Modular Type Constructors

Conservatively Composing Typed Language Fragments

Cyrus Omar and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15213, USA
{comar,aldrich}@cs.cmu.edu

Abstract. Researchers commonly describe typed programming languages in fragments or as simple calculi, leaving to language designers the task of composing these to form complete languages. Metatheoretic results must be established monolithically for each such language, guided only informally by the results derived for simpler calculi. We argue that, as the language design space grows, mechanisms for defining and composing typed language fragments that provide strong modular reasoning principles are needed.

In this paper, we begin from first principles with a minimal calculus, $@\lambda$, specified as a bidirectionally typed translation semantics: external terms are simultaneously typechecked and given a translation to a fixed typed internal language. The external language is made extensible by indexing the typing judgement by a *tycon context*. Each tycon defines the semantics of its associated operators using a fixed static language where types are values. This language is constructed to ensure that several strong semantic guarantees follow from reasoning that can be performed in the "closed world" of a fixed tycon context, notably *type safety* and *conservativity*: that the *type invariants* maintained under a minimal tycon context will be conserved under any further extended context. Violations are caught during typechecking by lifting typed compilation techniques into the semantics and enforcing an abstraction barrier around each tycon using type abstraction, so these *modular type constructors* can be reasoned about modularly, i.e. like modules in an ML-like language.

1 Introduction

Typed programming languages are often described in *fragments*, each defining distinct contributions to a language's concrete and abstract syntax, static semantics and dynamic semantics. For example, in his textbook, Harper organizes fragments around type constructors, describing each in a different chapter [6]. Complete languages are then identified by a set of type constructors, e.g. $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1\, \times\, +\}$ is the language that builds in partial function types, polymorphic types, recursive types, nullary and binary product types and binary sum types (its syntax is shown in Figure 1, discussed further below). Another common pattern is for a researcher to describe a new fragment by defining a complete calculus, perhaps including a "catch-all" constant and base type to stand for all other terms and types that may be of interest.

Metatheoretic reasoning techniques for programming languages generally assume that a complete specification of a language is being considered, however. To use these

techniques, every *combination* of fragments must be treated as its own monolithic language for which metatheorems must be established anew, guided only informally by results derived for the smaller systems from which they are, notionally, composed.

Luckily, fragment composition is not an everyday programming task because fragments like these are "general purpose" in that they make it possible to construct *isomorphic embeddings* of many other fragments as "libraries". For example, lists need not be built in "primitively" because they can be placed in isomorphism with the polymorphic recursive sum of products $\forall(\alpha.\mu(t.1 + (\alpha \times t)))$. Languages providing datatypes alá ML are perhaps most directly oriented around such embeddings.

Although these fragments certainly appear to occupy a "sweet spot" in the design space, "general purpose" does not mean "all-purpose" and situations do continue to arise where using these fragments to establish an isomorphic embedding that preserves a desirable fragment's static and dynamic semantics (including bounds specified by a cost semantics) is not possible. Embeddings can also sometimes be unsatisfyingly *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 cognitive factors. Each time a researcher seeks to address one of these problems by designing a new language construct, a new *dialect* of the language is born. Within the ML lineage, dialects that go beyond core ML 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), structurally typed or row polymorphic records [3], records with update and extension operators [9], mutable fields [9], and "methods" (i.e. pure objects [16]).[1] Sum-like types are also exposed in various ways: finite datatypes, open datatypes [10], hierarchically open datatypes [13], polymorphic variants [9] and ML-style exception types. Other generally useful fragments are also built in, e.g. `sprintf` in the OCaml dialect statically distinguishes format strings from strings.
2. **Specialized Fragments:** Fragments that track specialized static invariants to provide stronger correctness guarantees, manage unwieldy lower-level abstractions or control cost are also often introduced in dialects, e.g. for data parallelism [4], distributed programming [14], reactive programming [11], authenticated data structures [12], databases [15], units of measure [8] and string sanitation [5].
3. **Foreign Fragments:** A safe and natural foreign function interface (FFI) can be a valuable feature (particularly given this proliferation of dialects). However, this requires enforcing the type system of the foreign language in the calling language. For example, MLj builds in a safe FFI to Java [2].

This *dialect-oriented* state of affairs is unsatisfying for language designers, fragment providers and programmers alike. Language designers are burdened with needing to understand the complete metatheory of every fragment they add to their language to make sure it is not incompatible with existing fragments, so languages

[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]

internal types $\tau ::= \tau \rightharpoonup \tau \mid \alpha \mid \forall(\alpha.\tau) \mid t \mid \mu(t.\tau) \mid 1 \mid \tau \times \tau \mid \tau + \tau$
internal terms $\iota ::= x \mid \lambda[\tau](x.\iota) \mid \iota(\iota) \mid \mathsf{fix}[\tau](x.\iota) \mid \Lambda(\alpha.\iota) \mid \iota[\tau] \mid \mathsf{fold}[t.\tau](\iota) \mid \mathsf{unfold}(\iota)$
$\mid \ () \mid (\iota, \iota) \mid \mathsf{letpair}(\iota; x, y.\iota) \mid \mathsf{inl}[\tau](\iota) \mid \mathsf{inr}[\tau](\iota) \mid \mathsf{case}(\iota; x.\iota; x.\iota)$
internal typing contexts $\Gamma ::= \emptyset \mid \Gamma, x : \tau$
internal type formation contexts $\Delta ::= \emptyset \mid \Delta, \alpha \mid \Delta, t$

Fig. 1. Syntax of $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1\, \times\, +\}$, our internal language (IL).

change rarely. This decreases the impact of many potentially useful innovations designed by fragment providers (e.g. many academic researchers) by making them unavailable to programmers. At best, programmers can choose from either a dialect supporting, for example, a principled approach to distributed programming, or one that builds in support for statically reasoning about units of measure. There may not be an available dialect supporting both. Using different dialects separately for different components of a program is untenable: components written in different dialects cannot always interface safely (an FFI, above, is needed).

These problems do not arise when a fragment can be expressed as an isomorphic embedding (i.e. as a library) because modern *module systems* enforce abstraction barriers that ensure that the isomorphism need only be established in the "closed world" of the module. This is useful because it does not impose proof obligations on clients in the "open world". For example, a module defining the semantics of sets in ML can hold the representation of sets abstract, ensuring that any invariants maintained by the functions in the module (e.g. uniqueness) will hold even when other modules are in use.

Because embedding of this form are not always possible, as in the examples enumerated above, so mechanisms are needed that make it possible to modularly define new fragments that more directly influence the static and dynamic semantics of a language. Such mechanisms could ultimately be integrated directly into a language, blurring the distinction between fragments and libraries and decreasing the need for new dialects. Importing fragments that introduce new syntax and semantics would be as safe and easy as importing a new module is today.

Contributions In this paper, we take substantial steps towards this goal by constructing a minimal but powerful core calculus, @λ (the "actively typed" lambda calculus). This calculus is structured in a manner similar to the Harper-Stone semantics for Standard ML [7], consisting of an *external language* (EL) governed by a (bidirectionally) typed translation semantics targeting a fixed *internal language* (IL). Rather than building in a monolithic set of external type constructors, however, the typechecking judgement is indexed by a *tycon context*. Each tycon defines the semantics of its associated operators via functions written in a *static language* (SL). Types are values in the SL.

The bulk of the paper, in Sec. 2, introduces @λ by building up two 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 [5]. We then examine the key metatheoretic properties of the calculus in Sec. 3, beginning with *type safety* and *unicity of typing*, touching on *decidability of typechecking* and finally considering properties that relate to composition of tycon definitions: *hygiene*, *stability of typing* and a key modularity result, which we refer to as *conservativity*: any invariants that

external terms $e ::= x \mid \lambda(x.e) \mid e(e) \mid \mathsf{fix}(x.e) \mid e : \sigma \mid \mathsf{intro}[\sigma](\overline{e}) \mid \mathsf{targ}[\mathbf{op}; \sigma](e; \overline{e}) \mid \mathsf{other}[\iota]$
op arguments $\overline{e} ::= \cdot \mid \overline{e}, e$
external typing contexts $\Upsilon ::= \emptyset \mid \Upsilon, x \Rightarrow \sigma$

Fig. 2. Syntax of the external language (EL). We write x to range over variables and **op** over operator names. The form $\mathsf{other}[\iota]$ is a technical device (and would have no concrete syntax).

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, the approach we take to guarantee conservativity relies on type abstraction, in this case in the internal language. As a result, we are able to borrow the same results that underly modular reasoning in languages like ML to reason about extensions to the semantics itself. Finally, we describe related work in Sec. 4.

2 @λ

We will begin by giving an overview of the basic organization of @λ and introduce the main judgements in Section 2.1, discuss how types are constructed and tycons are defined in Section 2.2, then describe how the semantics of external terms are controlled by tycons in Section 2.3.

2.1 Overview

External Language Programs in @λ are written as *external terms*, e. The syntax of external terms is shown in Figure 2. The static and dynamic semantics of external terms are specified simultaneously as a *bidirectionally typed translation semantics*. The two central judgements in this paper 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". 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.

Bidirectional typechecking, also sometimes called *local type inference* [17], is increasingly being used in modern languages (e.g. Scala) because it eliminates the need for type annotations in many circumstances while remaining decidable in more situations than Hindley-Milner style inference and providing what are widely perceived to be higher quality error messages, due to the locality of reasoning. It will also give us a clean way to reuse a fixed set of introductory forms, discussed in Sec. 2.3.

Internal Language Internal terms, ι, together with internal types, τ, form the *typed internal language*. @λ relies on a typed internal language supporting type abstraction (i.e. universal quantification over types). In this paper, we use $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$, the syntax for which is shown in Figure 1. This is representative of a standard intermediate

$$
\begin{array}{lll}
\textbf{kinds} & \kappa ::= & \kappa \to \kappa \mid \boldsymbol{\alpha} \mid \forall(\boldsymbol{\alpha}.\kappa) \mid \boldsymbol{k} \mid \mu_{\text{ind}}(\boldsymbol{k}.\kappa) \mid 1 \mid \kappa \times \kappa \mid \kappa + \kappa \mid \mathsf{Ty} \mid \mathsf{ITy} \mid \mathsf{ITm} \\
\textbf{static terms} & \sigma ::= & \boldsymbol{x} \mid \lambda \boldsymbol{x}{::}\kappa.\sigma \mid \sigma(\sigma) \mid \Lambda(\boldsymbol{\alpha}.\sigma) \mid \sigma\,[\kappa] \mid \mathsf{fold}[\boldsymbol{k}.\kappa](\sigma) \mid \mathsf{rec}[\kappa](\sigma; \boldsymbol{x}.\sigma) \\
& & \mid\ () \mid (\sigma, \sigma) \mid \mathsf{letpair}(\sigma; \boldsymbol{x}, \boldsymbol{y}.\sigma) \mid \mathsf{inl}[\kappa](\sigma) \mid \mathsf{inr}[\kappa](\sigma) \mid \mathsf{case}(\sigma; \boldsymbol{x}.\sigma; \boldsymbol{x}.\sigma) \\
& & \mid\ c\langle\sigma\rangle \mid \mathsf{otherty}[m; \tau] \mid \mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma; \sigma) \\
& & \mid\ \blacktriangleright(\hat{\tau}) \mid \triangleright(\hat{\iota}) \mid \mathsf{ana}[n](\sigma) \mid \mathsf{syn}[n] \mid \mathsf{raise}[\kappa] \\
& m, n ::= & 0 \mid n + 1 \\
\textbf{trans. IL} & \hat{\tau} ::= & \blacktriangleleft(\sigma) \mid \mathsf{trans}(\sigma) \mid \hat{\tau} \rightharpoonup \hat{\tau} \mid \ldots \\
& \hat{\iota} ::= & \triangleleft(\sigma) \mid \mathsf{anatrans}[n](\sigma) \mid \mathsf{syntrans}[n] \mid x \mid \lambda[\hat{\tau}](x.\hat{\iota}) \mid \ldots \\
\multicolumn{3}{l}{\textbf{kinding contexts } \boldsymbol{\Gamma} ::= \emptyset \mid \boldsymbol{\Gamma}, \boldsymbol{x} :: \kappa} \\
\multicolumn{3}{l}{\textbf{kind formation contexts } \boldsymbol{\Delta} ::= \emptyset \mid \boldsymbol{\Delta}, \boldsymbol{\alpha} \mid \boldsymbol{\Delta}, \boldsymbol{k}} \\
\multicolumn{3}{l}{\textbf{argument environments } \mathcal{A} ::= \overline{e}; \Upsilon; \Phi}
\end{array}
$$

Fig. 3. Syntax of the static language (SL).

language as might be found for a language like ML. The IL has a fixed semantics, so different choices of IL represent different dialects of @λ.

