Modularly Composing Typed Language Fragments

Abstract

Researchers often describe type systems as fragments or simple calculi, leaving to language designers the task of composing these to form complete programming languages. This is not a systematic process: metatheoretic results must be established anew for each composition, guided only notionally by metatheorems derived for simpler systems. As the language design space grows, mechanisms that provide stronger modular reasoning principles than this are needed.

In this paper, we begin from first principles with a core calculus, @λ, specified like many full-scale languages: as a bidirectionally typed translation semantics. Only the $\rightarrow$ type constructor (tycon) is built in; all other external tycons (we show a variant on record types and constrained string types) are defined by extending a *tycon context*. Each tycon defines the semantics of its associated term-level operators (e.g. record projection) using functions written in a static language where types and translations are values. The semantics provide strong metatheoretic guarantees, notably *type safety* and *conservativity*: that all *tycon invariants* will be conserved under extension. Mechanized proofs are not needed: problems are caught during typechecking by lifting typed compilation techniques into the semantics and enforcing barriers around tycons using type abstraction, the same principle that underlies reasoning with ML-style modules.

1. Introduction

Typed programming languages are most often described as being composed of *fragments*, each contributing to the language's concrete syntax, abstract syntax, static semantics and dynamic semantics. In his textbook, Harper organizes fragments around type constructors, describing each in a different chapter [12]. Languages are then identified by a set of type constructors, e.g. $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$ is the language that includes partial function types, polymorphic types, recursive types, nullary and binary product types and binary sum types (its syntax is shown in Figure 1, discussed below).

Another related approach is to describe fragments with a simple calculus having a "catch-all" constant and base type to stand notionally for all other terms and types that may also be included in some future complete language.

In contrast, the usual metatheoretic reasoning techniques for programming languages (e.g., rule induction) operate on complete language specifications. Each combination of fragments must formally be treated as its own monolithic language for which metatheorems must be established anew, guided only informally by those derived for the smaller systems from which the language is notionally composed.

This is not an everyday problem for programmers only because fragments like those mentioned above are "general purpose": they make it possible to *isomorphically embed* many other fragments as "libraries". For example, list types need not be built in because they are isomorphic to the type $\forall(\alpha.\mu(t.1 + (\alpha \times t))))$ (datatypes in ML combine these into a single declaration construct).

Universality properties (e.g. "Turing-completeness") can usually be invoked to guarantee that an embedding that preserves a desirable fragment's dynamics is possible, but an isomorphic embedding must also preserve the static semantics and, if defined, performance bounds specified using a cost semantics. This is not always possible. Embeddings are also sometimes too *complex*, as measured by the cost of the extralinguistic computations that are needed to map in and out of the embedding and, if these must be performed mentally by programmers, considering various human factors. Each time a fragment like this is needed, a new *dialect* of a language must be constructed. Within the ML lineage, for example, dialects abound:

1. **General Purpose Fragments:** A number of variations on product types, for example, have been introduced in dialects: n-ary tuples, labeled tuples, records (identified up to reordering), records with width and depth subtyping [6], records with update operators[1] [15], records with mutable fields [15], and records with "methods" (i.e. pure objects [22]). Sum-like types are also exposed in various ways: finite datatypes, open datatypes [16], hierarchically open datatypes [19], polymorphic variants [15] and ML-style exception types. Combinations of these manifest themselves as class-based object systems [15].

[1] The Haskell wiki notes that "No, extensible records are not implemented in GHC. The problem is that the record design space is large, and seems to lack local optima. [...] As a result, nothing much happens." [1]

2. **Specialized Fragments:** Fragments that track specialized static invariants to provide stronger correctness or security guarantees, manage unwieldy lower-level abstractions and run-time systems or control cost are also frequently introduced in dialects, e.g. for data parallelism [7], distributed programming [20], reactive programming [17], authenticated data structures [18], databases [21], units of measure [14] and regular string sanitation [10].

3. **Foreign Fragments:** A safe and natural foreign function interface (FFI) can be valuable (particularly given this proliferation of dialects). This requires enforcing the type system of the foreign language in the calling language. For example, MLj builds in a safe FFI to Java [3].

This *dialect-oriented* state of affairs is unsatisfying. While programmers can choose from dialects supporting, e.g., a principled approach to distributed programming, or one that builds in support for statically reasoning about units of measure, one that supports both fragments may not be available. Using different dialects separately for different components of a program is untenable: components written in different dialects cannot always interface safely (i.e. a safe FFI, item 3 above, is needed between every pair of dialects).

These problems do not arise for a fragment that can be expressed as an isomorphic embedding (i.e. as a library) because modern *module systems* can enforce abstraction barriers that ensure that the isomorphism need only be established in the "closed world" of the module. For example, a module defining sets in ML can hold the representation of sets abstract, ensuring that any invariants maintained by the functions in the module (e.g. uniqueness, if using a list representation) will hold no matter which other modules are
citation separately in use by a client.

When library-based embeddings are not possible, as in the examples above, mechanisms are needed that make it possible to define and reason in a similarly modular manner about direct extensions to the semantics of a language. Such a mechanism could ultimately be integrated directly into the language, blurring the distinction between fragments and libraries and decreasing the need for new dialects.

Contributions In this paper, we take foundational steps towards this goal by constructing a simple but surprisingly powerful core calculus, $@\lambda$ (the "actively typed" lambda calculus). Its semantics are structured like those of many modern languages, consisting of an *external language* (EL) governed by a typed translation semantics targeting a much simpler *internal language* (IL). Rather than building in a monolithic set of external type constructors, however, the semantics are indexed by a *tycon context*. Each tycon defines the semantics of its operators via functions written in a *static language* (SL) where types and translations are values.

We will begin by giving an overview of the organization and main judgements of $@\lambda$ in Sec. 2, then discuss how types are constructed and introduce our two main examples, one defining labeled product types with a functional update operator, and the other regular string types, based on a recent core calculus style specification [10], in Sec. 3. We describe how tycons control the semantics of their associated term-level operator constructors (opcons) in Sec. 4. We next give the key metatheoretic properties of the calculus in Sec. 5, including *type safety* and a key modularity result, which we call *conservativity*: any invariants that can be established about all values of a type under *some* tycon context (i.e. in some "closed world") are conserved in any further extended tycon context (i.e. in the "open world"). Interestingly, type system providers need not necessarily provide mechanized proofs to maintain these guarantees. Instead, the approach we take relies on type abstraction in the internal language. As a result, we are able to borrow the same results that underly modular reasoning in simply-typed languages like ML to reason modularly about language fragments. We conclude by discussing related and future work in Sec. 6.

2. Overview of $@\lambda$

External Language Programmers interface with $@\lambda$ by writing *external terms*, e. The abstract syntax of external terms is shown in Figure 2 (we will give various concrete desugarings in Sec. 4). The static and dynamic semantics are given simultaneously as a *bidirectionally typed translation semantics*, i.e. the key judgements take the form:

$$\Upsilon \vdash_\Phi e \Rightarrow \sigma^+ \rightsquigarrow \iota^+ \quad \text{and} \quad \Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota^+$$

These are pronounced "e (synthesizes / analyzes against) type σ and has translation ι under typing context Υ and tycon context Φ". Note that our specifications in this paper are intended to be algorithmic: we indicate "outputs" when introducing judgement forms by *mode annotations*, $^+$; these are not part of the judgement's syntax. In particular, note that the type is an "output" only in the synthetic judgement.

This basic separation of the EL and IL is commonly used for full-scale language specifications, e.g. the Harper-Stone semantics for Standard ML [13]. The internal language is purposely kept small, e.g. defining only simple products, to simplify metatheoretic reasoning and compilation. The EL then specifies various useful higher-level constructs, e.g. record types, by translation to the IL. In $@\lambda$, the EL builds in only function types. All other external constructs are defined in the *tycon context*, described in the Sec. 3.

We choose bidirectional typechecking, also sometimes called *local type inference* [23], for two main reasons. The first is once again to justify the practicality of our approach: local type inference is increasingly being used in modern languages (e.g. Scala) because it eliminates the need for type citation annotations in many situations while remaining decidable in more situations than whole-function type inference and providing what are widely perceived to be higher quality error messages. Secondly, it will give us a clean way to citation from Dunfield reuse the generalized introductory form, $\mathsf{intro}[\sigma](\overline{e})$, and its

internal types

$\tau \quad ::= \quad \tau \rightharpoonup \tau \mid \alpha \mid \forall(\alpha.\tau) \mid t \mid \mu(t.\tau) \mid 1 \mid \tau \times \tau \mid \tau + \tau$

internal terms

$\iota \quad ::= \quad x \mid \lambda[\tau](x.\iota) \mid \iota(\iota) \mid \mathsf{fix}[\tau](x.\iota) \mid \Lambda(\alpha.\iota) \mid \iota[\tau]$
$\quad\quad \mid \quad \mathsf{fold}[t.\tau](\iota) \mid \mathsf{unfold}(\iota) \mid () \mid (\iota, \iota) \mid \mathsf{fst}(\iota) \mid \mathsf{snd}(\iota)$
$\quad\quad \mid \quad \mathsf{inl}[\tau](\iota) \mid \mathsf{inr}[\tau](\iota) \mid \mathsf{case}(\iota; x.\iota; x.\iota)$

internal typing contexts $\Gamma ::= \emptyset \mid \Gamma, x : \tau$

internal type formation contexts $\Delta ::= \emptyset \mid \Delta, \alpha \mid \Delta, t$

Figure 1. Syntax of $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$, our internal language (IL). Metavariable x ranges over term variables and α and t both range over type variables.

external terms

$e \quad ::= \quad x \mid \lambda(x.e) \mid e(e) \mid \mathsf{fix}(x.e) \mid e : \sigma$
$\quad\quad \mid \quad \mathsf{intro}[\sigma](\overline{e}) \mid \mathsf{targ}[\mathbf{op}; \sigma](e; \overline{e})$

argument lists $\overline{e} ::= \cdot \mid \overline{e}, e$

external typing contexts $\Upsilon ::= \emptyset \mid \Upsilon, x \Rightarrow \sigma$

Figure 2. Syntax of the external language (EL).

