Modular Type Constructors

Type System Fragments as Safely Composable Libraries

Abstract

Abstraction providers sometimes need to directly introduce new types and operators into existing languages. Ideally, such typed language fragments could be separately defined and safely composed. Unfortunately, existing tools and techniques do not come equipped with modular reasoning principles comparable to those available for library-based embeddings packaged using a modern module system. Each combination of fragments is semantically its own dialect of the language and metatheoretic and compiler correctness results must be established monolithically. Recent work has started to address this problem, e.g. by showing how to modularly reason about concrete syntax. Our focus here is on safely composing separately defined type system fragments. We organize each fragment around a type constructor, as is usual practice when describing type systems, and sidestep the difficulties of abstract syntax by statically delegating semantic control over a small, fixed abstract syntax in a type-directed manner to static logic associated with a relevant type constructor. The paper is organized around a minimal calculus, $@\lambda$ that permits surprisingly practical examples. We establish several strong semantic guarantees, notably *type safety*, *stability of typing* under extension and *conservativity*: that the *type invariants* that a finite set of fragments maintain are conserved under extension. This involves lifting typed compilation techniques into the semantics and enforcing an abstraction barrier around each fragment using a form of "internal" type abstraction. Conservativity then follows from classical parametricity results, so these *modular type constructors* can be reasoned about like modules.

1. Introduction

Typed programming languages are often described in *fragments*, each defining contributions to a language's concrete syntax, abstract syntax, static semantics and dynamic semantics. In his textbook, Harper organizes fragments around type constructors, each introduced in a different chapter [14]. A language is 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). It is left implied that the metatheory developed separately is conserved when fragments like these are composed to form a language.

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 can be placed in isomorphism with polymorphic recursive sum of products, $\forall(\alpha.\mu(t.1 + (\alpha \times t))))$. Languages providing datatypes in the style of ML are perhaps most directly oriented around embeddings like this (preserving type disequality requires adding some form of generativity, as ML datatypes also expose).

Unfortunately, situations do sometimes arise where using these fragments to establish an isomorphic embedding that preserves a desirable fragment's static and dynamic semantics (including bounds specified by its 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, considering cognitive metrics like error message comprehensibility.

When an embedding is too complex, abstraction providers have a few options, discussed in Sec. 4. When an embedding is not possible, however, or when these options are insufficient, providers are often compelled to introduce these fragments by extending an existing language, thereby forming a new *dialect*. To save effort, they may do so by forking existing artifacts or leverage tools like compiler generators or language frameworks, also reviewed in Sec. 4. Within the ML lineage, dialects that go beyond $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1\, \times\, +\}$ 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 [7], records with update and extension operators [17], mutable fields [17], and "methods" (i.e. pure objects [2, 25]).[1] Sum types are also exposed in various ways: finite datatypes, open datatypes [18], hierarchically open datatypes [22], polymorphic variants [17] 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 [8], distributed programming [23], reactive programming [20], authenticated data structures [21], databases [24] and units of measure [16].
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 [4].

[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. And all reasonable variants break backward compatibility. As a result, nothing much happens." [1]

This dialect-oriented state of affairs is, we argue, unsatisfying: a programmer can choose either a dialect supporting a principled approach to distributed programming, or one that builds in support for statically reasoning about units of measure, but there may not be an available dialect supporting both. Combining dialects is non-trivial in general, as we will discuss. Using different dialects separately for different components of a program is also untenable: components written in different dialects cannot always interface safely with one another (i.e. an FFI, item 3 above, is needed).

These problems do not arise when a fragment can be exposed as an isomorphic embedding because modern *module systems* can 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 situations where an embedding is not evident, however, mechanisms are needed that make specifying, implementing and reasoning about direct fragment composition similarly modular. 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 easy as importing a new module is today. In the limit, the community could rely on modularly mechanized metatheory and compiler correctness results, rather than requiring heroic efforts on the part of individual research groups as the situation exists 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 [15], consisting of an *external language* (EL) governed by a typed translation semantics targeting a fixed *internal language* (IL). Rather than building in a monolithic set of 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.

We introduce @λ by example in Sec. 2, demonstrating its surprising expressive power and detailing its semantics, then examine its key metatheoretic properties in Sec. 3, beginning with *type safety* and *unicity of typing* and touching on *decidability of typechecking*. We then consider 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 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").

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). We conclude by discussing some limitations, proposing some richer variants of the calculus and discussing related work (Sec. 4).

internal types

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

internal terms

$\iota ::= x \mid \mathsf{fix}[\tau](x.\iota) \mid \lambda[\tau](x.\iota) \mid \mathsf{ap}(\iota;\iota) \mid \Lambda(\alpha.\iota) \mid \iota[\tau]$
$\mid \mathsf{fold}[t.\tau](\iota) \mid \mathsf{unfold}(\iota) \mid \mathsf{triv} \mid \mathsf{pair}(\iota;\iota) \mid \mathsf{letpair}(\iota; x, y.\iota)$
$\mid \mathsf{inl}[\tau](\iota) \mid \mathsf{inr}[\tau](\iota) \mid \mathsf{case}(\iota; x.\iota; x.\iota) \mid \triangleleft(\sigma)$

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

internal type variable contexts $\Delta ::= \emptyset \mid \Delta, \alpha \quad \Theta ::= \emptyset \mid \Theta, t$

term substitutions $\gamma ::= \emptyset \mid \gamma, \iota/x$

type substitutions $\delta ::= \emptyset \mid \delta, \tau/\alpha$

Figure 1. Syntax of $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1\, \times\, +\}$, our internal language (IL). We write internal types and terms in blue when they may contain the indicated "unquote" forms, discussed later.

