Modular Composing Typed Language Fragments

Abstract

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

In this paper, we take a foundational approach, specifying an extensible typed translation semantics, $@\lambda$. Only the $\rightarrow$ type constructor is built in; all others (we discuss constrained strings and variants of record types) are defined by extending a *tycon context*. Each tycon defines associated term-level *opcons* (e.g. row projection) using a static language where types and translations are values. The semantics come with modular metatheoretic guarantees, notably *type safety* and *conservativity*: that *tycon-specific invariants* established in any "closed world" are conserved in the "open world". Remarkably, extension providers do not need to provide the semantics with mechanized specifications or proofs. Instead, these guarantees arise from a form of translation validation that, taking inspiration from the ML module system, uses type abstraction to check that the opcons associated with a tycon respect the *translation independence* of all others.

1. Introduction

Typed programming languages are usually described as consisting of *fragments*, each contributing separately to the concrete and abstract syntax and static and dynamic semantics of the language [Harper 2012]. Harper organizes such fragments around *type constructors*, e.g. $\rightarrow$ for total function types, describing each in a different chapter of his book. Languages are then identified by a set of type constructors, e.g. $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$ is the language of partial (i.e. possibly non-terminating) function types, polymorphic types, recursive types, nullary and binary product types and binary sum types (its syntax is shown in Figure 1, discussed below). Another common practice is to describe a fragment using a simple calculus having a "catch-all" constant and base type to stand notionally for all "other" terms and types.

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

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

Establishing an isomorphic embedding of a fragment is not always possible, however, because it requires preserving both the static and dynamic semantics and, if defined, performance bounds specified by a cost semantics, of all operators. This is possible if they have "function-like" semantics, as the list operators do. But in $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$, it is impossible to introduce record types as a library because this requires introducing row projection operators, written `#lbl` in ML, one for each of the infinite set of row labels `lbl` (`#` is thus an *operator constructor*). Each time such a situation occurs, a new *dialect* is needed. The ML lineage, like others, faces a proliferation of dialects that go beyond "core ML":

1. **General Purpose Fragments:** Many variations on product types, for example, exist: n-ary tuples, labeled tuples, records (identified up to reordering), and records with width and depth subtyping [Cardelli 1984] and functional update operators[1] [Leroy et al. 2013]. Sum-

[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." [GHC]

like types are also exposed variously: standard datatypes, open datatypes [Löh and Hinze 2006; Millstein et al. 2004], polymorphic variants [Leroy et al. 2013] and exception types [Milner et al. 1990]. Combinations occur as class-based object systems [Leroy et al. 2013].

2. **Specialized Fragments:** Fragments providing more specialized operators are also commonly introduced in dialects, e.g. for distributed programming [Murphy et al. 2008], concurrency [Rossberg et al. 2006], reactive programming [Mandel and Pouzet 2005], databases [Ohori and Ueno 2011], units of measure [Kennedy 2009] and regular string sanitation [Fulton et al. 2014].
3. **Foreign Fragments:** A safe and natural foreign function interface (FFI) can be valuable. This requires enforcing the type system of the foreign language in the calling language. For example, MLj builds in a safe FFI to Java [Benton and Kennedy 1999].

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

These problems with composition do not arise when a fragment is expressed as an isomorphic embedding (i.e. as a library) because modern *module systems* can enforce abstraction barriers that ensure that the isomorphism needs only to be established in the "closed world" of the module. For example, a module for sets in ML can hold its representation type abstract, ensuring that any invariants necessary to maintain the isomorphism need only be maintained by the functions in the module (e.g. uniqueness, if using a list representation). These then continue to hold no matter which other modules are in use by a client [Harper 1997].

The alarming proliferation of dialects above suggests that mechanisms that make it possible to define and reason in a similarly modular, localized manner about direct extensions to the semantics of a language are needed. For example, once a language is extended with *regular string types* as described in [Fulton et al. 2014], all terms having a regular string type like RSTR$\langle$`/.+/`$\rangle$ should continue to behave as non-empty strings no matter which other extensions are in use.

Contributions In this paper, we take foundational steps toward this goal of modular language metatheory by constructing a simple but surprisingly powerful calculus, @λ (the "actively typed" lambda calculus). Its semantics consist of an extensible *external language* (EL) governed by a typed translation semantics targeting a simple and fixed *internal language* (IL). In particular, rather than building in a monolithic set of external type and operator constructors, the semantics are indexed by a *tycon context*. Each tycon,

internal types

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

internal terms

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

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

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

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

e.g. LPROD defining labeled products, directly specifies the semantics of its associated opcons, e.g. # for row projection, via *static functions*, i.e. functions written in a *static language* (SL), where types and translations are values.

We call a tycon associated with opcons in this manner a *modular tycon* because all *translation invariants* maintained by the opcons associated with that tycon, e.g. the regular string invariant just mentioned, are necessarily maintained in any further extended tycon context, i.e. in the "open world", due to a remarkably simple translation validation procedure that maintains *translation independence* between tycons using type abstraction in the IL. This is, encouragingly, the same fundamental principle underlying representation independence in ML-style module systems. As in ML, mechanized specifications and proofs are not needed.

2. Overview of @λ

Internal Language At the heart of our semantics is a typed internal language supporting type abstraction (i.e. universal quantification over types) [Reynolds 1994]. We use $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$, Figure 1, as representative of a typical intermediate language for a typed language. We assume an internal statics specified by judgements for type assignment $\Delta\ \Gamma \vdash \iota : \tau^{+}$, type formation $\Delta \vdash \tau$ and typing context formation $\Delta \vdash \Gamma$, and an internal dynamics specified as a structural operational semantics with a stepping judgement $\iota \mapsto \iota^{+}$ and a value judgement ι `val`.[2] Both the static and dynamic semantics of the IL can be found in any standard textbook covering typed lambda calculi (e.g. [Harper 2012] or [Pierce 2002]), so we assume familiarity.

External Language Programs "execute" as internal terms, but programmers interface with @λ by writing *external terms*, e. The abstract syntax of external terms is shown in Figure 2 and we introduce various concrete desugarings as we go on. The semantics are specified as a *bidirectionally typed translation semantics*, i.e. the key judgements have the following form, pronounced "Under typing context Υ and tycon context Φ, e (synthesizes / analyzes against) type σ and has translation ι":

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

[2] Our specifications are intended to be algorithmic: we indicate "outputs" when introducing judgement forms by *mode annotations*, $^{+}$.

external terms

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

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

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

Figure 2. Abstract syntax of the external language (EL).

We distinguish situations where the type is an "output" from those where it must be provided as an "'input" using such a bidirectional approach, also known as *local type inference* [Pierce and Turner 2000], for two reasons. The first is to justify the practicality of our approach: local type inference is increasingly being used in modern languages (e.g. Scala [Odersky et al. 2001]) because it eliminates the need for type annotations in many places and provides high quality error messages. Secondly, it will give us a clean way to reuse the abstract introductory form, $\mathsf{intro}[\sigma](\overline{e})$, and its associated desugarings, at many types.

For example, consider the following types:

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

The *regular string types* σ_{title} and σ_{conf} classify values that behave as strings known to be in a specified regular language [Fulton et al. 2014], i.e. σ_{title} classifies non-empty strings and σ_{conf} classifies strings having the format of a typical conference name. The *labeled product type* σ_{paper} then describes a conference paper by defining two *rows*, each having one of the regular string types just described. Regular string types are defined by tycon context Φ_{rstr} and labeled products by Φ_{lprod}, both introduced in Sec. 3.