kinds

$\kappa \quad ::= \quad \kappa \to \kappa \mid \boldsymbol{\alpha} \mid \forall(\boldsymbol{\alpha}.\kappa) \mid \boldsymbol{k} \mid \mu_{\text{ind}}(\boldsymbol{k}.\kappa) \mid 1 \mid \kappa \times \kappa \mid \kappa + \kappa$
$\quad\quad \mid \quad \mathsf{Ty} \mid \mathsf{ITy} \mid \mathsf{ITm}$

static terms

$\sigma \quad ::= \quad \boldsymbol{x} \mid \lambda \boldsymbol{x}{::}\kappa.\sigma \mid \sigma(\sigma) \mid \Lambda(\boldsymbol{\alpha}.\sigma) \mid \sigma[\kappa]$
$\quad\quad \mid \quad \mathsf{fold}[\boldsymbol{k}.\kappa](\sigma) \mid \mathsf{rec}[\kappa](\sigma; \boldsymbol{x}.\sigma)$
$\quad\quad \mid \quad () \mid (\sigma, \sigma) \mid \mathsf{fst}(\sigma) \mid \mathsf{snd}(\sigma)$
$\quad\quad \mid \quad \mathsf{inl}[\kappa](\sigma) \mid \mathsf{inr}[\kappa](\sigma) \mid \mathsf{case}(\sigma; \boldsymbol{x}.\sigma; \boldsymbol{x}.\sigma)$
$\quad\quad \mid \quad c\langle\sigma\rangle \mid \mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma; \sigma)$
$\quad\quad \mid \quad \blacktriangleright(\hat{\tau}) \mid \rhd(\hat{\iota}) \mid \mathsf{ana}[n](\sigma) \mid \mathsf{syn}[n] \mid \mathsf{raise}[\kappa]$

translational internal types and terms

$\hat{\tau} \quad ::= \quad \blacktriangleleft(\sigma) \mid \mathsf{trans}(\sigma) \mid \hat{\tau} \rightharpoonup \hat{\tau} \mid \ldots$

$\hat{\iota} \quad ::= \quad \lhd(\sigma) \mid \mathsf{anatrans}[n](\sigma) \mid \mathsf{syntrans}[n] \mid x \mid \lambda[\hat{\tau}](x.\hat{\iota}) \mid \ldots$

kinding contexts $\boldsymbol{\Gamma} ::= \emptyset \mid \boldsymbol{\Gamma}, \boldsymbol{x} :: \kappa$

kind formation contexts $\boldsymbol{\Delta} ::= \emptyset \mid \boldsymbol{\Delta}, \boldsymbol{\alpha} \mid \boldsymbol{\Delta}, \boldsymbol{k}$

argument environments $\mathcal{A} ::= \overline{e}; \Upsilon; \Phi$

Figure 3. Syntax of the static language (SL). Metavariable $\boldsymbol{x}$ ranges over static term variables, $\boldsymbol{\alpha}$ and $\boldsymbol{k}$ over kind variables and n over natural numbers.

associated desugarings, at many types. For example, regular string types will be able to use standard string literal syntax.

Unlike the Harper-Stone semantics, where external and internal terms were governed by a common type system, in @λ each external type, σ, maps onto an internal type, τ, called the *type translation* of σ. This mapping is specified by the type translation judgement, $\vdash_\Phi \sigma \leadsto \tau$, which will be described in Sec. 3.4. For this reason, this specification style may also be compared to specifications for the first stage of a type-directed compiler, e.g. the TIL compiler for Standard ML [27], here lifted "one level up" into the semantics of the language itself. As we will see, type safety follows from a property analagous to a correctness condition that arises in typed compilers. Modular reasoning will be based on holding the type translation of σ abstract "outside" the tycon.

External typing contexts Υ map variables to types, so the type translation judgement induces judgement $\vdash_\Phi \Upsilon \leadsto \Gamma$.

Internal Language @λ relies on a typed internal language supporting type abstraction (i.e. universal quantification over types). We use $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$, the syntax for which is shown in Figure 1, as representative of any intermediate language for a typed functional language.

We assume the statics of the IL are specified in the standard way by judgements for type formation $\Delta \vdash \tau$, typing context formation $\Delta \vdash \Gamma$ and type assignment $\Delta\ \Gamma \vdash \iota : \tau^+$. In examples, we will omit leading $\emptyset$, used as the base case for finite mappings, and $\cdot$, used as the base case for finite sequences (e.g. writing $\Gamma_{\text{test}} := x : \tau$).

The internal dynamics are specified as a structural operational semantics with a stepping judgement $\iota \mapsto \iota^+$ and a value judgement $\iota\ \mathtt{val}$. The multi-step judgement $\iota \mapsto^* \iota^+$ is the reflexive, transitive closure of the stepping judgement and the evaluation judgement $\iota \Downarrow \iota'$ is defined iff $\iota \mapsto^* \iota'$ and $\iota'\ \mathtt{val}$. Both the static and dynamic semantics of the IL can be found in any standard textbook covering typed lambda calculi (we directly follow [12]), so we assume familiarity and key metatheoretic properties.

Static Language The workhorse of @λ is the *static language*, which itself forms a typed lambda calculus where *kinds*, κ, classify *static terms*, σ. The syntax of the SL is given in Figure 3. The portion of the SL covered by the first row of kinds and first four rows of static terms in the syntax forms an entirely standard functional programming language consisting of total functions, universal quantification over kinds, inductive kinds (constrained by the standard positivity condition to prevent non-termination), and products and sums. The reader can consider these as forming a total subset of ML or a simply-typed subset of Coq. The semantics also directly follow [12], so we omit the details of this core here. Only three new kinds are needed for the SL to serve its role as the language used to statically compute over types and type and term translations: Ty (Sec. 3), ITy (Sec. 3.4) and ITm (Sec. 4.1).

The kinding judgement takes the form $\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash_\Phi^n \sigma :: \kappa^+$, where $\boldsymbol{\Delta}$ and $\boldsymbol{\Gamma}$ are analagous to Δ and Γ and analagous kind and kinding context formation judgements $\boldsymbol{\Delta} \vdash \kappa$ and $\boldsymbol{\Delta} \vdash \boldsymbol{\Gamma}$ are defined. The natural number n is used as a technical device in our semantics to ensure that the forms shown as being indexed by n in the syntax only arise in a controlled manner internally to prevent "out of bounds" issues, as we will discuss; they would have no corresponding concrete syntax so n can be assumed 0 in user-defined terms.

The dynamic semantics of static terms is defined as a structural operational semantics by a stepping judgement $\sigma \mapsto_\mathcal{A} \sigma^+$, a value judgement $\sigma\ \mathtt{val}_\mathcal{A}$ and an error judgement $\sigma\ \mathtt{err}_\mathcal{A}$. Here, $\mathcal{A}$ ranges over *argument environments*, which we will return to when considering opcons in Sec. 4. The multi-step judgement $\sigma \mapsto_\mathcal{A}^* \sigma^+$ is the reflexive, transitive closure of the stepping judgement. The normalization judgement $\sigma \Downarrow_\mathcal{A} \sigma'$ is defined iff $\sigma \mapsto_\mathcal{A}^* \sigma'$ and $\sigma'\ \mathtt{val}_\mathcal{A}$.

$$\frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty} \times \mathsf{Ty}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \rightharpoonup\langle\sigma\rangle :: \mathsf{Ty}}\ \text{(k-parr)} \qquad \frac{\mathsf{tycon}\ \textsc{tc}\ \{\theta\} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\} \in \Phi \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa_{\text{tyidx}}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \textsc{tc}\langle\sigma\rangle :: \mathsf{Ty}}\ \text{(k-ty)} \qquad \frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma_{\text{tyidx}} :: \mathsf{Nat} \times \mathsf{ITy}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{other}[m]\langle\sigma_{\text{tyidx}}\rangle :: \mathsf{Ty}}\ \text{(k-otherty)}$$

Figure 5. Kinding rules for types, which take the form $c\langle\sigma_{\text{tyidx}}\rangle$ where c is a tycon and σ_{tyidx} is the type index.

$$\begin{array}{llcl}
\textbf{tycons} & c & ::= & \rightharpoonup \mid \textsc{tc} \mid \mathsf{other}[m] \\
\textbf{tycon contexts} & \Phi & ::= & \cdot \mid \Phi, \mathsf{tycon}\ \textsc{tc}\ \{\theta\} \sim \psi \\
\textbf{tycon structures} & \theta & ::= & \mathsf{trans} = \sigma\ \mathsf{in}\ \omega \\
\textbf{opcon structures} & \omega & ::= & \mathsf{ana\ intro} = \sigma \mid \omega; \mathsf{syn}\ \mathbf{op} = \sigma \\
\textbf{tycon sigs} & \psi & ::= & \mathsf{tcsig}[\kappa]\ \{\chi\} \\
\textbf{opcon sigs} & \chi & ::= & \mathsf{intro}[\kappa] \mid \theta; \mathbf{op}[\kappa]
\end{array}$$

Figure 4. Syntax of tycons. Metavariables $\textsc{tc}$ and **op** range over user-defined tycon and opcon names, respectively, and m ranges over natural numbers.

3. Types

External types, or simply *types*, are static values of kind Ty. The introductory form for kind Ty is $c\langle\sigma\rangle$, where c is a *tycon* and σ is the *type index*. The syntax for tycons given in Figure 4 specifies that c is either the built-in tycon governing partial functions, $\rightharpoonup$, a user-defined tycon name written in small caps, $\textsc{tc}$, or an "other" tycon, $\mathsf{other}[m]$, indexed by a natural number m to allow arbitrarily many such tycons. The kinding rules governing the form $c\langle\sigma\rangle$ are shown in Figure 5. The dynamics are simple (see supplement): the index is eagerly normalized and errors propagate. We write $\sigma\ \mathtt{type}_\Phi$ iff $\emptyset\ \emptyset \vdash^0_\Phi \sigma :: \mathsf{Ty}$ and $\sigma\ \mathtt{val}_{\cdot;\emptyset;\Phi}$.

The rule (k-parr) specifies that the type index of partial function types must be a pair of types. We thus say that $\rightharpoonup$ has *index kind* $\mathsf{Ty} \times \mathsf{Ty}$. To recover a conventional syntax, we can introduce a desugaring from $\sigma_1 \rightharpoonup \sigma_2$ to $\rightharpoonup\langle(\sigma_1, \sigma_2)\rangle$.

All user-defined tycons must be defined in the *tycon context*, Φ, which is simply a list of tycon definitions. Each tycon defines a name, $\textsc{tc}$, a *tycon structure*, θ, and a *tycon signature*, ψ. We will return to the tycon structure below. Tycon signatures have the form $\mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\}$, where κ_{tyidx} is the tycon's index kind and χ is the *opcon signature*, which we discuss in Sec. 4. The first premise of (k-ty) extracts the index kind and the second checks the type index against it.

The rule (k-otherty) governs types constructed by an "other" tycon. These will serve only as technical devices to stand in for types other than those in a given tycon context in Sec. 5. The index of such a type must pair a natural number with a static value of kind ITy, discussed in Sec. 3.4.