2. @λ, By Example

Programs in @λ are written as *external terms*, e. The concrete syntax of external terms is essentially conventional, as suggested by the left side of the example in Figure 2. It desugars (purely syntactically) to a subtantially more uniform abstract syntax, shown in Figure 3 and discussed as we go on. We review recent techniques that permit modularly introducing new desugarings like these in Sec. 4.

The static and dynamic semantics are specified simultaneously by a *bidirectionally typed translation semantics* targeting an *internal language*, with terms ι. The two key judgements have 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 with translation ι under typing context Υ and tycon context Φ". We indicate "outputs" when introducing judgement forms by a *mode annotation*, shown; these are not part of the judgement's syntax. The rules, shown in Figure 11, will be discussed incrementally as we explain the example in Figure 2. Bidirectional typechecking is also sometimes called *local type inference* [26].

Internal Language @λ relies on a *typed internal language* with support for type abstraction. In this paper, we use $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1\, \times\, +\}$ (Figure 1), though our results do not depend on this precise choice of internal type constructors, only on the general forms of the judgements. We assume the internal statics are specified in the standard way by judgements for internal type formation $\Delta\ \Theta \vdash \tau$, typing context formation $\Delta \vdash \Gamma$ and type assignment $\Delta\ \Gamma \vdash \iota : \tau^+$. The internal dynamics are also a standard structural operational semantics with judgements $\iota \mapsto \iota^+$ and ι `val` (cf. [14]).

We also here define a syntax for simultaneous substitutions, of terms for variables, γ, and of types for polymorphic type variables, δ, and assume standard judgements ensuring that these are valid with respect to a valid context, $\Delta \vdash \gamma : \Gamma$ and $\vdash \delta : \Delta$. We apply these substitutions to terms, types and typing contexts using the syntax $[\gamma]\iota$, $[\delta]\iota$, $[\delta]\tau$ and $[\delta]\Gamma$ (in some prior work, application of a substitution like this is written using a hatted form, e.g. $\hat{\gamma}(\iota)$; we intend the same semantics but use a notation more consistent with standard substitution, e.g. $[\iota/x]\iota'$). We will omit leading $\emptyset$ and $\cdot$ in examples; in specifications, the former is used for metatheoretic finite mappings, the latter for metatheoretic ordered lists.

This style of specification has some similarities to the Harper-Stone *typed elaboration semantics* for Standard ML [15]. There, however, external and internal terms were governed by a common type system. In @λ, internal types classify only internal terms. 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 [32], here lifted "one level up" into the language. We will see how the techniques developed to reason about typed compilation are lifted into our semantics to compositionally reason about type safety, which requires that the translation of every well-typed external term is a well-typed internal term, below.

```
1  type V = LPROD {venue : RSTR /([A-Z]+) \d{4}/}
2  let v : V = {venue="EXMPL 2015"}
3  let prefix = "01.0001/" : RSTR /\d{2}\.\d{4}\//
4  fun paper (title : RSTR /.+/) (id : RSTR /\d+/) =
5    v.with(title = title, doi = prefix.concat(id))
6  let pid : RSTR /\d{3}/ = "005"
7  let p = paper "M Theory" (pid.coerce[/\d+/])
8  let c = p#venue#0
9  c.case(λc : RSTR /EXMPL/.c.!rc; v.!ri)
```

```
letstatic[ty[LPROD](list1[Lbl × Ty] (⌊venue⌋, ty[RSTR](⌊/([A-Z]+) \d{4}/⌋)))](V.
let(asc[V](lit[list1[Lbl] ⌊venue⌋](lit[⌊"EXMPL 2015"⌋](·))); v.
let(asc[ty[RSTR](⌊/\d{2}\.\d{4}\//⌋)](lit[⌊"01.0001/"⌋](·)); prefix.
let(λ[ty[RSTR](⌊/.+/⌋)](title.λ[ty[RSTR](⌊/\d+/⌋)](id.
  targ[with; list2[Lbl] ⌊title⌋ ⌊doi⌋](v; title, targ[concat; ()](prefix; id)))); paper.
let(asc[ty[RSTR](⌊/\d{3}/⌋)](lit[⌊"005"⌋](·)); pid.
let(ap(ap(paper; lit[⌊"M Theory"⌋](·)); targ[coerce; ⌊/\d+/⌋](pid; ·)); p.
let(targ[#; ⌊0⌋](targ[#; ⌊venue⌋](p; ·); ·); c.
targ[case; ()](c; λ[ty[RSTR](⌊/EXMPL/⌋)](c.targ[!rc; ()](c; ·)), targ[!ri; ()](v; ·)))))))))
```

Figure 2. An example external term, e_{ex}, in concrete syntax (left), desugared to abstract syntax (right), with static terms shown in green in examples (only). The notation $\lfloor \ldots \rfloor$ stands for an embedding of the indicated `label`, `/regular expression/`, `"string"` or numeral into the SL, with derived kinds Lbl, Rx, Str and Nat, not shown. The signatures of helper functions, e.g. ***nil*** and ***list***n, are shown in Figure 13.