We next define a function, e_{ex}, that takes a paper title and produces a paper in a conference named "EXMPL 2015":

$$
\begin{array}{lll}
e_{\text{ex}} & := & \lambda \mathit{title}{:}\sigma_{\text{title}}.(e_{\text{paper}} : \sigma_{\text{paper}}) \\
e_{\text{paper}} & := & \{\texttt{title}{=}\mathit{title}, \texttt{conf}{=}\texttt{"EXMPL 2015"}\}
\end{array}
$$

We will detail the syntax and semantics in Sec. 4. To briefly summarize: because of the type ascription σ_{paper} in e_{ex}, semantic control over e_{paper} will be delegated to the *intro opcon definition* of LPROD. It will decide to analyze the the row value of `title` against σ_{title} and the row value of `conf`, a string literal, against σ_{conf}, causing control to pass similarly to RSTR. Satisfied that the term is well-typed, these will generate translations. Thus, we will be able to derive:

$$\emptyset \vdash_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} e_{\text{ex}} \Rightarrow (\sigma_{\text{title}} \rightharpoonup \sigma_{\text{paper}}) \rightsquigarrow \iota_{\text{ex}}$$

where $\iota_{\text{ex}} := \lambda \mathit{title}{:}\mathsf{str}.(\mathit{title}, (\texttt{"EXMPL 2015"}_{\text{IL}}, ()))$. Note that labels need not appear, and $\texttt{"EXMPL 2015"}_{\text{IL}}$ is an internal string (of internal type str, defined suitably). The trailing unit value arises only because it simplifies our exposition. The type annotation on the internal function could be determined because types also have translations, specified by the *type translation judgement* $\vdash_\Phi \sigma \;\texttt{type} \rightsquigarrow \tau^+$, read "$\sigma$ is a type under Φ with translation τ". For example, σ_{title} and σ_{conf} have type translations str and σ_{paper} in turn has $\mathsf{str} \times (\mathsf{str} \times 1)$. Selectively holding type translations abstract will be the key to modular metatheoretic reasoning. For example, LPROD cannot claim that a row has a regular string type but produce a translation inconsistent with this claim: the translation $(\texttt{""}_{\text{IL}}, (\texttt{"EXMPL"}_{\text{IL}}, ()))$ is invalid for a term of type σ_{paper}, despite being of internal type $\mathsf{str} \times (\mathsf{str} \times 1)$. In fact, it will be checked against $\alpha_1 \times (\alpha_2 \times 1)$ (Sec. 4.3).

kinds

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

static terms

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

translational internal types and terms

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

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

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

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

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

Static Language As suggested above, the main novelty of $@\lambda$ is that the types and term and type translations do not arise from a fixed specification. Rather, they are *statically computed* by tycon definitions using a *static language* (SL). The SL is itself a typed lambda calculus where *kinds*, κ, serve as the "types" of *static terms*, σ. Its syntax is shown in Figure 3. The portion of the SL covered by the first row of kinds and static terms, some of which are elided, forms a standard functional language consisting of total functions, polymorphic and inductive kinds, and products and sums [Harper 2012]. It can be seen as a total subset of ML.

Only three new kinds are needed for the SL to serve its role: 1) Ty, classifying types (Sec. 3); 2) ITy, classifying *quoted translational internal types*, used to compute type translations in Sec. 3.4; and (3) ITm, classifying *quoted translational internal terms*, used to compute term translations in Sec. 4.1. The forms $\mathsf{ana}[n](\sigma)$ and $\mathsf{syn}[n]$ will allow tycon definitions to request analysis or synthesis of operator arguments, linking the SL with the EL in Sec. 4.2.

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 judgements $\boldsymbol{\Delta} \vdash \kappa$ and $\boldsymbol{\Delta} \vdash \boldsymbol{\Gamma}$ are defined. The natural number n is simply a bound used to prevent "out of bounds" references to arguments. The computational behavior of static terms (i.e. the *static dynamics*) is defined by a stepping judgement $\sigma \mapsto_{\mathcal{A}} \sigma^+$, a value judgement $\sigma\ \texttt{val}_{\mathcal{A}}$ and an *error raised* judgement $\sigma\ \texttt{err}_{\mathcal{A}}$. $\mathcal{A}$ ranges over *argument environments*, which we return to in Sec. 4.2. The multi-step judgement $\sigma \mapsto^*_{\mathcal{A}} \sigma^+$ is the reflexive, transitive closure of the stepping judgement and the normalization judgement $\sigma \Downarrow_{\mathcal{A}} \sigma'$ is derivable iff $\sigma \mapsto^*_{\mathcal{A}} \sigma'$ and $\sigma'\ \texttt{val}_{\mathcal{A}}$.

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

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

$$\frac{\emptyset \vdash \kappa\ \mathsf{eq} \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa \times \mathsf{ITy}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{other}[m;\kappa]\langle\sigma\rangle :: \mathsf{Ty}}\ \text{(k-ty-other)}$$

Figure 4. Kinding rules for types, which take the form $c\langle\sigma\rangle$ where c is a tycon and σ is the type index.

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

Figure 5. Syntax of tycons. Metavariables $\textsc{tc}$, $\mathbf{op}$ and m range over extension tycon and opcon names and natural numbers, respectively. We omit leading $\cdot$ in examples.

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

$\Phi_{\text{rstr}} := \mathsf{tycon}\ \textsc{rstr}\ \{$	$\Phi_{\text{lprod}} := \mathsf{tycon}\ \textsc{lprod}\ \{$
$\mathsf{trans} = \sigma_{\text{rstr/schema}}\ \mathsf{in}$	$\mathsf{trans} = \sigma_{\text{lprod/schema}}\ \mathsf{in}$ (Sec 3.4)
$\mathsf{ana\ intro} = \sigma_{\text{rstr/intro}};$	$\mathsf{ana\ intro} = \sigma_{\text{lprod/intro}};$ (Sec 4.1)
$\mathsf{syn}\ \mathbf{conc} = \sigma_{\text{rstr/conc}};$	$\mathsf{syn}\ \# = \sigma_{\text{lprod/prj}};$ (Sec 4.4)
$\mathsf{syn}\ \mathbf{case} = \sigma_{\text{rstr/case}};$	$\mathsf{syn}\ \mathbf{conc} = \sigma_{\text{lprod/conc}};$ (Sec 4.4)
$\cdots\} \sim \psi_{\text{rstr}}$	$\cdots\} \sim \psi_{\text{lprod}}$

Figure 6. Example tycon signatures and definitions.

3. Types

Types are static values of kind Ty, i.e. we write $\sigma\ \mathtt{type}_\Phi$ iff $\sigma\ \mathtt{val}_\mathcal{A}$ and $\emptyset\ \emptyset \vdash^n_\Phi \sigma :: \mathsf{Ty}$. All types are of the form $c\langle\sigma\rangle$, where c is a *tycon* and σ is the *type index*. Three kinding rules govern this form, shown in Figure 4, one for each of the three forms for tycons given in Figure 5. The dynamics are simple and tycon-independent: the index is eagerly normalized and errors propagate (see supplement for the complete rules, lemmas and proofs from this paper).

Function Types In our example, we assumed a desugaring from $\sigma_1 \rightharpoonup \sigma_2$ to $\rightharpoonup\langle(\sigma_1, \sigma_2)\rangle$. The rule (k-ty-parr) specifies that the type index of partial function types must be a pair of types. We thus say that $\rightharpoonup$ has *index kind* $\mathsf{Ty} \times \mathsf{Ty}$.

Extension Types For types constructed by an *extension tycon*, written $\textsc{tc}$, rule (k-ty-ext) checks for a *tycon definition* for $\textsc{tc}$ in the tycon context, Φ. The syntax of tycon contexts is shown in Figure 5 and our main examples are in Figure 6. For now, the only relevant detail is that each tycon defines a *tycon signature*, ψ, which in turn defines the index kind used in the second premise of (k-ty-ext).

Types constructed by $\textsc{rstr}$, e.g. σ_{title} and σ_{conf}, specify regular expressions as their indices. The tycon signature of $\textsc{rstr}$, ψ_{rstr} in Figure 6, thus specifies index kind Rx, which classifies static regular expression patterns (defined as an inductive sum kind in the usual way). We wrote the type indices in our example assuming a standard concrete syntax. Recent work has shown how to define such type-specific, or here kind-specific, syntax composably [Omar et al. 2014].

Under the composed context $\Phi_{\text{rstr}}\Phi_{\text{lprod}}$, we defined the labeled product type σ_{paper}. The index kind of $\textsc{lprod}$, given by ψ_{lprod} in Figure 6, is $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$, where list kinds are defined in the usual way, and Lbl classifies static representations of row labels, and we again used kind-specific syntax to approximate a conventional syntax for row specifications.

Other Types Rule (k-ty-other) governs types constructed by a tycon of the form $\mathsf{other}[m;\kappa]$. These will serve only as technical devices to stand in for tycons other than those in a given tycon context in Theorem 5. The natural number m serves to ensure there are arbitrarily many of these tycons. The index pairs any value of *equality kind* κ, discussed in Sec. 3.3, with a value of kind ITy, discussed in Sec. 3.4.