Examples Our first example of a user-defined tycon is $\textsc{rstr}$. It has index kind Rx, which classifies static regular expression patterns (defined as an inductive sum kind in the usual way). Types constructed by $\textsc{rstr}$ will classify *regular strings*, which are statically known to be in the regular language specified by the type index [10]. For example, $\sigma_{\text{title}} := \textsc{rstr}\langle\mathtt{/.+/}\rangle$ will classify non-empty strings and $\sigma_{\text{conf}} := \textsc{rstr}\langle\mathtt{/[A\text{-}Z]+\ \backslash d\backslash d\backslash d\backslash d/}\rangle$ will classify conference names. The type indices are here written using standard concrete syntax for concision; recent work has specified how to define type-specific (or here, kind-specific) syntax like this composably in libraries [22]. We define a tycon context containing only the definition of $\textsc{rstr}$, $\Phi_{\text{rstr}} := \mathsf{tycon}\ \textsc{rstr}\ \{\theta_{\text{rstr}}\} \sim \mathsf{tcsig}[\mathsf{Rx}]\ \{\chi_{\text{rstr}}\}$.

Our second example is the tycon $\textsc{lprod}$, which will define a variant of labeled product type (labeled products are like record types, but maintain a row ordering; record types are also definable in a manner discussed in the supplement, but maintaining an ordering simplifies our discussion). We choose the index kind of $\textsc{lprod}$ to be $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$, where list kinds are defined as inductive sums in the usual way, and Lbl classifies static representations of row labels. The tycon context containing only $\textsc{lprod}$'s definition is $\Phi_{\text{lprod}} := \mathsf{tycon}\ \textsc{lprod}\ \{\theta_{\text{lprod}}\} \sim \mathsf{tcsig}[\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]]\ \{\chi_{\text{lprod}}\}$. In the tycon context containing both tycon definitions, $\Phi_{\text{rstr}}\Phi_{\text{lprod}}$, we can define a labeled product type classifying conference papers, $\sigma_{\text{paper}} := \textsc{lprod}\langle\{\mathtt{title} : \sigma_{\text{title}}, \mathtt{conf} : \sigma_{\text{conf}}\}\rangle$. Note that $\sigma_{\text{paper}}\ \mathtt{type}_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}}$ and we again use kind-specific syntax, in this case for Lbl and $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$.

$$\boxed{\vdash \Phi}$$

$$\frac{\vdash \Phi \qquad \textsc{tc} \notin \mathrm{dom}(\Phi) \qquad \emptyset \vdash \kappa_{\text{tyidx}}\ \mathsf{eq} \qquad \emptyset\ \emptyset \vdash^0_\Phi \sigma_{\text{schema}} :: \kappa_{\text{tyidx}} \to \mathsf{ITy} \qquad \vdash_{\Phi, \mathsf{tycon}\ \textsc{tc}\ \{\mathsf{trans}=\sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\}} \omega \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\}}{\vdash \Phi, \mathsf{tycon}\ \textsc{tc}\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\}}\ \text{(tcc-ext)}$$

Figure 6. Tycon context well-definedness. We omit the trivial case for when $\Phi = \cdot$ for concision.

3.1 Static Type Case Analysis

Types in $@\lambda$ can be thought of as arising from a distinguished "open datatype" defined by the tycon context [16]. Consistent with this view, a type σ can be case analyzed using $\mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2)$. If the value of σ is constructed by c, its type index is bound to $\boldsymbol{x}$ and the branch σ_1 is taken. For totality, a default branch, σ_2, must also be provided. For example, the kinding rule for when c is user-defined is below.

$$\frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty} \qquad \mathsf{tycon}\ \textsc{tc}\ \{\theta\} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\} \in \Phi \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma}, \boldsymbol{x} :: \kappa_{\text{tyidx}} \vdash^n_\Phi \sigma_1 :: \kappa \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma_2 :: \kappa}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{tycase}[\textsc{tc}](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) :: \kappa}\ \text{(k-tycase)}$$

The rule for $c{=}\rightharpoonup$ is analagous, but no rule for $c{=}\mathsf{other}[m]$ is defined (such types always take the default branch).

The dynamics (see supplement) are straightforwardly consistent with the intuition above.

3.2 Tycon Context Well-Definedness

The tycon context well-definedness judgement, $\vdash \Phi$, shown in Figure 6, requires that all tycon names are unique and performs additional checks, described below.

3.3 Type Equivalence

The first check simplifies the handling of type equivalence: type index kinds must be *equality kinds*, i.e. those for which semantic equivalence implies syntactic equality at normal form. We define these by the judgement $\boldsymbol{\Delta} \vdash \kappa \; \texttt{eq}$ (see supplement). Equality kinds are similar to equality types as found in Standard ML. The main implication of this choice is that type indices cannot contain static functions.

3.4 Type Translations

Recall that every type σ must have a type translation, τ. Each tycon in the tycon context computes translations for the types it constructs as a function of each type's index by specifying a *translation schema* in the tycon structure, θ. A tycon with index kind κ_{tyidx} defines a translation schema of kind $\kappa_{\text{tyidx}} \to \mathsf{ITy}$, checked by (tcc-ext).

The kind ITy has a single introductory form, $\blacktriangleright(\hat{\tau})$, where $\hat{\tau}$ is a *translational internal type*. Each form in the syntax for internal types, τ, corresponds to a form in the syntax of translational internal types, $\hat{\tau}$. For example, our translation schema for RSTR simply chooses to ignore the type index and represent all regular strings internally as strings, of internal type abbreviated str. We abbreviate the corresponding translational internal type $\hat{\mathsf{str}}$ and define the translation schema as $\theta_{\text{rstr}} := \mathsf{trans} = \lambda \boldsymbol{tyidx}{::}\mathsf{Rx}.\blacktriangleright(\hat{\mathsf{str}})$ in ω_{rstr}.

The kinding and dynamics for these shared forms simply proceed recursively, e.g.

$$\text{(k-ity-prod)}\quad \frac{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \blacktriangleright(\hat{\tau}_1) :: \mathsf{ITy} \qquad \boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \blacktriangleright(\hat{\tau}_2) :: \mathsf{ITy}}{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2) :: \mathsf{ITy}}$$

The syntax for translational internal types additionally includes an "unquote" form, $\blacktriangleleft(\sigma)$, so that they can be constructed compositionally, as well as a form, $\mathsf{trans}(\sigma)$, that allows one type's translation to refer to another:

$$\text{(k-ity-unquote)}\quad \frac{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{ITy}}{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \blacktriangleright(\blacktriangleleft(\sigma)) :: \mathsf{ITy}} \qquad \text{(k-ity-trans)}\quad \frac{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty}}{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \blacktriangleright(\mathsf{trans}(\sigma)) :: \mathsf{ITy}}$$

The unquote form is eliminated during normalization, while references to the translation of a type are retained in values of kind ITy. The key rules in the dynamics are:

$$\text{(s-ity-unquote-elim)}\quad \frac{\blacktriangleright(\hat{\tau}) \; \texttt{val}_{\mathcal{A}}}{\blacktriangleright(\blacktriangleleft(\blacktriangleright(\hat{\tau}))) \mapsto_{\mathcal{A}} \blacktriangleright(\hat{\tau})} \qquad \text{(s-ity-trans-val)}\quad \frac{\sigma \; \texttt{val}_{\mathcal{A}}}{\blacktriangleright(\mathsf{trans}(\sigma)) \; \texttt{val}_{\mathcal{A}}}$$

We choose a translation schema for LPROD that generates nested binary product types by recursing over the type index and referring to the translations of the types therein (though this is not the only workable choice, e.g. we could also have used a list). We assume $\boldsymbol{listrec} :: \forall(\boldsymbol{\alpha}_1.\forall(\boldsymbol{\alpha}_2.\mathsf{List}[\boldsymbol{\alpha}_1] \to \boldsymbol{\alpha}_2 \to (\boldsymbol{\alpha}_1 \to \boldsymbol{\alpha}_2 \to \boldsymbol{\alpha}_2) \to \boldsymbol{\alpha}_2))$ in defining $\theta_{\text{lprod}} := \mathsf{trans} = \sigma_{\text{lprod/trans}}$ in ω_{lprod} where $\sigma_{\text{lprod/trans}} :=$

$$\begin{array}{l}\lambda \boldsymbol{tyidx}{::}\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}].\boldsymbol{listrec}\,[\mathsf{Lbl} \times \mathsf{Ty}]\,[\mathsf{ITy}]\;\boldsymbol{tyidx} \blacktriangleright(1) \\ \quad (\lambda \boldsymbol{h}{:}\mathsf{Lbl} \times \mathsf{Ty}.\lambda \boldsymbol{r}{:}\mathsf{ITy}.\blacktriangleright(\mathsf{trans}(\mathsf{snd}(\boldsymbol{h})) \times \blacktriangleleft(\boldsymbol{r}))\end{array}$$

Evaluating this translation schema with the index of σ_{paper}, for example, produces the value $\sigma_{\text{paper/trans}} := \blacktriangleright(\hat{\tau}_{\text{paper/trans}})$ where $\hat{\tau}_{\text{paper/trans}} := \mathsf{trans}(\sigma_{\text{title}}) \times (\mathsf{trans}(\sigma_{\text{conf}}) \times 1)$. Note that we do not include logic to optimize away the trailing unit type for simplicity (and to again emphasize that many translations are possible for any given type).

3.4.1 Selective Type Translation Abstraction

References to translations of other types are maintained in values of kind ITy like this to allow us to selectively hold them abstract, which will be the key to our main results. This can be thought of as analagous to the process in ML by which the true identity of an abstract type in a module is held abstract outside the module until after typechecking. The judgement $\hat{\tau} \parallel \mathcal{D} \hookrightarrow^c_\Phi \tau^+ \parallel \mathcal{D}^+$ relates a normalized translational internal type $\hat{\tau}$ to an internal type τ, called a *selectively abstracted type translation* because references to translations of types constructed by a tycon other than the "delegated tycon", c, are replaced by a corresponding type variables, α. The *type translation store* $\mathcal{D} ::= \emptyset \mid \mathcal{D}, \sigma \leftrightarrow \tau/\alpha$ maintains the corresponding between types, their actual translations and the unique type variables which appear in their place. For example, $\hat{\tau}_{\text{paper/trans}} \parallel \emptyset \hookrightarrow^{\text{LPROD}}_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \tau_{\text{paper/abs}} \parallel \mathcal{D}_{\text{paper/abs}}$ where $\tau_{\text{paper/abs}} := \alpha_1 \times (\alpha_2 \times 1)$ and $\mathcal{D}_{\text{paper/abs}} := \sigma_{\text{title}} \leftrightarrow \mathsf{str}/\alpha_1, \sigma_{\text{conf}} \leftrightarrow \mathsf{str}/\alpha_2$.