external terms

$$
\begin{array}{lcl}
e & ::= & \mathsf{letstatic}[\sigma](\boldsymbol{x}.e) \mid \mathsf{asc}[\sigma](e) \\
& \mid & x \mid \mathsf{let}(e; x.e) \mid \mathsf{fix}(x.e) \mid \lambda[\sigma](x.e) \mid \mathsf{ap}(e; e) \\
& \mid & \mathsf{lit}[\sigma](\overline{e}) \mid \mathsf{targ}[\mathbf{op}; \sigma](e; \overline{e}) \mid \mathsf{other}[\iota] \qquad \overline{e} ::= \cdot \mid \overline{e}, e
\end{array}
$$

external typing contexts

$$\Upsilon ::= \emptyset \mid \Upsilon, x \Rightarrow \sigma$$

Figure 3. Abstract syntax of the external language (EL). Terms in gray are technical devices with no corresponding concrete syntax.

kinds

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

static terms

$$
\begin{array}{lcl}
\sigma & ::= & \boldsymbol{x} \mid \lambda \boldsymbol{x}{:}\kappa.\sigma \mid \sigma\ \sigma \mid \Lambda(\boldsymbol{\alpha}.\sigma) \mid \sigma\,[\kappa] \\
& \mid & \mathsf{fold}[\boldsymbol{k}.\kappa](\sigma) \mid \mathsf{rec}[\kappa](\sigma; \boldsymbol{x}.\sigma) \\
& \mid & () \mid (\sigma, \sigma) \mid \mathsf{letpair}(\sigma; \boldsymbol{x}, \boldsymbol{y}.\sigma) \\
& \mid & \mathsf{inl}[\kappa](\sigma) \mid \mathsf{inr}[\kappa](\sigma) \mid \mathsf{case}(\sigma; \boldsymbol{x}.\sigma; \boldsymbol{x}.\sigma) \\
& \mid & \mathsf{ty}[c](\sigma) \mid \mathsf{otherty}[m; \tau] \mid \mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma; \sigma) \\
& \mid & \blacktriangleright(\tau) \mid \mathsf{rep}(\sigma) \mid \triangleright(\iota) \\
& \mid & \mathsf{ana}[n](\sigma) \mid \mathsf{syn}[n] \\
& \mid & \mathsf{raise}[\kappa] \\
m, n & ::= & 0 \mid n + 1
\end{array}
$$

kinding contexts $\boldsymbol{\Gamma} ::= \emptyset \mid \boldsymbol{\Gamma}, \boldsymbol{x} : \kappa$ **kind variable contexts** $\boldsymbol{\Delta} ::= \emptyset \mid \boldsymbol{\Delta}, \boldsymbol{\alpha}$ $\boldsymbol{\Theta} ::= \emptyset \mid \boldsymbol{\Theta}, \boldsymbol{k}$

Figure 4. Syntax of the static language (SL).

Kinds and Static Terms 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 4. 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 closely follow [14] so we again omit some details).

The kinding judgement, specified in Figure 14, takes the form $\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^{n}_{\boldsymbol{\Xi}} \sigma : \kappa^{+}$, where $\boldsymbol{\Delta}$ and $\boldsymbol{\Gamma}$ are analagous to Δ and Γ (and analagous well-formedness judgements are defined, also omitted). The natural number n is used as a technical device in our metatheory to prevent terms of the form $\mathsf{ana}[n](\sigma)$ and $\mathsf{syn}[n]$ from arising if not permitted; these forms, discussed later, never need to be written directly (they would have no corresponding concrete syntax), so n can assumed 0 for all user code. $\boldsymbol{\Xi}$ is described below. The normalization judgement (specified in Figure 15) takes the form $\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma^{+} \parallel \mathcal{D}^{+}\ \mathcal{G}^{+}$, where $\mathcal{D}$, $\mathcal{G}$ and $\mathcal{C}$ are also technical devices that will be described later; they too can be ignored from the perspective of user code. We write $\sigma \Downarrow \sigma'$ iff $\sigma \parallel \emptyset \cdot \Downarrow_{\rightharpoonup;\emptyset;\emptyset} \sigma' \parallel \emptyset \cdot$. *Static values* are normalized static terms. Normalization can also raise an error (to indicate a type error in an external term, or a problem in a tycon definition, as we will discuss), indicated by the judgement $\sigma \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\mathcal{C}}$. We omit error propagation rules.

External Types are Static Values External types (or simply *types*) are static values of kind Ty. The form $\mathsf{ty}[c](\sigma)$ introduces a type by "applying" a *tycon*, c, to a static term, σ, called the *type index*. There is one built-in tycon, $\rightharpoonup$, classifying possibly partial external functions, discussed below. All other types are constructed by tycons defined in the tycon context and identified by name. We write tycon names in small caps, e.g. LPROD, and use TC as a metavariable ranging over these (Figure 5). We will return to the other introductory form for types, $\mathsf{otherty}[m; \tau]$, which also serves only as a technical device, later.

Two user-defined types are constructed on line 1 of Figure 2: a labeled product type declaring a single field labeled `venue` of type `RSTR /([A-Z]+) \d{4}/`, which classifies *regular strings*, known statically to be in the regular language of the indicated regular expression, which can contain nested parenthesized groups.