3.1 Type Case Analysis

Types might be thought of as arising from a distinguished "open datatype" [Löh and Hinze 2006] defined by the tycon context. Consistent with this view, a type σ can be case analyzed using $\mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2)$. If the value of σ is constructed by c, its type index is bound to $\boldsymbol{x}$ and branch σ_1 is taken. For totality, a default branch, σ_2, must be provided. For example, the kinding rule for extension tycons is:

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

We will see an example of its use in an opcon definition in Sec. 4.4. The rule for $c{=}\rightharpoonup$ is analagous, but, importantly, no rule for $c{=}\mathsf{other}[m;\kappa]$ is defined (these types always take the default branch and their indices cannot be examined).

3.2 Tycon Contexts

The tycon context well-definedness judgement, $\vdash \Phi$ is given in Figure 7 (omitting the trivial rule for $\Phi = \cdot$).

3.3 Type Equivalence

The first of the three checks in (tcc-ext), and the check in (k-ty-other), simplifies type equivalence: type index kinds must be *equality kinds*, i.e. those for which semantically equivalent values are syntactically equal. Equality kinds are defined by the judgement $\boldsymbol{\Delta} \vdash \kappa\ \mathsf{eq}$ (see supplement) and are exactly analagous to equality types as in Standard ML [Milner et al. 1990]. Arrow kinds are not equality kinds.

3.4 Type Translations

Recall that a type, σ, defines a translation, τ. Extension tycons compute translations for the types they construct as a function of each type's index by specifying a *translation schema* in the tycon structure, θ. The translation schema must have kind $\kappa_{\text{tyidx}} \rightarrow \mathsf{ITy}$, checked by the third premise

(tcc-ext)

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

Figure 7. Tycon context well-definedness.

of (tcc-ext). Terms of kind ITy are introduced by a *quotation form*, $\blacktriangleright(\hat{\tau})$, where $\hat{\tau}$ is a *translational internal type*. Each form of internal type, τ, has a corresponding form of translational internal type. For example, regular string types have type translations abbreviated str. Abbreviating the corresponding translational internal type $\hat{\mathsf{str}}$, we define the translation schema of RSTR as $\sigma_{\text{rstr/trans}} := \lambda \boldsymbol{tyidx}{::}\mathsf{Rx}.\blacktriangleright(\hat{\mathsf{str}})$.

The syntax for $\hat{\tau}$ also includes an "unquote" form, $\blacktriangleleft(\sigma)$, so that they can be constructed compositionally, as well as a form, $\mathsf{trans}(\sigma)$, that refers to another type's translation. These have the following simple kinding rules:

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

(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 semantics for the shared forms propagates the quotation marker recursively to ensure these are reached, e.g.

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

These are needed in the translation schema for LPROD, which generates nested binary product types by folding over the type index and referring to the translations of the types therein. We assume the standard ***fold*** function in defining:

$$\sigma_{\text{lprod/trans}} := \lambda \boldsymbol{tyidx}{::}\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}].\boldsymbol{fold\ tyidx}\ \blacktriangleright(1)$$
$$(\lambda \boldsymbol{h}{:}\mathsf{Lbl} \times \mathsf{Ty}.\lambda \boldsymbol{r}{:}\mathsf{ITy}.\blacktriangleright(\mathsf{trans}(\mathsf{snd}(\boldsymbol{h})) \times \blacktriangleleft(\boldsymbol{r}))$$

Applying this translation schema to the index of σ_{paper}, for example, produces the value $\sigma_{\text{paper/trans}} := \blacktriangleright(\hat{\tau}_{\text{paper/trans}})$ where $\hat{\tau}_{\text{paper/trans}} := \mathsf{trans}(\sigma_{\text{title}}) \times (\mathsf{trans}(\sigma_{\text{conf}}) \times 1)$. Note that references to translations of types are retained in values of kind ITy, while unquote forms are eliminated (see supplement for the full semantics of quotations).

3.4.1 Selective Type Translation Abstraction

References to type translations are maintained in values like this to allow us to selectively hold them abstract. This can be thought of as analagous to the process in ML by which the true identity of an abstract type in a module is held abstract outside the module until after typechecking. The judgement $\hat{\tau} \parallel \mathcal{D} \looparrowright^c_\Phi \tau^+ \parallel \mathcal{D}^+$ relates a normalized translational internal type $\hat{\tau}$ to an internal type τ, called a *selectively abstracted type translation* because references to translations of types *constructed by a tycon other than the delegated tycon*, c, are replaced by a corresponding type variable, α. For example, $\hat{\tau}_{\text{paper/trans}} \parallel \emptyset \looparrowright^{\text{LPROD}}_{\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \tau_{\text{paper/abs}} \parallel \mathcal{D}_{\text{paper/abs}}$ where $\tau_{\text{paper/abs}} := \alpha_1 \times (\alpha_2 \times 1)$.

The *type translation store* $\mathcal{D} ::= \emptyset \mid \mathcal{D}, \sigma \leftrightarrow \tau/\alpha$ maintains the correspondence between types, their actual translations and the distinct type variables appearing in their place, e.g. $\mathcal{D}_{\text{paper/abs}} := \sigma_{\text{title}} \leftrightarrow \mathsf{str}/\alpha_1, \sigma_{\text{conf}} \leftrightarrow \mathsf{str}/\alpha_2$. The judgement $\mathcal{D} \rightsquigarrow \delta^+ : \Delta^+$ constructs the n-ary *type substitution*, $\delta ::= \emptyset \mid \delta, \tau/\alpha$, and corresponding internal type formation context, Δ, implied by the type translation store $\mathcal{D}$. For example, $\mathcal{D}_{\text{paper/abs}} \rightsquigarrow \delta_{\text{paper/abs}} : \Delta_{\text{paper/abs}}$ where $\delta_{\text{paper/abs}} := \mathsf{str}/\alpha_1, \mathsf{str}/\alpha_2$ and $\Delta_{\text{paper/abs}} := \alpha_1, \alpha_2$.

We can apply type substitutions to internal types, terms and typing contexts, written $[\delta]\tau$, $[\delta]\iota$ and $[\delta]\Gamma$, respectively. For example, $[\delta_{\text{paper/abs}}]\tau_{\text{paper/abs}}$ is $\tau_{\text{paper}} := \mathsf{str} \times (\mathsf{str} \times 1)$, i.e. the actual type translation of σ_{paper}. Indeed, we can now define the type translation judgement, $\vdash_\Phi \sigma\ \mathtt{type} \rightsquigarrow \tau$, mentioned in Sec. 2. We simply determine any selectively abstracted translation, then apply the implied substitution:

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

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

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

Otherwise, it is held abstract via a fresh type variable added to the store (the supplement gives rule (abs-ty-stored) for retrieving it if already there):

(abs-ext-not-delegated-new)
$$\frac{\begin{array}{c}c \neq \textsc{tc} \qquad \textsc{tc}\langle\sigma_{\text{tyidx}}\rangle \notin \text{dom}(\mathcal{D}) \\ \mathsf{tycon}\ \textsc{tc}\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \psi \in \Phi \\ \sigma_{\text{schema}}(\sigma_{\text{tyidx}}) \Downarrow_{\cdot;\emptyset;\Phi} \blacktriangleright(\hat{\tau}) \qquad \hat{\tau} \parallel \mathcal{D} \looparrowright^c_\Phi \tau \parallel \mathcal{D}' \\ \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \Delta \vdash \tau \qquad (\alpha\ \text{fresh})\end{array}}{\mathsf{trans}(\textsc{tc}\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \looparrowright^c_\Phi \alpha \parallel \mathcal{D}', \textsc{tc}\langle\sigma_{\text{tyidx}}\rangle \leftrightarrow [\delta]\tau/\alpha}$$

The translations of "other" types are given directly in their indices (in Sec. 5, we will replace some extension types with other types, and this is how we preserve their translations):

(abs-other-delegated)
$$\frac{\hat{\tau} \parallel \mathcal{D} \looparrowright^{\mathsf{other}[m;\kappa]}_\Phi \tau \parallel \mathcal{D}' \qquad \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \Delta \vdash \tau}{\mathsf{trans}(\mathsf{other}[m;\kappa]\langle(\sigma, \blacktriangleright(\hat{\tau}))\rangle) \parallel \mathcal{D} \looparrowright^{\mathsf{other}[m;\kappa]}_\Phi \tau \parallel \mathcal{D}'}$$

Rule (abs-other-not-delegated-new) is analagous to (abs-ext-not-delegated-new), and is shown in the supplement.

