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 by specifying a language, $@\lambda$, in a style that follows 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 constrained strings and a variation on record types) are defined by extending a *tycon context*. Each tycon defines the semantics of its associated term-level *opcons* (e.g. row projection) using a static language where types and translations are values. The semantics guarantees *type safety* and, critically, *conservativity*: that all *tycon-specific invariants* that can be established in a "closed world" will necessarily be conserved in the "open world". Type system providers need not provide mechanized proofs. Instead, these guarantees require only a form of translation validation that uses type abstraction to check that tycons maintain *translation independence*, analagous to representation independence in ML-style module systems.

1. Introduction

Typed programming languages are most often described as being composed from *fragments*, each contributing to the language's concrete syntax, abstract syntax, static semantics and dynamic semantics. In his textbook, Harper organizes such fragments around type constructors, describing each in a different chapter [17]. Languages are then identified by a set of these type constructors, e.g. $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$ is the language of 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 common practice is to describe a fragment using 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 (e.g. [15]).

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") are often enough to guarantee that an embedding that preserves a desirable fragment's dynamic semantics can be constructed, but an isomorphic embedding must also preserve the static semantics and, if defined, performance bounds specified by a cost semantics. This is not always possible. For example, in $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$, it is impossible to introduce record types as a library because you need row projection operators, in ML written `#lbl`, one for each of the infinite set of row labels `lbl` (`#` itself is thus an *operator constructor*). Each time a fragment like must be introduced directly, rather than as a library, a new *dialect* of the language arises. Within the ML lineage, dialects that go beyond "core ML" abound:

1. **General Purpose Fragments:** Many variations on product types, for example, exist: n-ary tuples, labeled tuples, records (identified up to reordering), records with width and depth subtyping [8], records with update operators[1] [21], records with mutable fields [21], and records with "methods" (i.e. pure objects [31]). Sum-like types are also exposed variously: standard datatypes, open datatypes [23, 26], polymorphic variants [21] and exception types [16]. Combinations of these manifest themselves as class-based object systems [21].

[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 are also frequently introduced in dialects, e.g. for distributed programming [28], reactive programming [24], authenticated data structures [25], databases [30], units of measure [20] and regular string sanitation [15], amongst many other examples.
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 [5].

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 fragment expressed as an isomorphic embedding (i.e. as a library) because modern *module systems* can enforce abstraction barriers that ensure that the isomorphism needs only to 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 necessary to maintain the isomorphism need only be maintained by the functions in the module (e.g. uniqueness, if using a list representation). These then continue to hold no matter which other modules are in use by a client [16]. Other languages provide similar forms of *abstract data types* to localize reasoning [22].

Mechanisms are needed that make it possible to define and reason in a similarly modular, localized manner about direct extensions to the semantics of a language. For example, if a language is extended with *regular string types* as described in [15], all terms having a regular string type like $\textsc{rstr}\langle\texttt{/.+/}\rangle$ should continue to behave as non-empty strings no matter which other extensions are in use.

Contributions In this paper, we take foundational steps toward 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, e.g. LPROD defining labeled products or RSTR defining regular strings, determines the semantics of its associated opcons (e.g. **#** for row projection, or **conc** for concatenation) via *static functions*, i.e. functions written in a *static language* (SL), where types and translations are values.

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

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

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.

All translation invariants maintained by these opcons in any "closed world", e.g. the regular string invariant just mentioned, are guaranteed to be maintained in any further extended tycon context, i.e. in the "open world", due to a simple static check that maintains *translation independence* between tycons using type abstraction in the IL, the same fundamental mechanism underlying representation independence in ML-style module systems. As in ML, mechanized specifications and proofs are not needed.

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 and we will introduce various concrete desugarings in Sec. 4. The semantics are specified as a *bidirectionally typed translation semantics*, i.e. the key judgements have the form:

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

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*, $^+$. Note that the type is an "output" only for the synthetic judgement.

This separation of the EL from the IL is commonly used for full-scale language specifications, e.g. the Harper-Stone semantics for Standard ML [18] or the semantics of Wyvern [31]. The internal language is purposely kept small, e.g. defining only binary 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*, Φ, described starting in the Sec. 3.

We choose bidirectional typechecking, also sometimes called *local type inference* [32], 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 [29]) because it eliminates the need for type annotations in many situations while being simpler and decidable in more situations than whole-function type inference and providing what are widely perceived to be higher

external terms

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

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).

quality error messages [19]. Secondly, it gives us a clean way to reuse the generalized introductory form, $\mathsf{intro}[\sigma](\overline{e})$, and its associated desugarings, at many types [31]. For example, regular string types can 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, $\vdash_\Phi \sigma \rightsquigarrow \tau$, which will be described in Sec. 3.4. For example, regular string types will translate to internal type abbreviated `str`, and labeled product types will translate to nested binary product types, though we will emphasize that there are other valid choices.

This specification style may thus also be compared to specifications for the first stage of a type-directed compiler, e.g. the TIL compiler for Standard ML [40], 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 we also need the judgement $\vdash_\Phi \Upsilon \rightsquigarrow \Gamma$.