Index Kinds The rule (*k-ty*) in Figure 14 requires that the type index must be of the *index kind* of the tycon, given by the *index kind context*, $\boldsymbol{\Xi}$, a simple mapping from tycons to kinds (Figure 5). The initial index kind context, $\boldsymbol{\Xi}_0$, is always included, defining the index kind of $\rightharpoonup$ as $\mathsf{Ty} \times \mathsf{Ty}$. The example index kind context in Figure 6, $\boldsymbol{\Xi}_{\text{ex}}$, specifies that the index kind of LPROD is $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$, and the index kind of RSTR is Rx. We assume $\mathsf{List}[\kappa]$, Lbl and Rx are embedded into the static language in the standard way. The concrete syntax used on line 1 desugars to static terms of these kinds (this sugar could be defined modularly as *kind-specific syntax* by adapting the technique described in Sec. 4 [25]). If a tycon context, Φ, is valid, it uniquely determines a valid index kind context, as specified in Figure 7 by the judgement $\vdash \Phi \sim \boldsymbol{\Xi}^{+}$, detailed below. In our example, $\vdash \Phi_{\text{ex}} \sim \boldsymbol{\Xi}_{\text{ex}}$.

Type-Level Computation The labeled product type on line 1 is bound to the static variable $\boldsymbol{V}$ in the remainder of the external term. This desugars to the more general external construct $\mathsf{letstatic}[\sigma](\boldsymbol{x}.e)$, on the right. The rules (*syn-slet*) and (*ana-slet*), shown in Figure 11 (here, the former applies) begin by assigning a kind to σ under the index kind context determined by Φ, then substitute its value into the external term e to continue with synthesis or analysis. Static variables, written in bold, are distinct from external and internal variables. The rule (*n-ty*) in Figure 15 shows that normalizing $\mathsf{ty}[c](\sigma)$ simply involves normalizing σ. Because types are static values but need not be written directly in normal form, this is a form of *type-level computation of higher kind*, though our SL has a more general role than a standard type-level language.

Type Constructor Contexts A tycon context, Φ, is a list of tycon definitions, ϕ (Figure 5). Each tycon definition is annotated with a *tycon signature*, ψ, which specifies the index kind of the tycon as well as an *opcon signature*, Ω, which relates to the *opcon structure*, ω, discussed below. The tycons used in our example are defined in Figures 8 and 9. In Figure 7, the judgement $\vdash_{\boldsymbol{\Xi}} \phi \sim \text{TC}^{+}[\kappa^{+}_{\text{tyidx}}]$ ensures that their names are unique, their signatures are valid, and that the opcon structure respects the opcon signature.

tycons $c ::= \rightharpoonup \mid \text{TC}$
tycon contexts
$\Phi ::= \cdot \mid \Phi, \phi$
tycon defs
$\phi ::= \text{tycon TC } \{\omega\} : \psi$
opcon structures
$\omega ::= \text{rep} = \sigma \mid \omega, \text{ana lit} = \sigma \mid \omega, \text{syn } \mathbf{op} = \sigma$

index kind contexts
$\Xi_0 := \rightharpoonup[\text{Ty} \times \text{Ty}]$
$\Xi ::= \Xi_0 \mid \Xi, \text{TC}[\kappa]$
tycon sigs
$\psi ::= \text{tcsig}[\kappa]\ \{\Omega\}$
opcon sigs
$\Omega ::= \cdot \mid \Omega, \text{lit}[\kappa] \mid \Omega, \mathbf{op}[\kappa]$

Figure 5. Syntax of tycons and opcons + sigs and contexts.

$\Phi_{\text{ex}} := \phi_{\text{rstr}}, \phi_{\text{lprod}}$ $\quad \Xi_{\text{ex}} := \Xi_0, \text{RSTR}[\text{Rx}], \text{LPROD}[\text{List}[\text{Lbl} \times \text{Ty}]]$
ϕ_{rstr} (Figure 8) $\quad \psi_{\text{rstr}} := \text{tcsig}[\text{Rx}]\ \{\Omega_{\text{rstr}}\}$
ϕ_{lprod} (Figure 9) $\quad \psi_{\text{lprod}} := \text{tcsig}[\text{List}[\text{Lbl} \times \text{Ty}]]\ \{\Omega_{\text{lprod}}\}$
$\Omega_{\text{rstr}} := \text{lit}[\text{Str}], \mathbf{concat}[1], \#[\text{Nat}], \mathbf{coerce}[\text{Rx}], \mathbf{case}[1], \mathbf{!rc}[1]$
$\Omega_{\text{lprod}} := \text{lit}[\text{List}[\text{Lbl}]], \#[\text{Lbl}], \mathbf{with}[\text{List}[\text{Lbl}]], \mathbf{!ri}[1]$

Figure 6. The contexts and signatures for Figures 8 and 9.

$$\frac{}{\vdash \cdot \sim \Xi_0}\ (tcc\text{-}emp) \qquad \frac{\vdash \Phi \sim \Xi \quad \vdash_\Xi \phi \sim \text{TC}[\kappa_{\text{tyidx}}]}{\vdash \Phi, \phi \sim \Xi, \text{TC}[\kappa_{\text{tyidx}}]}\ (tcc\text{-}ext) \qquad \boxed{\vdash \Phi \sim \Xi^+}$$

$$\frac{\text{TC} \notin \text{dom}(\Xi) \quad \vdash \psi \sim \kappa_{\text{tyidx}} \quad \vdash_\Xi \text{TC}\ \{\omega\} : \psi}{\vdash_\Xi \text{tycon TC } \{\omega\} : \psi \sim \text{TC}[\kappa_{\text{tyidx}}]}\ (tcdef\text{-}ok) \qquad \boxed{\vdash_\Xi \phi \sim \text{TC}^+[\kappa^+]}$$

$$\frac{\emptyset \vdash \kappa_{\text{tyidx}}\ \text{eq} \quad \vdash \Omega}{\vdash \text{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega\} \sim \kappa_{\text{tyidx}}}\ (tcsig\text{-}ok) \qquad \boxed{\vdash \psi \sim \kappa^+}$$