The translations of function types are not held abstract, so that lambdas, which are built in, can be the sole binding construct in the EL:

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

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

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

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

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

$$\text{(ana-fix)}\ \frac{\Upsilon, x : \sigma \vdash_\Phi e \Leftarrow \sigma \leadsto \iota \qquad \vdash_\Phi \sigma\ \mathtt{type} \leadsto \tau}{\Upsilon \vdash_\Phi \mathsf{fix}(x.e) \Leftarrow \sigma \leadsto \mathsf{fix}[\tau](x.\iota)}$$

$$\text{(ana-lam)}\ \frac{\Upsilon, x : \sigma_1 \vdash_\Phi e \Leftarrow \sigma_2 \leadsto \iota \qquad \vdash_\Phi \sigma_1\ \mathtt{type} \leadsto \tau_1}{\Upsilon \vdash_\Phi \lambda x.e \Leftarrow \rightharpoonup\langle(\sigma_1, \sigma_2)\rangle \leadsto \lambda x{:}\tau_1.\iota}$$

$$\text{(syn-lam)}\ \frac{\emptyset\,\emptyset \vdash^0_\Phi \sigma_1 :: \mathsf{Ty} \qquad \sigma_1 \Downarrow_{\cdot;\emptyset;\Phi} \sigma_1' \qquad \Upsilon, x : \sigma_1' \vdash_\Phi e \Rightarrow \sigma_2 \leadsto \iota \qquad \vdash_\Phi \sigma_1'\ \mathtt{type} \leadsto \tau_1}{\Upsilon \vdash_\Phi \lambda x{:}\sigma_1.e \Rightarrow \rightharpoonup\langle(\sigma_1', \sigma_2)\rangle \leadsto \lambda x{:}\tau_1.\iota}$$

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

$$\text{(ana-intro)}\ \frac{\begin{array}{c}\mathsf{tycon}\ \mathrm{TC}\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\} \in \Phi \\ \mathsf{intro}[\kappa_{\text{tmidx}}] \in \chi \qquad \emptyset\,\emptyset \vdash^0_\Phi \sigma_{\text{tmidx}} :: \kappa_{\text{tmidx}} \\ \mathsf{ana\ intro} = \sigma_{\text{def}} \in \omega \qquad |\overline{e}| = n \qquad \mathsf{args}(n) = \sigma_{\text{args}} \\ \sigma_{\text{def}}(\sigma_{\text{tyidx}})(\sigma_{\text{tmidx}})(\sigma_{\text{args}}) \Downarrow_{\overline{e};\Upsilon;\Phi} \triangleright(\hat{\iota}) \\ \vdash^{\mathrm{TC}}_{\overline{e};\Upsilon;\Phi} \hat{\iota} : \mathrm{TC}\langle\sigma_{\text{tyidx}}\rangle \leadsto \iota\end{array}}{\Upsilon \vdash_\Phi \mathsf{intro}[\sigma_{\text{tmidx}}](\overline{e}) \Leftarrow \mathrm{TC}\langle\sigma_{\text{tyidx}}\rangle \leadsto \iota}$$

$$\text{(syn-targ)}\ \frac{\begin{array}{c}\Upsilon \vdash_\Phi e_{\text{targ}} \Rightarrow \mathrm{TC}\langle\sigma_{\text{tyidx}}\rangle \leadsto \iota_{\text{targ}} \\ \mathsf{tycon}\ \mathrm{TC}\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \mathsf{tcsig}[\kappa_{\text{tyidx}}]\ \{\chi\} \in \Phi \\ \mathbf{op}[\kappa_{\text{tmidx}}] \in \chi \qquad \emptyset\,\emptyset \vdash^0_\Phi \sigma_{\text{tmidx}} :: \kappa_{\text{tmidx}} \\ \mathsf{syn}\ \mathbf{op} = \sigma_{\text{def}} \in \omega \qquad |e_{\text{targ}}; \overline{e}| = n \qquad \mathsf{args}(n) = \sigma_{\text{args}} \\ \sigma_{\text{def}}(\sigma_{\text{tyidx}})(\sigma_{\text{tmidx}})(\sigma_{\text{args}}) \Downarrow_{(e_{\text{targ}};\overline{e});\Upsilon;\Phi} (\sigma, \triangleright(\hat{\iota})) \\ \vdash^{\mathrm{TC}}_{(e_{\text{targ}};\overline{e});\Upsilon;\Phi} \hat{\iota} : \sigma \leadsto \iota\end{array}}{\Upsilon \vdash_\Phi \mathsf{targop}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e}) \Rightarrow \sigma \leadsto \iota}$$

$$\text{(ana-intro-other)}\ \frac{\begin{array}{c}|\overline{e}| = n \qquad \emptyset\,\emptyset \vdash^n_\Phi \triangleright(\hat{\iota}) :: \mathsf{ITm} \qquad \triangleright(\hat{\iota})\ \mathtt{val}_{\overline{e};\Upsilon;\Phi} \\ \vdash^{\mathsf{other}[m;\kappa]}_{\overline{e};\Upsilon;\Phi} \hat{\iota} : \mathsf{other}[m;\kappa]\langle\sigma_{\text{tyidx}}\rangle \leadsto \iota\end{array}}{\Upsilon \vdash_\Phi \mathsf{intro}[\triangleright(\hat{\iota})](\overline{e}) \Leftarrow \mathsf{other}[m;\kappa]\langle\sigma_{\text{tyidx}}\rangle \leadsto \iota}$$

$$\text{(syn-targ-other)}\ \frac{\begin{array}{c}\Upsilon \vdash_\Phi e_{\text{targ}} \Rightarrow \mathsf{other}[m;\kappa]\langle\sigma_{\text{tyidx}}\rangle \leadsto \iota_{\text{targ}} \\ |e_{\text{targ}}; \overline{e}| = n \qquad \emptyset\,\emptyset \vdash^n_\Phi \triangleright(\hat{\iota}) :: \mathsf{ITm} \qquad \triangleright(\hat{\iota})\ \mathtt{val}_{\overline{e};\Upsilon;\Phi} \\ \sigma\ \mathtt{type}_\Phi \qquad \vdash^{\mathsf{other}[m;\kappa]}_{(e_{\text{targ}};\overline{e});\Upsilon;\Phi} \hat{\iota} : \sigma \leadsto \iota\end{array}}{\Upsilon \vdash_\Phi \mathsf{targop}[\mathbf{op}; (\sigma, \triangleright(\hat{\iota}))](e_{\text{targ}}; \overline{e}) \Rightarrow \sigma \leadsto \iota}$$

Figure 8. Typing

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

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

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

Figure 9. Opcon structure kinding against tycon signatures

4. External Terms

Now that we have established how types are constructed and how type translations are computed, we are ready to give the semantics for external terms, shown in Figure 8.

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

Rules (syn-var) states that variables synthesize types, as is standard. Functions operate in the standard manner given our definitions of types and type translations (used to generate annotations in the IL). We use Plotkin's fixpoint operator for general recursion (cf. [Harper 2012]), and define it only analytically with rule (ana-fix). We also define an analytic form of lambda without a type ascription to emphasize that bidirectional typing allows you to omit type ascriptions in analytic positions.

4.1 Generalized Intro Operations

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

Before discussing rules (ana-intro) and (ana-intro-other), we note that we can recover a variety of standard concrete introductory forms by desugaring. For example, the string literal form, `"s"`, desugars to $\mathsf{intro}[\texttt{"s"}_{\text{SL}}](\cdot)$, i.e. the term index is the corresponding static value of kind Str and there are no arguments. Similarly, we define a generalized labeled collection form, $\{\mathtt{lbl}_1 = e_1, \ldots, \mathtt{lbl}_n = e_n\}$, that desugars to $\mathsf{intro}[[\mathtt{lbl}_1, \ldots, \mathtt{lbl}_n]](e_1; \ldots; e_n)$, i.e. a static list constructed from the row labels is the term index and the corresponding row values are the arguments. Additional desugarings are discussed in the supplement. The literal form in e_{paper}, from Sec. 2, for example, desugars to $e_{\text{paper}} := \mathsf{intro}[[\mathtt{title}, \mathtt{conf}]](\mathit{title}; \mathsf{intro}[\texttt{"EXMPL 2015"}_{\text{SL}}](\cdot))$.

Let us now derive the typing judgement in Sec. 2. We first apply (syn-lam), which will ask $(e_{\text{paper}} : \sigma_{\text{paper}})$ to synthesize a type in the typing context $\Upsilon_{\text{ex}} = \mathit{title} : \sigma_{\text{title}}$. Then (ascribe) will analyze e_{paper} against σ_{paper} via (ana-intro).