Internal Language @λ requires a typed internal language supporting type abstraction (i.e. universal quantification over types) [35]. We use $\mathcal{L}\{\rightharpoonup \forall \, \mu \, 1 \times +\}$, the syntax for which is shown in Figure 1, as representative of a typical 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 $\Delta \; \Gamma \vdash \iota : \tau^+$. The internal dynamics are specified as a structural operational semantics with a stepping judgement $\iota \mapsto \iota^+$ and a value judgement ι `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 ι' `val`. Both the static and dynamic semantics of the IL can be found in any standard textbook covering typed lambda calculi (we directly follow [17]), so we assume familiarity and omit the details.

Static Language The main novelty of @λ is the *static language*, which itself forms a typed lambda calculus where we use the word *kind*, κ, to refer to the "types" of *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 static terms, some of which are elided for concision, forms an entirely standard total functional programming language

kinds

$$
\begin{array}{lll}
\kappa & ::= & \kappa \rightarrow \kappa \mid \boldsymbol{\alpha} \mid \forall(\boldsymbol{\alpha}.\kappa) \mid \boldsymbol{k} \mid \mu_{\text{ind}}(\boldsymbol{k}.\kappa) \mid 1 \mid \kappa \times \kappa \mid \kappa + \kappa \\
 & \mid & \mathsf{Ty} \mid \mathsf{ITy} \mid \mathsf{ITm}
\end{array}
$$

static terms

$$
\begin{array}{lll}
\sigma & ::= & \boldsymbol{x} \mid \boldsymbol{\lambda x}{::}\kappa.\sigma \mid \sigma(\sigma) \mid \Lambda(\boldsymbol{\alpha}.\sigma) \mid \sigma[\kappa] \mid \cdots \mid \mathsf{raise}[\kappa] \\
 & \mid & c\langle\sigma\rangle \mid \mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma; \sigma) \\
 & \mid & \blacktriangleright(\hat{\tau}) \mid \rhd(\hat{\iota}) \mid \mathsf{ana}[n](\sigma) \mid \mathsf{syn}[n]
\end{array}
$$

translational internal types and terms

$$
\begin{array}{lll}
\hat{\tau} & ::= & \blacktriangleleft(\sigma) \mid \mathsf{trans}(\sigma) \mid \hat{\tau} \rightharpoonup \hat{\tau} \mid \ldots \\
\hat{\iota} & ::= & \lhd(\sigma) \mid \mathsf{anatrans}[n](\sigma) \mid \mathsf{syntrans}[n] \mid x \mid \lambda[\hat{\tau}](x.\hat{\iota}) \mid \ldots
\end{array}
$$

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.

consisting of total functions, universal quantification over kinds, inductive kinds, and products and sums [17]. The reader can consider these as forming a total subset of ML [16] or a simply-typed subset of Coq and we will assume standard conveniences from such languages (e.g. let bindings, pattern matching and inference of type parameters when applying polymorphic functions) in examples for concision. Only three new kinds are needed for the SL to serve its role as the language used to control typing and translation of the EL: Ty, classifying types (Sec. 3), ITy, classifying *translational internal types* (Sec. 3.4) and ITm, classifying *translational internal terms* (Sec. 4.1).

The kinding judgement takes the form $\boldsymbol{\Delta} \; \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa^+$, 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. All such contexts in @λ are identified up to exchange and contraction and obey weakening [17]. The natural number n is used as a technical device in our semantics to ensure that the forms shown as being indexed by n arise in a controlled manner 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 σ $\mathtt{val}_\mathcal{A}$ and an error raised judgement σ $\mathtt{err}_\mathcal{A}$. $\mathcal{A}$ 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 σ' $\mathtt{val}_\mathcal{A}$.

3. Types

External types, or simply *types*, are static values of kind Ty. We write σ $\mathtt{type}_\Phi$ iff $\emptyset \; \emptyset \vdash^0_\Phi \sigma :: \mathsf{Ty}$ and σ $\mathtt{val}_{\cdot;\emptyset;\Phi}$. The introductory form for kind Ty is $c\langle\sigma\rangle$, where c is a *tycon* and σ is the *type index*. The dynamics are simple: the index is eagerly normalized and errors propagate (see supplement for the complete set of rules in this paper).

(k-parr)
$$\frac{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty} \times \mathsf{Ty}}{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \rightharpoonup\langle\sigma\rangle :: \mathsf{Ty}}$$

(k-tc)
$$\frac{\mathsf{tycon}\ \textsc{tc}\ \{\theta\} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\} \in \Phi \qquad \boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa_{\text{tyidx}}}{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \textsc{tc}\langle\sigma\rangle :: \mathsf{Ty}}$$

(k-otherty)
$$\frac{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \sigma_{\text{tyidx}} :: \kappa \times \mathsf{ITy}}{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{other}[m;\kappa]\langle\sigma_{\text{tyidx}}\rangle :: \mathsf{Ty}}$$

Figure 4. 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.

tycons	c	$::=$	$\rightharpoonup \mid \textsc{tc} \mid \mathsf{other}[m;\kappa]$
tycon contexts	Φ	$::=$	$\cdot \mid \Phi, \mathsf{tycon}\ \textsc{tc}\ \{\theta\} \sim \psi$
tycon structures	θ	$::=$	$\mathsf{trans} = \sigma\ \mathsf{in}\ \omega$
opcon structures	ω	$::=$	$\mathsf{ana\ intro} = \sigma \mid \omega; \mathsf{syn}\ \mathbf{op} = \sigma$
tycon sigs	ψ	$::=$	$\mathsf{tcsig}[\kappa]\ \{\chi\}$
opcon sigs	χ	$::=$	$\mathsf{intro}[\kappa] \mid \chi; \mathbf{op}[\kappa]$

Figure 5. Syntax of tycons. Metavariables $\textsc{tc}$ and **op** range over extension tycon and opcon names, respectively, and m ranges over natural numbers.