We assume the internal statics are specified in the standard way by judgements for internal type formation $\Delta \vdash \tau$, typing context formation $\Delta \vdash \Gamma$ and type assignment $\Delta\ \Gamma\ \vdash\ \iota\ :\ \tau^+$. The typing and type formation contexts obey standard structural properties (i.e. weakening, exchange, contraction). Throughout the paper, we will omit leading $\emptyset$ (used as the base case for finite mappings) and $\cdot$ (used as the base case for finite sequences) in examples (e.g. writing $\Gamma_1 := x : \tau$ rather than $\Gamma_1 := \emptyset, x : \tau$).

The internal dynamics are also a standard structural operational semantics with stepping judgement $\iota \mapsto \iota^+$ and value judgement $\iota\ \mathtt{val}$. The judgement $\iota \mapsto^*\ \iota^+$ is the reflexive, transitive closure of the stepping judgement. The semantics of the IL can be found in any standard textbook covering typed lambda calculi (we closely follow [6]), so we assume familiarity and omit the details.

Static Language The workhorse of our calculus 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. We note that the core of the SL is a total functional programming language based on several standard fragments: total functions, quantification over kinds, inductive kinds (constrained by a positivity condition to prevent non-termination), and products and sums. These all also closely follow [6], so we again omit some details. Only three new kinds are introduced: Ty, ITy and ITm. We will describe these shortly. We write static term variables and kind variables in bold for clarity.

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 well-formedness judgements $\boldsymbol{\Delta} \vdash \kappa$ and $\boldsymbol{\Delta} \vdash \boldsymbol{\Gamma}$ are also defined. The natural number n is used as a technical device in our semantics to prevent certain forms (those indexed by n in the syntax above) from arising except internally in the semantics (they would have no corresponding concrete syntax); n can be assumed 0 in all user-defined terms. The dynamic semantics of static terms is defined as a structural operational semantics by the stepping judgement $\sigma \mapsto_{\mathcal{A}} \sigma^+$, the value judgement $\sigma\ \mathtt{val}_{\mathcal{A}}$ and the error judgement $\sigma\ \mathtt{err}_{\mathcal{A}}$. Here, $\mathcal{A}$ ranges over *argument environments*, which we will also return to later; we omit the subscript $\mathcal{A}$ in cases where $\mathcal{A} = \cdot; \emptyset; \cdot$. We define the judgement $\sigma \mapsto^*_{\mathcal{A}} \sigma^+$ as the reflexive, transitive closure of the stepping judgement and the evaluation judgement $\sigma \Downarrow_{\mathcal{A}} \sigma'$ iff $\sigma \mapsto^*_{\mathcal{A}} \sigma'$ and $\sigma'\ \mathtt{val}_{\mathcal{A}}$.

tycons	$c ::= \rightharpoonup \mid \text{TC}$		
tycon contexts	$\Phi ::= \cdot \mid \Phi, \mathsf{tycon}\ \text{TC}\ \{\theta\} : \psi$		
tycon structures	$\theta ::= \mathsf{trans} = \sigma\ \mathsf{in}\ \omega$	**tycon sigs**	$\psi ::= \mathsf{tcsig}[\kappa]\ \{\chi\}$
opcon structures	$\omega ::= \mathsf{ana\ intro} = \sigma \mid \omega, \mathsf{syn\ op} = \sigma$	**opcon sigs**	$\chi ::= \mathsf{intro}[\kappa] \mid \theta, \mathbf{op}[\kappa]$

Fig. 4. Syntax of type constructors. Metavariable TC ranges over user-defined tycon names.

2.2 Types and Tycon Contexts

External types, or simply *types*, are static values of kind Ty. We define the auxiliary relation $\sigma\ \mathtt{type}_\Phi$ for convenience:

Definition 1. $\sigma\ \mathtt{type}_\Phi$ *iff* $\emptyset\ \emptyset \vdash^0_\Phi \sigma :: \mathsf{Ty}$ *and* $\sigma\ \mathtt{val}$.

There are two introductory forms for types: $c\langle\sigma\rangle$, where c is a tycon and σ is the *type index*, and $\mathsf{otherty}[m;\tau]$. The syntax for tycons, shown in Fig. 4, specifies that c can either be the built-in tycon governing partial functions, $\rightharpoonup$, or a user-defined tycon name, written in small caps, TC. There are three kinding rules governing these forms:

$$\text{(k-parr)}\ \frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty} \times \mathsf{Ty}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \rightharpoonup\langle\sigma\rangle :: \mathsf{Ty}} \qquad \text{(k-ty)}\ \frac{\mathsf{tycon}\ \text{TC}\ \{\theta\} : \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\} \in \Phi \quad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa_{\text{tyidx}}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \text{TC}\langle\sigma\rangle :: \mathsf{Ty}} \qquad \text{(k-otherty)}\ \frac{\emptyset \vdash \tau}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{otherty}[m;\tau] :: \mathsf{Ty}}$$

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 might introduce a desugaring from $\sigma_1 \rightharpoonup \sigma_2$ to $\rightharpoonup\langle(\sigma_1, \sigma_2)\rangle$.

All other tycons must be defined in the *tycon context*, Φ, ostensibly by the users of the language, rather than its designers. Each tycon specifies a *tycon signature*, ψ, of the form $\mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\}$, where κ_{tyidx} is the tycon's index kind (we will return to the opcon signature, χ, in Sec. 2.3). The first premise of the rule (k-ty) extracts the index kind and the second checks the type index against it.

For example, we will define a type constructor RSTR classifying *regular strings*, which are known to be in a regular language specified by a regular expression, closely following [5]. The index kind of RSTR is Rx, which we assume is defined as an inductive sum kind in the standard way. We can write its signature, ψ_{rstr}, and define a tycon context containing only its definition, Φ_{rstr}, as follows:

$$\begin{aligned}\psi_{\text{rstr}} &:= \mathsf{tcsig}[\mathsf{Rx}]\ \{\chi_{\text{rstr}}\}\\ \Phi_{\text{rstr}} &:= \mathsf{tycon}\ \text{RSTR}\ \{\theta_{\text{rstr}}\} : \psi_{\text{rstr}}\end{aligned}$$

We then have that, e.g., $\emptyset\ \emptyset \vdash^0_{\Phi_{\text{rstr}}} \sigma_{\text{title}} :: \mathsf{Ty}$ and $\emptyset\ \emptyset \vdash^0_{\Phi_{\text{rstr}}} \sigma_{\text{conf}} :: \mathsf{Ty}$, where

$$\begin{aligned}\sigma_{\text{title}} &:= \text{RSTR}\langle\texttt{/.+/}\rangle\\ \sigma_{\text{conf}} &:= \text{RSTR}\langle\texttt{/[A-Z]+ \textbackslash d\textbackslash d\textbackslash d\textbackslash d/}\rangle\end{aligned}$$

The type index is here written using standard concrete syntax for regular expressions for concision. In previous work, we have shown how to define type-specific (here, kind-specific) syntax like this composably in libraries [16].

The second example we will define is the tycon LPROD, which will define a variant of labeled product type (which are like record types, but maintain a row ordering). The index kind of LPROD is $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$, assuming lists are defined in the usual way, and Lbl classifies static representations of syntactic labels. We can write the tycon signature of LPROD, ψ_{lprod}, and the tycon context containing only its definition, Φ_{lprod}, as follows:

$$\psi_{\text{lprod}} := \mathsf{tcsig}[\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]]\,\{\chi_{\text{lprod}}\}$$
$$\Phi_{\text{lprod}} := \mathsf{tycon}\ \text{LPROD}\ \{\theta_{\text{lprod}}\} : \psi_{\text{lprod}}$$

In a tycon context containing both these tycon definitions, $\Phi_{\text{rstr}}\Phi_{\text{lprod}}$, we can derive that $\emptyset\ \emptyset \vdash^0_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \sigma_{\text{paper}} :: \mathsf{Ty}$ where:

$$\sigma_{\text{paper}} := \text{LPROD}\langle\{\mathtt{title} : \sigma_{\text{title}}, \mathtt{conf} : \sigma_{\text{conf}}\}\rangle$$

We again use kind-specific syntax, here for Lbl and $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$, for concision, and to again demonstrate how standard syntax for types can be recovered composably despite the uniform abstract syntax for types we use in our specification of the language.

The remaining introductory form for types, $\mathsf{otherty}[m; \tau]$, is a technical device that will be used to quantify over all types other than those in a given tycon context. As there may be arbitrarily many of these, the term is indexed by a natural number, m. It is also indexed by a closed internal type, τ, which serves as its translation, discussed below.

The operational semantics for these forms is straightforward: the type index is recursively evaluated to a value and errors are propagated:

$$\text{(s-ty-step)}\ \frac{\sigma \mapsto_{\mathcal{A}} \sigma'}{c\langle\sigma\rangle \mapsto_{\mathcal{A}} c\langle\sigma'\rangle} \qquad \text{(s-ty-err)}\ \frac{\sigma\ \mathtt{err}_{\mathcal{A}}}{c\langle\sigma\rangle\ \mathtt{err}_{\mathcal{A}}} \qquad \text{(s-ty-v)}\ \frac{\sigma\ \mathtt{val}_{\mathcal{A}}}{c\langle\sigma\rangle\ \mathtt{val}_{\mathcal{A}}} \qquad \text{(s-otherty-v)}\ \frac{}{\mathsf{otherty}[m;\tau]\ \mathtt{val}_{\mathcal{A}}}$$

Type Case Analysis A type σ can be case analyzed against a known tycon c 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.

$$\text{(k-tycase-parr)}\ \frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty} \quad \boldsymbol{\Delta}\ \boldsymbol{\Gamma}, \boldsymbol{x} :: \mathsf{Ty} \times \mathsf{Ty} \vdash^n_\Phi \sigma_1 :: \kappa \quad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma_2 :: \kappa}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{tycase}[\rightharpoonup](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) :: \kappa}$$

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

The operational semantics are straightforward:

$$\text{(s-tycase-step)}\ \frac{\sigma \mapsto_{\mathcal{A}} \sigma'}{\mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) \mapsto_{\mathcal{A}} \mathsf{tycase}[c](\sigma'; x.\sigma_1; \sigma_2)} \qquad \text{(s-tycase-err)}\ \frac{\sigma\ \mathtt{err}_{\mathcal{A}}}{\mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2)\ \mathtt{err}_{\mathcal{A}}}$$

$$\text{(s-tycase-match)}\ \frac{c\langle\sigma\rangle\ \mathtt{val}_{\mathcal{A}}}{\mathsf{tycase}[c](c\langle\sigma\rangle; \boldsymbol{x}.\sigma_1; \sigma_2) \mapsto_{\mathcal{A}} [\sigma/x]\sigma_1} \qquad \text{(s-tycase-fail)}\ \frac{\sigma \neq c\langle\sigma'\rangle}{\mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) \mapsto_{\mathcal{A}} \sigma_2}$$

Put another way, types can be thought of as arising from a distinguished "open datatype" defined by the tycon context. We will see how case analysis can be useful for defining the semantics of operators associated with tycons in Sec. 2.3.

Tycon Context Formation The tycon context formation judgement, $\vdash\ \Phi$, requires that tycon names are unique (we assume some extralinguistic mechanism is used for avoiding naming conflicts) and checks that each tycon structure, θ, is valid against its tycon signature, ψ. We consider the premises of (tcc-ext) below:

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

Type Equivalence To simplify the handling of type equivalence, type index kinds must be *equality kinds*: those for which semantic equivalence implies syntactic equality of values. We define these by the judgement $\boldsymbol{\Delta} \vdash \kappa\ \mathsf{eq}$, appearing as a premise in (tcc-ext). 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.

$$\frac{\boldsymbol{k} \in \boldsymbol{\Delta}}{\boldsymbol{\Delta} \vdash \boldsymbol{k}\ \mathsf{eq}}\ \text{(keq-k)} \qquad \frac{\boldsymbol{\Delta}, \boldsymbol{k} \vdash \kappa\ \mathsf{eq}}{\boldsymbol{\Delta} \vdash \mu_{\text{ind}}(\boldsymbol{k}.\kappa)\ \mathsf{eq}}\ \text{(keq-ind)} \qquad \frac{}{\boldsymbol{\Delta} \vdash 1\ \mathsf{eq}}\ \text{(keq-unit)}$$

$$\frac{\boldsymbol{\Delta} \vdash \kappa_1\ \mathsf{eq} \quad \boldsymbol{\Delta} \vdash \kappa_2\ \mathsf{eq}}{\boldsymbol{\Delta} \vdash \kappa_1 \times \kappa_2\ \mathsf{eq}}\ \text{(keq-prod)} \qquad \frac{\boldsymbol{\Delta} \vdash \kappa_1\ \mathsf{eq} \quad \boldsymbol{\Delta} \vdash \kappa_2\ \mathsf{eq}}{\boldsymbol{\Delta} \vdash \kappa_1 + \kappa_2\ \mathsf{eq}}\ \text{(keq-sum)} \qquad \frac{}{\boldsymbol{\Delta} \vdash \mathsf{Ty}\ \mathsf{eq}}\ \text{(keq-ty)}$$

Type Translations Our style of specifying the meaning of external terms directly in terms of a translation to the IL is similar to the Harper-Stone *typed elaboration semantics* for Standard ML [7]. There, however, external and internal terms were governed by a common type system. In @λ, each external type maps onto an internal type, called its *translation*, as specified by the judgement $\vdash_{\Phi}\ \sigma\ \rightsquigarrow\ \tau$, defined below. This 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 [18], here lifted "one level up" into the semantics of the language itself.