$\lambda \textbf{\textit{tyidx}}$::Rx.$\lambda \textbf{\textit{tmidx}}$::Str.$\lambda \textbf{\textit{args}}$::List[Arg].
let $\textbf{\textit{aok}}$ :: 1 = $\textbf{\textit{arity0 args}}$ in
let $\textbf{\textit{rok}}$ :: 1 = $\textbf{\textit{rmatch tyidx tmidx}}$ in
$\textbf{\textit{str\_of\_Str tmidx}}$

Figure 10. The intro opcon definition for RSTR, $\sigma_{\text{rstr/intro}}$.

The first premise of (ana-intro) simply finds the tycon definition for the tycon of the type provided for analysis, in this case LPROD. The second premise extracts the *intro term index kind*, κ_{tmidx}, from the *opcon signature*, χ, it specifies. This is simply the kind of term index expected by the tycon, e.g. in Figure 6, LPROD specifies List[Lbl], so that it can use the labeled collection form, while RSTR specifies Str, so that it can use the string literal form. The third premise checks the provided term index against this kind.

The fourth premise extracts the *intro opcon definition* from the *opcon structure*, ω, of the tycon structure, calling it σ_{def}. This is a static function that is applied, in the seventh premise, to determine whether the term is well-typed, raising an error if not or computing a translation if so. Its kind is checked by the judgement $\vdash_\Phi \omega \sim \psi$, which was the final premise of the rule (tcc-ext) and is defined in Figure 9. Rule (ocstruct-intro) specifies that it has access to the type index, the term index and an interface to the list of arguments, discussed below, and returns a *quoted translational internal term* of kind ITm, analagous to ITy. The intro opcon definitions for RSTR and LPROD are given in Figures 10 and 11.

Though the latter will be encountered first in our example derivation, let us first consider the intro opcon definition in Figure 10 because it is more straightforward. It will be used to analyze the row value intro["EXMPL 2015"$_{\text{SL}}$]$(\cdot)$ against σ_{conf}. It begins by making sure that no arguments were passed in using the helper function $\textbf{\textit{arity0}}$ :: List[Arg] $\rightarrow$ 1 defined such that any non-empty list will raise an error, via the static term raise[1]. In practice, the tycon provider would specify an error message here. Next, it checks the string provided as the term index against the regular expression given as the type index using $\textbf{\textit{rmatch}}$ :: Rx $\rightarrow$ Str $\rightarrow$ 1, which we assume is defined in the usual way and again raises an error on failure. Finally, the *translational internal string* corresponding to the static string provided as the term index is generated via the helper function $\textbf{\textit{str\_of\_Str}}$ :: Str $\rightarrow$ ITm.

Static terms of kind ITm are introduceed by the quotation form, $\triangleright(\hat{\iota})$, where $\hat{\iota}$ is a *translational internal term*. This is analagous to the form $\blacktriangleright(\hat{\tau})$ for ITy in Sec. 3.4. Each form in the syntax of ι has a corresponding form in the syntax for $\hat{\iota}$ and the kinding rules and dynamics simply recurse through these in the same manner as in Sec. 3.4. There is also an analagous unquote form, $\triangleleft(\sigma)$. The two interesting forms of translational internal term are anatrans$[n](\sigma)$ and syntrans$[n]$. These stand in for the translation of argument n, the first if it arises via analysis against type σ and the second if it arises via type synthesis. Before giving the rules, let us motivate the mechanism with the intro opcon definition for LPROD, shown in Figure 11.

$\lambda \textbf{\textit{tyidx}}$:List[Lbl $\times$ Ty].$\lambda \textbf{\textit{tmidx}}$:List[Lbl].$\lambda \textbf{\textit{args}}$:List[Arg].
let $\textbf{\textit{inhabited}}$: 1 = $\textbf{\textit{uniqmap tyidx}}$ in
$\textbf{\textit{fold3 tyidx tmidx args}}$ $\triangleright(())$
$\lambda \textbf{\textit{rowtyidx}}$:Lbl $\times$ Ty.$\lambda \textbf{\textit{rowtmidx}}$:Lbl.$\lambda \textbf{\textit{rowarg}}$:Arg.$\lambda \textbf{\textit{r}}$:ITm.
letpair $(\textbf{\textit{rowlbl}}, \textbf{\textit{rowty}})$ = $\textbf{\textit{rowtyidx}}$ in
let $\textbf{\textit{lok}}$ :: 1 = $\textbf{\textit{lbleq rowlbl rowtmidx}}$ in
let $\textbf{\textit{rowtr}}$ :: ITm = $\textbf{\textit{ana rowarg rowty}}$ in
$\triangleright((\triangleleft(\textbf{\textit{rowtr}}), \triangleleft(\textbf{\textit{r}})))$

Figure 11. The intro opcon definition for LPROD, $\sigma_{\text{lprod/intro}}$.

The first line checks that the type provided is inhabited, in this case by checking that there are no duplicate labels via the helper function $\textbf{\textit{uniqmap}}$ :: List[Lbl $\times$ Ty] $\rightarrow$ 1, raising an error if there are (we briefly discuss alternative strategies in the supplement). The rest of the definition folds over the three lists provided as input: the list mapping row labels to types provided as the type index, the list of labels provided as the term index, and the list of argument interfaces, which give access to the corresponding row values. We assume a straightforward helper function, $\textbf{\textit{fold3}}$, 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 checks, for each row, that the label provided in the term index matches the label in the type index using helper function $\textbf{\textit{lbleq}}$ :: Lbl $\rightarrow$ Lbl $\rightarrow$ 1. Then, we request type analysis of the corresponding argument, $\textbf{\textit{rowarg}}$, against the type in the type index, $\textbf{\textit{rowty}}$, by writing $\textbf{\textit{ana rowarg rowty}}$. Here, $\textbf{\textit{ana}}$ is a helper function defined in Sec. 4.2 below that triggers type analysis of the provided argument. If this succeeds, it evaluates to a translational internal term of the form $\triangleright$(anatrans$[n](\sigma)$), where n is the position of $\textbf{\textit{rowarg}}$ in $\textbf{\textit{args}}$ and σ is the value of $\textbf{\textit{rowty}}$. If analysis fails, it raises an error. The final line constructs a nested tuple based on the row value's translation and the recursive result. Taken together, the translational internal term that will be generated for our example involving e_{paper} above is $\hat{\iota}_{\text{paper}}$:= (anatrans$[0](\sigma_{\text{title}})$, (anatrans$[1](\sigma_{\text{conf}})$, ())), i.e. it simply recalls that the two arguments were analyzed against σ_{title} and σ_{conf}, without yet inserting their translations directly. This will be done after *translation validation*, triggered by the final premise of (ana-intro) and described in Sec. 4.3.

4.2 Argument Interfaces

The kind of *argument interfaces* is Arg := (Ty $\rightarrow$ ITm) $\times$ (1 $\rightarrow$ Ty $\times$ ITm), i.e. a product of functions, one for analysis and the other for synthesis. The helpers $\textbf{\textit{ana}}$ and $\textbf{\textit{syn}}$ only project them out, e.g. $\textbf{\textit{ana}}$:= $\lambda \textbf{\textit{arg}}$::Arg.fst($\textbf{\textit{arg}}$). To actually perform analysis or synthesis, we must provide a link between the dynamics of the static language and the EL's typing rules. This is purpose of the static forms ana$[n](\sigma)$ and syn$[n]$. When consider an argument list of length n, written $|\overline{e}| = n$, the opcon definition will receive a static list of length n where the jth entry is the argument interface $(\lambda \textbf{\textit{ty}}$::Ty.ana$[j](\textbf{\textit{ty}})$, $\lambda\_$::1.syn$[j])$. This *argument interface list* is generated by the judgement args$(n) = \sigma_{\text{args}}$.

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

Figure 12. Kinding for the SL-EL interface.

The index n on the kinding judgement is an upper bound on the argument index of terms of the form $\mathsf{ana}[n](\sigma)$ and $\mathsf{syn}[n]$, enforced in Figure 12. Thus, if $\mathsf{args}(n) = \sigma_{\text{args}}$ then $\emptyset\ \emptyset \vdash^n_\Phi \sigma_{\text{args}} :: \mathsf{List}[\mathsf{Arg}]$. The rule (ocstruct-intro) ruled out writing either of these forms explicitly in an opcon definition by checking against bound $n = 0$. This is to prevent out-of-bounds errors: tycon providers can only access these forms via the argument interface list, which has the correct length.