$\Phi_{\text{rstr}} := \mathsf{tycon}\ \textsc{rstr}\ \{$
$\quad\mathsf{trans} = \sigma_{\text{rstr/trans}}\ \mathsf{in}$
$\quad\mathsf{ana\ intro} = \sigma_{\text{rstr/intro}};$
$\quad\mathsf{syn}\ \mathbf{conc} = \sigma_{\text{rstr/conc}};$
$\quad\mathsf{syn}\ \mathbf{case} = \sigma_{\text{rstr/case}};$
$\quad\cdots\} \sim \psi_{\text{rstr}}$

$\Phi_{\text{lprod}} := \mathsf{tycon}\ \textsc{lprod}\ \{$
$\quad\mathsf{trans} = \sigma_{\text{lprod/trans}}\ \mathsf{in}$
$\quad\mathsf{ana\ intro} = \sigma_{\text{lprod/intro}};$
$\quad\mathsf{syn}\ \# = \sigma_{\text{lprod/prj}};$
$\quad\mathsf{syn}\ \mathbf{conc} = \sigma_{\text{lprod/conc}};$
$\quad\cdots\} \sim \psi_{\text{lprod}}$

$\psi_{\text{rstr}} := \mathsf{tcsig}[\mathsf{Rx}]\ \{\mathsf{intro}[\mathsf{Str}]; \mathbf{conc}[1]; \mathbf{case}[\mathsf{StrPattern}]; \ldots\}$
$\psi_{\text{lprod}} := \mathsf{tcsig}[\mathsf{List}[\mathsf{Lbl}\times\mathsf{Ty}]]\ \{\mathsf{intro}[\mathsf{List}[\mathsf{Lbl}]]; \#[\mathsf{Lbl}]; \mathbf{conc}[1]; \ldots\}$

Figure 6. Example tycon contexts and signatures.

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

Figure 7. Tycon context well-definedness.

The syntax for tycons given in Figure 5 specifies that c can have one of three forms, so there are three corresponding kinding rules for types, shown in Figure 4.

Arrow Types 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* $\mathsf{Ty} \times \mathsf{Ty}$. We can introduce a desugaring from $\sigma_1 \rightharpoonup \sigma_2$ to $\rightharpoonup\langle(\sigma_1, \sigma_2)\rangle$.

Extension Types For types constructed by an extension tycon, written in small caps, $\textsc{tc}$, the rule (k-tc) extracts the index kind of $\textsc{tc}$ from the tycon context, Φ, which is simply a list of tycon definitions, $\mathsf{tycon}\ \textsc{tc}\ \{\theta\} \sim \psi$, where θ is the *tycon structure* and ψ is the *tycon signature*. Tycon signatures have the form $\mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\}$, where κ_{tyidx} is the tycon's index kind. We cover the remaining components of tycon definitions and signatures as we continue. Two examples of tycon contexts (each defining only one tycon)[2] and their signatures are shown in Figure 6.

Our first example is $\textsc{rstr}$. Its signature specifies that 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 terms that behave as *regular strings*, i.e. they are statically known to be in the regular language specified by the type index [15]. For example, $\sigma_{\text{title}} := \textsc{rstr}\langle\texttt{/.+/}\rangle$ will classify non-empty strings and $\sigma_{\text{conf}} := \textsc{rstr}\langle\texttt{/[A-Z]+ \textbackslash d\textbackslash d\textbackslash d\textbackslash d/}\rangle$, shown in Sec. 1, 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 [31].

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. In the tycon context containing both tycon definitions, $\Phi_{\text{rstr}}\Phi_{\text{lprod}}$, we can define a labeled product type classifying conference papers, $\sigma_{\text{paper}} := \textsc{lprod}\langle\{\texttt{title} : \sigma_{\text{title}}, \texttt{conf} : \sigma_{\text{conf}}\}\rangle$. Note that $\sigma_{\text{paper}}\ \texttt{type}_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}}$ and we again use kind-specific syntax, in this case for $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$.

Other Types The rule (k-otherty) governs types constructed by $\mathsf{other}[m;\kappa]$. 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 must pair a term of kind κ with a static term of kind ITy, discussed in Sec. 3.4.

3.1 Type Case Analysis

Types in @λ can be thought of as arising from a distinguished "open datatype" defined by the tycon context [23]. 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.

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

The rule for $c{=}\rightharpoonup$ is analagous, but, importantly, no rule for $c{=}\mathsf{other}[m;\kappa]$ is defined.