Each tycon determines the translations of the types it constructs as a function of each type's index by specifying a *translation schema* in the tycon structure. For a tycon with index kind κ_{tyidx}, the translation schema has 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 follows:

$$\theta_{\text{rstr}} := \mathsf{trans} = \lambda \boldsymbol{tyidx}{::}\mathsf{Rx}.\blacktriangleright(\hat{\mathsf{str}})\ \mathsf{in}\ \omega_{\text{rstr}}$$

The kinding rules for these shared forms simply proceed recursively, e.g.:

$$\frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n}_{\Phi} \blacktriangleright(\hat{\tau}_1) :: \mathsf{ITy} \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n}_{\Phi} \blacktriangleright(\hat{\tau}_2) :: \mathsf{ITy}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n}_{\Phi} \blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2) :: \mathsf{ITy}}\ \text{(k-ity-lam)} \qquad \frac{}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n}_{\Phi} \blacktriangleright(\alpha) :: \mathsf{ITy}}\ \text{(k-ity-alpha)}$$

The operational semantics for shared forms are also straightforwardly recursive:

$$\text{(s-ity-lam-step-1)}\ \frac{\blacktriangleright(\hat{\tau}_1) \mapsto_{\mathcal{A}} \blacktriangleright(\hat{\tau}_1')}{\blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2) \mapsto_{\mathcal{A}} \blacktriangleright(\hat{\tau}_1' \times \hat{\tau}_2)} \qquad \text{(s-ity-lam-step-2)}\ \frac{\blacktriangleright(\hat{\tau}_1)\ \mathtt{val}_{\mathcal{A}} \qquad \blacktriangleright(\hat{\tau}_2) \mapsto_{\mathcal{A}} \blacktriangleright(\hat{\tau}_2')}{\blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2) \mapsto_{\mathcal{A}} \blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2')} \qquad \text{(s-ity-lam-err-1)}\ \frac{\blacktriangleright(\hat{\tau}_1)\ \mathtt{err}_{\mathcal{A}}}{\blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2)\ \mathtt{err}_{\mathcal{A}}}$$

$$\text{(s-ity-lam-err-2)}\ \frac{\blacktriangleright(\hat{\tau}_2)\ \mathtt{err}_{\mathcal{A}}}{\blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2)\ \mathtt{err}_{\mathcal{A}}} \qquad \text{(s-ity-lam-v)}\ \frac{\blacktriangleright(\hat{\tau}_1)\ \mathtt{val}_{\mathcal{A}} \qquad \blacktriangleright(\hat{\tau}_2)\ \mathtt{val}_{\mathcal{A}}}{\blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2)\ \mathtt{val}_{\mathcal{A}}} \qquad \text{(s-ity-alpha-v)}\ \frac{}{\blacktriangleright(\alpha)\ \mathtt{val}_{\mathcal{A}}}$$

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 refers to the translation of type σ. For example, our translation schema for LPROD translates labeled product types to nested binary product types computed by folding over the type index and referring to the translations of the types therein. We assume a standard list recursor, ***listfold*** $:: \forall(\boldsymbol{\alpha}_1.\forall(\boldsymbol{\alpha}_2.\mathsf{List}[\boldsymbol{\alpha}_1] \to \boldsymbol{\alpha}_2 \to (\boldsymbol{\alpha}_1 \to \boldsymbol{\alpha}_2 \to \boldsymbol{\alpha}_2) \to \boldsymbol{\alpha}_2))$:

$$\begin{aligned}\theta_{\text{lprod}} := \mathsf{trans} = \lambda&\textbf{\textit{tyidx}}{::}\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}].\\ &\textbf{\textit{listfold}}\,[\mathsf{Lbl} \times \mathsf{Ty}]\,[\mathsf{ITy}]\ \textbf{\textit{tyidx}}\ \blacktriangleright(1)\\ &\quad(\lambda\boldsymbol{h}{:}\mathsf{Lbl} \times \mathsf{Ty}.\lambda\boldsymbol{r}{:}\mathsf{ITy}.\mathsf{letpair}\ (\textbf{\textit{hlbl}}, \textbf{\textit{hty}}) = \boldsymbol{h}\ \mathsf{in}\\ &\qquad\blacktriangleright(\mathsf{trans}(\textbf{\textit{hty}}) \times \blacktriangleleft(\boldsymbol{r}))\\ \mathsf{in}\ &\omega_{\text{rstr}}\end{aligned}$$

For example, evaluating this translation schema with the index of σ_{paper} produces $\sigma_{\text{paper/trans}} := \blacktriangleright(\hat{\tau}_{\text{paper/abstrans}})$ where: $\hat{\tau}_{\text{paper/abstrans}} := \mathsf{trans}(\sigma_{\text{title}}) \times (\mathsf{trans}(\sigma_{\text{conf}}) \times 1)$. We do not include logic to optimize away the trailing unit type for simplicity.

The kinding rules for the unquote and translation forms are straightforward:

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

The unquote form is eliminated during evaluation:

$$\text{(s-ity-unquote-step)}\ \frac{\sigma \mapsto_{\mathcal{A}} \sigma'}{\blacktriangleright(\blacktriangleleft(\sigma)) \mapsto_{\mathcal{A}} \blacktriangleright(\blacktriangleleft(\sigma'))} \qquad \text{(s-ity-unquote-err)}\ \frac{\sigma\ \mathtt{err}_{\mathcal{A}}}{\blacktriangleright(\blacktriangleleft(\sigma))\ \mathtt{err}_{\mathcal{A}}} \qquad \text{(s-ity-unquote-elim)}\ \frac{\blacktriangleright(\hat{\tau})\ \mathtt{val}_{\mathcal{A}}}{\blacktriangleright(\blacktriangleleft(\blacktriangleright(\hat{\tau}))) \mapsto_{\mathcal{A}} \blacktriangleright(\hat{\tau})}$$

References to the translation of a type, however, are retained in values of kind ITy:

$$\text{(s-ity-trans-step)}\ \frac{\sigma \mapsto_{\mathcal{A}} \sigma'}{\blacktriangleright(\mathsf{trans}(\sigma)) \mapsto_{\mathcal{A}} \blacktriangleright(\mathsf{trans}(\sigma'))} \qquad \text{(s-ity-trans-err)}\ \frac{\sigma\ \mathtt{err}_{\mathcal{A}}}{\blacktriangleright(\mathsf{trans}(\sigma))\ \mathtt{err}_{\mathcal{A}}} \qquad \text{(s-ity-trans-val)}\ \frac{\sigma\ \mathtt{val}_{\mathcal{A}}}{\blacktriangleright(\mathsf{trans}(\sigma))\ \mathtt{val}_{\mathcal{A}}}$$

This allows us to selectively hold these translations abstract and will be the key to our main results. The selective type abstraction judgement $\hat{\tau} \parallel \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\Phi} \tau^+ \parallel \mathcal{D}^+$ relates a normalized translational internal type $\hat{\tau}$ to an internal type τ, called an *abstract type translation* because all references to translations of types constructed by a user-defined tycon other than those constructed by a "delegated tycon", TC, are replaced by universally quantified type variables, α. Each unique type has a unique type variable

associated with it, tracked by the *type translation store*, $\mathcal{D}$, which maps from types, σ, to their (concrete) translation, τ, and the type variable, α, which appears in its place in the abstract type translation: $\mathcal{D} ::= \emptyset \mid \mathcal{D}, \sigma \leftrightarrow \tau/\alpha$.

For example, $\hat{\tau}_{\text{paper/trans}} \parallel \emptyset \twoheadrightarrow^{\text{LPROD}}_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \tau_{\text{paper/abstrans}} \parallel \mathcal{D}_{\text{paper/abstrans}}$ where:

$$\begin{array}{ll} \tau_{\text{paper/abstrans}} & := \alpha_1 \times (\alpha_2 \times 1) \\ \mathcal{D}_{\text{paper/abstrans}} & := \sigma_{\text{title}} \leftrightarrow \mathsf{str}/\alpha_1, \sigma_{\text{conf}} \leftrightarrow \mathsf{str}/\alpha_2 \end{array}$$

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 are a standard type theoretic construct, $\delta ::= \emptyset \mid \delta, \tau/\alpha$.

For example, $\mathcal{D}_{\text{paper/abstrans}} \rightsquigarrow \delta_{\text{paper/abstrans}} : \Delta_{\text{paper/abstrans}}$ where:

$$\begin{array}{ll} \delta_{\text{paper/abstrans}} & := \mathsf{str}/\alpha_1, \mathsf{str}/\alpha_2 \\ \Delta_{\text{paper/abstrans}} & := \alpha_1, \alpha_2 \end{array}$$

This judgement is defined by the following straightforward rules:

$$\text{(tstore-emp)}\ \frac{}{\emptyset \rightsquigarrow \emptyset : \emptyset} \qquad \text{(tstore-ext)}\ \frac{\mathcal{D} \rightsquigarrow \delta : \Delta}{(\mathcal{D}, \sigma \leftrightarrow \tau/\alpha) \rightsquigarrow (\delta, \tau/\alpha) : (\Delta, \alpha)}$$

We can apply type substitutions to internal types, terms and typing contexts, written $[\delta]\tau$, $[\delta]\iota$ and $[\delta]\Gamma$, respectively. For example, $[\delta_{\text{paper/abstrans}}]\tau_{\text{paper/abstrans}}$ is:

$$\tau_{\text{paper}} := \mathsf{str} \times (\mathsf{str} \times 1)$$

This process of selectively holding the translations of all types constructed by any tycon other than the "delegated tycon" abstract will be critical for conservativity to hold below. 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 is complete. Here, the translation of the type is being held abstract, rather than the type itself, and type translations are computed from indices, rather than declared "literally".

The selective type abstraction judgement is defined by recursing generically over sub-terms until terms of the form $\mathsf{trans}(\sigma)$ are encountered. For example,

$$\text{(abs-iparr)}\ \frac{\hat{\tau}_1 \parallel \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\Phi} \tau_1 \parallel \mathcal{D}' \qquad \hat{\tau}_2 \parallel \mathcal{D}' \twoheadrightarrow^{\text{TC}}_{\Phi} \tau_2 \parallel \mathcal{D}''}{\hat{\tau}_1 \times \hat{\tau}_2 \parallel \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\Phi} \tau_1 \times \tau_2 \parallel \mathcal{D}''} \qquad \text{(abs-alpha)}\ \frac{}{\alpha \parallel \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\Phi} \alpha \parallel \mathcal{D}}$$

The translation of partial function types is not held abstract, so that lambdas can be used as the sole binding construct in the language:

$$\text{(abs-parr)}\ \frac{\mathsf{trans}(\sigma_1) \parallel \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\Phi} \tau_1 \parallel \mathcal{D}' \qquad \mathsf{trans}(\sigma_2) \parallel \mathcal{D}' \twoheadrightarrow^{\text{TC}}_{\Phi} \tau_2 \parallel \mathcal{D}''}{\mathsf{trans}(\rightharpoonup\langle(\sigma_1, \sigma_2)\rangle) \parallel \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\Phi} \tau_1 \rightharpoonup \tau_2 \parallel \mathcal{D}''}$$

The translation of user-defined types 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 $\mathcal{D}'$.