The static dynamics of $\mathsf{ana}[n](\sigma)$ are interesting. After normalizing σ, the argument environment, which contains the arguments themselves and the typing and tycon contexts, $\mathcal{A} ::= \overline{e}; \Upsilon; \Phi$, is consulted to analyze the nth argument against σ. If this succeeds, $\triangleright(\mathsf{anatrans}[n](\sigma))$ is generated:

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

If it fails, an error is raised:

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

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

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

The kinding rules also prevent these translational forms from being well-kinded when $n = 0$ and, like $\mathsf{trans}(\sigma)$ in Sec. 3.4, they are retained in values of kind ITm.

4.3 Translation Validation

The judgement $\vdash^c_{\mathcal{A}} \hat\iota : \sigma \rightsquigarrow \iota^+$, defined by a single rule in Figure 13 and appearing as the final premise of (ana-intro) and the other rules described below, is pronounced "translational internal term $\hat\iota$ generated by an opcon associated with tycon c under argument environment $\mathcal{A}$ for an operation with type σ is valid, so translation ι is produced". For example,

$$\vdash^{\text{LPROD}}_{(\mathit{title};\texttt{"EXMPL 2015"});\Upsilon_{\text{ex}};\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \hat\iota_{\text{paper}} : \sigma_{\text{paper}} \rightsquigarrow \iota_{\text{paper}}$$

The purpose of translation validation is to check that the generated translation will be well-typed *no matter what the translations of types other than those constructed by c are.*

The first premise generates the selectively abstracted type translation for σ given that c was the delegated tycon as described in Sec. 3.4.1. In our running example, this is $\tau_{\text{paper/abs}}$, i.e. $\alpha_0 \times (\alpha_1 \times 1)$.

The judgement $\hat\iota \parallel \mathcal{D}\ \mathcal{G} \hookrightarrow^c_{\mathcal{A}} \iota^+ \parallel \mathcal{D}^+\ \mathcal{G}^+$, appearing as the next premise, relates a translational internal term $\hat\iota$ to an internal term ι called a *selectively abstracted term translation*, because all references to the translation of an argument (having any type) are replaced with a corresponding variable, x, which will be of the selectively abstracted type translation of the type of that argument. For our example, $\hat\iota_{\text{paper}} \parallel \mathcal{D}_{\text{paper/abs}}\ \emptyset \hookrightarrow^{\text{LPROD}}_{(\mathit{title};\texttt{"EXMPL 2015"});\Upsilon_{\text{ex}};\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \iota_{\text{paper/abs}} \parallel \mathcal{D}_{\text{paper/abs}}\ \mathcal{G}_{\text{paper/abs}}$ where $\iota_{\text{paper/abs}} := (x_0, (x_1, ()))$.

$$\boxed{\vdash^c_{\mathcal{A}} \hat\iota : \sigma \rightsquigarrow \iota^+}$$

$$\text{(validate-tr)}\quad \frac{\begin{array}{c}\mathsf{trans}(\sigma) \parallel \emptyset \hookrightarrow^c_\Phi \tau_{\text{abs}} \parallel \mathcal{D} \qquad \hat\iota \parallel \mathcal{D}\ \emptyset \hookrightarrow^c_{\overline{e};\Upsilon;\Phi} \iota_{\text{abs}} \parallel \mathcal{D}'\ \mathcal{G} \\ \mathcal{D}' \rightsquigarrow \delta : \Delta_{\text{abs}} \qquad \mathcal{G} \rightsquigarrow \gamma : \Gamma_{\text{abs}} \qquad \Delta_{\text{abs}}\ \Gamma_{\text{abs}} \vdash \iota_{\text{abs}} : \tau_{\text{abs}}\end{array}}{\vdash^c_{\overline{e};\Upsilon;\Phi} \hat\iota : \sigma \rightsquigarrow [\delta][\gamma]\iota_{\text{abs}}}$$

Figure 13. Translation Validation

The type translation store, $\mathcal{D}$, discussed previously, and term translation store, $\mathcal{G}$, track these correspondences. Term translation stores have syntax $\mathcal{G} ::= \emptyset \mid \mathcal{G}, n : \sigma \rightsquigarrow \iota/x : \tau$. Each entry can be read "argument n having type σ and translation ι appears as variable x with type τ". In the example above, $\mathcal{G}_{\text{paper/abs}} := 0 : \sigma_{\text{title}} \rightsquigarrow \mathit{title}/x_0 : \alpha_0, 1 : \sigma_{\text{conf}} \rightsquigarrow \texttt{"EXMPL 2015"}_{\text{IL}}/x_1 : \alpha_1$.

To derive this, the judgement proceeded recursively along shared forms, as in Sec. 3.4.1. The interesting rule for argument translations derived via analysis is below ($\mathsf{syntrans}[n]$ is analagous; see supplement):

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

The third premise of (validate-tr) generates the type substitution and type formation contexts implied by the final type translation store as described in Sec. 3.4.1. Similarly, each term translation store $\mathcal{G}$ implies an internal term substitution, $\gamma ::= \emptyset \mid \gamma, \iota/x$, and corresponding Γ by the judgement $\mathcal{G} \rightsquigarrow \gamma : \Gamma$, appearing as the fourth premise. Here, $\gamma_{\text{paper/abs}} := \mathit{title}/x_0, \texttt{"EXMPL 2015"}_{\text{IL}}/x_1$ and $\Gamma_{\text{paper/abs}} := x_0 : \alpha_0, x_1 : \alpha_1$.

The critical fifth premise checks the selectively abstracted term translation $\iota_{\text{paper/abs}}$ against the selectively abstracted type translation $\tau_{\text{paper/abs}}$ under these contexts via the internal statics. Here, $\Delta_{\text{paper/abs}}\ \Gamma_{\text{paper/abs}} \vdash \iota_{\text{paper/abs}} : \tau_{\text{paper/abs}}$, i.e.:

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

In summary, the translation of the labeled product e_{paper} generated by LPROD is checked with the references to term and type translations of regular strings replaced by variables and type variables, respectively. But because the definition treated arguments parametrically, the check succeeds.

Applying the substitutions $\gamma_{\text{paper/abs}}$ and $\delta_{\text{paper/abs}}$ in the conclusion of the rule, we arrive at the actual term translation $\iota_{\text{paper}} := (\mathit{title}, (\texttt{"EXMPL 2015"}_{\text{IL}}, ()))$. Note that ι_{paper} has type τ_{paper} under the translation of Υ_{ex}, i.e. $\vdash \Upsilon_{\text{ex}}\ \mathtt{ctx} \rightsquigarrow \Gamma_{\text{ex}}$ where $\Gamma_{\text{ex}} := \mathit{title} : \mathsf{str}$. This relationship will always hold, and implies type safety (Sec. 5).

Had we attempted to "smuggle out" a value of regular string type that violated the regular string invariant, e.g. generating $\hat{\iota}_{\text{bad}} := (\texttt{""}, (\texttt{""}, ()))$, it would fail, because even though $(\texttt{""}_{\text{IL}}, (\texttt{""}_{\text{IL}}, ())) : \mathsf{str} \times \mathsf{str} \times 1$, it is not the case that $(\texttt{""}_{\text{IL}}, (\texttt{""}_{\text{IL}}, ())) : \alpha_0 \times (\alpha_1 \times 1)$. We call this property *translation independence*.

4.4 Generalized Targeted Operations

All non-introductory operations go through the form for *targeted operations*, $\mathsf{targop}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e})$, 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. Concrete desugarings for this form include $e_{\text{targ}}.\mathbf{op}\langle\sigma_{\text{tmidx}}\rangle(\overline{e})$ (and variants where the term index or arguments are omitted), projection syntax for use by record-like types, $e_{\text{targ}}\#\texttt{lbl}$, which desugars to $\mathsf{targop}[\#; \texttt{lbl}](e_{\text{targ}}; \cdot)$, and $e_{\text{targ}} \cdot e_{\text{arg}}$, which desugars to $\mathsf{targop}[\mathbf{conc}; ()](e_{\text{targ}}; e_{\text{arg}})$. We show other desugarings, e.g. case analysis, in the supplement.