3.2 Tycon Context Well-Definedness

The tycon context well-definedness judgement, $\vdash \Phi$, requires that all tycon names are unique and performs three additional checks, described below (we omit (tcc-emp)).

[2] In examples, we omit leading $\emptyset$, used as the base case for finite mappings, and $\cdot$, used as the base case for finite sequences, for concision.

3.3 Type Equivalence

The first check simplifies type equivalence: type index kinds must be *equality kinds*, i.e. those for which semantically equivalent values are syntactically equal. We define these by the judgement $\mathbf{\Delta} \vdash \kappa\ \texttt{eq}$ (see supplement). Arrow kinds are not equality kinds, so type indices cannot contain static functions. Equality kinds are analagous to equality types as found in Standard ML [27].

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} must define 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 schema for RSTR will simply choose to ignore the type index and translate regular strings to strings, of internal type abbreviated str. We abbreviate the corresponding translational internal type $\hat{\mathsf{str}}$ and define $\sigma_{\text{rstr/trans}} := \lambda \boldsymbol{tyidx}{::}\mathsf{Rx}.\blacktriangleright(\hat{\mathsf{str}})$. The kinding and dynamics for shared forms proceed recursively, e.g.

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

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:

(k-ity-unquote)
$$\frac{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{ITy}}{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \blacktriangleright(\blacktriangleleft(\sigma)) :: \mathsf{ITy}}$$

(k-ity-trans)
$$\frac{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty}}{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \blacktriangleright(\mathsf{trans}(\sigma)) :: \mathsf{ITy}}$$

The unquote form is eliminated during normalization, while references to type translations are retained in values:

(s-ity-unquote-elim)
$$\frac{\blacktriangleright(\hat{\tau})\ \texttt{val}_{\mathcal{A}}}{\blacktriangleright(\blacktriangleleft(\blacktriangleright(\hat{\tau}))) \mapsto_{\mathcal{A}} \blacktriangleright(\hat{\tau})}$$

(s-ity-trans-val)
$$\frac{\sigma\ \texttt{val}_{\mathcal{A}}}{\blacktriangleright(\mathsf{trans}(\sigma))\ \texttt{val}_{\mathcal{A}}}$$

These forms are needed in the translation schema for LPROD, which generates nested binary product types (though we could also have used a list) by recursing over the type index and referring to the translations of the types therein. We assume the standard $\boldsymbol{listrec} :: \forall(\boldsymbol{\alpha}_1.\forall(\boldsymbol{\alpha}_2.\mathsf{List}[\boldsymbol{\alpha}_1] \to \boldsymbol{\alpha}_2 \to (\boldsymbol{\alpha}_1 \to \boldsymbol{\alpha}_2 \to \boldsymbol{\alpha}_2) \to \boldsymbol{\alpha}_2))$ in defining $\sigma_{\text{lprod/trans}} :=$

$$\lambda \boldsymbol{tyidx}{::}\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}].\boldsymbol{listrec}\ \boldsymbol{tyidx}\ \blacktriangleright(1)$$
$$(\lambda \boldsymbol{h}{:}\mathsf{Lbl} \times \mathsf{Ty}.\lambda \boldsymbol{r}{:}\mathsf{ITy}.\blacktriangleright(\mathsf{trans}(\mathsf{snd}(\boldsymbol{h})) \times \blacktriangleleft(\boldsymbol{r}))$$

Applying this translation schema to the index of σ_{paper}, for example, produces $\sigma_{\text{paper/trans}} := \blacktriangleright(\hat{\tau}_{\text{paper/trans}})$ where $\hat{\tau}_{\text{paper/trans}} := \mathsf{trans}(\sigma_{\text{title}}) \times (\mathsf{trans}(\sigma_{\text{conf}}) \times 1)$. Note that our schema did not remove the trailing unit type for simplicity (and to again emphasize that this choice is local).

3.4.1 Selective Type Translation Abstraction

References to type translations are maintained in values of kind ITy like this to allow us to selectively hold them abstract, which will be the key to translation independence. 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 variable, α. The *type translation store* $\mathcal{D} ::= \emptyset \mid \mathcal{D}, \sigma \leftrightarrow \tau/\alpha$ maintains this correspondence between types, their actual translations and the distinct type variables which appear in their place, e.g. $\hat{\tau}_{\text{paper/trans}} \parallel \emptyset \hookrightarrow^{\text{LPROD}}_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \tau_{\text{paper/abs}} \parallel \mathcal{D}_{\text{paper/abs}}$ where $\tau_{\text{paper/abs}} := \alpha_1 \times (\alpha_2 \times 1)$ and $\mathcal{D}_{\text{paper/abs}} := \sigma_{\text{title}} \leftrightarrow \mathsf{str}/\alpha_1, \sigma_{\text{conf}} \leftrightarrow \mathsf{str}/\alpha_2$.