Each type translation store induces a *type substitution*, δ, and a corresponding internal type formation context, Δ, according to the judgement $\mathcal{D} \rightsquigarrow \delta : \Delta$. Type substitutions simply define an n-ary substitution for type variables, $\delta ::= \emptyset \mid \delta, \tau/\alpha$. For example, $\mathcal{D}_{\text{paper/abs}} \rightsquigarrow \delta_{\text{paper/abs}} : \Delta_{\text{paper/abs}}$ where $\delta_{\text{paper/abs}} := \mathsf{str}/\alpha_1, \mathsf{str}/\alpha_2$ and $\Delta_{\text{paper/abs}} := \alpha_1, \alpha_2$. We can apply type substitutions to internal types, terms and typing contexts, written $[\delta]\tau$, $[\delta]\iota$ and $[\delta]\Gamma$, respectively. For example, $[\delta_{\text{paper/abs}}]\tau_{\text{paper/abs}}$ is $\tau_{\text{paper}} := \mathsf{str} \times (\mathsf{str} \times 1)$, i.e. the actual type translation of σ_{paper}. Indeed, we can now give the rule defining the type translation judgement, $\vdash_\Phi \sigma \rightsquigarrow \tau$, mentioned in Sec. 2. We simply determine any selectively abstract translation, then apply the induced substitution:

$$\text{(conc-ty-trans)}\quad \frac{\sigma \; \mathsf{type}_\Phi \qquad \mathsf{trans}(\sigma) \parallel \emptyset \hookrightarrow^{\text{TC}}_\Phi \tau \parallel \mathcal{D} \qquad \mathcal{D} \rightsquigarrow \delta : \Delta}{\vdash_\Phi \sigma \rightsquigarrow [\delta]\tau}$$

The rules for the selective type abstraction judgement are straightforward. We first recurse generically over sub-terms of $\hat{\tau}$ until sub-terms of form $\mathsf{trans}(\sigma)$ are encountered (see supplement). The translation of partial function types is direct and is not held abstract, so that lambdas can be used as the sole binding construct in the EL:

(abs-parr)

$$\frac{\mathsf{trans}(\sigma_1) \parallel \mathcal{D} \hookrightarrow^{c}_{\Phi} \tau_1 \parallel \mathcal{D}' \qquad \mathsf{trans}(\sigma_2) \parallel \mathcal{D}' \hookrightarrow^{c}_{\Phi} \tau_2 \parallel \mathcal{D}''}{\mathsf{trans}(\rightharpoonup\langle(\sigma_1, \sigma_2)\rangle) \parallel \mathcal{D} \hookrightarrow^{c}_{\Phi} \tau_1 \rightharpoonup \tau_2 \parallel \mathcal{D}''}$$

The translation of a user-defined type constructed by the delegated tycon is determined by calling the translation schema and checking that the type translation it generates refers only to type variables generated from $\mathcal{D}'$:

(abs-tc-local)

$$\frac{\begin{array}{c}\mathsf{tycon}\ \mathrm{TC}\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \psi \in \Phi \\ \sigma_{\text{schema}}\ \sigma_{\text{tyidx}} \Downarrow \blacktriangleright(\hat{\tau}) \\ \hat{\tau} \parallel \mathcal{D} \hookrightarrow^{\mathrm{TC}}_{\Phi} \tau \parallel \mathcal{D}' \qquad \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \Delta \vdash \tau\end{array}}{\mathsf{trans}(\mathrm{TC}\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \hookrightarrow^{\mathrm{TC}}_{\Phi} \tau \parallel \mathcal{D}'}$$

The translation of a user-defined type constructed by any tycon other than the delegated tycon is added to the store (the supplement has the simple rule for retriving it once there):

(abs-tc-foreign-new)

$$\frac{\begin{array}{c} c \neq \mathrm{TC} \qquad \mathrm{TC}\langle\sigma_{\text{tyidx}}\rangle \notin \mathrm{dom}(\mathcal{D}) \\ \mathsf{tycon}\ \mathrm{TC}\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \psi \in \Phi \\ \sigma_{\text{schema}}\ \sigma_{\text{tyidx}} \Downarrow \blacktriangleright(\hat{\tau}) \qquad \hat{\tau} \parallel \mathcal{D} \hookrightarrow^{\mathrm{TC}}_{\Phi} \tau \parallel \mathcal{D}' \\ \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \Delta \vdash \tau \qquad (\alpha \text{ fresh})\end{array}}{\mathsf{trans}(\mathrm{TC}\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \hookrightarrow^{c}_{\Phi} \alpha \parallel \mathcal{D}', \mathrm{TC}\langle\sigma_{\text{tyidx}}\rangle \leftrightarrow [\delta]\tau/\alpha}$$

The translation of an "other" type is given directly in its index (rule (abs-other-foreign-new) is in the supplement):

(abs-other-local)

$$\frac{\hat{\tau} \parallel \mathcal{D} \hookrightarrow^{\mathsf{other}[m]}_{\Phi} \tau \parallel \mathcal{D}'}{\mathsf{trans}(\mathsf{other}[m]\langle(\sigma_{\text{nat}}, \blacktriangleright(\hat{\tau}))\rangle) \parallel \mathcal{D} \hookrightarrow^{\mathsf{other}[m]}_{\Phi} \tau \parallel \mathcal{D}'}$$

4. Typing and Translation of External Terms

Having established how types are constructed, and how they determine selectively abstract and from there actual type translations, we can finally give the typing and translation rules for external terms, shown in Figure 7.

Because we are defining a bidirectional type system, a subsumption rule is needed to allow synthetic terms to be analyzed against an equivalent type. Per Sec. 3.3, equivalent types must be syntactically identical at normal form, and we consider analysis only if $\sigma\ \mathtt{type}_{\Phi}$, so the rule (subsume) is straightforward. To use an analytic term in a synthetic position, the programmer must provide a type ascription, written $e : \sigma$. The ascription is kind checked and normalized to a type before being used for analysis by rule (ascribe).

Variables and functions behave in the standard manner given our definitions of types and type translations (used to generate ascriptions in the IL, which is intrinsically rather than bidirectionally typed). We use Plotkin's fixpoint operator for general recursion (cf. [12]), and define both lambdas and fixpoints only in analytic positions for simplicity.

4.1 Generalized Introductory Operations

The translation of the generalized introductory form, $\mathsf{intro}[\sigma_{\text{tmidx}}](\overline{e})$, is determined by the tycon of the type it is being analyzed against as a function of the type's index, the *term index*, σ_{tmidx}, and the *argument list*, $\overline{e}$.

Before discussing rules (ana-intro) and (ana-intro-other), we note that we can recover a variety of standard concrete introductory forms by a purely syntactic desugaring to this abstract form (and thus allow their use at more than one type). For example, for regular strings we can use the string literal form, `"s"`, which desugars to $\mathsf{intro}[\texttt{"s"}_{\text{SL}}](\cdot)$, i.e. the term index is the corresponding static value of kind Str, indicated by a subscript for clarity. Similarly, for labeled products, records, objects and so on, we can define a generalized labeled collection form, $\{\mathtt{lbl}_1 = e_1, \ldots, \mathtt{lbl}_n = e_n\}$, that desugars to $\mathsf{intro}[[\mathtt{lbl}_1, \ldots, \mathtt{lbl}_n]](e_1; \ldots; e_n)$, i.e. a list constructed from the row labels is the term index and the corresponding row values are the arguments. In both cases, the term index captures static portions of the concrete form and the arguments capture all external sub-terms. Additional desugarings are shown in the supplement and a technique based on [22] could be introduced to allow tycon providers to define more such desugarings composably.

Let us derive $\Upsilon_{\text{ex}} \vdash_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} e_{\text{ex}} \Leftarrow \sigma_{\text{paper}} \rightsquigarrow \iota_{\text{ex}}$ where $\Upsilon_{\text{ex}} := \mathit{title} \Rightarrow \sigma_{\text{title}}$ and $e_{\text{ex}} := \{\mathtt{title} = \mathit{title}, \mathtt{conf} = \texttt{"EXMPL 2015"}\}$. The translation will be $\iota_{\text{ex}} := (\mathit{title}, (\texttt{"EXMPL 2015"}_{\text{IL}}, ()))$, where $\texttt{"EXMPL 2015"}_{\text{IL}}$ is an internal string (of internal type str).

The first premise of (ana-intro) extracts the tycon definition for the tycon of the type the intro form is being analyzed against. In this example, this is LPROD. We will use this as the *delegated tycon* in the final premises of the rule.

The second premise extracts the *intro term index kind*, κ_{tmidx}, from the *opcon signature*, χ, and the third premise checks the provided term index against this kind. This is simply the kind of term index expected by the tycon, e.g., LPROD specifies $\mathsf{List}[\mathsf{Lbl}]$, so that it can use the labeled collection form, while RSTR specifies an intro index kind of Str, so that it can use the string literal form.

The fourth premise extracts the *intro opcon definition* from the *opcon structure*, ω, of the tycon structure, calling it σ_{def}. This is a static function that is applied, in the seventh premise, to determine whether the term is well-typed, raising an error if not or determining the translation of the term if so. The function has access to the type index, the term index and an interface to the list of arguments, and its kind is checked by the judgement $\vdash_{\Phi} \omega \sim \psi$, which appeared as the final premise of the rule (tcc-ext) and is defined in Figure 8.

For example, the opcon structure of RSTR is $\omega_{\text{rstr}} := \mathsf{ana\ intro} = \sigma_{\text{rstr/intro}}; \omega_{\text{rstr/targops}}$ where $\sigma_{\text{rstr/intro}} :=$

$$\begin{array}{l}\lambda \boldsymbol{\mathit{tyidx}}{::}\mathsf{Rx}.\lambda \boldsymbol{\mathit{tmidx}}{::}\mathsf{Str}.\lambda \boldsymbol{\mathit{args}}{::}\mathsf{List}[\mathsf{Arg}]. \\ \quad \mathsf{let}\ \boldsymbol{\mathit{aok}} :: 1 = \boldsymbol{\mathit{arity0\ args}}\ \mathsf{in} \\ \quad \mathsf{let}\ \boldsymbol{\mathit{rok}} :: 1 = \boldsymbol{\mathit{rmatch\ tyidx\ tmidx}}\ \mathsf{in}\ \boldsymbol{\mathit{str\_of\_Str\ tmidx}}\end{array}$$

Because regular strings are implemented as strings, this intro opcon definition is straightforward. It begins by making sure that no arguments were passed in (we will return to arguments and the fifth and sixth premises of (ana-intro)

$\boxed{\Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota}$ $\boxed{\Upsilon \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota}$

(subsume)
$$\frac{\Upsilon \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota}{\Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota}$$

(ascribe)
$$\frac{\emptyset\,\emptyset \vdash^0_\Phi \sigma :: \mathsf{Ty} \qquad \sigma \Downarrow \sigma' \qquad \Upsilon \vdash_\Phi e \Leftarrow \sigma' \rightsquigarrow \iota}{\Upsilon \vdash_\Phi e : \sigma \Rightarrow \sigma' \rightsquigarrow \iota}$$

(syn-var)
$$\frac{x \Rightarrow \sigma \in \Upsilon}{\Upsilon \vdash_\Phi x \Rightarrow \sigma \rightsquigarrow x}$$

(ana-fix)
$$\frac{\Upsilon, x \Rightarrow \sigma \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota \qquad \vdash_\Phi \sigma \rightsquigarrow \tau}{\Upsilon \vdash_\Phi \mathsf{fix}(x.e) \Leftarrow \sigma \rightsquigarrow \mathsf{fix}[\tau](x.\iota)}$$

