}_2 \curlyvee \bar{\sigma}_1' \rightharpoonup \bar{\sigma}_2'}$$

NT-I-UNQUOTE

$$\frac{\tau \Downarrow \tau'}{\blacktriangleleft(\tau) \curlyvee \blacktriangleleft(\tau')}$$

NT-I-REPOF

$$\frac{\tau \Downarrow \tau'}{\mathsf{repof}(\tau) \curlyvee \mathsf{repof}(\tau')}$$

(omitted forms have similarly recursive rules)

Figure 12. Normalization semantics for type-level terms, including those inside translational IL. There is nothing clever happening here.

References

[1] N. Benton and A. Kennedy. Interlanguage working without tears: Blending sml with java. In *ACM SIGPLAN Notices*, volume 34, pages 126–137. ACM, 1999.

[2] M. Chakravarty, R. Leshchinskiy, S. Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] J.-Y. Girard. Une extension de l'interpretation de gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. *Studies in Logic and the Foundations of Mathematics*, 63:63–92, 1971.

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

[10] R. Harper and C. Stone. A type-theoretic interpretation of standard ml. In *IN PROOF, LANGUAGE AND INTERACTION: ESSAYS IN HONOUR OF ROBIN MILNER*. MIT Press, 2000.

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

[12] L. Mandel and M. Pouzet. Reactiveml: a reactive extension to ml. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93. ACM, 2005.

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

[14] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 411–423, New York, NY, USA, 2014. ACM.

[15] T. Murphy, VII., K. Crary, and R. Harper. Type-safe distributed programming with ml5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC'07, pages 108–123, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] A. Ohori and K. Ueno. Making standard ml a practical database programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 307–319, New York, NY, USA, 2011. ACM.

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

[18] J. H. Reppy. Concurrent ml: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, 1993.

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

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

[21] 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.

[22] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[23] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. 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 111–121. ACM, 2010.

[24] 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.

[25] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. Specifying properties of concurrent computations in clf. *Electronic Notes in Theoretical Computer Science*, 199:67–87, 2008.