The judgement $\mathcal{D} \rightsquigarrow \delta : \Delta$ constructs the n-ary *type substitution*, $\delta ::= \emptyset \mid \delta, \tau/\alpha$, and corresponding internal type formation context, Δ, implied by the type translation store $\mathcal{D}$. 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:

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

The rules for the selective type translation abstraction judgement recurse generically over shared forms in $\hat{\tau}$. Only sub-terms of form $\mathsf{trans}(\sigma)$ are interesting. The translations of function types are not held abstract, so that lambdas can serve 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 an extension type constructed by the delegated tycon is determined by calling the translation schema and checking that the type translation it generates is closed except for type variables tracked by $\mathcal{D}'$:

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

The translation of an extension type constructed by any tycon other than the delegated tycon is held abstract via a fresh type variable added to the store (the supplement has the rule (abs-stored) for retrieving it once there):

(abs-tc-not-delegated-new)

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

The translation of an "other" type is given in its index (rule (abs-other-not-delegated-new) is in the supplement):

(abs-other-delegated)

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

4. External Terms

Having established how types are constructed, and how they determine selectively abstracted and from there actual type translations, we can finally give the typing and translation rules for external terms, shown in Figure 8.

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). We use Plotkin's fixpoint operator for general recursion (cf. [17]), and define both lambdas and fixpoints only analytically for simplicity.

4.1 Generalized Introductory Operations

The translation of the *generalized intro operation*, written $\text{intro}[\sigma_{\text{tmidx}}](\overline{e})$, 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, $\texttt{"s"}$, which desugars to $\text{intro}[\texttt{"s"}_{\text{SL}}](\cdot)$, 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 $\text{intro}[[\mathtt{lbl}_1, \ldots, \mathtt{lbl}_n]](e_1; \ldots; e_n)$, 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 [31] 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 List[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 is 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 9. For example, $\sigma_{\text{rstr/intro}} :=$

λ***tyidx***::Rx.λ***tmidx***::Str.λ***args***::List[Arg].
let ***aok*** :: 1 = ***arity0 args*** in
let ***rok*** :: 1 = ***rmatch tyidx tmidx*** in ***str_of_Str tmidx***

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) with the next example), using the helper function ***arity0*** :: List[Arg] $\rightarrow$ 1 defined such that any non-empty list will raise an error, via the static term 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*** :: Rx $\rightarrow$ Str $\rightarrow$ 1, which we assume is defined in the usual way and again raises an error on failure. Finally, the *translational internal string* corresponding to the static string provided as the term index is generated via the helper function ***str_of_Str*** :: Str $\rightarrow$ 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

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

$$\text{(subsume)}\ \frac{\Upsilon \vdash_\Phi e \Rightarrow \sigma \leadsto \iota}{\Upsilon \vdash_\Phi e \Leftarrow \sigma \leadsto \iota}
\qquad
\text{(ascribe)}\ \frac{\begin{array}{c}\emptyset\,\emptyset \vdash^0_\Phi \sigma :: \mathsf{Ty} \qquad \sigma \Downarrow_{\cdot;\emptyset;\Phi} \sigma' \\ \Upsilon \vdash_\Phi e \Leftarrow \sigma' \leadsto \iota\end{array}}{\Upsilon \vdash_\Phi e : \sigma \Rightarrow \sigma' \leadsto \iota}$$

$$\text{(syn-var)}\ \frac{x \Rightarrow \sigma \in \Upsilon}{\Upsilon \vdash_\Phi x \Rightarrow \sigma \leadsto x}
\qquad
\text{(ana-fix)}\ \frac{\begin{array}{c}\Upsilon, x \Rightarrow \sigma \vdash_\Phi e \Leftarrow \sigma \leadsto \iota \\ \vdash_\Phi \sigma \leadsto \tau\end{array}}{\Upsilon \vdash_\Phi \mathsf{fix}(x.e) \Leftarrow \sigma \leadsto \mathsf{fix}[\tau](x.\iota)}$$

$$\text{(ana-lam)}\ \frac{\Upsilon, x \Rightarrow \sigma_1 \vdash_\Phi e \Leftarrow \sigma_2 \leadsto \iota \qquad \vdash_\Phi \sigma_1 \leadsto \tau_1}{\Upsilon \vdash_\Phi \lambda(x.e) \Leftarrow \rightharpoonup\langle(\sigma_1, \sigma_2)\rangle \leadsto \lambda[\tau_1](x.\iota)}
\qquad
\text{(syn-ap)}\ \frac{\Upsilon \vdash_\Phi e_1 \Rightarrow \rightharpoonup\langle(\sigma_1, \sigma_2)\rangle \leadsto \iota_1 \qquad \Upsilon \vdash_\Phi e_2 \Leftarrow \sigma_2 \leadsto \iota_2}{\Upsilon \vdash_\Phi e_1(e_2) \Rightarrow \sigma_2 \leadsto \iota_1(\iota_2)}$$

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

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

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

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

Figure 8. Typing

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

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

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

Figure 9. Opcon structure kinding against tycon signatures

```
λtyidx:List[Lbl × Ty].λtmidx:List[Lbl].λargs:List[Arg].
  let inhabited : 1 = uniqmap tyidx in
  listrec3 [Lbl × Ty] [Lbl] [Arg] [ITm] tyidx tmidx args ▷(())
    λrowtyidx:Lbl × Ty.λrowtmidx:Lbl.λrowarg:Arg.λr:ITm.
      letpair (rowlbl, rowty) = rowtyidx in
      let lok :: 1 = lbleq rowlbl rowtmidx in
      let rowtr :: ITm = ana rowarg rowty in
      ▷((◁(rowtr), ◁(r)))
```

Figure 10. The intro opcon definition for LPROD.

