$@\lambda$: 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 $@\lambda$, 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 $@\lambda$ 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 $@\lambda$, 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 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 a technique similar to Church encodings [**?**]. While of substantial theoretical (and pedagogical) interest, this fact does not render irrelevant the construction of a language like $\mathcal{L}\{\rightarrow \forall\ \texttt{nat} + \times\}$. 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 the practicality of 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.

Finding a weak encoding of a language fragment in terms of a collection of others is useful if *implementing the dynamic semantics correctly* is one's main interest, but adding new fragments directly to a language can make statically reasoning about programs more precise, support a more natural programming style and endow the language with a more favorable cost semantics. Consistent with this view, typed programming languages like ML expose, for example, both n-ary product types and record types, despite the fact that any particular product or record type is weakly definable using polymorphic functions (or simply in terms of the other). The compiler for such a language, on the other hand, *is* mainly concerned with implementing the dynamic semantics correctly, and indeed, compilers often use the fact that weak definability results are quite readily derivable to their advantage, translating the constructs of their external language (EL) to those of a much simpler internal language (IL) after typechecking is complete.

Having established why a minimal language like $\mathcal{L}\{\rightarrow\ \forall\}$, while suitable as an internal language, needs to be extended with additional fragments before it is suitable for use as a human-facing external language, we might now ask whether modern "general-purpose" languages like ML are fundamentally different. These languages have been used successfully for a wide variety of computing tasks, admitting many more strong encodings than a calculus like $\mathcal{L}\{\rightarrow\ \forall\}$. It is certainly not our intention to argue against the notion that the particular fragments they have chosen to privilege occupy “sweet spots” in the design space. But in fact, situations continue to arise in both research and practice where the direct introduction of a new fragment is desirable because it can still only be weakly defined in terms of existing fragments, or a strong encoding is possible but 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, prototypic delegation or specified field ordering (that is, labeled tuples) are either awkward to strongly define 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 `printf` 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 a bounty of 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 language fragments are generally disseminated as distinct *dialects* of an existing general-purpose language, constructed either as a fork, a feature behind a compiler flag or, using tools like compiler generators, DSL frameworks or language workbenches. 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 language composition operations, so even fragmentary frameworks induce an unbounded, combinatoric collection of proof obligations that providers have little choice but to burden clients with.

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 attending to burdensome “link-time” proof obligations.

We draw inspiration from this approach to design a mechanism that makes it possible to extend the static and dynamic semantics of an external portion of the language with new type and operator constructors from within. The semantics of the language as a whole constrains these extensions to maintain type safety and decidability of type checking and type equality. Moreover, an important class of lemmas that play a key role in proofs that an extension enables strong encodings 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 language guarantees that they will be conserved in the “open world” by enforcing representation independence at extension boundaries.

Our mechanism essentially tasks the type-level language, in a controlled manner, with implementing portions of the logic that would normally be implemented in the front-end of the compiler. When typechecking a user-defined operator invocation, the semantics invoke a type-level function associated with the operator’s constructor to *compute* both a type and a translation to a fixed typed internal language (together called a *derivate*). Language fragments that can be weakly defined in terms of the typed internal language can be made strongly definable in the external language by introducing, in this way, the necessary constructors and static semantics. We call this an *actively typed semantics*.

We will begin in Sec. **??** by introducing a minimal calculus with an actively typed semantics, $@\lambda$. The external language builds in only the $\rightarrow$ fragment. We then use the `nat` fragment of Gödel’s $\mathbf{T}$ and a fragment defining n-ary products as pedagogical examples to explain how extensions are written and motivate the constraints that the semantics must impose to make the guarantees that we claim. We then examine the semantics in greater detail and state and sketch proofs of strong safety and conservativity theorems in Sec. **??**.