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

The translation of a user-defined type constructed by any tycon other than the delegated tycon is added to the store, or retrieved from it if already there (using a store ensures that equal types map to equal abstract type variables, even if their translation is requested multiple times). The translation of $\mathsf{otherty}[m;\tau]$ is simply τ. It is always held abstract, because it is designed to behave indistinguishably from a type from the "future", as we will discuss.

$$\text{(abs-tc-foreign-new)}\quad \frac{\text{TC} \neq \text{TC}' \qquad \text{TC}'\langle\sigma_{\text{tyidx}}\rangle \notin \text{dom}(\mathcal{D}) \qquad \mathsf{tycon}\ \text{TC}'\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} : \psi \in \Phi \qquad \sigma_{\text{schema}}\ \sigma_{\text{tyidx}} \Downarrow \blacktriangleright(\hat{\tau}) \qquad \hat{\tau} \parallel \mathcal{D} \looparrowright^{\text{TC}}_{\Phi} \tau \parallel \mathcal{D}' \qquad \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \Delta \vdash \tau \qquad (\alpha\ \text{fresh})}{\mathsf{trans}(\text{TC}'\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \looparrowright^{\text{TC}}_{\Phi} \alpha \parallel \mathcal{D}', \text{TC}'\langle\sigma_{\text{tyidx}}\rangle \leftrightarrow \tau/\alpha}$$

$$\text{(abs-tc-other-new)}\quad \frac{\mathsf{otherty}[m;\tau] \notin \text{dom}(\mathcal{D}) \qquad (\alpha\ \text{fresh})}{\mathsf{trans}(\mathsf{otherty}[m;\tau]) \parallel \mathcal{D} \looparrowright^{\text{TC}}_{\Phi} \alpha \parallel \mathcal{D}, \mathsf{otherty}[m;\tau] \leftrightarrow \tau/\alpha} \qquad \text{(abs-tc-stored)}\quad \frac{\sigma \leftrightarrow \tau/\alpha \in \mathcal{D}}{\mathsf{trans}(\sigma) \parallel \mathcal{D} \looparrowright^{\text{TC}}_{\Phi} \alpha \parallel \mathcal{D}}$$

Finally, we can give the rule defining the concrete translation judgement, $\vdash_{\Phi} \sigma \rightsquigarrow \tau$, mentioned at the beginning of this subsection. We simply determine any selectively abstract translation, then apply the substitution induced by the store:

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

Typing Contexts The translation judgement for external typing contexts $\vdash_{\Phi} \Upsilon \rightsquigarrow \Gamma$ simply checks that Υ actually maps variables to types, then generates the corresponding internal typing context Γ. We follow standard practice in assuming implicitly that terms are congruent up to α-equivalence so all variables in typing contexts can be assumed unique implicitly. We also have that external typing contexts obey the standard structural congruences: weakening, exchange and contraction.

$$\text{(etctx-emp)}\quad \frac{}{\vdash_{\Phi} \emptyset \rightsquigarrow \emptyset} \qquad \text{(etctx-ext)}\quad \frac{\vdash_{\Phi} \Upsilon \rightsquigarrow \Gamma \qquad \sigma\ \mathsf{type}_{\Phi} \qquad \vdash_{\Phi} \sigma \rightsquigarrow \tau}{\vdash_{\Phi} \Upsilon, x \Rightarrow \sigma \rightsquigarrow \Gamma, x : \tau}$$

2.3 Typing and Translation of External Terms

Having established how types are constructed, and how they determine translations to internal types, we can finally give the typing and translation rules for external terms.

Variables and Functions Variables and functions behave in the standard manner. We use Plotkin's fixpoint operator to support recursive (i.e. partial) functions (cf. [6]), and define both only in analytic positions for simplicity:

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

$$\text{(syn-ap)}\quad \frac{\Upsilon \vdash_\Phi e_1 \Rightarrow \rightharpoonup\langle(\sigma_1, \sigma_2)\rangle \rightsquigarrow \iota_1 \qquad \Upsilon \vdash_\Phi e_2 \Leftarrow \sigma_2 \rightsquigarrow \iota_2}{\Upsilon \vdash_\Phi e_1(e_2) \Rightarrow \sigma_2 \rightsquigarrow \iota_1(\iota_2)} \qquad \text{(ana-fix)}\quad \frac{\Upsilon, x \Rightarrow \sigma \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota \qquad \vdash_\Phi \sigma \rightsquigarrow \tau}{\Upsilon \vdash_\Phi \mathsf{fix}(x.e) \Leftarrow \sigma \rightsquigarrow \mathsf{fix}[\tau](x.\iota)}$$

Subsumption 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 above, equivalent types must be syntactically identical, so the rule is straightforward:

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

Type Ascriptions 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 evaluated to a type before being used for analysis:

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

Generalized Introductory Operations The meaning 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 arguments, $\overline{e}$.

Note that we can recover a variety of standard introductory forms (and allow their use at more than one type) by a purely syntactic desugaring to this form. The term index captures all static portions of the concrete form and the arguments capture all external sub-terms. Some example desugarings are defined below (using kind-specific syntax for static terms as above and writing n and `"s"` for static numbers and strings):

Description	**Concrete Form**	**Desugared Form**
sequences	$(e_1, \ldots, e_n)$ or $[e_1, \ldots, e_n]$	$\mathsf{intro}[()](e_1; \ldots; e_n)$
labeled sequences	$\{\mathtt{lbl}_1 = e_1, \ldots, \mathtt{lbl}_n = e_n\}$	$\mathsf{intro}[[\mathtt{lbl}_1, \ldots, \mathtt{lbl}_n]](e_1; \ldots; e_n)$
label application	$\mathtt{lbl}\langle e_1, \ldots, e_n\rangle$	$\mathsf{intro}[\mathtt{lbl}](e_1, \ldots, e_n)$
numerals	n	$\mathsf{intro}[n](\cdot)$
labeled numerals	$n\mathtt{lbl}$	$\mathsf{intro}[(n, \mathtt{lbl})](\cdot)$
strings	`"s"`	$\mathsf{intro}[\mathtt{"s"}](\cdot)$

As an example, we will derive $\Upsilon_{\text{example}} \vdash_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} e_{\text{example}} \Leftarrow \sigma_{\text{paper}} \rightsquigarrow \iota_{\text{example}}$ where the labeled product type σ_{paper} and regular string type σ_{title} was defined in Sec. 2.2 and:

$$\begin{array}{rl} \Upsilon_{\text{example}} & := \mathit{title} \Rightarrow \sigma_{\text{title}} \\ e_{\text{example}} & := \{\texttt{title} = \mathit{title}, \texttt{conf} = \texttt{"EXMPL 2015"}\} \\ & \quad \hookrightarrow \mathsf{intro}[[\texttt{title}, \texttt{conf}]](\mathit{title}; \mathsf{intro}[\texttt{"EXMPL 2015"}](\cdot)) \\ \iota_{\text{example}} & := (\mathit{title}, (\texttt{"EXMPL 2015"}, ())) \end{array}$$

The rule governing this form is below:

(ana-intro)

$$\frac{\begin{array}{c} \mathsf{tycon}\ \text{TC}\ \{\mathsf{trans} = \_\ \mathsf{in}\ \omega\} : \mathsf{tcsig}[\_]\ \{\chi\} \in \Phi \\ \mathsf{intro}[\kappa_{\text{tmidx}}] \in \chi \qquad \emptyset\ \emptyset \vdash^{0}_{\Phi} \sigma_{\text{tmidx}} :: \kappa_{\text{tmidx}} \\ \mathsf{ana}\ \mathsf{intro} = \sigma_{\text{def}} \in \omega \qquad \mathsf{args}(\overline{e}) = \sigma_{\text{args}} \qquad \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \Downarrow_{\overline{e};\Upsilon;\Phi} \rhd(\hat{\iota}) \\ \hat{\iota} \parallel \emptyset\ \emptyset \twoheadrightarrow^{\text{TC}}_{\overline{e};\Upsilon;\Phi} \iota \parallel \mathcal{D}\ \mathcal{G} \qquad \mathsf{trans}(\text{TC}\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\Phi} \tau \parallel \mathcal{D}' \\ \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma \qquad \Delta\ \Gamma \vdash \iota : \tau \end{array}}{\Upsilon \vdash_{\Phi} \mathsf{intro}[\sigma_{\text{tmidx}}](\overline{e}) \Leftarrow \text{TC}\langle\sigma_{\text{tyidx}}\rangle \rightsquigarrow [\delta][\gamma]\iota}$$

The first premise of (ana-intro) extracts the tycon definition for the tycon of the type the literal is being analyzed against, which is the delegated tycon, from Φ. In this example, the delegated tycon is LPROD.

The second premise extracts the *intro index kind*, κ_{tmidx}, from the *opcon signature*, χ, of the delegated tycon's tycon signature. The third premise checks the provided term index against this kind. For example, LPROD specifies $\mathsf{List}[\mathsf{Lbl}]$, so that it can use the labeled sequence form, while RSTR specifies an intro index kind of Str, so that it can use the string literal form above:

$$\begin{array}{rl} \chi_{\text{lprod}} & := \mathsf{intro}[\mathsf{List}[\mathsf{Lbl}]], \chi_{\text{lprod/targops}} \\ \chi_{\text{rstr}} & := \mathsf{intro}[\mathsf{Str}], \chi_{\text{rstr/targops}} \end{array}$$

The fourth premise extracts the *intro opcon definition* from the *opcon structure*, ω, of the delegated tycon's tycon structure, calling it σ_{def}. This is the static function that is invoked to determine the translation of the term, returning a *translational internal term* of kind ITm given the type index σ_{tyidx} of kind κ_{tyidx}, the term index σ_{tmidx}, of kind κ_{tmidx}, and an *argument interface list*, σ_{args}, of kind $\mathsf{List}[\mathsf{Arg}]$, derived from $\overline{e}$ by the judgement $\mathsf{args}(\overline{e}) =_n \sigma$, appearing as the fifth premise and described below.

For example, the intro opcon definition for RSTR is shown below:

$$\begin{array}{rl} \omega_{\text{rstr}} := & \mathsf{ana}\ \mathsf{intro} = (\lambda \textbf{\textit{tyidx}}{::}\mathsf{Rx}.\lambda \textbf{\textit{tmidx}}{::}\mathsf{Str}.\lambda \textbf{\textit{args}}{::}\mathsf{List}[\mathsf{Arg}]. \\ & \quad \mathsf{let}\ \textbf{\textit{aok}} :: 1 = \textbf{\textit{arity0 args}}\ \mathsf{in} \\ & \quad \mathsf{let}\ \textbf{\textit{rok}} :: 1 = \textbf{\textit{rmatch tyidx tmidx}}\ \mathsf{in} \\ & \quad \rhd((\lhd(\textbf{\textit{str\_of\_Str tmidx}}), \lhd(\textbf{\textit{nat\_of\_Nat}}\ (\textbf{\textit{len tmidx}}))))), \\ & \omega_{\text{rstr/targops}} \end{array}$$

The judgement, $\vdash_{\Phi} \omega \sim \psi$, which appeared as the final premise of the rule (tcc-ext) above, checks that such an intro opcon definition is well-kinded:

(ocstruct-intro)

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

This example intro opcon definition begins by making sure that no arguments were passed in, using the helper function ***arity0*** :: $\mathsf{List}[\mathsf{Arg}] \to 1$ defined such that any non-empty input will raise an error, via the static term $\mathsf{raise}[1]$. The kinding rules and operational semantics of this exception-like form are standard:

$$\text{(k-raise)}\ \frac{\boldsymbol{\Delta} \vdash \kappa}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{raise}[\kappa] :: \kappa} \qquad \text{(s-raise)}\ \frac{}{\mathsf{raise}[\kappa]\ \mathtt{err}_\mathcal{A}}$$

In practice, the tycon provider would specify an error message, but here we are satisfied simply signaling an error in this way.

Next, this intro opcon definition 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 *translational internal term* corresponding to the term index is generated using the helper functions ***str_of_Str*** :: $\mathsf{Str} \to \mathsf{ITm}$ and ***nat_of_Nat*** :: $\mathsf{Nat} \to \mathsf{ITm}$ to generate translational internal terms corresponding to the provided static terms and ***len*** :: $\mathsf{Str} \to \mathsf{Nat}$ to statically compute the length of the string.