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

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

$$\frac{}{\vdash \cdot}\ (ocs\text{-}rep) \qquad \frac{\emptyset \vdash \kappa\ \text{eq}}{\vdash \text{lit}[\kappa]}\ (ocs\text{-}lit) \qquad \frac{\vdash \Omega \quad \mathbf{op} \notin \text{dom}(\Omega) \quad \emptyset \vdash \kappa\ \text{eq}}{\vdash \Omega, \mathbf{op}[\kappa]}\ (ocs\text{-}targ) \qquad \boxed{\vdash \Omega}$$

$\kappa_{\text{arg}} := (1 \to (\text{Ty} \times \text{ITm})) \times (\text{Ty} \to \text{ITm})$ $\qquad \boxed{\vdash_\Xi \text{TC}\ \{\omega\} : \psi}$

$$\frac{\emptyset\ \emptyset \vdash^0_\Xi \sigma_{\text{rep}} : \kappa_{\text{tyidx}} \to \text{ITy}}{\vdash_\Xi \text{TC}\ \{\text{rep} = \sigma_{\text{rep}}\} : \text{tcsig}[\kappa_{\text{tyidx}}]\ \{\cdot\}}\ (oc\text{-}rep)$$

$$\frac{\vdash_\Xi \text{TC}\ \{\omega\} : \text{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega\} \quad \emptyset\ \emptyset \vdash^0_{\Xi, \text{TC}[\kappa_{\text{tyidx}}]} \sigma_{\text{def}} : \kappa_{\text{tyidx}} \to \kappa \to \text{List}[\kappa_{\text{arg}}] \to \text{ITm}}{\vdash_\Xi \text{TC}\ \{\omega, \text{ana lit} = \sigma_{\text{def}}\} : \text{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega, \text{lit}[\kappa]\}}\ (oc\text{-}lit)$$

$$\frac{\vdash_\Xi \text{TC}\ \{\omega\} : \text{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega\} \quad \emptyset\ \emptyset \vdash^0_{\Xi, \text{TC}[\kappa_{\text{tyidx}}]} \sigma_{\text{def}} : \kappa_{\text{tyidx}} \to \text{ITm} \to \kappa \to \text{List}[\kappa_{\text{arg}}] \to (\text{Ty} \times \text{ITm})}{\vdash_\Xi \text{TC}\ \{\omega, \text{syn } \mathbf{op} = \sigma_{\text{def}}\} : \text{tcsig}[\kappa_{\text{tyidx}}]\ \{\Omega, \mathbf{op}[\kappa]\}}\ (oc\text{-}targ)$$

Figure 7. Type constructor kinding.

```
tycon RSTR {
  rep = λtyidx:Rx.▶(μ(s.str × list[s]))
  ana lit = λtyidx:Rx.λtmidx:Str.λargs:List[κarg].
    let aok : 1 = arity0 args in
    letpair (chars, groups) = rmatch tyidx tmidx in
      ▷(fold[s.str × list[s]](
        (◁(str_of_Str chars), ◁(rstrs_of_Strs groups)))),
  syn concat = λttyidx:Rx.λttr:ITm.λtmidx:1.λargs:List[κarg].
    letpair (targ, a) = arity2 args in letpair (aty, atr) = syn a in
    tycase[RSTR](aty; atyidx.(ty[RSTR](rconcat ttyidx atyidx),
      ▷(rsconcat ◁(ttr) ◁(atr))); raise[Ty × ITm]),
  syn # = λttyidx:Rx.λttr:ITm.λtmidx:Nat.λargs:List[κarg].
    let aok : 1 = arity0 args in
    let rg : Rx = rgroupn ttyidx tmidx in
      (ty[RSTR](rg), ▷(rsgroupn ◁(nat_of_Nat tmidx) ◁(ttr))),
  syn coerce = λttyidx:Rx.λttr:ITm.λtmidx:Rx.λargs:List[κarg].
    let aok : 1 = arity0 args in
    let slok : 1 = rsublang tmidx ttyidx in
    let rtr : ITm = rx_of_Rx tmidx in
      (ty[RSTR](tmidx), ▷(rsregroup ◁(rtr) ◁(ttr))),
  syn case = λttyidx:Rx.λttr:ITm.λtmidx:1.λargs:List[κarg].
    letpair (a1, a2) = arity2 args in letpair (a1ty, a1tr) = syn a1 in
    tycase[⇀](a1ty; a1tyidx.letpair(a1tyin, a1tyout) = a1tyidx in
    tycase[RSTR](a1tyin; r.let rtr : ITm = rx_of_Rx r in
    let a2tr : ITm = ana a2 a1tyout in
      (a1tyout, ▷(rscheck[◀(rep(a1tyout))] ◁(rtr) ◁(ttr)
        ◁(a1tr) λ[1](_.◁(a2tr))))
    ; raise[Ty × ITm]); raise[Ty × ITm])
  syn !rc = λttyidx:Rx.λttr:ITm.λtmidx:1.λargs:List[κarg].
    (▷(ty[RSTR](⌊/\d+/⌋)), str_of_Str ⌊"13"⌋)} : ψrstr
```