organization of rest of sections

2. $@\lambda$

In this section, we will develop an “actively typed” version of the simply-typed lambda calculus with simply-kinded type-level computation called $@\lambda$. More specifically, the level of types, τ, will itself form a simply-typed lambda calculus. *Kinds* classify type-level terms in the same way that types conventionally classify expressions. Types become just one kind of type-level value (which we will write Ty, though it is also variously written $\star$, `T` and `Type` in various settings). Rather than there being a fixed set of type constructors, we allow the programmer to declare new type constructors, and give the static and dynamic semantics of their associated operators, by writing type-level functions. In the semantics for this calculus, our kind system combined with techniques borrowed from the typed compilation literature and a form of type abstraction allow us to prove strong type safety, decidability and conservativity theorems.

The syntax of $@\lambda$ is given in Fig. 1. An example of a program containing an extension that implements the static and dynamic semantics of Gödel’s $\mathbf{T}$ is given in Fig. 2. We do not, of course, deny that natural numbers can be strongly encoded with a similar usage and asymptotic performance profile (but with potentially relevant additional function call overhead) by existing means (e.g. by an abstract type). We will provide more sophisticated examples where this is not the case below.

2.1 Programs

A program, ρ, consists of a series of static declarations followed by an external term, e. The syntax for external terms contains three core forms: variables, functions, 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

$$
\begin{array}{rlll}
\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}[\tau_{\text{idx}}](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 \mid \bar{\mathrm{z}} \mid \iota_1 \oplus \iota_2 \mid \mathsf{if}\ \iota_1 \equiv_{\mathbb{Z}} \iota_2\ \mathsf{then}\ \iota_3\ \mathsf{else}\ \iota_4 \\
 & & \mid & () \mid (\iota_1,\iota_2) \mid \mathsf{fst}(\iota) \mid \mathsf{snd}(\iota) \mid \mathsf{inl}[\sigma_2](\iota_1) \mid \mathsf{inr}[\sigma_1](\iota_2) \mid \mathsf{case}(e;x.e_1;x.e_2) \\
\textbf{internal types} & \sigma & ::= & \sigma_1 \rightharpoonup \sigma_2 \mid \mathbb{Z} \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 []_\kappa \mid \tau_1 :: \tau_2 \mid \mathsf{fold}(\tau_1;\tau_2;\mathbf{h},\mathbf{t},\mathbf{r}.\tau_3) \\
 & & \mid & \bar{\mathrm{z}} \mid \tau_1 \oplus \tau_2 \mid \textit{label} \mid () \mid (\tau_1,\tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau) \\
 & & \mid & \mathsf{inl}[\kappa_2](\tau_1) \mid \mathsf{inr}[\kappa_1](\tau_2) \mid \mathsf{case}(\tau;x.\tau_1;x.\tau_2) \\
\text{equality} & & \mid & \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \\
\text{types} & & \mid & \textsc{Tycon}[\tau] \mid \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Tycon}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
\text{derivates} & & \mid & \llbracket \tau_{\text{itm}}\ \mathsf{as}\ \tau_{\text{ty}} \rrbracket \mid \mathsf{let}\ \llbracket \mathbf{x}\ \mathsf{as}\ \mathbf{t} \rrbracket = \tau\ \mathsf{in}\ \tau_1 \\
\text{reified IL} & & \mid & \triangleright(\bar{\iota}) \mid \blacktriangleright(\bar{\sigma}) \\
 & \bar{\iota} & ::= & x \mid \mathsf{fix}\ x{:}\bar{\sigma}\ \mathsf{is}\ \bar{\iota} \mid \cdots \mid \triangleleft(\tau) \mid \mathsf{trans}(\llbracket \tau_{\text{itm}}\ \mathsf{as}\ \tau_{\text{ty}} \rrbracket) \\
 & \bar{\sigma} & ::= & \bar{\sigma}_1 \rightharpoonup \bar{\sigma}_2 \mid \cdots \mid \blacktriangleleft(\tau) \mid \mathsf{rep}(\tau) \\
 & & & \\
\textbf{kinds} & \kappa & ::= & \kappa_1 \to \kappa_2 \mid \mathsf{list}[\kappa] \mid \mathbb{Z} \mid \mathsf{Lbl} \mid 1 \mid \kappa_1 \times \kappa_2 \mid \kappa_1 + \kappa_2 \mid \mathsf{Ty} \mid \mathsf{D} \mid \mathsf{ITm} \mid \mathsf{ITy}
\end{array}
$$