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

Figure 6 showed portions of the opcon signatures of $\textsc{rstr}$ and $\textsc{lprod}$. The opcons associated with $\textsc{rstr}$ are taken directly from Fulton et al.'s specification of regular string types [Fulton et al. 2014], with the exception of **case**, which generalizes case analysis as defined there to arbitrary string patterns, based on the form of desugaring we show in the supplement. The opcons associated with $\textsc{lprod}$ are also straightforward: **#** projects out the row with the provided label and **conc** concatenates two labeled products (updating common rows with the value from the right argument). Note that both $\textsc{rstr}$ and $\textsc{lprod}$ can define concatenation.

The fifth premise of (syn-targ) extracts the *targeted opcon definition* of **op** from the opcon structure, ω. Like the intro opcon definition, this is a static function that generates a translational internal term on the basis of the target tycon's type index, the term index and an argument interface list. Targeted opcon definitions additionally synthesize a type. The rule (ocstruct-targ) in Figure 9 ensures that it is well-kinded. For example, the definition of $\textsc{rstr}$'s **conc** opcon is shown in Figure 14 (others will be in the supplement).

The helper function ***arity2*** checks that two arguments, including the target argument, were provided. We then request synthesis of both arguments. We can ignore the type synthesized by the first because by definition it is a regular string type with type index ***tyidx***. We case analyze the second against $\textsc{rstr}$, to extract its index regular expression (raising an error if it is not of regular string type). We then synthesize the resulting regular string type, using the helper

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

Figure 14. A targeted opcon definition, $\sigma_{\text{rstr/conc}}$.

function ***rxconcat*** $:: \mathsf{Rx} \to \mathsf{Rx} \to \mathsf{Rx}$ which generates the synthesized type's index by concatenating the indices of the argument's types consistent with the specification in [Fulton et al. 2014]. Finally the translation is generated using helper function $\mathit{sconcat} : \mathsf{str} \to \mathsf{str} \to \mathsf{str}$, the translational term for which we assume has been substituted in directly.

The last premise of (syn-targ) again performs translation validation as described above. The only difference relative to (ana-intro) is that that we check the term translation against the synthesized type but the delegated tycon is that of the type synthesized by the target argument.

4.5 Operations Over Other Types

The rules (ana-intro-other) and (syn-targ-other) are used to introduce and simulate targeted operations on terms of all types constructed by any "other" tycon. In both cases, the term index, rather than the tycon context, directly specifies the translational internal term to be used.

5. Metatheory

We will now state the key metatheoretic properties of $@\lambda$. The proofs are in the supplement.

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

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

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

The case in the proof of Theorem 2 for $\sigma = \mathsf{syn}[n]$ requires that the following theorem be mutually defined.

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

Type Safety Type safety in a typed translation semantics requires that well-typed external terms translate to well-typed internal terms. Type safety for the IL [Harper 2012] then implies that evaluation cannot go wrong. To prove this, we must in fact prove a stronger theorem: that a term's translation has its type's translation under the typing context's translation (the analagous notion is *type-preserving compilation* in type-directed compilers [Tarditi et al. 1996]):

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

Proof Sketch. The interesting cases are (ana-intro), (ana-intro-other), (syn-trans) and (syn-trans-other); the latter two arise via subsumption. The result follows directly from the translation validation process, combined with lemmas that state that all variables in Δ_{abs} and Γ_{abs} in (tr-validate) have well-formed/well-typed substitutions in δ and γ, so applying in the conclusion them gives a well-typed term. $\square$

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

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

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

Proof Sketch. We have fixed the tycon context Φ_{rstr}, so we can essentially treat the calculus like a type-directed compiler for a calculus with only two tycons, $\rightharpoonup$ and RSTR, plus some "other" one. Such a calculus and compiler specification was given in [Fulton et al. 2014], so we must simply show that the opcon definitions in RSTR adequately satisfy these specification using standard techniques for the SL, a simply-typed functional language [Chlipala 2007]. The only "twist" is that the rule (syn-targ-other) can synthesize a regular string type paired with any translational term $\hat{\tau}$. But it will be validated against $\tau_{\text{abs}} = \alpha$ because rule (abs-ext-not-delegated-new) applies in that case. Thus, the invariants cannot be violated by direct application of parametricity in the IL (i.e. this case can always be dispatched via a "free theorem") [Wadler 1989]. $\square$

Another way to interpret this argument is that "other" types simulate all types that might arise from "the future" in that they can have any valid type translation (given directly in the type index) and their operators can produce any valid term translation (given directly in the term index). Because they are distinct from all "present" tycons, our translation validation procedure ensures that they can be reasoned about uniformly – they cannot possibly violate present invariants because they do not even know what type of term they need to generate. The only way to generate a term of type α is via an argument, which inductively obeys all tycon invariants.

Theorem 5 (Conservativity)**.** *If* $\vdash \Phi$ *and* $\text{TC} \in dom(\Phi)$ *and a tycon invariant for* TC *holds under* Φ*:*

- *For all* $\Upsilon, e, \sigma_{tyidx}$*, if* $\Upsilon \vdash_\Phi e \Leftarrow \text{TC}\langle\sigma_{tyidx}\rangle \rightsquigarrow \iota$ *and* $\vdash_\Phi \Upsilon\ \texttt{ctx} \rightsquigarrow \Gamma$ *and* $\vdash_\Phi \text{TC}\langle\sigma_{tyidx}\rangle \rightsquigarrow \tau$ *then* $P(\Gamma, \sigma_{tyidx}, \iota)$.

then for all $\Phi' = \Phi, \mathsf{tycon}\ \text{TC}'\ \{\theta'\} \sim \mathsf{tcsig}[\kappa']\ \{\chi'\}$ *such that* $\vdash \Phi'$*, the same tycon invariant holds under* Φ'*:*

- *For all* $\Upsilon, e, \sigma_{tyidx}$*, if* $\Upsilon \vdash_{\Phi'} e \Leftarrow \text{TC}\langle\sigma_{tyidx}\rangle \rightsquigarrow \iota$ *and* $\vdash_{\Phi'} \Upsilon \rightsquigarrow \Gamma$ *and* $\vdash_{\Phi'} \text{TC}\langle\sigma_{tyidx}\rangle \rightsquigarrow \tau$ *then* $P(\Gamma, \sigma_{tyidx}, \iota)$.

(if proposition $P(\Gamma, \sigma, \iota)$ *is* modular*, defined below)*

Proof Sketch. Our proof of the more general property is a realization of this intuition that "other" types simulate future types. We simply map every well-typed term under Φ' to a well-typed term under Φ with the same translation, and if the term has a type constructed by a tycon in Φ, e.g. TC, the new term has a type constructed by that tycon with the same type translation, and only a slightly different type index. In particular, the mapping's effect on static terms is to replace all types constructed by TC' with a type constructed by $\mathsf{other}[m; \kappa'_{\text{tyidx}}]$. If $P(\Gamma, \sigma_{\text{tyidx}}, \iota)$ is preserved under this transformation on σ_{tyidx} then we can simply invoke the existing proof of the tycon invariant. We call such propositions *modular*. Non-modular propositions are uninteresting because they distinguish tycons "from the future". Our regular string proposition is clearly modular because regular expressions don't contain types at all.

On external terms, the mapping replaces all intro and targeted terms that delegated to TC' with equivalent ones that pass through rules (ana-intro-other) and (syn-targ-other) by pre-applying the intro and targeted opcon definitions to generate the term indices. $\square$

6. Related Work and Discussion

We are not the first to use a semantics distinguishing the EL from a smaller IL for a language specification. For example, the Harper-Stone semantics for Standard ML takes a similar approach, though the EL and IL are governed by a common, and fixed, type system there [Harper and Stone 2000]. Our specification style is also comparable to that of the initial stage of a type-directed compiler, e.g. the TIL compiler for Standard ML [Tarditi et al. 1996], here lifted "one level up" into the semantics of the language itself and made extensible.

Language-integrated static term rewriting systems, like Template Haskell [Sheard and Peyton Jones 2002] and Scala's static macros [Burmako 2013], can be used to decrease complexity when an isomorphic embedding into the underlying type system is already possible. Similarly, when an embedding that preserves a fragment's static semantics exists, but a different embedding better approaches a desired cost semantics, term rewriting can also be used to perform "optimizations". Care is needed when this changes the type of a term. Type abstraction has been used for this purpose in work on *lightweight modular staging* (LMS) [Rompf and Odersky 2012]. In all of these cases, the type system remains fixed.