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, shown in Figure 10.

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. An analagous technique could be used to implement record types by requiring that the index be sorted (see supplement).

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, ***listrec3***, 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 triggers type analysis of the provided argument, producing a translational internal term of the form $\triangleright(\mathsf{anatrans}[n](\sigma))$, where n is the position of ***rowarg*** in ***args*** and σ is the value of ***rowty***, upon success or raising 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 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 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* is a static list of length n where the ith entry is $(\lambda \boldsymbol{ty}::\mathsf{Ty}.\mathsf{ana}[i](\boldsymbol{ty}), \lambda\_::1.\mathsf{syn}[i])$. It is generated by the judgement $\mathsf{args}(n) = \sigma_{\text{args}}$, where n is the length of the argument list, written $|\overline{e}| = n$.

Recall that the kinding judgement is indexed by n. This 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)
$$\frac{n' < n}{\boldsymbol{\Delta}\,\boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{syn}[n'] :: \mathsf{Ty} \times \mathsf{ITm}}$$

Thus, if $\mathsf{args}(n) = \sigma_{\text{args}}$ then $\emptyset\,\emptyset \vdash^n_\Phi \sigma_{\text{args}} :: \mathsf{List}[\mathsf{Arg}]$.

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, only accessing them via the argument interface list, which is guaranteed to have the correct length.

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 normalizing σ, the argument environment, which contains the arguments themselves and the typing and tycon contexts, $\mathcal{A} ::= \overline{e}; \Upsilon; \Phi$, 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:

(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 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 argument translations abstract by replacing them with variables. The judgement $\hat{\iota} \parallel \mathcal{D}\,\mathcal{G} \hookrightarrow^c_{\mathcal{A}} \iota^+ \parallel \mathcal{D}^+\,\mathcal{G}^+$, appearing as the eighth premise of (ana-intro), relates a translational internal term $\hat{\iota}$ to an internal term ι called a *selectively abstracted term translation*, because all references to the translation of an argument (having any type) are replaced with a corresponding variable and, as in Sec. 3.4.1, all references to the translation of a type constructed by an extension tycon other than the "delegated tycon" c are replaced with a corresponding abstract type variable. The type translation store, $\mathcal{D}$, discussed previously, and term translation store, $\mathcal{G}$, track these correspondences. Term translation stores have 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", e.g.

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

where $\iota_{\text{ex/abs}} := (x_0, (x_1, ()))$ and $\mathcal{G}_{\text{ex/abs}} := 0 : \sigma_{\text{title}} \rightsquigarrow title/x_0 : \alpha_0, 1 : \sigma_{\text{conf}} \rightsquigarrow \texttt{"EXMPL 2015"}_{\text{IL}}/x_1 : \alpha_1$ and $\mathcal{D}_{\text{ex/abs}}$ is $\mathcal{D}_{\text{paper/abs}}$ from Sec. 3.4.1.

This judgement proceeds recursively along shared forms, like the selectively abstracted type translation judgement in Sec. 3.4.1. The key rule for references to argument translations derived via analysis is below ($\mathsf{syntrans}[n]$ is analagous; the full rules are in the supplement). Note that we are rederiving the translation already determined in (s-ana-success) for simplicity (in practice, this might be cached):

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

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 $\mathcal{G} \rightsquigarrow \gamma : \Gamma$, 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 a selectively abstracted type translation for the type provided for analysis, also using the tycon of the type provided for analysis as the delegated tycon and starting with the same store to ensure that equal types have equal type variables. 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). 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 the references to term and type translations of regular strings replaced by variables and type variables, respectively. But because our definition treated arguments parametrically, the check succeeds.

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}, defined previously. Note that ι_{ex} has type τ_{paper} under the translation of Υ_{ex}, i.e. $\vdash \Upsilon_{\text{ex}} \leadsto \Gamma_{\text{ex}}$ where $\Gamma_{\text{ex}} := title : \mathsf{str}$.

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, it is analagous. It is used as a technical device in Sec. 5.

4.2 Generalized Targeted Operations

All non-introductory opcons associated with extension 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.

Concrete desugarings include $e_{\text{targ}}.\mathbf{op}\langle\sigma_{\text{tmidx}}\rangle(\overline{e})$ (and variants where the term index or arguments are omitted), projection syntax for use by record-like types, $e_{\text{targ}}\#\mathtt{lbl}$, which desugars to $\mathsf{targ}[\mathbf{prj}; \mathtt{lbl}](e_{\text{targ}}; \cdot)$, and $e_{\text{targ}} \cdot e_{\text{arg}}$, 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 @λ. 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 it.

Figure 6 showed portions of the opcon signatures of RSTR and LPROD. The opcons associated with RSTR are taken directly from Fulton et al.'s specification of regular string types [15], 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 rows with the value from the right argument), and **drop** creates a new labeled product from the target with some rows dropped. Note that both RSTR and LPROD 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 9 ensures that no tycon defines an opcon twice and that the opcon definitions are well-kinded. For example, $\sigma_{\text{rstr/conc}} =$

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

The helper function ***arity2*** checks that two arguments, including the target argument, were provided. 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.