(ana-lam)
$$\frac{\Upsilon, x \Rightarrow \sigma_1 \vdash_\Phi e \Leftarrow \sigma_2 \rightsquigarrow \iota \qquad \vdash_\Phi \sigma_1 \rightsquigarrow \tau_1}{\Upsilon \vdash_\Phi \lambda(x.e) \Leftarrow \rightharpoonup\langle(\sigma_1, \sigma_2)\rangle \rightsquigarrow \lambda[\tau_1](x.\iota)}$$

(syn-ap)
$$\frac{\Upsilon \vdash_\Phi e_1 \Rightarrow \rightharpoonup\langle(\sigma_1, \sigma_2)\rangle \rightsquigarrow \iota_1 \qquad \Upsilon \vdash_\Phi e_2 \Leftarrow \sigma_2 \rightsquigarrow \iota_2}{\Upsilon \vdash_\Phi e_1(e_2) \Rightarrow \sigma_2 \rightsquigarrow \iota_1(\iota_2)}$$

(ana-intro)
$$\frac{\begin{array}{c}\mathsf{tycon}\ \textsc{tc}\ \{\mathsf{trans} = \_\ \mathsf{in}\ \omega\} \sim \mathsf{tcsig}[\_]\ \{\chi\} \in \Phi \\ \mathsf{intro}[\kappa_{\text{tmidx}}] \in \chi \qquad \emptyset\,\emptyset \vdash^0_\Phi \sigma_{\text{tmidx}} :: \kappa_{\text{tmidx}} \\ \mathsf{ana\ intro} = \sigma_{\text{def}} \in \omega \qquad |\overline{e}| = n \qquad \mathsf{args}(n) = \sigma_{\text{args}} \\ \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \Downarrow_{\overline{e};\Upsilon;\Phi} \rhd(\hat{\iota}) \\ \hat{\iota} \parallel \emptyset\,\emptyset \looparrowright^{\textsc{tc}}_{\overline{e};\Upsilon;\Phi} \iota_{\text{abs}} \parallel \mathcal{D}\,\mathcal{G} \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma_{\text{abs}} \\ \mathsf{trans}(\textsc{tc}\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \looparrowright^{\textsc{tc}}_\Phi \tau_{\text{abs}} \parallel \mathcal{D}' \qquad \mathcal{D}' \rightsquigarrow \delta : \Delta_{\text{abs}} \\ \Delta_{\text{abs}}\ \Gamma_{\text{abs}} \vdash \iota_{\text{abs}} : \tau_{\text{abs}}\end{array}}{\Upsilon \vdash_\Phi \mathsf{intro}[\sigma_{\text{tmidx}}](\overline{e}) \Leftarrow \textsc{tc}\langle\sigma_{\text{tyidx}}\rangle \rightsquigarrow [\delta][\gamma]\iota}$$

(syn-targ)
$$\frac{\begin{array}{c}\Upsilon \vdash_\Phi e_{\text{targ}} \Rightarrow \textsc{tc}\langle\sigma_{\text{tyidx}}\rangle \rightsquigarrow \iota_{\text{targ}} \\ \mathsf{tycon}\ \textsc{tc}\ \{\mathsf{trans} = \_\ \mathsf{in}\ \omega\} \sim \mathsf{tcsig}[\_]\ \{\chi\} \in \Phi \\ \mathbf{op}[\kappa_{\text{tmidx}}] \in \chi \qquad \emptyset\,\emptyset \vdash^0_\Phi \sigma_{\text{tmidx}} :: \kappa_{\text{tmidx}} \\ \mathsf{syn}\ \mathbf{op} = \sigma_{\text{def}} \in \omega \qquad |e_{\text{targ}}; \overline{e}| = n \qquad \mathsf{args}(n) = \sigma_{\text{args}} \\ \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \Downarrow_{(e_{\text{targ}};\overline{e});\Upsilon;\Phi} (\sigma, \rhd(\hat{\iota})) \\ \hat{\iota} \parallel \emptyset\,\emptyset \looparrowright^{\textsc{tc}}_{(e_{\text{targ}};\overline{e});\Upsilon;\Phi} \iota_{\text{abs}} \parallel \mathcal{D}\,\mathcal{G} \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma_{\text{abs}} \\ \mathsf{trans}(\sigma) \parallel \mathcal{D} \looparrowright^{\textsc{tc}}_\Phi \tau_{\text{abs}} \parallel \mathcal{D}' \qquad \mathcal{D}' \rightsquigarrow \delta : \Delta_{\text{abs}} \\ \Delta_{\text{abs}}\ \Gamma_{\text{abs}} \vdash \iota_{\text{abs}} : \tau_{\text{abs}}\end{array}}{\Upsilon \vdash_\Phi \mathsf{targ}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e}) \Rightarrow \sigma \rightsquigarrow [\delta][\gamma]\iota_{\text{abs}}}$$

(ana-intro-other)
$$\frac{\begin{array}{c}\emptyset\,\emptyset \vdash^0_\Phi \sigma_{\text{def}} :: \mathsf{List}[\mathsf{Arg}] \to \mathsf{ITm} \\ |\overline{e}| = n \qquad \mathsf{args}(n) = \sigma_{\text{args}} \qquad \sigma_{\text{def}}\ \sigma_{\text{args}} \Downarrow_{\overline{e};\Upsilon;\Phi} \rhd(\hat{\iota}) \\ \hat{\iota} \parallel \emptyset\,\emptyset \looparrowright^{\mathsf{other}[m]}_{\overline{e};\Upsilon;\Phi} \iota_{\text{abs}} \parallel \mathcal{D}\,\mathcal{G} \\ \mathsf{trans}(\mathsf{other}[m]\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \looparrowright^{\mathsf{other}[m]}_\Phi \tau_{\text{abs}} \parallel \mathcal{D}' \\ \mathcal{D}' \rightsquigarrow \delta : \Delta_{\text{abs}} \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma_{\text{abs}} \qquad \Delta_{\text{abs}}\ \Gamma_{\text{abs}} \vdash \iota_{\text{abs}} : \tau_{\text{abs}}\end{array}}{\Upsilon \vdash_\Phi \mathsf{intro}[\sigma_{\text{def}}](\overline{e}) \Leftarrow \mathsf{other}[m]\langle\sigma_{\text{tyidx}}\rangle \rightsquigarrow [\delta][\gamma]\iota_{\text{abs}}}$$

(syn-targ-other)
$$\frac{\begin{array}{c}\Upsilon \vdash_\Phi e_{\text{targ}} \Rightarrow \mathsf{other}[m]\langle\sigma_{\text{tyidx}}\rangle \rightsquigarrow \iota_{\text{targ}} \\ \emptyset\,\emptyset \vdash^0_\Phi \sigma_{\text{def}} :: \mathsf{List}[\mathsf{Arg}] \to (\mathsf{Ty} \times \mathsf{ITm}) \\ |e_{\text{targ}}; \overline{e}| = n \qquad \mathsf{args}(n) = \sigma_{\text{args}} \qquad \sigma_{\text{def}}\ \sigma_{\text{args}} \Downarrow_{(e_{\text{targ}};\overline{e});\Upsilon;\Phi} (\sigma, \rhd(\hat{\iota})) \\ \hat{\iota} \parallel \emptyset\,\emptyset \looparrowright^{\mathsf{other}[m]}_{(e_{\text{targ}};\overline{e});\Upsilon;\Phi} \iota_{\text{abs}} \parallel \mathcal{D}\,\mathcal{G} \\ \mathsf{trans}(\sigma) \parallel \mathcal{D} \looparrowright^{\mathsf{other}[m]}_\Phi \tau_{\text{abs}} \parallel \mathcal{D}' \\ \mathcal{D}' \rightsquigarrow \delta : \Delta_{\text{abs}} \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma_{\text{abs}} \qquad \Delta_{\text{abs}}\ \Gamma_{\text{abs}} \vdash \iota_{\text{abs}} : \tau_{\text{abs}}\end{array}}{\Upsilon \vdash_\Phi \mathsf{targ}[\mathbf{op}; \sigma_{\text{def}}](e_{\text{targ}}; \overline{e}) \Rightarrow \sigma \rightsquigarrow [\delta][\gamma]\iota_{\text{abs}}}$$

Figure 7. Typing

$\boxed{\vdash_\Phi \omega \sim \psi}$

(ocstruct-intro)
$$\frac{\mathsf{intro}[\kappa_{\text{tmidx}}] \in \chi \qquad \emptyset \vdash \kappa_{\text{tmidx}} \qquad \emptyset\,\emptyset \vdash^0_\Phi \sigma_{\text{def}} :: \kappa_{\text{tyidx}} \to \kappa_{\text{tmidx}} \to \mathsf{List}[\mathsf{Arg}] \to \mathsf{ITm}}{\vdash_\Phi \mathsf{ana\ intro} = \sigma_{\text{def}} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\}}$$

(ocstruct-targ)
$$\frac{\vdash_\Phi \omega \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\} \qquad \mathbf{op} \notin \text{dom}(\chi) \qquad \emptyset \vdash \kappa_{\text{tmidx}} \qquad \emptyset\,\emptyset \vdash^0_\Phi \sigma_{\text{def}} :: \kappa_{\text{tyidx}} \to \kappa_{\text{tmidx}} \to \mathsf{List}[\mathsf{Arg}] \to (\mathsf{Ty} \times \mathsf{ITm})}{\vdash_\Phi \omega; \mathsf{syn}\ \mathbf{op} = \sigma_{\text{def}} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi, \mathbf{op}[\kappa_{\text{tmidx}}]\}}$$

Figure 8. Checking opcon structures against tycon signatures.

with the next example), using the helper function ***arity0*** :: $\mathsf{List}[\mathsf{Arg}] \to 1$ defined such that any non-empty list will raise an error, via the static term $\mathsf{raise}[1]$. In practice, the tycon provider would specify an error message here. Next, it checks the string provided as the term index against the regular expression given as the type index using ***rmatch*** :: $\mathsf{Rx} \to \mathsf{Str} \to 1$, which we assume is defined in the usual way and again raises an error on failure. Finally, a translation corresponding to the term index is generated via helper function ***str_of_Str*** :: $\mathsf{Str} \to \mathsf{ITm}$.