When new static distinctions are needed within an existing type, but new operators are not necessary, one solution is to develop an overlying system of *refinement types*

[Freeman and Pfenning 1991]. For example, a refinement of integers might distinguish negative integers. Proposals for "pluggable type systems" describe composing such systems [Bracha 2004; Andreae et al. 2006]. Refinements of abstract types can be used for representation independence, but note that the type being refined is not held abstract. Were it to be, the system could be seen in ways as a degenerate mode of use of our work: we further cover the cases when new operators are needed. For example, labeled tuples couldn't be seen as refinements of nested pairs because label-indexed row projection operators simply don't exist.

Many *language frameworks* exist that simplify dialect implementation (cf. [Erdweg et al. 2013]). These sometimes do not support forming languages from fragments due to the "expression problem" (EP) [Wadler 1998; Reynolds 1975]. We sidestep the most serious consequences of the EP by leaving our abstract syntax entirely fixed, instead delegating to tycons. Fewer tools require knowledge of all external tycons in a typed translation semantics. Some language frameworks do address the EP, e.g. by encoding terms and types as open datatypes [Löh and Hinze 2006], but this makes it quite difficult to reason modularly, particularly about metatheoretic properties specific to typed languages, like type safety and tycon invariants. Our key insight is to instead associate term-level opcons with tycons, which then become the fundamental constituents of the semantics (consistent with Harper's informal notation from Sec. 1).

As discussed, our treatment of concrete syntax defers to recent work on *type-specific languages*, which takes a similar split bidirectional approach for composably introducing syntax [Omar et al. 2014]. We focus on semantics.

Proof assistants can be used to specify and mechanize the metatheory of languages, but also usually require a complete specification (this has been identified as a key challenge [Aydemir et al. 2005]). Techniques for composing specifications and proofs exist [Delaware et al. 2013, 2011; Schwaab and Siek 2013], but they require additional proofs at composition-time and provide no guarantees that *all* fragments having some modularly checkable property can safely and conservatively be composed, as in our work. Several authors, notably Chlipala [Chlipala 2010], suggest proof automation as a heuristic solution to the problem.

In contrast, in $@\lambda$, fragment providers need not provide the semantics with mechanized specfications or proofs to benefit from rigorous modular reasoning principles. Instead, under a fixed tycon context, the calculus can be reasoned about like a very small type-directed compiler [Tarditi et al. 1996; Chlipala 2007; Dave 2003]. Errors in reasoning can only lead to failure at typechecking time, via our chief contribution: a novel form of *translation validation* [Pnueli et al. 1998]. Incorrect opcon definitions relative to a specification (e.g. [Fulton et al. 2014] for regular strings) can at worst weaken expected invariants at that tycon, like incorrectly implemented modules in ML. Thus, modular tycons can reasonably be tested "in the field" without concern about the reliability of the semantics as a whole. To eliminate even these localized failure modes, we plan to introduce *optional* specification and proof mechanization into the SL (by basing it on a dependently typed language like Coq, rather than ML). Because types are values in the SL, the SL can be seen as building on the concept of *type-level computation*.

find appropriate citation

Type abstraction, encouragingly, also underlies modular reasoning in ML-like languages [Harper 2012, 1997] and languages with other forms of ADTs [Liskov and Zilles 1974] like Scala [Amin et al. 2014]. Indeed, proofs of tycon invariants can rely on existing parametricity theorems [Wadler 1989]. Our work is thus reminiscent of work on elaborating an ML-style module system into System $\mathbf{F}_\omega$ [Rossberg et al. 2010]. Unlike in module systems, type translations (analagous to the choice of representation for an abstract type) are statically *computed* based on a type index, rather than statically *declared*. Moreover, there can be arbitrarily many operators because they arise by providing a term index to an opcon, and their semantics can be complex because a static function computes the types and translations that arise. In contrast, modules and ADTs can only specify a fixed number of operations, and each must have function type. Note also that these are not competing mechanisms: we did not specify quantification over external types here for simplicity, but we conjecture, based on the finding that polymorphism is conservative over simple types, [Breazu-Tannen and Meyer 1990], that this is complementary and thus $@\lambda$ could serve as the core of a language with an ML-style module system and polymorphism. Another related direction is to explore *tycon functors*, which would abstract over tycons with the same signature to support modularly tunable cost semantics.

7. Conclusion

We have specified a simple but surprisingly powerful typed translation semantics, $@\lambda$, where new type constructors can be introduced "from within". The corresponding operator constructors determine types and translations "actively", i.e. using static functions. A simple form of translation validation based on existing notions of type abstraction ensures that opcons associated with one tycon maintain translation independence from all others, guaranteeing that a wide class of important properties can be reasoned about modularly. Implementation in several settings is ongoing.

A limitation of our approach is that it supports only fragments with the standard "shape" of typing judgement. Fragments that require new forms of scoped contexts (e.g. symbol contexts [Harper 2012]) cannot presently be defined. Relatedly, the language controls variable binding, so, for example, linear type systems cannot be defined. Another limitation is that opcons cannot directly invoke one another (e.g. a **len** opcon on regular strings could not construct a natural number). We conjecture that these are not fundamental limitations and expect $@\lambda$ to serve as a foundation for future efforts that increase expressiveness while maintaining the strong modularity guarantees established here.

References

GHC/FAQ. http://www.haskell.org/haskellwiki/GHC:FAQ#Extensible_Records.

N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 233–249. ACM, 2014. ISBN 978-1-4503-2585-1. URL http://dl.acm.org/citation.cfm?id=2660193.

C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA*, OOPSLA '06, pages 57–74, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167479. URL http://doi.acm.org/10.1145/1167473.1167479.

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

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. ISBN 1-58113-111-9. doi: 10.1145/317636.317791. URL http://doi.acm.org/10.1145/317636.317791.

G. Bracha. Pluggable type systems, Oct. 2004. URL http://pico.vub.ac.be/%7Ewdmeuter/RDL04/papers/Bracha.pdf. OOPSLA Workshop on Revival of Dynamic Languages.

V. Breazu-Tannen and A. R. Meyer. Polymorphism is conservative over simple types. In *Logical foundations of functional programming*, pages 297–314. Addison-Wesley Longman Publishing Co., Inc., 1990.

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

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. ISBN 3-540-13346-1. URL http://dl.acm.org/citation.cfm?id=1096.1098.

A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 54–65. ACM, 2007. ISBN 978-1-59593-633-2. URL http://doi.acm.org/10.1145/1250734.1250742.

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

M. A. Dave. Compiler verification: A bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, Nov. 2003. ISSN 0163-5948. doi: 10.1145/966221.966235. URL http://doi.acm.org/10.1145/966221.966235.

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

B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Meta-theory à la carte. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 207–218. ACM, 2013. ISBN 978-1-4503-1832-7. URL http://dl.acm.org/citation.cfm?id=2429069.

S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. The state of the art in language workbenches. In *Software Language Engineering*. 2013.

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

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.

R. Harper. Programming in standard ml, 1997.

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

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.

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. ISBN 978-3-642-17684-5. URL http://dx.doi.org/10.1007/978-3-642-17685-2.

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.

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

A. Löh and R. Hinze. Open data types and open functions. In *PPDP*, pages 133–144. ACM, 2006. ISBN 1-59593-388-3. URL http://doi.acm.org/10.1145/1140335.1140352.

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

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. URL http://doi.acm.org/10.1145/1018203.1018207.

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

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. ISBN 3-540-78662-7, 978-3-540-78662-7. URL http://dl.acm.org/citation.cfm?id=1793574.1793585.

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

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. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034815. URL http://doi.acm.org/10.1145/2034773.2034815.

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

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000. ISSN 0164-0925. doi: 10.1145/345099.345100. URL http://doi.acm.org/10.1145/345099.345100.

A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998. URL citeseer.ist.psu.edu/article/pnueli98translation.html.

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

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

T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, June 2012. ISSN 0001-0782 (print), 1557-7317 (electronic). doi: http://dx.doi.org/10.1145/2184319.2184345.

A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, Feb. 2006.

A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. In A. Kennedy and N. Benton, editors, *TLDI*, pages 89–102. ACM, 2010. ISBN 978-1-60558-891-9. URL http://dl.acm.org/citation.cfm?id=1708016.

C. Schwaab and J. G. Siek. Modular type-safety proofs in agda. In M. Might, D. V. Horn, A. A. 0001, and T. Sheard, editors, *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 3–12. ACM, 2013. ISBN 978-1-4503-1860-0. URL http://dl.acm.org/citation.cfm?id=2428116.

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

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.

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

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