The only introductory form for kind ITm is $\triangleright(\hat{\iota})$, where $\hat{\iota}$ is a translational internal term. This form is analagous to the introductory form for kind ITy described previously, $\blacktriangleright(\hat{\tau})$, where $\hat{\tau}$ is a translational internal type. Each form in the syntax of ι has a corresponding form in the syntax for $\hat{\iota}$ and the kinding rules simply recurse through these generically, i.e. as shown for variables and lambda functions below:

$$\text{(k-itm-var)}\ \frac{}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \triangleright(x) :: \mathsf{ITm}} \qquad \text{(k-itm-lam)}\ \frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \blacktriangleright(\hat{\tau}) :: \mathsf{ITy} \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \triangleright(\hat{\iota}) :: \mathsf{ITm}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \triangleright(\lambda[\hat{\tau}](x.\hat{\iota})) :: \mathsf{ITm}}$$

The operational semantics for these shared forms are similarly recursive:

$$\text{(s-itm-var-v)}\ \frac{}{\triangleright(x)\ \mathtt{val}_\mathcal{A}} \qquad \text{(s-itm-lam-step-1)}\ \frac{\blacktriangleright(\hat{\tau}) \mapsto_\mathcal{A} \blacktriangleright(\hat{\tau}')}{\triangleright(\lambda[\hat{\tau}](x.\hat{\iota})) \mapsto_\mathcal{A} \triangleright(\lambda[\hat{\tau}'](x.\hat{\iota}))} \qquad \text{(s-itm-lam-step-2)}\ \frac{\blacktriangleright(\hat{\tau})\ \mathtt{val}_\mathcal{A} \qquad \triangleright(\hat{\iota}) \mapsto_\mathcal{A} \triangleright(\hat{\iota}')}{\triangleright(\lambda[\hat{\tau}](x.\hat{\iota})) \mapsto_\mathcal{A} \triangleright(\lambda[\hat{\tau}](x.\hat{\iota}'))}$$

$$\text{(s-itm-lam-err-1)}\ \frac{\blacktriangleright(\hat{\tau})\ \mathtt{err}_\mathcal{A}}{\triangleright(\lambda[\hat{\tau}](x.\hat{\iota}))\ \mathtt{err}_\mathcal{A}} \qquad \text{(s-itm-lam-err-2)}\ \frac{\triangleright(\hat{\iota})\ \mathtt{err}_\mathcal{A}}{\triangleright(\lambda[\hat{\tau}](x.\hat{\iota}))\ \mathtt{err}_\mathcal{A}} \qquad \text{(s-itm-lam-v)}\ \frac{\blacktriangleright(\hat{\tau})\ \mathtt{val}_\mathcal{A} \qquad \triangleright(\hat{\iota})\ \mathtt{val}_\mathcal{A}}{\triangleright(\lambda[\hat{\tau}](x.\hat{\iota}))\ \mathtt{val}_\mathcal{A}}$$

Like translational internal types, there is also an unquote form, $\triangleleft(\sigma)$, that permits translational internal terms to be constructed compositionally:

$$\text{(k-itm-unquote)}\ \frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{ITm}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \triangleright(\triangleleft(\sigma)) :: \mathsf{ITm}}$$

These unquote forms are eliminated in the course of evaluation:

$$\text{(s-itm-unquote-step)}\ \frac{\sigma \mapsto_\mathcal{A} \sigma'}{\triangleright(\triangleleft(\sigma)) \mapsto_\mathcal{A} \triangleright(\triangleleft(\sigma'))} \qquad \text{(s-itm-unquote-err)}\ \frac{\sigma\ \mathtt{err}_\mathcal{A}}{\triangleright(\triangleleft(\sigma))\ \mathtt{err}_\mathcal{A}} \qquad \text{(s-itm-unquote-elim)}\ \frac{\triangleright(\hat{\iota})\ \mathtt{val}_\mathcal{A}}{\triangleright(\triangleleft(\triangleright(\hat{\iota}))) \mapsto_\mathcal{A} \triangleright(\hat{\iota})}$$

The final two forms of translational internal term are $\mathsf{anatrans}[n](\sigma)$ and $\mathsf{syntrans}[n]$. These represent the translation of argument n, the first if it arises via analysis against type σ and the second if it arises by synthesis. They are not intended to be written directly by tycon providers, arising only internally in the semantics of argument interface lists. Before giving the rules, let us motivate the mechanism by showing the intro opcon definition for LPROD:

$$
\begin{array}{l}
\omega_{\text{lprod}} := \mathsf{ana\ intro} = \lambda\textbf{\textit{tyidx}}{:}\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}].\lambda\textbf{\textit{tmidx}}{:}\mathsf{List}[\mathsf{Lbl}].\lambda\textbf{\textit{args}}{:}\mathsf{List}[\mathsf{Arg}]. \\
\quad \mathsf{let}\ \textbf{\textit{inhabited}} : 1 = \textbf{\textit{uniqmap tyidx}} \\
\quad \textbf{\textit{listrec3}}\,[\mathsf{Lbl} \times \mathsf{Ty}]\,[\mathsf{Lbl}]\,[\mathsf{Arg}]\,[\mathsf{ITm}]\ \textbf{\textit{tyidx tmidx args}}\ \triangleright(()) \\
\quad\quad \lambda\textbf{\textit{rowtyidx}}{:}\mathsf{Lbl} \times \mathsf{Ty}.\lambda\textbf{\textit{rowtmidx}}{:}\mathsf{Lbl}.\lambda\textbf{\textit{rowarg}}{:}\mathsf{Arg}.\lambda\textbf{\textit{r}}{:}\mathsf{ITm}. \\
\quad\quad\quad \mathsf{letpair}\ (\textbf{\textit{rowlbl}}, \textbf{\textit{rowty}}) = \textbf{\textit{rowtyidx}} \\
\quad\quad\quad \mathsf{let}\ \textbf{\textit{lok}} :: 1 = \textbf{\textit{lbleq rowlbl rowtmidx}} \\
\quad\quad\quad \mathsf{let}\ \textbf{\textit{rowtr}} :: \mathsf{ITm} = \textbf{\textit{ana rowarg rowty}} \\
\quad\quad\quad \triangleright((\triangleleft(\textbf{\textit{rowtr}}), \triangleleft(\textbf{\textit{r}}))), \\
\quad \omega_{\text{lprod/targops}}
\end{array}
$$

The first line of the intro opcon definition for LPROD checks that the type index is well-formed, 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 more general technique: types with indices that are to be considered "malformed" can simply be left uninhabited. An analagous technique could be used to implement record types, leaving indices where the labels did not appear sorted uninhabited. In both cases, we could provide a static helper function of kind $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}] \to \mathsf{Ty}$ that checked well-formedness immediately before constructing the requested type.

The rest of this intro 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***.

We define Arg, the kind of *argument interfaces*, as a product of functions, one for analysis and the other for synthesis, and the helper functions ***ana*** and ***syn*** simply project the corresponding function out:

fix products

$$
\begin{array}{l}
\mathsf{Arg} := (\mathsf{Ty} \to \mathsf{ITm}) \times (1 \to \mathsf{Ty} \times \mathsf{ITm}) \\
\textbf{\textit{ana}} := \lambda\textbf{\textit{arg}}{::}\mathsf{Arg}.\mathsf{fst}(\textbf{\textit{arg}}) \\
\textbf{\textit{syn}} := \lambda\textbf{\textit{arg}}{::}\mathsf{Arg}.\mathsf{snd}(\textbf{\textit{arg}})
\end{array}
$$

As mentioned above, the argument interface list is constructed from the argument list by the judgement $\mathsf{args}(\overline{e}) =_n \sigma_{\text{args}}$, defined as follows:

$$\text{(args-z)}\ \frac{}{\mathsf{args}(\cdot) =_0 \textbf{\textit{nil}}[\mathsf{Arg}]} \qquad \text{(args-s)}\ \frac{\mathsf{args}(\overline{e}) =_n \sigma}{\mathsf{args}(\overline{e}; e) =_{n+1} \textbf{\textit{rcons}}[\mathsf{Arg}]\ \sigma\ (\lambda \textbf{\textit{ty}}{::}\mathsf{Ty}.\mathsf{ana}[n](\textbf{\textit{ty}}), \lambda\_{::}1.\mathsf{syn}[n])}$$

We assume that the definitions of the standard helper functions $\textbf{\textit{nil}} :: \forall(\boldsymbol{\alpha}.\mathsf{List}[\boldsymbol{\alpha}])$ and $\textbf{\textit{rcons}} :: \forall(\boldsymbol{\alpha}.\mathsf{List}[\boldsymbol{\alpha}] \to \boldsymbol{\alpha} \to \mathsf{List}[\boldsymbol{\alpha}])$, which adds an item to the end of a list, have been substituted into these rules.

The nth element of the argument interface list simply wraps the static terms $\mathsf{ana}[n](\sigma)$ and $\mathsf{syn}[n]$. Recall that the kinding judgement is indexed by n, which is an upper bound on the argument index of such terms. This is enforced in the kinding rules:

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

The following lemma thus characterizes the argument interface list judgement:

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

The rule (ocstruct-intro) ruled out writing either of these forms explicitly in an opcon definition by checking against the bound $n = 0$. But accessing them via the argument interface list is possible because the following straightforward lemma holds (see appendix for proofs):

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

The operational semantics for these couple the dynamics of the static language to 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. If it fails, an error is raised. We write the judgement $[\Upsilon \vdash_{\Phi} e \Leftarrow\!\!\!\!\!\!\cdot\ \ \sigma]$ to indicate that e fails to analyze against σ. Because we do not here define this judgement itself inductively, we also allow that this premise be left out entirely, resulting in a non-deterministic but sufficient semantics for our metatheory:

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

$$\text{(s-ana-err)}\ \frac{\sigma\ \mathtt{err}_{\mathcal{A}}}{\mathsf{ana}[n](\sigma)\ \mathtt{err}_{\mathcal{A}}} \qquad \text{(s-ana-fail)}\ \frac{\sigma\ \mathtt{val} \qquad \mathsf{nth}[n](\overline{e}) = e \qquad [\Upsilon \vdash_{\Phi} e \Leftarrow\!\!\!\!\!\!\cdot\ \ \sigma]}{\mathsf{ana}[n](\sigma)\ \mathtt{err}_{\overline{e};\Upsilon;\Phi}}$$

The semantics for $\mathsf{syn}[n]$ are analagous, evaluating to a pair containing the translational internal term $\triangleright(\mathsf{syntrans}[n])$ as well as the synthesized type:

$$\text{(s-syn-success)}\ \frac{\mathsf{nth}[n](\overline{e}) = e \qquad \Upsilon \vdash_{\Phi} e \Rightarrow \sigma \rightsquigarrow \iota}{\mathsf{syn}[n] \mapsto_{\overline{e};\Upsilon;\Phi} (\sigma, \triangleright(\mathsf{syntrans}[n]))} \qquad \text{(s-syn-fail)}\ \frac{\mathsf{nth}[n](\overline{e}) = e \qquad [\Upsilon \vdash_{\Phi} e \Rightarrow\!\!\!\!\!\!\cdot\ \ ]}{\mathsf{syn}[n]\ \mathtt{err}_{\overline{e};\Upsilon;\Phi}}$$

The kinding rules also prevent the translational internal term forms generated by these operations from being written directly when $n = 0$:

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

(k-itm-syntrans)
$$\frac{n' < n}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n}_{\Phi} \triangleright(\mathsf{syntrans}[n']) :: \mathsf{ITm}}$$

Like the translational internal type form $\mathsf{trans}(\sigma)$, these two translational internal term forms are retained in values of kind ITm:

(s-itm-anatrans-step)
$$\frac{\sigma \mapsto_{\mathcal{A}} \sigma'}{\triangleright(\mathsf{anatrans}[n](\sigma)) \mapsto_{\mathcal{A}} \triangleright(\mathsf{anatrans}[n](\sigma'))}$$

(s-itm-anatrans-err)
$$\frac{\sigma\ \mathtt{err}_{\mathcal{A}}}{\triangleright(\mathsf{anatrans}[n](\sigma))\ \mathtt{err}_{\mathcal{A}}}$$

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

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

The final line of the intro opcon definition for LPROD constructs a nested tuple as the translation. Taken together, the translational internal term that will be derived for our example involving e_{example} above is:

$$\hat{\iota}_{\text{example}} := (\mathsf{anatrans}[0](\sigma_{\text{title}}), (\mathsf{anatrans}[1](\sigma_{\text{conf}}), ()))$$

The reason that the translations themselves were not inserted yet (even though they were derived in rule (s-ana-success) above) is again because we want to be able to selectively hold these abstract. The selective term abstraction judgement $\hat{\iota} \parallel \mathcal{D}\ \mathcal{G} \looparrowright^{\text{TC}}_{\mathcal{A}} \iota \parallel \mathcal{D}'\ \mathcal{G}'$, appearing as the seventh premise of (ana-intro), relates a translational internal term $\hat{\iota}$ to an internal term ι called the corresponding abstract internal term, where all references to the translation of a type constructed by a user-defined tycon other than the "delegated tycon" TC are replaced with an abstract type variable, and all references to the translation of an argument (having any type) are replaced with a variable. 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, we have that

$$\hat{\iota}_{\text{example}} \parallel \emptyset\ \emptyset \looparrowright^{\text{LPROD}}_{\overline{e},\Upsilon,\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \iota_{\text{example/abs}} \parallel \mathcal{D}_{\text{example/abs}}\ \mathcal{G}_{\text{example/abs}}$$

where

$$\begin{aligned}
\iota_{\text{example/abs}} &:= (x_0, (x_1, ())) \\
\mathcal{G}_{\text{example/abs}} &:= 0 : \sigma_{\text{title}} \rightsquigarrow \mathit{title}/x_0 : \alpha_0, 1 : \sigma_{\text{conf}} \rightsquigarrow \texttt{"EXMPL 2015"}/x_1 : \alpha_1 \\
\mathcal{D}_{\text{example/abs}} &:= \sigma_{\text{title}} \leftrightarrow \mathsf{str}/\alpha_1, \sigma_{\text{conf}} \leftrightarrow \mathsf{str}/\alpha_2
\end{aligned}$$