Figure 1. Syntax of Core @λ. Here, x ranges over external and internal language variables, **t** ranges over type-level variables, $\textsc{Tycon}$ 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 the standard total binary operations over integers (e.g. addition). The form $\mathsf{trans}(\llbracket \tau_{\text{itm}}\ \mathsf{as}\ \tau_{\text{ty}} \rrbracket)$ is used internally by the semantics (it need not be supported by a parser).

$$
\begin{array}{ll}
\mathsf{tycon}\ \textsc{Nat}\ \mathsf{of}\ 1\ \{\mathsf{schema}\ \lambda\mathbf{idx}{:}1.\blacktriangleright(\mathbb{Z}); & (1) \\
\quad \mathsf{opcon}\ \boldsymbol{z}\ \mathsf{of}\ 1\ (\lambda\mathbf{idx}{:}1.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{D}].\mathbf{if\_empty\ args}\ \llbracket\triangleright(0)\ \mathsf{as}\ \textsc{Nat}[()]\rrbracket); & (2) \\
\quad \mathsf{opcon}\ \boldsymbol{s}\ \mathsf{of}\ 1\ (\lambda\mathbf{idx}{:}1.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{D}]. & (3) \\
\qquad \mathbf{pop\_final\ args}\ \lambda\mathbf{x}{:}\mathsf{ITm}.\lambda\mathbf{t}{:}\mathsf{Ty}. & (4) \\
\qquad \mathbf{check\_type\ t}\ \textsc{Nat}[()]\ \llbracket\triangleright(\triangleleft(\mathbf{x})+1)\ \mathsf{as}\ \textsc{Nat}[()]\rrbracket); & (5) \\
\quad \mathsf{opcon}\ \boldsymbol{rec}\ \mathsf{of}\ 1\ (\lambda\mathbf{idx}{:}1.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{D}]. & (6) \\
\qquad \mathbf{pop\ args}\ \lambda\mathbf{x1}{:}\mathsf{ITm}.\lambda\mathbf{t1}{:}\mathsf{Ty}.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{D}]. & (7) \\
\qquad \mathbf{pop\ args}\ \lambda\mathbf{x2}{:}\mathsf{ITm}.\lambda\mathbf{t2}{:}\mathsf{Ty}.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{D}]. & (8) \\
\qquad \mathbf{pop\_final\ args}\ \lambda\mathbf{x3}{:}\mathsf{ITm}.\lambda\mathbf{t3}{:}\mathsf{Ty}. & (9) \\
\qquad \mathbf{check\_type\ t1}\ \textsc{Nat}[()]\ (& (10) \\
\qquad \mathbf{check\_type\ t3}\ \textsc{Arrow}[(\textsc{Nat}[()],\textsc{Arrow}[(\mathbf{t2},\mathbf{t2})])] & (11) \\
\qquad \llbracket\triangleright((\mathsf{fix}\ f{:}\mathbb{Z}\rightharpoonup\mathsf{rep}(\mathbf{t2})\ \mathsf{is}\ \lambda x{:}\mathbb{Z}. & (12) \\
\qquad\quad \mathsf{if}\ x\equiv_{\mathbb{Z}} 0\ \mathsf{then}\ \triangleleft(\mathbf{x2})\ \mathsf{else}\ \triangleleft(\mathbf{x3})\ (x-1)\ (f\ (x-1)))\ \triangleleft(\mathbf{x1}))\ \mathsf{as}\ \mathbf{t2}\rrbracket)) & (13) \\
\}; & (14) \\
\mathsf{let}\ \mathbf{nat} : \mathsf{Ty} = \textsc{Nat}[()]\ \mathsf{in} & (15) \\
\mathsf{let}\ plus : \textsc{Arrow}[(\mathbf{nat},\textsc{Arrow}[(\mathbf{nat},\mathbf{nat})])] = \lambda x{:}\mathbf{nat}.\lambda y{:}\mathbf{nat}. & (16) \\
\quad \textsc{Nat}.\boldsymbol{rec}[()](x;y;\lambda p{:}\mathbf{nat}.\lambda r{:}\mathbf{nat}.\textsc{Nat}.\boldsymbol{s}[()](r))\ \mathsf{in} & (17) \\
\mathsf{let}\ two : \mathbf{nat} = \textsc{Nat}.\boldsymbol{s}[()](\textsc{Nat}.\boldsymbol{s}[()](\textsc{Nat}.\boldsymbol{z}[()]()))\ \mathsf{in} & (18) \\
\textsc{Arrow}.\boldsymbol{ap}[()](plus;\textsc{Arrow}.\boldsymbol{ap}[()](two;two)) & (19)
\end{array}
$$