The remaining premises of (syn-targ) are analagous to the corresponding premises in (ana-intro), with the only difference being that we check the selectively abstracted term translation against the selectively abstracted type translation of the synthesized type, but the delegated tycon is that of the type synthesized by the target argument.

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 mutually defined.

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 [17] then implies that evaluation cannot go wrong. To prove this, we must prove a stronger theorem, *type-preserving translation*, analagous to type-preserving compilation in the typed compilation literature [40]:

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

Proof Sketch. 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 $\mathcal{D} \rightsquigarrow \delta : \Delta_{\text{abs}}$ and $\mathcal{G} \rightsquigarrow \gamma : \Gamma_{\text{abs}}$ then $\emptyset \vdash \delta : \Delta_{\text{abs}}$ and $\Delta_{\text{abs}}\, \Gamma \vdash \gamma : \Gamma_{\text{abs}}$, i.e. that all variables in Δ_{abs} and Γ_{abs} have well-formed/well-typed substitutions in δ and γ, and so applying them 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 earlier 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.

if space, bring unicity back

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

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

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* $\emptyset \vdash_{\Phi_{rstr}} e \Leftarrow \textsc{rstr}\langle /r/ \rangle \rightsquigarrow \iota$ *and* $\iota \Downarrow \iota'$ *then* $\iota' =$ `"s"` *and* `"s"` *is in the regular language* $\mathcal{L}(r)$.

Proof Sketch. The proof is not unusually difficult because we have fixed the tycon context Φ_{rstr}, so we can essentially treat the calculus like a type-directed compiler for a calculus with only two tycons, $\rightharpoonup$ and $\textsc{rstr}$, plus one "other" one. Such a calculus and compiler specification was given in [15], so we must simply show that the opcon definitions in $\textsc{rstr}$ adequately satisfy these specification using standard techniques for the SL, a simply-typed functional language [9]. The only "twist" is that the rule (syn-targ-other) can synthesize a regular string type. But if so, ι_{abs} will be checked against $\tau_{\text{abs}} = \alpha$. Thus, the invariants cannot be violated by direct application of relational parametricity in the IL (i.e. a "free theorem") [41]. □

The reason why (syn-targ-other) is never a problem in proving a tycon invariant – *translation independence* of tycons – 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* $\Upsilon \vdash_\Phi e \Leftarrow \textsc{tc}\langle\sigma_{tyidx}\rangle \rightsquigarrow \iota$ *and* $\vdash_\Phi \Upsilon \rightsquigarrow \Gamma$ *and* $\vdash_\Phi \textsc{tc}\langle\sigma_{tyidx}\rangle \rightsquigarrow \tau$ *then* $P(\Gamma, \sigma_{tyidx}, \iota)$.

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

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

(if proposition $P(\Gamma, \sigma_{tyidx}, \iota)$ *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 new term has a type constructed by that tycon with the same type translation, and only a slightly different type index. In particular, the mapping's effect on static terms is to replace all types constructed by $\textsc{tc}'$ with a type constructed by $\text{other}[m; \kappa'_{\text{tyidx}}]$ for some m corresponding to $\textsc{tc}'$ and the index kind of $\textsc{tc}'$. If $P(\Gamma, \sigma_{\text{tyidx}}, \iota)$ is preserved under this transformation then we can simply invoke the existing proof of the tycon invariant. We call such propositions *modular*. Non-modular propositions are uninteresting because they have knowledge of tycons "from the future".

On external terms, the mapping replaces all intro and targeted 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 selective translation abstraction are preserved under the mapping, defined inductively in the supplement. Note that for this reason all propositions decidable by the SL are modular. □

6. Related Work and Discussion

Language-integrated static term rewriting systems, like Template Haskell [39] and Scala's static macros [7], can be used to decrease complexity when an isomorphic embedding into the underlying type system is possible. Similarly, when an embedding that preserves a desired static semantics exists, but a different embedding preserves the cost semantics, term rewriting can also be used to perform "optimizations", achieving an isomorphism. Care is needed when this changes the type of a term. Type abstraction has been used for this purpose in *lightweight modular staging* (LMS) [36]. In both cases, the type system is fixed (e.g. in LMS, Scala's).

When new static distinctions are needed within an existing type, but new operators are not necessary, one solution is to develop an overlying system of *refinement types* [14]. For example, a refinement of integers might distinguish negative integers. Proposals for "pluggable type systems" describe composing such systems [3, 6]. Refinements of abstract types can be used for representation independence, but note that the type being refined is not held abstract. Were it to be, the system could be seen in some ways as a degenerate mode of use of our work: we further cover the cases when new operators are needed. For example, the LPROD opcons use labels and types, which exist only statically. Thus, labeled tuples cannot be seen as refinements of nested pairs because the row projection operators don't exist.