The rules for the selective term abstraction judgement follow recursively for shared forms, e.g. for variables and lambdas:

(abs-var)
$$\frac{}{x \parallel \mathcal{D}\ \mathcal{G} \looparrowright^{\text{TC}}_{\mathcal{A}} x \parallel \mathcal{D}\ \mathcal{G}}$$

(abs-lam)
$$\frac{\hat{\tau} \parallel \mathcal{D} \looparrowright^{\text{TC}}_{\Phi} \tau \parallel \mathcal{D}' \qquad \hat{\iota} \parallel \mathcal{D}'\ \mathcal{G} \looparrowright^{\text{TC}}_{\overline{e};\Upsilon;\Phi} \iota \parallel \mathcal{D}''\ \mathcal{G}'}{\lambda[\hat{\tau}](x.\hat{\iota}) \parallel \mathcal{D}\ \mathcal{G} \looparrowright^{\text{TC}}_{\overline{e};\Upsilon;\Phi} \lambda[\tau](x.\iota) \parallel \mathcal{G}'\ \mathcal{D}''}$$

The rules for references to argument translations operate as follows:

$$\text{(abs-anatrans-stored)}\quad \frac{n : \sigma \rightsquigarrow \iota/x : \tau \in \mathcal{G}}{\mathsf{anatrans}[n](\sigma) \parallel \mathcal{G}\ \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\mathcal{A}} x \parallel \mathcal{G}\ \mathcal{D}} \qquad \text{(abs-syntrans-stored)}\quad \frac{n : \sigma \rightsquigarrow \iota/x : \tau \in \mathcal{G}}{\mathsf{syntrans}[n] \parallel \mathcal{G}\ \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\mathcal{A}} x \parallel \mathcal{G}\ \mathcal{D}}$$

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

$$\text{(abs-syntrans-new)}\quad \frac{n \notin \text{dom}(\mathcal{G}) \qquad \mathsf{nth}[n](\overline{e}) = e \qquad \Upsilon \vdash_{\Phi} e \Rightarrow \sigma \rightsquigarrow \iota \qquad \mathsf{trans}(\sigma) \parallel \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\Phi} \tau \parallel \mathcal{D}' \qquad (x \text{ fresh})}{\mathsf{syntrans}[n] \parallel \mathcal{G}\ \mathcal{D} \twoheadrightarrow^{\text{TC}}_{\overline{e};\Upsilon;\Phi} x \parallel \mathcal{G}, n : \sigma \rightsquigarrow \iota/x : \tau\ \mathcal{D}'}$$

After computing an abstract term translation, the eight premise computes an abstract type translation for the type the intro term is being analyzed against:

$$\mathsf{trans}(\sigma_{\text{conf}}) \parallel \mathcal{D}_{\text{example/abs}} \twoheadrightarrow^{\text{LPROD}}_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \tau_{\text{example/abs}} \parallel \mathcal{D}_{\text{example/abs}}$$

where $\tau_{\text{example/abs}} := \alpha_0 \times (\alpha_1 \times 1)$ (alpha-equivalent to $\tau_{\text{conf/abstrans}}$ in Sec. 2.2).

Each term translation store $\mathcal{G}$ induces an internal term substitution, written in the standard way $\gamma ::= \emptyset \mid \gamma, \iota/x$, and a corresponding internal typing context Γ, as specified by the judgement $\mathcal{G} \rightsquigarrow \gamma : \Gamma$ d defined by the following rules:

$$\text{(ttrs-emp)}\quad \frac{}{\emptyset \rightsquigarrow \emptyset : \emptyset} \qquad \text{(ttrs-ext)}\quad \frac{\mathcal{G} \rightsquigarrow \gamma : \Gamma}{(\mathcal{G}, n : \sigma \rightsquigarrow \iota/x : \tau) \rightsquigarrow (\gamma, \iota/x) : (\Gamma, x : \tau)}$$

In this case, we have the ninth premise, $\mathcal{G}_{\text{example/abs}} \rightsquigarrow \gamma_{\text{example/abs}} : \Gamma_{\text{example/abs}}$, where

$$\gamma_{\text{example/abs}} := title/x_0, \texttt{"EXMPL 2015"}/x_1$$
$$\Gamma_{\text{example/abs}} := x_0 : \alpha_0, x_1 : \alpha_1$$

The tenth premise extracts a type substitution, $\delta_{\text{example/abs}}$, and type formation context, $\Delta_{\text{example/abs}}$, from $\mathcal{D}_{\text{example/abs}}$, again equivalent to those defined in Sec. 2.2.

Finally, the tenth premise checks that the abstract term translation is consistent with the abstract type translation: $\Delta_{\text{example/abs}}\ \Gamma_{\text{example/abs}} \vdash \iota_{\text{example/abs}} : \tau_{\text{example/abs}}$, i.e.

$$(\alpha_0, \alpha_1)\ (x_0 : \alpha_0, x_1 : \alpha_1) \vdash (x_0, (x_1, ())) : \alpha_0 \times (\alpha_1 \times \mathtt{1})$$

Observe that the translation of the labeled product e_{example} containing regular strings is being checked with all references to term and type translations of regular strings replaced by variables and type variables, respectively. Nevertheless, because our logic for labeled products has been treating regular strings parametrically, the check succeeds.

Applying the substitutions $\gamma_{\text{example/abs}}$ and $\delta_{\text{example/abs}}$ in the conclusion of the rule, we have our final translation, ι_{example}, i.e. $(title, (\texttt{"EXMPL 2015"}, ()))$.

Note that $\Gamma_{\text{example}} \vdash \iota_{\text{example}} : \tau_{\text{conf}}$, where $\vdash_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \Upsilon_{\text{example}} \rightsquigarrow \Gamma_{\text{example}}$ and $\vdash_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \sigma_{\text{conf}} \rightsquigarrow \tau_{\text{conf}}$ as described in Sec. 2.2. In fact, this relationship between the

translation of an external term and the translation of the type it has will always hold given the check we performed on the abstract term and type transations just described due to parametricity properties. Combined with type safety of the IL this will imply type safety of $@\lambda$. We give the details in Sec. 3.

Had the translational term generated by the intro opcon definition been, e.g., $(\mathsf{anatrans}[0](\sigma_{\text{title}}), (\texttt{"TEST"}, ()))$, then the corresponding abstract term would be $(x_0, (\texttt{"TEST"}, ()))$ and the check would fail because $(\alpha_0, \alpha_1)\ (x_0 : \alpha_0) \not\vdash \texttt{"TEST"} : \alpha_1$. Had we not gone through the machinations of holding types and terms abstract, however, the check would succeed, because str would no longer have been held abstract as α_1. This form of "type translation independence" is precisely analagous to the representation independence properties that underly abstraction theorems for module systems based on abstract types and will similarly serve to ensure that the type invariants maintained by RSTR are conserved when new tycons are defined. In this case the invariant is that only strings that are in the regular language specified in the type index can be generated from an external term of regular string type. Note that "TEST" is not in the regular language specified by `/[A-Z]+ \d\d\d\d/`, so it is critical that it not be allowed, despite being consistent with the type translation of σ_{conf}, i.e. it is a string. Such an invariant could not be maintained if the type system were treated as merely a "bag of rules". We consider this more rigorously in Sec. 3.

Note also that the domains of $\Upsilon_{\text{example}}$ and $\Gamma_{\text{example/abs}}$ 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 $\Upsilon_{\text{example}}$ can (and do – $title$ in this example) occur in arguments to the operation, but the translations of the arguments only appear after the substitution $\gamma_{\text{example/abs}}$ has been applied. We assume that applying a substitution is capture-avoiding in the usual manner (i.e. consistent with a locally nameless implementation).

Generalized Targeted Operations All non-introductory opcons associated with user-defined tycons are used via a form for *targeted operations*, $\mathsf{targ}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e})$, where **op** is the opcon name, σ_{tmidx} is the term index, e_{targ} is the *target argument* and $\overline{e}$ are the remaining arguments. Some examples of desugarings to this form are below:

Description	**Concrete Form**	**Desugared Form**
index projection	$e_{\text{targ}}\texttt{\#}n$	$\mathsf{targ}[\mathbf{idx}; n](e_{\text{targ}}; \cdot)$
label projection	$e_{\text{targ}}\texttt{\#lbl}$	$\mathsf{targ}[\mathbf{prj}; \texttt{lbl}](e_{\text{targ}}; \cdot)$
explicit invocation	$e_{\text{targ}}{\cdot}\mathbf{op}[\sigma_{\text{tmidx}}](\overline{e})$	$\mathsf{targ}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e})$
	$e_{\text{targ}}{\cdot}\mathbf{op}(\overline{e})$	$\mathsf{targ}[\mathbf{op}; ()](e_{\text{targ}}; \overline{e})$
	$e_{\text{targ}}{\cdot}\mathbf{op}(\texttt{lbl}_1 = e_1, \ldots, \texttt{lbl}_n = e_n)$	$\mathsf{targ}[\mathbf{op}; [\texttt{lbl}_1, \ldots, \texttt{lbl}_n]](e_{\text{targ}}; e_1; \ldots; e_n)$
labeled case analysis	$e_{\text{targ}}{\cdot}\mathbf{case}\,\{$	$\mathsf{targ}[\mathbf{case}; [\sigma_1, \ldots, \sigma_n]](e_{\text{targ}};$
	$\mid\ \sigma_1\langle x_1, \ldots, x_k\rangle \Rightarrow e_1$	$\lambda(x_1 \ldots . \lambda(x_k . e_1));$
	$\mid\ \ldots$	$\ldots;$
	$\mid\ \sigma_n\langle x_1, \ldots, x_k\rangle \Rightarrow e_n\}$	$\lambda(x_1 \ldots . \lambda(x_k . e_n)))$

Whereas introductory operations were strictly analytic, targeted operations are always synthetic in $@\lambda$. The type and translation is determined by the tycon of the type synthesized by the target argument (i.e. this tycon is used as the delegated tycon).

The rule given below is otherwise similar to (ana-intro) in its key machinations:

$$\frac{\begin{array}{c}\text{(syn-targ)}\\ \Upsilon \vdash_\Phi e_{\text{targ}} \Rightarrow \text{TC}\langle\sigma_{\text{tyidx}}\rangle \rightsquigarrow \iota_{\text{targ}} \qquad \mathsf{tycon}\ \text{TC}\ \{\mathsf{trans} = \_\ \mathsf{in}\ \omega\} : \mathsf{tcsig}[\_]\ \{\chi\} \in \Phi\\ \mathbf{op}[\kappa_{\text{tmidx}}] \in \chi \qquad \emptyset\ \emptyset \vdash^0_\Phi \sigma_{\text{tmidx}} :: \kappa_{\text{tmidx}}\\ \mathsf{syn}\ \mathbf{op} = \sigma_{\text{def}} \in \omega \qquad \mathsf{args}(e_{\text{targ}}; \overline{e}) = \sigma_{\text{args}} \qquad \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \Downarrow_{(e_{\text{targ}};\overline{e});\Upsilon;\Phi} (\sigma, \triangleright(\hat{\iota}))\\ \hat{\iota} \parallel \emptyset\ \emptyset \overset{\text{TC}}{\hookrightarrow}_{\overline{e};\Upsilon;\Phi} \iota \parallel \mathcal{D}\ \mathcal{G} \qquad \mathsf{trans}(\sigma) \parallel \mathcal{D} \overset{\text{TC}}{\hookrightarrow}_\Phi \tau \parallel \mathcal{D}'\\ \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma \qquad \Delta\ \Gamma \vdash \iota : \tau\end{array}}{\Upsilon \vdash_\Phi \mathsf{targ}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e}) \Rightarrow \sigma \rightsquigarrow [\delta][\gamma]\iota}$$

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 targeted opcon signatures with the tycons RSTR and LPROD:

$$\begin{array}{ll}\chi_{\text{rstr/targops}} & := \mathbf{concat}[1], \mathbf{case}[\mathsf{List}[\mathsf{StrPattern}]], \mathbf{coerce}[\mathsf{Rx}], \mathbf{check}[\mathsf{Rx}], \mathbf{replace}[\mathsf{Rx}]\\ \chi_{\text{lprod/targops}} & := \mathbf{prj}[\mathsf{Lbl}], \mathbf{with}[\mathsf{List}[\mathsf{Lbl}]], \mathbf{drop}[\mathsf{List}[\mathsf{Lbl}]]\end{array}$$

The opcons associated with RSTR are taken directly from Fulton et al.'s specification of regular string types [5], with the exception of **case**, which generalizes the case analysis operator given there to arbitrary string patterns, which we assume are encoded in some suitable way by kind StrPattern. The opcons associated with LPROD are also straightforward: **prj** projects out a labeled value from a labeled product, **with** creates a new labeled product based on the target with some fields updated or added, and **drop** creates a new labeled product from the target with some fields dropped. Note that regular string types cannot directly be embedded into ML-like languages and the analogs of **with** and **drop** are only found in some dialects of ML. For concision, we will only show **concat** and **prj** here; more details on the others can be found in the appendix.