Figure 2. Gödel's **T** in @λ, used to calculate 2+2. The simple helper functions **if_empty**, **pop**, **pop_final**, **check_type** all have return kind $\mathsf{D}+1$ (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 by manually performing the indicated substitutions (or added to the semantics).

type-level terms then *type checking* the external term and, simultaneously, *translating* it to a term, ι, in a *typed internal language*. In our calculus, the internal language contains 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 give a nod to practicality on contemporary machines). In practice, the internal language could be any intermediate language for which type safety and decidability of typechecking are known, as @λ requires these to establish the corresponding theorems about the language as a whole.

The *active typing judgement* relates 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 that the available type and operator constructors are not fixed by the language, but rather are tracked by the constructor context, Φ, and that a translation is simultaneously derived. The dynamic semantics of external terms are defined by their translation to the internal language. For this judgement, the two contexts and the external term can be thought of as input, while the type and translation are output (we do not consider a bidirectional approach in this work, though this may be a promising avenue for future research).

2.2 Types

User-defined type constructors are declared at the top-level of a program (or package, in a practical implementation) using tycon. Each type constructor in the program must have a unique name, written e.g. $\textsc{Nat}$. We do not intrinsically consider the issue of namespacing, under the assumption that globally unique names can be generated by some extrinsic mechanism (e.g. a URI-based scheme). A type constructor must also declare an *index kind*, κ_{idx}. Types themselves are type-level values of kind Ty and are introduced by applying a previously declared type constructor to a type-level term of its index kind, $\textsc{Tycon}[\tau_{\text{idx}}]$. The kind Ty also has an elimination form: a type can be case analyzed against a type constructor in scope to extract its index. Like open datatypes, there is no longer a notion of exhaustiveness. To maintain totality, we require the default case.

To permit the implementation of interesting type systems, the type-level language includes several other kinds of data alongside types. We lift several standard functional constructs 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 ($\mathbb{Z}$). We also include labels (Lbl), written in a slanted font, e.g. *myLabel*, which are atomic string-like values that can only be compared, here only for equality, 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, because there is only one natural number type, $\textsc{Nat}[()]$, but we will show examples of type constructors that are indexed in more interesting ways in later portions of this work. For example, TUPLE is indexed by a list of types and LABELEDTUPLE is indexed by a list pairing labels with types.

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 evaluation to normal form 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. We leave introducing a richer notion of equivalence of types that extension providers can extend as future work, as the metatheoretic guarantee that typing respects type equivalence would be quite a bit more complex in such a setting.

cite

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, $\kappa_1 \rightarrow \kappa_2$. Type constructor application can be wrapped in a type-level function to emulate first-class type constructors (and indeed, such a wrapper could be generated automatically, though we avoid this for simplicity in our semantics). Equivalence at arrow kind does not coincide with β-equivalence, so type-level functions cannot appear in type indices. 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. Indeed, one might imagine that a practical implementation of our calculus would start with a pure, total subset of the term language of ML to construct the type-level language (consistent with its early development as a "metalanguage" and, as we will see, the role of our type-level language as a DSL for implementing type systems and translators, which functional languages are widely considered well-suited for as-is). 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.