Figure 8. The definition of the RSTR tycon, ϕ_{rstr}.

```
tycon LPROD {
  rep = λtyidx:List[Lbl × Ty].listrec[Lbl × Ty][ITy] tyidx ▶(1)
    (λh:Lbl × Ty.λr:ITy.letpair (_, hty) = h in
      ▶(◀(r) × ◀(rep(hty)))),
  ana lit = λtyidx:List[Lbl × Ty].λtmidx:List[Lbl].λargs:List[κarg].
    let inhabited : 1 = uniqmap tyidx in
    listrec3[Lbl × Ty][Lbl][κarg][ITm] tyidx tmidx args ▷(triv)
      λh1:Lbl × Ty.λh2:Lbl.λh3:κarg.λr:ITm.
        letpair (rowlbl, rowty) = h1 in let lok : 1 = lbleq rowlbl h2 in
        let rowtm : ITm = ana h3 rowty in ▷((◁(r), ◁(rwtm))),
  syn # = λttyidx:List[Lbl × Ty].λttr:ITm.λtmidx:Lbl.λargs:List[κarg].
    let aok : 1 = arity0 args in
    letpair (rownum, rowty) = lookup ttyidx tmidx in
      (rowty, prjnth rownum ttr)),
  syn with =
  λttyidx:List[Lbl × Ty].λttr:ITm.λtmidx:List[Lbl].λargs:List[κarg].
    letpair (olist, otr) = listrec2[Lbl][κarg][List[Lbl × Ty] × ITm]
      tmidx args (ttyidx, ttr)
      (λh1:Lbl.λh2:κarg.λr:List[Lbl × Ty] × ITm.
        letpair (rowty, rowtr) = syn h2 in letpair (rlist, rtr) = r in
          (append[Lbl × Ty] rlist (h1, rowty),
            ▷((◁(rtr), ◁(rowtr))))) in
    let inhabited : 1 = uniqmap olist in
    (ty[LPROD](olist), otr)
  syn !ri = λttyidx:Rx.λttr:ITm.λtmidx:1.λargs:List[κarg].
    (▷(ty[RSTR](⌊/\d+/⌋)), ▷(fold[s.str × list[s]](
      (◁(str_of_Str ⌊"oops"⌋), ◁(rstrs_of_Strs nil[Str])))))} : ψlprod
```

Figure 9. The definition of the LPROD tycon, ϕ_{lprod}.

Type Equivalence To simplify the handling of type equivalence, type index kinds must be *equality kinds*: those for which semantic equivalence coincides with syntactic equality at normal form. We define these by the judgement $\boldsymbol{\Delta} \vdash \kappa\ \text{eq}$, appearing as a premise of (*tcsig-ok*) in Figure 7. 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

Subsumption The rule (*subsume*), Figure 11, simply states that if a term synthesizes a type, it can be analyzed against that type. This relies on the fact that equivalent types are syntactically identical, and a metatheoretic property that states that the type synthesis judgement, assuming well-formed contexts, actually synthesizes a type (stated formally in Section 3). Subsumption will only be applied once in our example, on line 7, discussed below.

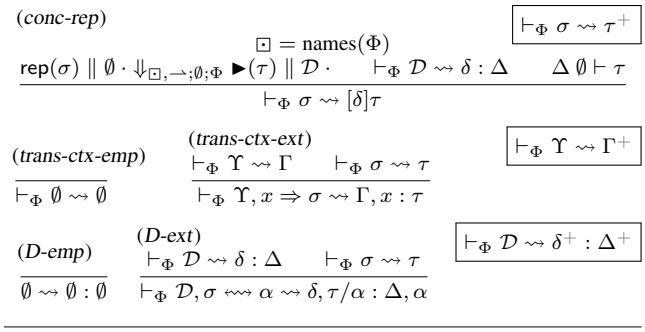

Figure 10. Concrete Representations

Representations Each external type has a corresponding internal type called its *concrete representation*. Each tycon definition specifies the concrete representations of the types it constructs with a *representation schema*, specified as the first component of the opcon structure. Rule (*oc-rep*) specifies that this is simply a static function mapping type indices to static terms of kind ITy, which are introduced by the quotation form $\blacktriangleright(\tau)$. Unquote forms, written $\blacktriangleleft(\sigma)$, normalize away recursively, shown in rules (*n-qity-**).

In Figure 8, the representation schema of RSTR simply ignores the type index, mapping all regular string types to the same internal type: a recursive type pairing a string with a list of other regular strings, corresponding to the "pre-extracted" parenthesized groups (internal types str and $\mathsf{list}[\tau]$ are defined in the usual way, and are distinguished from analagous kinds Str and List by initial capitalization). This is of course not the only valid choice of representation – one could simply use str (increasing the cost of group extraction, discussed below), or use a tree of offsets, or inspect the index to extract the number of groups, permitting the use of nested tuples (in practice, the IL might provide fixed length vectors).

In Figure 9, the representation schema of LPROD constructs a simple nested tuple representation based on the type index (the labels are not needed in the translation). Because the type index itself contains types, the representation schema must refer to the representations of these types. The static language provides the operator $\mathsf{rep}(\sigma)$ to extract knowledge about the representation of type σ. The kinding rule (*k-rep*) is simple, but the normalization semantics are more interesting. Rule (*n-rep-conc*) extracts the representation schema and passes it the type index, but it does not always apply. For types constructed by a tycon other than ones named in the *concrete rep whitelist*, $\boxdot$ (a simple list of tycons), an *abstract representation*, is generated. The first time a representation is requested, a fresh internal type variable, α, is generated and the correspondence is recorded in the *abstract rep store*, $\mathcal{D}$, as seen in rule (*n-rep-abs*). Later requests return this type variable. Requesting the representation is a simple effect. Given $\boldsymbol{V}$ from our example and abbreviating the index of the type of the field venue as σ_{rsidx}:

$$\mathsf{rep}(\boldsymbol{V}) \parallel \emptyset \cdot \Downarrow_{\rightharpoonup,\text{LPROD};\emptyset;\Phi_{\text{ex}}} \blacktriangleright(1 \times \alpha) \parallel \emptyset, \mathsf{ty}[\text{RSTR}](\sigma_{\text{rsidx}}) \leftrightsquigarrow \alpha \cdot$$