Many *language frameworks* exist that can simplify dialect implementation (cf. [13]). These sometimes do not support forming languages from fragments due to the "expression problem" (EP) [34, 42]. We sidestep the most serious consequences of the EP by leaving our abstract syntax entirely fixed, instead delegating to tycons. Fewer tools require knowledge of all external tycons in a typed translation semantics. As discussed previously, our treatment of concrete syntax in both the EL and SL defers to recent work on *type-specific languages*, which takes a similar bidirectional approach for composably introducing syntactic desugarings [31]. Some language frameworks do address the EP, e.g. by encoding terms and types as open datatypes [23], but this makes it quite difficult to reason modularly, particularly about metatheoretic properties specific to typed languages, like type safety and tycon invariants. Our key insight is to instead associate term-level opcons with tycons, which then become the fundamental constituents of the semantics (consistent with Harper's informal notation [17]).

Proof assistants can be used to specify and mechanize the metatheory of languages, but also usually require a complete specification (this has been identified as a key challenge [4]). Techniques for composing specifications and proofs exist [11, 12, 38], but they require additional proofs at composition-time and provide no guarantees that *all* fragments having some modularly checkable property can safely and conservatively be composed, as in our work. The authors, along with Chlipala [10], suggest proof automation. This is fundamentally a heuristic approach.

In contrast, with $@\lambda$ fragment providers need not provide the semantics with mechanized proofs to benefit from modular reasoning principles. Instead, under a fixed tycon context, the calculus can be reasoned about like a very small type-directed compiler [9, 40]. Errors in reasoning can only lead to failure at typechecking time, via a novel form of *translation validation* [33]. Incorrect opcon definitions (relative to a specification, e.g. [15] for regular strings) can at worst weaken expected invariants at that tycon, like incorrectly implemented modules in ML. Thus, modular tycons can reasonably be tested "in the field" without concern about the reliability of the system as a whole. To eliminate even these localized failure modes for "reliability-critical" tycons, we plan to introduce *optional* proof mechanization into the SL (by basing it on a dependently typed language like Coq).

Type abstraction, encouragingly, also underlies modular reasoning in ML-like languages [16, 17] and languages with other forms of ADTs [22] like Scala [2]. Indeed, proofs of tycon invariants can rely on existing parametricity theorems [41]. Our work is reminiscent of work on elaborating an ML-style module system into System $\mathbf{F}_\omega$ [37]. Unlike in module systems, type translations (analagous to the choice of representation for an abstract type) are statically *computed* based on a type index, rather than statically *declared*. Moreover, there can be arbitrarily many operators because they arise by providing a term index to an opcon, and their semantics can be complex because a static function computes the types and translations that arise. In contrast, modules and ADTs can only specify a fixed number of operations, and each must have function type. Note however that these are not competing mechanisms: we did not specify quantification over external types here for simplicity, but we conjecture that it is complementary and thus $@\lambda$ could serve as the core of a language with an ML-style module system. Another related direction is *tycon functors*, which would abstract over tycons with the same signature to support tunable cost semantics.

A limitation of our approach is that it supports only fragments with the standard "shape" of typing judgement. Fragments that require new forms of scoped contexts (e.g. symbol contexts [17]) or unscoped declarations cannot presently be defined. Relatedly, the language controls variable binding, so, for example, linear type systems, cannot be defined. Another limitation is that opcons cannot directly invoke one another (e.g. a **len** opcon on regular strings could not construct a natural number). We conjecture that these are not fundamental limitations and expect $@\lambda$ to serve as a minimal foundation for future efforts that increase expressiveness while maintaining the strong guarantees, like type safety and conservativity, established here.

References

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

[2] N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 233–249. ACM, 2014.

[3] 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.

[4] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer, 2005.

[5] 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.

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

[7] 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.

[8] 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.

[9] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 54–65. ACM, 2007.

[10] 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.

[11] B. Delaware, W. R. Cook, and D. S. Batory. Product lines of theorems. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (26th OOPSLA'11)*, pages 595–608, Portland, Oregon, USA, Oct. 2011. ACM Press.

[12] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Meta-theory à la carte. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 207–218. ACM, 2013.

[13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *Software Language Engineering*. 2013.

[14] 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.

[15] 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.

[16] R. Harper. Programming in standard ml, 1997.

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

[18] 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.

[19] S. L. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program*, 17(1):1–82, 2007.

[20] 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.

[21] 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.

[22] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Sigplan Notices*, volume 9, pages 50–59. ACM, 1974.

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

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

[25] 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.

[26] 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.

[27] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Aug. 1990.

[28] 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.

[29] Odersky, Zenger, and Zenger. Colored local type inference. In *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.

[30] 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.

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

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

[33] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998.

[34] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *Conference on New Directions on Algorithmic Languages*, Aug. 1975.

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

[36] 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.

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

[38] C. Schwaab and J. G. Siek. Modular type-safety proofs in agda. In M. Might, D. V. Horn, A. A. 0001, and T. Sheard, editors, *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 3–12. ACM, 2013.

[39] 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.

[40] 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.

[41] P. Wadler. Theorems for free! In *Proc. ACM Conf. Functional Programming and Computer Architecture*, pages 347–359, 1989.

[42] P. Wadler. The expression problem. *java-genericity mailing list*, 1998.