The only introductory form for kind ITm is $\rhd(\hat{\iota})$, where $\hat{\iota}$ is a *translational internal term*. This form is analagous to the introductory form for kind ITy described in Sec. 3.4, $\blacktriangleright(\hat{\tau})$. Each form in the syntax of ι has a corresponding form in the syntax for $\hat{\iota}$ and both the kinding rules and dynamics simply recurse through these in the same manner as shown in Sec. 3.4. There is also an analagous unquote form, $\lhd(\sigma)$. The final two forms of translational internal term are $\mathsf{anatrans}[n](\sigma)$ and $\mathsf{syntrans}[n]$. These stand in for the translation of argument n, the first if it arises via analysis against type σ and the second if it arises via type synthesis. Before giving the rules, let us motivate the mechanism with the intro opcon definition for LPROD. We have $\omega_{\text{lprod}} :=$ $\mathsf{ana\ intro} = \sigma_{\text{lprod/intro}}; \omega_{\text{lprod/targops}}$ where $\sigma_{\text{lprod/intro}} :=$

λ***tyidx***:$\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$.$\lambda$***tmidx***:$\mathsf{List}[\mathsf{Lbl}]$.$\lambda$***args***:$\mathsf{List}[\mathsf{Arg}]$.
  let ***inhabited*** : $1 =$ ***uniqmap tyidx***
  listrec3 $[\mathsf{Lbl} \times \mathsf{Ty}]\ [\mathsf{Lbl}]\ [\mathsf{Arg}]\ [\mathsf{ITm}]$ ***tyidx tmidx args*** $\rhd(())$
    λ***rowtyidx***:$\mathsf{Lbl} \times \mathsf{Ty}$.$\lambda$***rowtmidx***:$\mathsf{Lbl}$.$\lambda$***rowarg***:$\mathsf{Arg}$.$\lambda$***r***:$\mathsf{ITm}$.
      letpair (***rowlbl***, ***rowty***) = ***rowtyidx***
      let ***lok*** :: $1 =$ ***lbleq rowlbl rowtmidx***
      let ***rowtr*** :: $\mathsf{ITm} =$ ***ana rowarg rowty***
      $\rhd((\lhd($***rowtr***$), \lhd($***r***$)))$

The first line checks that the type provided is inhabited, in this case by checking that there are no duplicate labels via the helper function ***uniqmap*** :: $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}] \to 1$, raising an error if there are. An alternative strategy may have been to use an abstract kind that ensured that such type indices could not have been constructed, but to be compatible with our equality kind restriction, this would require support for abstract equality kinds, analagous to abstract equality types in SML. We chose not to formalize these for simplicity, and to demonstrate this general technique.

The rest of this opcon definition folds over the three lists provided as input: the list mapping labels to types provided as the type index, the list of labels provided as the term index, and the list of argument interfaces. We assume a straightforward helper function, ***listfold3***, that raises an error if the three lists are not of the same length. The base case is the translational empty product. The recursive case first checks that the label provided in the term index matches the label provided in the type index, using a helper function ***lbleq*** :: $\mathsf{Lbl} \to \mathsf{Lbl} \to 1$. Then, we request type analysis of the corresponding argument, ***rowarg***, against the type in the type index, ***rowty***, by writing ***ana rowarg rowty***. Here, ***ana*** is a helper function defined below that produces a translational internal term of the form $\triangleright(\mathsf{anatrans}[n](\sigma))$, where σ is the value of ***rowty***, if the argument is well-typed, and raises an error if not. The final line constructs a nested tuple based on this translation and the recursive result. Taken together, the translational internal term that will be generated for our example involving e_{ex} above is $\hat{\iota}_{\text{ex}} := (\mathsf{anatrans}[0](\sigma_{\text{title}}), (\mathsf{anatrans}[1](\sigma_{\text{conf}}), ()))$.

Argument Interface Lists We define Arg, the kind of *argument interfaces*, as a simple product of functions, $\mathsf{Arg} := (\mathsf{Ty} \to \mathsf{ITm}) \times (1 \to \mathsf{Ty} \times \mathsf{ITm})$, one for analysis and the other for synthesis, both having the expected kind for these operations. The helper functions ***ana*** and ***syn*** simply project the corresponding function out, ***ana*** $:= \lambda \boldsymbol{arg}{::}\mathsf{Arg}.\mathsf{fst}(\boldsymbol{arg})$ and ***syn*** $:= \lambda \boldsymbol{arg}{::}\mathsf{Arg}.\mathsf{snd}(\boldsymbol{arg})$.

The argument interface list generated by the judgement $\mathsf{args}(n) = \sigma_{\text{args}}$, where n is the length of the argument list, written $|\overline{e}| = n$, is then simply a static value of kind $\mathsf{List}[\mathsf{Arg}]$ of length n where the ith entry is $(\lambda \boldsymbol{ty}{::}\mathsf{Ty}.\mathsf{ana}[i](\boldsymbol{ty}), \lambda \_{::}1.\mathsf{syn}[i])$.

Recall that the kinding judgement is indexed by n, which is an upper bound on the argument index of terms of the form $\mathsf{ana}[n](\sigma)$ and $\mathsf{syn}[n]$. This is enforced in their kinding rules:

(k-ana)
$$\frac{n' < n \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{ana}[n'](\sigma) :: \mathsf{ITm}}$$

(k-syn)
$$\frac{n' < n}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{syn}[n'] :: \mathsf{Ty} \times \mathsf{ITm}}$$

The following lemma characterizes thus characterizes the argument list interface generation judgement:

Lemma 1. *If* $\mathsf{args}(n) = \sigma_{args}$ *then* $\emptyset\ \emptyset \vdash^n_\Phi \sigma_{args} :: \mathsf{List}[\mathsf{Arg}]$.

The rule (ocstruct-intro) ruled out writing either of these forms explicitly in an opcon definition by checking against the bound $n = 0$. This is to prevent out-of-bounds errors – tycon providers do not write these forms directly, but only access them via the argument interface list, guaranteed to have the correct length. This is safe because:

Lemma 2. *If* $\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n'}_\Phi \sigma :: \kappa$ *and* $n > n'$ *then* $\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa$.

These forms serve as the link between the dynamics of the static language and the statics of the external language. For $\mathsf{ana}[n](\sigma)$, after stepping the type σ to a value and propagating errors, the argument environment, which stores the arguments themselves and the typing and tycon contexts, $\overline{e}; \Upsilon; \Phi$, is consulted to retrieve the nth argument and analyze it against σ. If this succeeds, the translational internal term $\triangleright(\mathsf{anatrans}[n](\sigma))$ is generated to refer to it.

(s-ana-success)
$$\frac{\sigma\ \mathtt{val}_{\overline{e};\Upsilon;\Phi} \qquad \mathsf{nth}[n](\overline{e}) = e \qquad \Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota}{\mathsf{ana}[n](\sigma) \mapsto_{\overline{e};\Upsilon;\Phi} \triangleright(\mathsf{anatrans}[n](\sigma))}$$

If it fails, an error is raised:

(s-ana-fail)
$$\frac{\sigma\ \mathtt{val}_{\overline{e};\Upsilon;\Phi} \qquad \mathsf{nth}[n](\overline{e}) = e \qquad [\Upsilon \vdash_\Phi e \not\Leftarrow \sigma]}{\mathsf{ana}[n](\sigma)\ \mathtt{err}_{\overline{e};\Upsilon;\Phi}}$$

We write $[\Upsilon \vdash_\Phi e \not\Leftarrow \sigma]$ to indicate that e fails to analyze against σ. We do not define this inductively, so we also allow that this premise be omitted, leaving a non-deterministic semantics nevertheless sufficient for our metatheory.

The dynamics for $\mathsf{syn}[n]$ are analagous, evaluating to a pair $(\sigma, \triangleright(\mathsf{syntrans}[n]))$ where σ is the synthesized type.

The kinding rules also prevent these translational internal term forms from being well-kinded when $n = 0$. Like the translational internal type form $\mathsf{trans}(\sigma)$, these are retained in values of kind ITm:

(s-itm-anatrans-v)
$$\frac{\sigma\ \mathtt{val}_{\overline{e};\Upsilon;\Phi} \qquad \mathsf{nth}[n](\overline{e}) = e}{\triangleright(\mathsf{anatrans}[n](\sigma))\ \mathtt{val}_{\overline{e};\Upsilon;\Phi}}$$

(s-itm-syntrans-v)
$$\frac{\mathsf{nth}[n](\overline{e}) = e}{\triangleright(\mathsf{syntrans}[n])\ \mathtt{val}_{\overline{e};\Upsilon;\Phi}}$$

Selectively Abstracted Term Translations The reason for this is again because we will hold these abstract by replacing them with variables. The judgement $\hat{\iota} \parallel \mathcal{D}\ \mathcal{G} \looparrowright^c_{\mathcal{A}} \iota^+ \parallel \mathcal{D}^+\ \mathcal{G}^+$, appearing as the eighth premise of (ana-intro), relates a translational internal term $\hat{\iota}$ to an internal term ι called its *selectively abstracted term translation*, where all references to the translation of an argument (having any type) are replaced with a unique variable and all references to the translation of a type constructed by a user-defined tycon other than the "delegated tycon" $\textsc{tc}$ are replaced with an abstract type variable as described in Sec. 3.4.1. The type translation store, $\mathcal{D}$, discussed previously, and term translation store, $\mathcal{G}$, track these mappings. Term translation stores have the following syntax: $\mathcal{G} ::= \emptyset \mid \mathcal{G}, n : \sigma \rightsquigarrow \iota/x : \tau$. Each entry can be read "argument n having type σ and translation ι appears as variable x with type τ". For example,

$$\hat{\iota}_{\text{ex}} \parallel \emptyset\ \emptyset \looparrowright^{\text{LPROD}}_{(title, \texttt{"EXMPL 2015"}), \Upsilon, \Phi_{\text{rstr}}\Phi_{\text{lprod}}} \iota_{\text{ex/abs}} \parallel \mathcal{D}_{\text{ex/abs}}\ \mathcal{G}_{\text{ex/abs}}$$

where $\iota_{\text{ex/abs}} := (x_0, (x_1, ()))$, the term translation store is $\mathcal{G}_{\text{ex/abs}} := 0 : \sigma_{\text{title}} \leadsto title/x_0 : \alpha_0, 1 : \sigma_{\text{conf}} \leadsto$ $\texttt{"EXMPL 2015"}_{\text{IL}}/x_1 : \alpha_1$ and the type translation store $\mathcal{D}_{\text{ex/abs}}$ is alpha-equivalent to $\mathcal{D}_{\text{paper/abs}}$ from Sec. 3.4.1.

The rules for the abstracted term translation judgement follow recursively for shared forms just like those for the abstracted type translation judgement (see supplement). The rule for references to argument translations derived via type analysis operates as follows ($\mathsf{syntrans}[n]$ is analagous):

(abs-anatrans-new)

$$\frac{n \notin \text{dom}(\mathcal{G}) \qquad \mathsf{nth}[n](\overline{e}) = e \qquad \Upsilon \vdash_\Phi e \Leftarrow \sigma \leadsto \iota \qquad \mathsf{trans}(\sigma) \parallel \mathcal{D} \hookrightarrow^{c}_{\Phi} \tau \parallel \mathcal{D}' \qquad (x \text{ fresh})}{\mathsf{anatrans}[n](\sigma) \parallel \mathcal{D}\; \mathcal{G} \hookrightarrow^{c}_{\overline{e};\Upsilon;\Phi} x \parallel \mathcal{D}'\; \mathcal{G}, n : \sigma \leadsto \iota/x : \tau}$$

Like $\mathcal{D}$, each $\mathcal{G}$ induces an internal term substitution, $\gamma ::= \emptyset \mid \gamma, \iota/x$, and corresponding internal typing context Γ by the judgement $\mathcal{G} \leadsto \gamma : \Gamma$, appearing as the ninth premise. In this case, $\gamma_{\text{ex/abs}} := title/x_0, \texttt{"EXMPL 2015"}_{\text{IL}}/x_1$ and $\Gamma_{\text{ex/abs}} := x_0 : \alpha_0, x_1 : \alpha_1$.