2.3 Operators

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 the 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, but later examples will use more interesting indices. For example, the projection operators for tuples and labeled tuples use numbers and labels as indices, respectively. Note that neither operator constructors nor operators are first-class type-level values (we plan to investigate lifting the latter restriction) and we do not impose any restrictions on their index kinds.

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*, $\textsc{Tycon}.\boldsymbol{op}[\tau_{\text{idx}}](e_1; \ldots; e_n)$. 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.[1] 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 active typing judgement[2]: an internal term, called the translation, paired with a type. Derivates have kind D and introductory form $[\![\tau_{\text{trans}}\ \mathsf{as}\ \tau_{\text{ty}}]\!]$. The elimination form $\mathsf{let}\ [\![\mathbf{x}\ \mathsf{as}\ \mathbf{t}]\!] = \tau\ \mathsf{in}\ \tau'$ 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 $\rhd(\bar{\iota})$ 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* $\lhd(\tau)$ 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.

The definition of $\textsc{Nat}.\boldsymbol{z}$ is quite simple: it returns the derivate $[\![\rhd(0)\ \mathsf{as}\ \textsc{Nat}[()]]\!]$ 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** : $\mathsf{list}[\mathsf{D}] \rightarrow \mathsf{D} \rightarrow (\mathsf{D}+1)$, 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** : $\mathsf{list}[\mathsf{D}] \rightarrow (\mathsf{Ty} \rightarrow \mathsf{ITm} \rightarrow (\mathsf{D}+1)) \rightarrow (\mathsf{D}+1)$ "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 $\textsc{Nat}[()]$ using **check_type** : $\mathsf{Ty} \rightarrow \mathsf{Ty} \rightarrow \mathsf{D} \rightarrow (\mathsf{D}+1)$, 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: $[\![\rhd(\lhd(\mathbf{x})+1)\ \mathsf{as}\ \textsc{Nat}[()]]\!]$. 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

[1] Although our focus here is entirely on semantics, a brief note on syntax: in the expanded syntax, the trivial indices and empty argument lists can be omitted, so we could write `Nat.s(Nat.s(Nat.z))`. With the ability to "open" a type's operators into the context, we could shorten this still to `s(s(z))`. Alternatively, with the ability to define a TSL in a manner similar to that in Sec. **??**, we might instead just write `2`.

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

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 ARROW can be defined from within the language (and is included in the "prelude" constructor context), but only defines the operator constructor, ***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 NAT.***z2*** that produced the derivate $[\![\triangleright(\mathsf{inl}[\mathbb{Z}](()))\ \mathsf{as}\ \mathrm{NAT}[()]]\!]$, then there would be two different internal types, $\mathbb{Z}$ and $1 + \mathbb{Z}$, associated with a single external type, NAT$[()]$. This makes it impossible to operate compositionally on the translation of an external term of type NAT$[()]$, so our implementation of NAT.***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$[()]$ is $\mathbb{Z}$. 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 $\kappa_{\text{idx}} \rightarrow \mathsf{ITy}$, where κ_{idx} is the index kind of the type constructor. When the compiler needs to determine the representation type of the type TYCON$[\tau_{\text{idx}}]$ 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. $\mathsf{rep}(\tau)$ 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.

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 $\mathsf{rep}(\mathbf{t2})$. 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.

add this example from appendix of ESOP

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 $\mathbf{T}$. 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 $@\lambda$ implements the dynamic semantics of Gödel's $\mathbf{T}$. 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$[()]$ 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 NAT$[()]$ 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

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.