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 delegated tycon's type index, the term index and an argument interface list. Targeted opcon definitions pair the translation with the synthesized type as well. The rule (ocstruct-op) ensures that the opcon definition has a correct and well-formed kind:

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

For example, the opcon definition for **concat** is defined as follows:

$$\begin{array}{rl}\omega_{\text{rstr/targops}} := & \mathsf{syn}\ \mathbf{concat} = \lambda\textbf{\textit{tyidx}}{::}\mathsf{Rx}.\lambda\textbf{\textit{tmidx}}{::}1.\lambda\textbf{\textit{args}}{::}\mathsf{List}[\mathsf{Arg}].\\ & \quad \mathsf{letpair}\ (\textbf{\textit{arg1}}, \textbf{\textit{arg2}}) = \textbf{\textit{arity2 args}}\\ & \quad \mathsf{letpair}\ (\_, \textbf{\textit{tr1}}) = \textbf{\textit{syn arg1}}\\ & \quad \mathsf{letpair}\ (\textbf{\textit{ty2}}, \textbf{\textit{tr2}}) = \textbf{\textit{syn arg2}}\\ & \quad \mathsf{tycase}[\text{RSTR}](\textbf{\textit{ty2}}; \textbf{\textit{tyidx2}}.\\ & \qquad (\text{RSTR}\langle\textbf{\textit{rxconcat tyidx tyidx2}}\rangle, \triangleright(\mathit{sconcat} \triangleleft(\textbf{\textit{tr1}}) \triangleleft(\textbf{\textit{tr2}})))\\ & \qquad ; \mathsf{raise}[\mathsf{Ty} \times \mathsf{ITm}])\end{array}$$

The first line simply checks that exactly two total arguments, including the target term, were passed in using the helper function ***arity2***. 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 failing if not. We finally synthesize the resulting regular string type, using the helper function ***rxconcat*** :: $\mathsf{Rx} \to \mathsf{Rx} \to \mathsf{Rx}$ which generates the concatenated regular expression, and the translation, using an internal helper function $sconcat$: $\mathtt{str} \to \mathtt{str} \to \mathtt{str}$, the translational internal term for which we assume has been substituted in directly above.

This definition is invoked as the sixth premise of (syn-targ), generating the type ... and the translational internal term ... consistent with the rules described above. The remaining premises are identical to the corresponding premises in (ana-intro), with the only exception being that we check the abstract term translation against the abstract type translation of ... In this case... Note that because the delegated tycon and the tycon of the synthesized type are the same, the type is not held abstract, allowing our use of the helper function $sconcat$ to successfully typecheck. Had an identical translational internal term arisen from an opcon associated with LPROD, the corresponding abstract type translation would be a type variable, α, so the check would fail. This implies that the choice of representation, and of representation invariants, for regular strings is entirely at the disrection of the RSTR definition. More creative possibilities could be used with only local impact, e.g. traces through the regular language or ...

Other The final form of external term is $\mathsf{other}[m]\iota$. It can be analyzed against $\mathsf{otherty}[m;\tau]$ if ι has type τ in the translation of the current typing context. We will see how it is used as a technical device in the next section.

(ana-other)
$$\frac{\vdash_\Phi \Upsilon \rightsquigarrow \Gamma \qquad \emptyset\ \Gamma \vdash \iota : \tau}{\Upsilon \vdash_\Phi \mathsf{other}[\iota] \Leftarrow \mathsf{otherty}[m;\tau] \rightsquigarrow \iota}$$

3 Metatheory

Definition 2 (Argument Environment Formation). $\vdash^n \overline{e}; \Upsilon; \Phi$ *iff* $|\overline{e}| = n$ *and* $\vdash \Phi$ *and* $\vdash_\Phi \Upsilon \rightsquigarrow \Gamma$.

3.1 Static Language

Theorem 1 (Static Canonical Forms). *If* $\emptyset\ \emptyset \vdash^n_\Phi \sigma :: \kappa$ *and* $\vdash^n \mathcal{A}$ *and* $\sigma\ \mathtt{val}_\mathcal{A}$ *then:*

1. *TODO: arrow*
2. *TODO: unit*
3. *TODO: product*
4. *TODO: sum*
5. *TODO: inductive*
6. *TODO: universal*
7. *If* $\kappa = \mathsf{Ty}$ *then* $\sigma\ \mathtt{type}_\Phi$ *and*

   (a) $\sigma = \rightharpoonup\langle(\sigma_1, \sigma_2)\rangle$ *and* σ_1 $\texttt{type}_\Phi$ *and* σ_2 $\texttt{type}_\Phi$*; or*
   (b) $\sigma = \mathtt{TC}\langle\sigma_{tyidx}\rangle$ *and* $\mathsf{tycon}\ \mathtt{TC}\ \{\omega\} : \mathsf{tcsig}[\kappa_{tyidx}]\ \{\chi\} \in \Phi$ *and* $\emptyset\ \emptyset \vdash^0_\Phi \sigma :: \kappa_{tyidx}$ *and* σ_{tyidx} $\mathtt{val}$*; or*
   (c) $\sigma = \mathsf{otherty}[m; \tau]$ *and* $\emptyset \vdash \tau$.

8. *If* $\kappa = \mathsf{ITy}$ *then* $\sigma = \blacktriangleright(\hat{\tau})$ *and* $\emptyset\ \emptyset \vdash^0_\Phi \sigma :: \mathsf{ITy}$ *and* σ $\mathtt{val}$ *and* $\blacktriangleleft(\sigma') \notin \hat{\tau}$ *and*
   (a) *The outer form of* $\hat{\tau}$ *is shared with* τ *and if* $\hat{\tau}'$ *is a sub-term of* $\hat{\tau}$ *then* $\emptyset\ \emptyset \vdash^0_\Phi \blacktriangleright(\hat{\tau}') :: \mathsf{ITy}$ *and* $\blacktriangleright(\hat{\tau}')$ $\mathtt{val}$*; or*
   (b) $\hat{\tau} = \mathsf{trans}(\sigma')$ *and* σ' $\texttt{type}_\Phi$

9. *If* $\kappa = \mathsf{ITm}$ *then* $\sigma = \triangleright(\hat{\iota})$ *and no sub-terms of* $\hat{\iota}$ *have the form* $\triangleleft(\sigma')$ *and*
   (a) *The outer form of* $\hat{\iota}$ *is shared with* ι *and if* $\hat{\iota}'$ *is a sub-term of* $\hat{\iota}$ *then* $\emptyset\ \emptyset \vdash^n_\Phi \triangleright(\hat{\iota}') :: \mathsf{ITm}$ *and* $\triangleright(\hat{\iota}')$ $\mathtt{val}_\mathcal{A}$ *and if* $\hat{\tau}$ *is a sub-term of* $\hat{\iota}$ *then* $\emptyset\ \emptyset \vdash^0_\Phi \blacktriangleright(\hat{\tau}) :: \mathsf{ITy}$ *and* $\blacktriangleright(\hat{\tau})$ $\mathtt{val}$*; or*
   (b) $\hat{\iota} = \mathsf{anatrans}[n'](\sigma')$ *and* $n' < n$ *and* σ' $\texttt{type}_\Phi$*; or*
   (c) $\hat{\iota} = \mathsf{syntrans}[n']$ *and* $n' < n$.

Theorem 2 (Static Progress). *If* $\emptyset\ \emptyset \vdash^n_\Phi \sigma :: \kappa$ *and* $\vdash \Phi$ *and* $\vdash^n \mathcal{A}$ *then* σ $\mathtt{val}_\mathcal{A}$ *or* σ $\mathtt{err}_\mathcal{A}$ *or* $\sigma \mapsto_\mathcal{A} \sigma'$.

Proof. We proceed by rule induction on the kinding judgement. (k-parr)
(k-ty)
(k-otherty)
(k-tycase-parr)
(k-tycase)
(k-ity-lam)
(k-ity-alpha)
(k-ity-unquote)
(k-ity-trans)
(k-raise)
(k-itm-var)
(k-itm-lam)
(k-itm-unquote)
(k-itm-anatrans)
(k-itm-syntrans)

Theorem 3 (Static Preservation). *If* $\emptyset\ \emptyset \vdash^n_\Phi \sigma :: \kappa$ *and* $\vdash \Phi$ *and* $\vdash^n \mathcal{A}$ *and* $\sigma \mapsto_\mathcal{A} \sigma'$ *then* $\emptyset\ \emptyset \vdash^n_\Phi \sigma' :: \kappa$.

Proof. (s-ty-step) (2 cases) (s-tycase-step) (2 cases) (s-tycase-match) (2 cases) (s-tycase-fail-1) (s-tycase-fail-2) (s-ity-lam-step-1) (s-ity-lam-step-2) (s-ity-unquote-step) (s-ity-unquote-elim) (s-ity-trans-step) (s-itm-lam-step-1) (s-itm-lam-step-2) (s-itm-unquote-step) (s-itm-unquote-elim) (s-ana-step) (s-ana-success) (s-syn-success) (relies on type synthesis theorem) (s-itm-anatrans-step)

3.2 Types

Lemma 3 (Type Substitution Application). *If* $\vdash \delta : \Delta$ *and* $\Delta \vdash \tau$ *then* $\emptyset \vdash [\delta]\tau$.

Lemma 4 (Selective Type Abstraction). *If* $\vdash \Phi$ *and* $\emptyset\ \emptyset \vdash^{0}_{\Phi} \blacktriangleright(\hat{\tau}) :: \mathsf{ITy}$ *and* $\blacktriangleright(\hat{\tau})\ \mathtt{val}$ *and* $\mathcal{D} \rightsquigarrow \delta : \Delta$ *and* $\vdash \delta : \Delta$ *and* $\hat{\tau} \parallel \mathcal{D} \looparrowright^{\Phi}_{\mathrm{TC}} \tau \parallel \mathcal{D}'$ *then* $\mathcal{D}' \rightsquigarrow \delta' : \Delta'$ *and* $\delta \subseteq \delta'$ *and* $\Delta \subseteq \Delta'$ *and* $\vdash \delta' : \Delta'$ *and* $\Delta' \vdash \tau$.

Proof. (shared forms) (trans(st))

Lemma 5 (Type Translation). *If* $\vdash \Phi$ *and* $\sigma\ \mathtt{type}_{\Phi}$ *and* $\vdash_{\Phi} \sigma \rightsquigarrow \tau$ *then* $\emptyset \vdash \tau$.

Proof. By Lemma 3 and Lemma 4.

Lemma 6 (Typing Context Translation). *If* $\vdash \Phi$ *and* $\vdash_{\Phi} \Upsilon \rightsquigarrow \Gamma$ *then* $\emptyset \vdash \Gamma$.

Proof. We proceed by rule induction on typing context translation. Empty case trivial. Extended case follows by Lemma 5 and definition of internal typing context formation.

3.3 External Terms

Theorem 4 (Type Synthesis). *If* $\vdash \Phi$ *and* $\vdash_{\Phi} \Upsilon \rightsquigarrow \Gamma$ *and* $\Upsilon \vdash_{\Phi} e \Rightarrow \sigma \rightsquigarrow \iota$ *then* $\sigma\ \mathtt{type}_{\Phi}$ *and* $\vdash_{\Phi} \sigma \rightsquigarrow \tau$.

Proof. (var) (ap) (syn-targ)

Lemma 7 (Selective Term Abstraction). *If* $\vdash \Phi$ *and* $\vdash^{n} \mathcal{A}$ *and* $\emptyset\ \emptyset \vdash^{n}_{\Phi} \triangleright(\hat{\iota}) :: \mathsf{ITm}$ *and* $\triangleright(\hat{\iota})\ \mathtt{val}_{\mathcal{A}}$ *and* $\mathcal{D} \rightsquigarrow \delta : \Delta$ *and* $\vdash \delta : \Delta$ *and* $\emptyset \vdash \Gamma_{out}$ *and* $\mathcal{G} \rightsquigarrow \gamma : \Gamma$ *and* $\Delta\ \Gamma_{out} \vdash \gamma : \Gamma$ *and* $\hat{\iota} \parallel \mathcal{D}\ \mathcal{G} \looparrowright^{\mathrm{TC}}_{\mathcal{A}} \iota \parallel \mathcal{D}'\ \mathcal{G}'$ *then* $\mathcal{D}' \rightsquigarrow \delta' : \Delta'$ *and* $\mathcal{G}' \rightsquigarrow \gamma' : \Gamma'$ *and* $\delta \subseteq \delta'$ *and* $\Delta \subseteq \Delta'$ *and* $\gamma \subseteq \gamma'$ *and* $\Gamma \subseteq \Gamma'$ *and* $\vdash \delta' : \Delta'$ *and* $\Delta'\ \Gamma_{out} \vdash \gamma' : \Gamma'$.