The tenth premise of (ana-intro) determines an abstract type translation for the type provided for analysis, in this case $\tau_{\text{ex/abs}} := \alpha_0 \times (\alpha_1 \times 1)$ (alpha-equivalent to $\tau_{\text{paper/abs}}$ in Sec. 3.4.1), and the eleventh premise extracts a type substitution, $\delta_{\text{ex/abs}}$, and type formation context, $\Delta_{\text{ex/abs}}$, from $\mathcal{D}_{\text{ex/abs}}$, again equivalent to $\delta_{\text{conf/abs}}$ and $\Delta_{\text{conf/abs}}$ from Sec. 3.4.1.

Finally, the twelfth premise checks the abstracted term translation against the abstracted type translation. Here, we are checking $\Delta_{\text{ex/abs}}\; \Gamma_{\text{ex/abs}} \vdash \iota_{\text{ex/abs}} : \tau_{\text{ex/abs}}$, i.e.:

$$(\alpha_0, \alpha_1)\; (x_0 : \alpha_0, x_1 : \alpha_1) \vdash (x_0, (x_1, ())) : \alpha_0 \times (\alpha_1 \times 1)$$

In other words, the translation of the labeled product e_{ex} generated by LPROD is checked with all references to term and type translations of regular strings replaced by variables and type variables, respectively. Because our intro opcon definition treated arguments parametrically, the check succeeds. We describe an ill-behaved operator in Sec. 5.

Applying the substitutions $\gamma_{\text{ex/abs}}$ and $\delta_{\text{ex/abs}}$ in the conclusion of the rule, we arrive at the actual term translation ι_{ex}.

The rule (ana-intro-other) is used to introduce terms of a type constructed by an "other" tycon. The term index, rather than the tycon context, directly specifies the static function that maps the arguments to a translation. In all other respects, the rule is analagous. It is used as a technical device in our proof of conservativity in Sec. 5.

4.2 Generalized Targeted Operations

All non-introductory opcons associated with user-defined tycons go through another generalized form, in this case for *targeted operations*, $\mathsf{targ}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e})$, where **op** ranges over opcon names, σ_{tmidx} is the term index, e_{targ} is the *target argument* and $\overline{e}$ are the remaining arguments.

Desugarings include an explicit form, $e_{\text{targ}}.\mathbf{op}\langle\sigma_{\text{tmidx}}\rangle(\overline{e})$ (and variants of where the term index or the arguments are omitted), projection syntax for use by record-like types, $e_{\text{targ}}\texttt{\#lbl}$, which desugars to $\mathsf{targ}[\mathbf{prj}; \texttt{lbl}](e_{\text{targ}}; \cdot)$, and concatenative syntax $e_{\text{targ}} \cdot e_{\text{arg}}$, which desugars to $\mathsf{targ}[\mathbf{conc}; ()](e_{\text{targ}}; e_{\text{arg}})$. We show other desugarings, including case analysis, in the supplement.

Whereas introductory operations were analytic, targeted operations are synthetic in $@\lambda$. The type and translation are determined by the tycon of the type synthesized by the target argument. The rule (syn-targ) is otherwise similar to (ana-intro) in its structure. The first premise synthesizes a type, $\text{TC}\langle\sigma_{\text{tyidx}}\rangle$, for the target argument. The second premise extracts the tycon definition for TC from the tycon context. The third extracts the *operator index kind* from its opcon signature, and the fourth checks the term index against this kind. For example, we may associate the following opcon signatures with the tycons RSTR and LPROD:

$$\begin{aligned} \chi_{\text{rstr}} &:= \mathsf{intro}[\mathsf{Str}], \mathbf{conc}[1], \mathbf{case}[\mathsf{List}[\mathsf{StrPattern}]], \\ &\quad\; \mathbf{coerce}[\mathsf{Rx}], \mathbf{check}[\mathsf{Rx}], \mathbf{replace}[\mathsf{Rx}] \\ \chi_{\text{lprod}} &:= \mathsf{intro}[\mathsf{List}[\mathsf{Lbl}]], \mathbf{prj}[\mathsf{Lbl}], \mathbf{conc}[1], \mathbf{drop}[\mathsf{List}[\mathsf{Lbl}]] \end{aligned}$$

The opcons associated with RSTR are taken directly from Fulton et al.'s specification of regular string types [10], with the exception of **case**, which generalizes case analysis as defined there to arbitrary string patterns, which we discuss in the supplement. The opcons associated with LPROD are also straightforward: **prj** projects out the row with the provided label, **conc** concatenates two labeled products (updating common fields with the value of the right argument), and **drop** creates a new labeled product from the target with some fields dropped. Note that both regular strings and labeled products can define concatenation without conflict.

The fifth premise of (syn-targ) extracts the targeted opcon definition of **op** from the opcon structure, ω. Like the intro opcon definition, this is a static function that generates a translational internal term on the basis of the target tycon's type index, the term index and an argument interface list. Targeted opcon definitions additionally synthesize a type. The rule (ocstruct-targ) in Figure 8 ensures that no tycon defines an opcon twice and that the opcon definitions are well-kinded. For example, $\omega_{\text{rstr/targops}}$ defines:

$$\begin{array}{l} \mathsf{syn}\; \mathbf{conc} = \lambda \boldsymbol{tyidx}{::}\mathsf{Rx}.\lambda \boldsymbol{tmidx}{::}1.\lambda \boldsymbol{args}{::}\mathsf{List}[\mathsf{Arg}]. \\ \quad \mathsf{letpair}\; (\boldsymbol{arg1}, \boldsymbol{arg2}) = \boldsymbol{arity2\; args} \\ \quad \mathsf{letpair}\; (\_, \boldsymbol{tr1}) = \boldsymbol{syn\; arg1} \\ \quad \mathsf{letpair}\; (\boldsymbol{ty2}, \boldsymbol{tr2}) = \boldsymbol{syn\; arg2} \\ \quad \mathsf{tycase}[\text{RSTR}](\boldsymbol{ty2}; \boldsymbol{tyidx2}. \\ \qquad (\text{RSTR}\langle \boldsymbol{rxconcat\; tyidx\; tyidx2}\rangle, \rhd(sconcat \lhd(\boldsymbol{tr1}) \lhd(\boldsymbol{tr2}))) \\ \quad ; \mathsf{raise}[\mathsf{Ty} \times \mathsf{ITm}]) \end{array}$$

The helper function ***arity2*** checks that exactly two arguments, including the target argument. We then request synthesis of both arguments. We can ignore the type synthesized by the first because by definition it is a regular string type with type index ***tyidx***. We case analyze the second against RSTR, extracting its index regular expression if so and raising an error if not. We then synthesize the resulting regular string type, using the helper function ***rxconcat*** ::

$\mathsf{Rx} \to \mathsf{Rx} \to \mathsf{Rx}$ which generates the synthesized type index, and finally the translation, using an internal helper function $sconcat : \mathsf{str} \to \mathsf{str} \to \mathsf{str}$, the translational term for which we assume has been substituted in directly above.

The remaining premises of (syn-targ) are analagous to the corresponding premises in (ana-intro), with the only difference being that we check the abstract term translation against the abstract type translation of the synthesized type.

Like (ana-intro-other), rule (syn-targ-other) is used when the target synthesizes an "other" type. The mapping from the arguments to a type and translation is given directly in the term index (the op name is ignored).

5. Metatheory

We will now state the key metatheoretic properties of @λ. The full proofs are in the supplement.

Kind Safety Kind safety ensures that normalization of well-kinded static terms cannot go wrong. We can take a standard progress and preservation based approach.

Theorem 1 (Static Progress). *If* $\emptyset\ \emptyset \vdash^n_\Phi \sigma :: \kappa$ *and* $\vdash \Phi$ *and* $|\overline{e}| = n$ *and* $\vdash_\Phi \Upsilon \leadsto \Gamma$ *then* $\sigma\ \mathtt{val}_{\overline{e};\Upsilon;\Phi}$ *or* $\sigma\ \mathtt{err}_{\overline{e};\Upsilon;\Phi}$ *or* $\sigma \mapsto_{\overline{e};\Upsilon;\Phi} \sigma'$.

Theorem 2 (Static Preservation). *If* $\emptyset\ \emptyset \vdash^n_\Phi \sigma :: \kappa$ *and* $\vdash \Phi$ *and* $|\overline{e}| = n$ *and* $\vdash_\Phi \Upsilon \leadsto \Gamma$ *and* $\sigma \mapsto_{\overline{e};\Upsilon;\Phi} \sigma'$ *then* $\emptyset\ \emptyset \vdash^n_\Phi \sigma' :: \kappa$.

The case in the proof of Theorem 2 for $\mathsf{syn}[n]$ requires that the following theorem be simultaneously defined. The mutual induction is well-founded because the total number of argument lists in the terms being considered decreases.

Theorem 3 (Type Synthesis). *If* $\vdash \Phi$ *and* $\vdash_\Phi \Upsilon \leadsto \Gamma$ *and* $\Upsilon \vdash_\Phi e \Rightarrow \sigma \leadsto \iota$ *then* $\vdash_\Phi \sigma \leadsto \tau$ *(and thus* $\sigma\ \mathtt{type}_\Phi$*).*

Type Safety Type safety in a typed translation semantics requires that well-typed external terms translate to well-typed internal terms. Type safety for the IL [12] then implies that well-typed external terms cannot go wrong. To prove this, we must actually prove a stronger theorem, *type-preserving translation*, analagous to type-preserving compilation in the typed compilation literature [27]:

right citation for TIL?

Theorem 4 (Type Preserving Translation). *If* $\vdash \Phi$ *and* $\vdash_\Phi \Upsilon \leadsto \Gamma$ *and* $\vdash_\Phi \sigma \leadsto \tau$ *and* $\Upsilon \vdash_\Phi e \Leftarrow \sigma \leadsto \iota$ *then* $\emptyset\ \Gamma \vdash \iota : \tau$.