The judgement $\vdash_\Phi \mathcal{D} \rightsquigarrow \delta^+ : \Delta^+$ constructs a substitution and corresponding internal type variable context by generating the concrete representations for each stored type. For the above example:

$$\vdash_{\Phi_{\text{ex}}} \emptyset, \mathsf{ty}[\text{RSTR}](\sigma_{\text{rsidx}}) \leftrightsquigarrow \alpha \rightsquigarrow \emptyset, \mu(s.\mathsf{str} \times \mathsf{list}[s])/\alpha : \emptyset, \alpha$$

The judgement $\vdash_\Phi \sigma \rightsquigarrow \tau^+$ extracts the concrete representation by placing every tycon in Φ in the whitelist and checking that the representation is well-formed ($\rightharpoonup$ is always in the whitelist; $\mathcal{D}$ may still be non-empty because of $\mathsf{otherty}[m;\tau]$, also discussed later):

$$\mathsf{rep}(\boldsymbol{V}) \parallel \emptyset \cdot \Downarrow_{\rightharpoonup,\text{RSTR},\text{LPROD};\emptyset;\Phi_{\text{ex}}} \blacktriangleright(1 \times \mu(s.\mathsf{str} \times \mathsf{list}[s])) \parallel \emptyset \cdot$$

Variables We can now continue to line 2 of Figure 2, where we bind a term to a variable using the form $\mathsf{let}(e_1; x.e_2)$. The rules (*syn-let*) and (*ana-let*) give an essentially standard semantics. The translation is to lambda application. The concrete representation of the synthesized type for e_1 determines the type annotation. The (*syn-var*) rule is standard. The well-formedness judgement for external typing contexts, $\vdash_\Phi \Upsilon$, ensures that variables map to types (Figure 12). The judgement $\vdash_\Phi \Upsilon \rightsquigarrow \Gamma^+$ translates the types in Υ to their concrete representations (Figure 10).

Functions On lines 4-6, we see a function definition. The syntax and semantics of functions are fixed and built into the external language (so that the semantics can uniformly handle the tricky issue of variable binding). Only the argument types need to be provided explicitly; the return type is synthesized, as shown in the rules (*syn-lam*). Rule (*syn-ap*) is standard. Function types are written concretely as $\sigma_1 \rightharpoonup \sigma_2$ and abstractly as $\mathsf{ty}[\rightharpoonup]((\sigma_1, \sigma_2))$, i.e. the index kind of the built-in type constructor $\rightharpoonup$ is $\mathsf{Ty} \times \mathsf{Ty}$ (cf. $\boldsymbol{\Xi}_0$ in Figure 6). By the semantics of the operators described below, the full type synthesized by *paper*, written concretely, will be:

```
RSTR /..*/ ⇀ RSTR /\d\d*/ ⇀ LABELEDPROD {
  venue : RSTR /([A-Z]+) \d+/, title : RSTR /..*/,
  doi : RSTR /\d\d\.\d\d\d\d\/\d\d*/}
```

Like internal functions (and unlike static functions), external functions may be partial, via a fixpoint operator. Rule (*ana-fix*) follows Plotkin's PCF, cf. [14], here defined analytically.

Ascriptions Type ascriptions allow the programmer to explicitly specify the type a term should be analyzed against. For analytic terms appearing in synthetic positions, an ascription is necessary. In the abstract syntax, type ascriptions are always placed directly on terms using the form $\mathsf{asc}[\sigma](e)$ (in the concrete syntax, they may appear on the variable being bound for convenience). The rule (*syn-asc*) in Figure 11 specifies that the ascription must be a closed static term of kind Ty under the index kind context determined by the tycon context. It is normalized to a type before proceeding with analysis. This is also an essentially standard rule.

Literals The variable v binds a term introduced *literally*, under an ascription specifying that it should be analyzed against type $\boldsymbol{V}$. Inside this literal, we also see a literal, though with no ascription (it will be analyzed against the type specified in $\boldsymbol{V}$, as discussed below). All literal forms in the concrete syntax desugar to the same abstract form, $\mathsf{lit}[\sigma_{\text{tmidx}}](\overline{e})$. The *term index*, σ_{tmidx}, captures statically known portions of the literal as a single static term and the *argument list*, $\overline{e}$, captures all external sub-terms. For example, in the labeled product literal, desugaring constructs a list of field labels as the term index (using standard helper functions whose signatures are shown in Figure 13) and passes in the corresponding field values as arguments. For regular string literals, the string is lifted into the SL as the term index. There are no sub-terms, so the argument list is empty.

Literal terms can only appear in analytic positions in our calculus. More specifically, they are given meaning by the type constructor of the type they are being analyzed against. This tycon's opcon signature, Ω, determines the *literal term index kind* governing σ_{tmidx}. In Figure 6, ψ_{lprod} specifies Ω_{lprod} which in turn specifies the literal term index kind $\mathsf{List}[\mathsf{Lbl}]$, and ψ_{rstr} similarly specifies Ω_{rstr} and Str. The literal index kind must also be an equality kind, as specified by rule (*ocs-lit*) in Figure 7. The rule governing type analysis of literals is (*ana-lit*), shown in Figure 11. It begins by checking the provided term index against this kind (premises 1-4).