Proof. By rule induction on selective term abstraction judgement.
(shared forms) (unquote form not possibly by canonical forms) (anatrans) (syntrans)

Theorem 5 (Type-Preserving Translation). *If* $\vdash \Phi$ *and* $\vdash_{\Phi} \Upsilon \rightsquigarrow \Gamma$ *and either*

1. $\Upsilon \vdash_{\Phi} e \Leftarrow \sigma \rightsquigarrow \iota$ *and* $\sigma\ \mathtt{type}_{\Phi}$*; or*
2. $\Upsilon \vdash_{\Phi} e \Rightarrow \sigma \rightsquigarrow \iota$

then $\vdash_{\Phi} \sigma \rightsquigarrow \tau$ *then* $\emptyset\ \Gamma \vdash \iota : \tau$.

Proof. By rule induction on typing rules.
(var) (lam) (ap) (fix) (subsume) (ascribe) (ana-intro) (syn-targ) (other)

TODO: stability of typing TODO: Unicity of typing TODO: hygiene?

TODO: Make definitions out of these. The standard judgement $\Gamma \vdash \gamma : \Gamma'$ states that every binding $x : \tau$ in Γ' has a corresponding substitution ι/x in γ such that $\Gamma \vdash \iota : x$.

, and the judgement $\vdash \delta : \Delta$ checks that every type variable in Δ has a well-formed substitution in δ

3.4 Conservativity

Theorem 6 (Conservativity). *If* $\vdash \Phi$ *and* $\mathrm{TC}\langle\sigma_{tyidx}\rangle$ $\mathtt{type}_\Phi$ *and for all* e*, if* $\emptyset \vdash_\Phi e \Leftarrow \mathrm{TC}\langle\sigma_{tyidx}\rangle \rightsquigarrow \iota$ *and* $\iota \Downarrow \iota'$ *then* $P(\iota')$*, then if* $\vdash \Phi, \mathsf{tycon}\ \mathrm{TC}'\ \{\theta\} : \psi$ *then for all* e*, if* $\emptyset \vdash_{\Phi,\mathsf{tycon}\ \mathrm{TC}'\ \{\theta\}:\psi} e \Leftarrow \mathrm{TC}\langle\sigma_{tyidx}\rangle \rightsquigarrow \iota$ *and* $\iota \Downarrow \iota'$ *then* $P(\iota')$*.*

Lemma 8 (Other Substitution). *If* $\vdash \Phi, \mathsf{tycon}\ \mathrm{TC}\ \{\theta\} : \psi$ *and* σ $\mathtt{type}_\Phi$ *and* $\Upsilon \vdash_{\Phi,\mathsf{tycon}\ \mathrm{TC}\ \{\theta\}:\psi} e \Leftarrow \sigma \rightsquigarrow \iota$ *then there exists* e' *such that* $\Upsilon \vdash_\Phi e' \Leftarrow \sigma \rightsquigarrow \iota$*.*

4 Related Work

5 Conclusion

We combine several interesting type theoretic techniques, applying them to novel ends: 1) a bidirectional type system permits flexible reuse of a fixed syntax; 2) the SL serves both as an extension language and as the type-level language; we give it its own statics (i.e. a *kind system*); 3) we use a typed intermediate language and leverage corresponding *typed compilation* techniques, here lifted into the semantics of the EL; 4) we leverage internal type abstraction implicitly as an effect during normalization of the SL to enforce abstraction barriers between type constructors. As a result, conservativity follows from the same elegant parametricity results that underly abstraction theorems for module systems. Like modules, reasoning about these *modular type constructors* does not require mechanized specifications or proofs: correctness issues in the type constructor logic necessarily causes typechecking to fail, so even extensions that are not proven correct can be distributed and "tested" in the wild without compromising the integrity of an entire program (at worst, only values of types constructed by the tycon being tested may exhibit undesirable properties).

References

1. GHC/FAQ. `http://www.haskell.org/haskellwiki/GHC:FAQ#Extensible_Records`.
2. 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.
3. 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.
4. 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.
5. 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.
6. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.
7. 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.

8. 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.
9. 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.
10. A. Löh and R. Hinze. Open data types and open functions. In *PPDP*, pages 133–144. ACM, 2006.
11. L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP*, pages 82–93. ACM, 2005.
12. 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.
13. 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.
14. 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.
15. 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.
16. C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014.
17. B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.
18. 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.

A Appendix

A.1 More Examples

For example, the opcon definition for **prj** in LPROD is:

$$
\begin{array}{l}
\omega_{\text{lprod/targops}} := \mathsf{syn}\ \mathbf{prj} = \lambda \boldsymbol{tyidx}{::}\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}].\lambda \boldsymbol{tmidx}{::}\mathsf{Lbl}.\lambda \boldsymbol{args}{::}\mathsf{List}[\mathsf{Arg}]. \\
\quad \mathsf{let}\ \boldsymbol{self} :: \mathsf{Arg} = \boldsymbol{arity1}\ \boldsymbol{args} \\
\quad \mathsf{letpair}\ (\_, \boldsymbol{str}) = \boldsymbol{syn}\ \boldsymbol{self} \\
\quad \mathsf{letpair}\ (\boldsymbol{oty}, \boldsymbol{tr}) = \boldsymbol{listrec}\,[\mathsf{Lbl} \times \mathsf{Ty}]\,[\mathsf{Opt}[\mathsf{Ty}] \times \mathsf{ITm}]\ \boldsymbol{tyidx}\ (\boldsymbol{none}\,[\mathsf{Ty}], \boldsymbol{str}) \\
\quad\quad \lambda \boldsymbol{row}{::}\mathsf{Lbl} \times \mathsf{Ty}.\lambda \boldsymbol{rout}{::}\mathsf{Opt}[\mathsf{Ty}] \times \mathsf{ITm}. \\
\quad\quad\quad \mathsf{letpair}\ (\boldsymbol{rty}, \boldsymbol{rtr}) = \boldsymbol{rout} \\
\quad\quad\quad \boldsymbol{optcase}\,[\mathsf{Ty}]\,[\mathsf{Opt}[\mathsf{Ty}] \times \mathsf{ITm}]\ (\boldsymbol{rty})\ (\lambda\_{::}\mathsf{Ty}.\boldsymbol{rout})\ (\lambda\_{::}1. \\
\quad\quad\quad\quad \mathsf{letpair}\ (\boldsymbol{rowlbl}, \boldsymbol{rowty}) = \boldsymbol{row} \\
\quad\quad\quad\quad \boldsymbol{iflbleq}\ \boldsymbol{rowlbl}\ \boldsymbol{tmidx} \\
\quad\quad\quad\quad\quad (\lambda\_{::}1.(\boldsymbol{some}\,[\mathsf{Ty}]\ \boldsymbol{rowty}, \triangleright(\mathsf{fst}(\triangleleft(\boldsymbol{rtr}))))) \\
\quad\quad\quad\quad\quad (\lambda\_{::}1.(\boldsymbol{rty}, \triangleright(\mathsf{snd}(\triangleleft(\boldsymbol{rtr})))))) \\
\quad \boldsymbol{optcase}\ \boldsymbol{oty}\ (\lambda \boldsymbol{ty}{::}\mathsf{Ty}.(\boldsymbol{ty}, \boldsymbol{tr}))\ (\lambda\_{::}1.\mathsf{raise}[\mathsf{Ty} \times \mathsf{ITm}]), \\
\quad\quad \omega_{\text{lprod/targops/1}}
\end{array}
$$

The target term is passed in as the first argument. The first two lines simply check that no other arguments were passed in and (re-)synthesize the type and translation for the target. Then, we recurse over the type index looking for the provided label.

We maintain a translation and a type option, which remains ***none*** until the label is found. The translation involves projecting out the second component of the translation of the labeled product until the label is found, at which point we project out the first component and stop. If this process succeeds, we return the resulting type and translation. Otherwise, we raise a type error.

Lemma 9. *If* $\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n'}_{\Phi} \sigma :: \kappa$ *and* $n > n'$ *then* $\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n}_{\Phi} \sigma :: \kappa$.

Proof. We can proceed by straightforward rule induction, noting that all the kinding rules have been defined for all n except for the four most recently defined. In these cases, the transitive property of inequality on natural numbers implies that the same rule can directly be applied to derive the consequent in each case.

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

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

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

(ana-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-lit)
$$\frac{\begin{array}{c}\mathsf{tycon}\ \mathrm{TC}\ \{\omega\} : \mathsf{tcsig}[\kappa_{\mathrm{tyidx}}]\ \{\theta\} \in \Phi \\ \mathsf{intro}[\kappa_{\mathrm{tmidx}}] \in \theta \qquad \emptyset\,\emptyset \vdash^0_\Phi \sigma_{\mathrm{tmidx}} :: \kappa_{\mathrm{tmidx}} \\ \mathsf{anaintro} = \sigma_{\mathrm{def}} \in \omega \qquad \mathsf{args}(\overline{e}) = \sigma_{\mathrm{args}} \\ \sigma_{\mathrm{def}}\ \sigma_{\mathrm{tyidx}}\ \sigma_{\mathrm{tmidx}}\ \sigma_{\mathrm{args}} \Downarrow_{\overline{e};\Upsilon;\Phi} \rhd(\hat{\iota}) \\ \hat{\iota}...\delta : \Delta; \gamma : \Gamma; \iota \qquad \mathsf{trans}(\mathrm{TC}\langle\sigma_{\mathrm{tyidx}}\rangle)...\delta' : \Delta'; \tau \\ \Delta\Delta'\ \Gamma \vdash \iota : \tau\end{array}}{\Upsilon \vdash_\Phi \mathsf{intro}[\sigma_{\mathrm{tmidx}}](\overline{e}) \Leftarrow \mathrm{TC}\langle\sigma_{\mathrm{tyidx}}\rangle \rightsquigarrow [\delta][\delta'][\gamma]\iota}$$

(syn-targ)
$$\frac{\begin{array}{c}\Upsilon \vdash_\Phi e_{\mathrm{targ}} \Rightarrow \mathrm{TC}\langle\sigma_{\mathrm{tyidx}}\rangle \rightsquigarrow \iota_{\mathrm{targ}} \\ \mathsf{tycon}\ \mathrm{TC}\ \{\omega\} : \mathsf{tcsig}[\kappa_{\mathrm{tyidx}}]\ \{\Omega\} \in \Phi \\ \mathbf{op}[\kappa_{\mathrm{tmidx}}] \in \Omega \qquad \vdash \Phi \sim \Xi \qquad \emptyset\,\emptyset \vdash^0_\Xi \sigma_{\mathrm{tmidx}} :: \kappa_{\mathrm{tmidx}} \\ \mathsf{syn}\ \mathbf{op} = \sigma_{\mathrm{def}} \in \omega \qquad e_{\mathrm{targ}}; \overline{e} \looparrowright \sigma_{\mathrm{args}} \looparrowright \mathcal{G}_0; n \\ \sigma_{\mathrm{def}}\ \sigma_{\mathrm{tyidx}}\ (\textbf{\textit{tr}}\ \mathsf{syn}[0])\ \sigma_{\mathrm{tmidx}}\ (\textbf{\textit{atl}}\ \sigma_{\mathrm{args}}) \parallel \emptyset\ \mathcal{G}_0 \Downarrow_{\rightharpoonup,\mathrm{TC};\Upsilon;\Phi} (\sigma, \rhd(\iota_{\mathrm{abs}})) \parallel \mathcal{D}\ \mathcal{G} \\ \mathsf{trans}(\sigma) \parallel \mathcal{D} \cdot \Downarrow_{\rightharpoonup,\mathrm{TC};\emptyset;\Phi} \blacktriangleright(\tau_{\mathrm{abs}}) \parallel \mathcal{D}' \cdot \\ \vdash_\Phi \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma \qquad \Delta\ \Gamma \vdash \iota_{\mathrm{abs}} : \tau_{\mathrm{abs}}\end{array}}{\Upsilon \vdash_\Phi \mathsf{targ}[\mathbf{op}; \sigma_{\mathrm{tmidx}}](e_{\mathrm{targ}}; \overline{e}) \Rightarrow \sigma \rightsquigarrow [\delta][\gamma]\iota_{\mathrm{abs}}}$$

(ana-other)
$$\frac{\vdash_\Phi \mathsf{trans}(\Upsilon) \rightsquigarrow \Gamma \parallel \mathcal{D} \qquad \mathcal{D} \rightsquigarrow \delta; \Delta \qquad \Delta\ \Gamma \vdash \iota : \tau}{\Upsilon \vdash_\Phi \mathsf{other}[\iota] \Leftarrow \mathsf{otherty}[n; \tau] \rightsquigarrow [\delta]\iota}$$

Fig. 5. Typing