Proof Sketch. We induct on the typing judgement. The interesting cases are (ana-intro), (ana-intro-other), (syn-trans) and (syn-trans-other); the latter two arise via subsumption. The result follows directly from the final premise of each rule, combined with lemmas that state that if $\mathcal{D} \leadsto \delta : \Delta_{\text{abs}}$ and $\mathcal{G} \leadsto \gamma : \Gamma_{\text{abs}}$ then $\vdash \delta : \Delta$ and $\Delta_{\text{abs}}\ \Gamma \vdash \gamma : \Gamma_{\text{abs}}$, i.e. that all the variables in Δ_{abs} and Γ_{abs} have well-formed/well-typed substitutions in δ and γ, and thus that applying them in the conclusions of the rules gives a well-typed term. □

Hygienic Translation Note above that the domains of Υ (and thus Γ) and Γ_{abs} are disjoint. This serves to ensure *hygienic translation* – translations cannot refer to variables in the surrounding scope directly, so uniformly renaming a variable cannot change the meaning of a program. Variables in Υ can occur in arguments (e.g. $title$ in the previous example), but the translations of the arguments only appear *after* the substitution γ has been applied. We assume that substitution is capture-avoiding in the usual manner.

Stability Extending the tycon context does not change the meaning of any terms that were previously well-typed.

Theorem 5 (Stability). *Letting* $\Phi' := \Phi, \mathsf{tycon}\ \textsc{tc}\ \{\theta\} \sim \psi$, *if* $\vdash \Phi'$ *and* $\vdash_\Phi \Upsilon \leadsto \Gamma$ *and* $\vdash_\Phi \sigma \leadsto \tau$ *and* $\Upsilon \vdash_\Phi e \Leftarrow \sigma \leadsto \iota$ *then* $\vdash_{\Phi'} \Upsilon \leadsto \Gamma$ *and* $\vdash_{\Phi'} \sigma \leadsto \tau$ *and* $\Upsilon \vdash_{\Phi'} e \Leftarrow \sigma \leadsto \iota$.

Conservativity Extending the tycon context also conserves all *tycon invariants* maintained in any smaller tycon context. An example of a tycon invariant is the following:

Tycon Invariant 1 (Regular String Soundness). *If* $\emptyset \vdash_{\Phi_{rstr}} e \Leftarrow \textsc{rstr}\langle /\mathtt{r}/\rangle \leadsto \iota$ *and* $\iota \Downarrow \iota'$ *then* $\iota' = \mathtt{"s"}$ *and* $\mathtt{"s"}$ *is in the regular language* $\mathcal{L}(\mathtt{r})$.

Proof Sketch. The proof is not unusually difficult because we have fixed the tycon context Φ_{rstr}, so we can simply treat the calculus like a type-directed compiler for a calculus with only two tycons, $\rightharpoonup$ and $\textsc{rstr}$. Such a calculus and compiler specification was given by Fulton et al. [10] and extending this to our setting requires showing only that the opcon definitions in $\textsc{rstr}$ adequately capture these specifications using standard program correctness techniques for the SL, a simply-typed functional language. The only "wrinkle" is that the rule (syn-targ-other) can synthesize a regular string type. But if so, the abstracted translation will be checked against $\tau_{\text{abs}} = \alpha$, so only strings that could otherwise arise can result, by the abstraction theorem for the IL. citation

The reason why the rule (syn-targ-other) is never a problem in proving a tycon invariant turns out to be the same reason extending the tycon context conserves all tycon invariants. A newly introduced tycon defining a targeted operator that synthesizes a regular string type, e.g. σ_{paper}, and generating a translation that is not in the corresponding regular language, e.g. `""`, could be defined, but when used, the rule (syn-targ) would check the translation against $\tau_{\text{abs}} = \alpha$, which would fail. We can state this more generally:

Theorem 6 (Conservativity). *If* $\vdash \Phi$ *and* $\textsc{tc} \in dom(\Phi)$ *and a tycon invariant for* $\textsc{tc}$ *holds under* Φ*:*

- *If* $\emptyset \vdash_\Phi e \Leftarrow \textsc{tc}\langle\sigma_{tyidx}\rangle \leadsto \iota$ *and* $\vdash_\Phi \textsc{tc}\langle\sigma_{tyidx}\rangle \leadsto \tau$ *then* $P(\sigma_{tyidx}, \iota)$.

then for all $\Phi' = \Phi, \mathsf{tycon}\ \textsc{tc}'\ \{\theta'\} \sim \psi'$ *such that* $\vdash \Phi'$*, the same tycon invariant holds under* Φ'*:*

- *If* $\emptyset \vdash_{\Phi'} e \Leftarrow \textsc{tc}\langle\sigma_{tyidx}\rangle \leadsto \iota$ *and* $\vdash_{\Phi'} \textsc{tc}\langle\sigma_{tyidx}\rangle \leadsto \tau$ *then* $P(\sigma_{tyidx}, \iota)$.

(if proposition $P(\sigma_{tyidx}, \iota)$ *is* modular, *defined below)*

Proof Sketch. The proof maps every well-typed term under Φ' to a well-typed term under Φ with the same translation, and if the term has a type constructed by a tycon in Φ, e.g. $\textsc{tc}$, the mapping still has a type constructed by that tycon with the same type translation and thus we can apply the original tycon invariant. The only condition on $P(\sigma_{\text{tyidx}}, \iota)$ is that it be preserved under this mapping on σ_{tyidx}. We call such propositions *modular*. Non-modular propositions are uninteresting because the mapping's only effect on static terms is to replace all types constructed by $\textsc{tc}'$ with a unique type constructed by $\mathsf{other}[m]$, for an m uniquely associated with $\textsc{tc}'$. On external terms, the mapping replaces all introductory and targeted sub-terms associated with $\textsc{tc}'$ with an equivalent one that passes through the rules (ana-intro-other) and (syn-targ-other) by partially applying the intro and targeted opcon definitions to generate the term indices. Typing, kinding, static normalization and the selective translation abstraction judgements are all preserved under this mapping, defined inductively in the supplement. □

6. Related and Future Work

As discussed throughout the paper, our treatment of concrete syntax defers to recent work on *type-specific languages* [22]. As in our work, TSLs determine the meaning of introductory terms based on the type the term is analyzed against, but TSLs only support syntactic elaboration to a fixed semantics.

Language-integrated static term rewriting ("macro") systems, like Template Haskell [26] and Scala's static macros [5], can be used to decrease complexity when an isomorphic embedding into the underlying type system is possible, but cannot extend the type system directly.

When a simple embedding that preserves the static semantics exists, but a different embedding is needed to preserve a desired cost semantics, term rewriting can also be used to perform "optimizations", thus achieving an isomorphic embedding. Care is needed when the optimized value has a different type, to ensure that the invariants of the unoptimized term are preserved. Type abstraction has been used for this purpose in work on *lightweight modular staging* (LMS) [24]. This does not enable new type systems, i.e. in [24], Scala's type system must suffice.

When new static distinctions are needed within an existing type, but new operators are not necessary and cost is acceptable, one solution is to develop a system of *refinement types* for the language [9]. For example, a refinement of integers might distinguish negative integers. Proposals for "pluggable type systems" describe such type refinement systems [2, 4]. Refinements of abstract types can be used to hold representations abstract.

Our interest is in a more general situation, of which refinement types can be seen in ways as a degenerate case: when introducing a fragment requires defining new types and new operators, where the types and translations arising from the new operators are a function of statically valued information. For example, labeled products require new opcons that use labels and types, which exist only statically.

Many *language frameworks* and *compiler generators* exist to simplify dialect implementation (e.g.). These either citations do not support forming languages from separately defined fragments or provide few clear modular reasoning principles that make it possible to reason separately about them.

Proof assistants and logical frameworks can be used to mechanize dialect specification and metatheory, but they similarly require a complete specification. Chlipala has suggested using proof automation that covers a variety of language features to decrease the burden when a dialect is formed, but this is a fundamentally heuristic approach [8]. wrong citation Our approach provides a conservativity theorem. No proof terms are needed to guarantee modular reasoning: mistakes can only weaken the invariants at a single tycon. A promising avenue for future work beyond the scope of this paper is to introduce optional proof mechanization into the SL (by making it dependently kinded), and to mechanize the metatheory of @λ itself.

Type abstraction is, encouragingly, also the technique underlying modular reasoning in ML-like languages. Indeed, proofs of tycon invariants rely on the same abstraction theorem. Our work has some technical similarities to work on syntactic type abstraction [11] and on translating modules to System F [25]. Unlike in module systems, type translations (analagous to the choice of representation for an abstract type) can be *computed* based on a type index. Moreover, there can be arbitrarily many operators because they arise by providing a term index to an opcon. Their static and dynamic semantics can be complex because a static function computes the type and translation. In contrast, modules can only specify a fixed number of operations, and each must have function type. Note that although we did not specify quantification over external types here for simplicity, we conjecture that it is complementary (and thus @λ could serve as the core of a language with a ML-style module system). Another avenue for future work is tycon functors, which abstract over tycons with the same tycon signature.

One limitation of our approach is that it supports only fragments with the same "shape" of typing judgement. Fragments that require new forms of contexts (e.g. symbol contexts [12]) cannot be defined. The language controls variable binding, so new notions of binding, e.g. linear type systems, cannot be defined. Another limitation is that opcons cannot invoke one another (e.g. a "length" opcon on regular strings could not construct a natural number). We conjecture that these are not fundamental limitations.

References

[1] GHC/FAQ. `http://www.haskell.org/haskellwiki/GHC:FAQ#Extensible_Records`.

[2] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA*, OOPSLA '06, pages 57–74, New York, NY, USA, 2006. ACM.

[3] N. Benton and A. Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *ICFP '99*, pages 126–137, New York, NY, USA, 1999. ACM.

[4] G. Bracha. Pluggable type systems, Oct. 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[5] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[6] L. Cardelli. A semantics of multiple inheritance. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[7] 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.

[8] A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (37th POPL'10)*, pages 93–106, Madrid, Spain, Jan. 2010. ACM Press.

[9] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[10] N. Fulton, C. Omar, and J. Aldrich. Statically typed string sanitation inside a python. In *First International Workshop on Privacy and Security in Programming (PSP 2014)*. ACM, 2014.

[11] D. Grossman, J. G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst*, 22(6):1037–1080, 2000.

[12] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[13] 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.

[14] 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.

[15] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.01 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2013.

[16] A. Löh and R. Hinze. Open data types and open functions. In *PPDP*, pages 133–144. ACM, 2006.

[17] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP*, pages 82–93. ACM, 2005.

[18] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated Data Structures, Generically. In *POPL '14*, pages 411–423, New York, NY, USA, 2014. ACM.

[19] T. D. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst*, 26(5):836–889, 2004.

[20] 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.

[21] A. Ohori and K. Ueno. Making Standard ML a Practical Database Programming Language. In *ICFP*, ICFP '11, pages 307–319, New York, NY, USA, 2011. ACM.

[22] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014.

[23] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

[24] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, June 2012.

[25] A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. In A. Kennedy and N. Benton, editors, *TLDI*, pages 89–102. ACM, 2010.

[26] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[27] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI '96*, Philadelphia, PA, May 1996.