Literal Opcon Definitions The next premise of (*ana-lit*) extracts the *literal opcon definition*, σ_{def}, defined using the form $\mathsf{ana\ lit} = \sigma_{\text{def}}$ in the opcon structure, ω, of the tycon definition. This is responsible for deciding the translation of the literal as a function of the type index, the term index and the types and translations of the arguments.

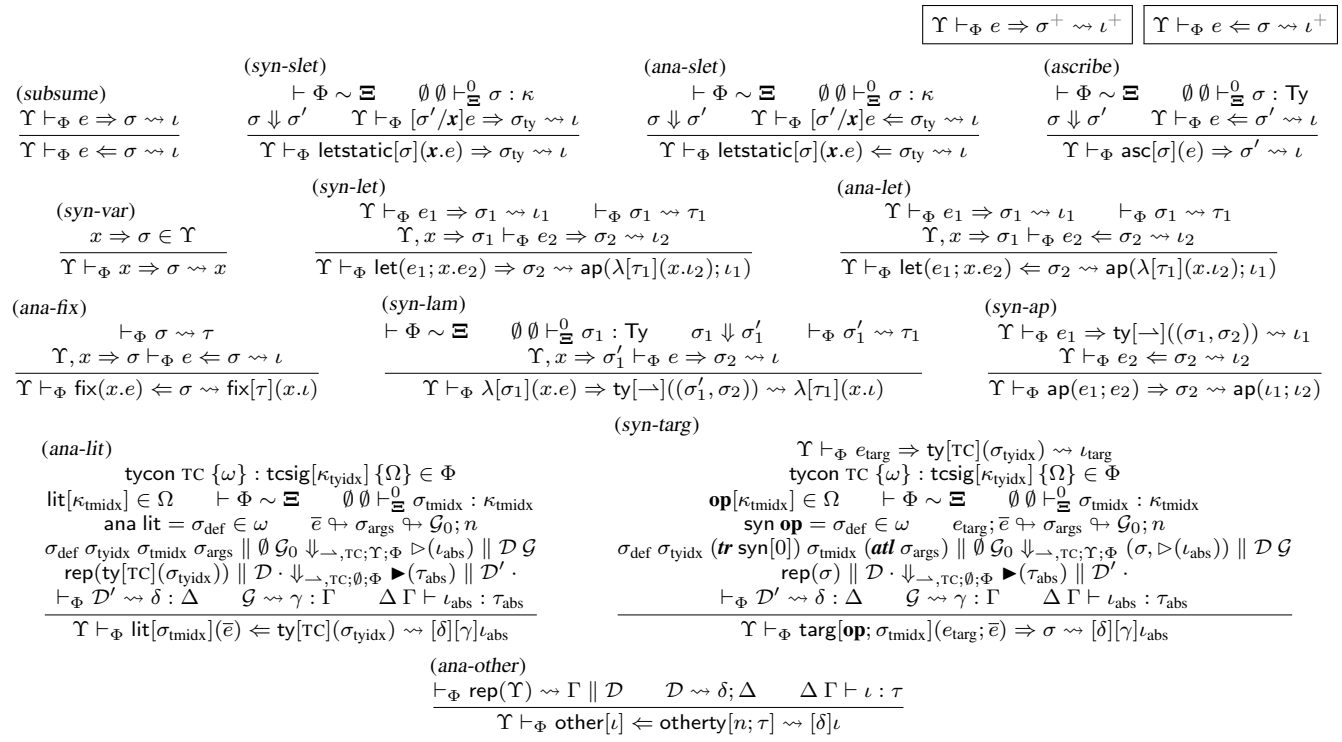

Figure 11. Typing

The relevant rule in Figure 7 is (*oc-lit*), which ensurses that σ_{def} has kind $\kappa_{\text{tyidx}} \to \kappa_{\text{tmidx}} \to \mathsf{List}[\kappa_{\text{arg}}] \to \mathsf{ITm}$. The kinds κ_{tyidx} and κ_{tmidx} are determined by the signature as just described. The kind ITm is analagous to ITy, and has one introductory "quotation" form, $\triangleright(\iota)$. Normalization produces an internal term without such forms (rules (*n-qitm-**) in Figure 15), i.e. a standard IL term. Note that these can only be opaquely composed, not inspected (there is no elim form) and ITy and ITm are not equality kinds.

Before discussing arguments and the remaining premises, let us turn to the literal opcon definition for RSTR in Figure 8, which will be called by the rule (*ana-lit*) to derive a translation for the field value on line 1, $e_{\text{venue}} := \mathsf{lit}[\lfloor\texttt{"EXMPL 2015"}\rfloor](\cdot)$:

$$\emptyset \vdash_{\Phi_{\text{ex}}} e_{\text{venue}} \Leftarrow \mathsf{ty}[\text{RSTR}](\lfloor\texttt{/([A-Z]+) \textbackslash d+/}\rfloor) \rightsquigarrow \mathsf{fold}[s.\mathsf{str} \times \mathsf{list}[s]](\mathsf{pair}(\lVert\texttt{"EXMPL 2015"}\rVert; list1[\mu(s.\mathsf{str} \times \mathsf{list}[s])]\ \mathsf{fold}[s.\mathsf{str} \times \mathsf{list}[s]](\mathsf{pair}(\lVert\texttt{"EXMPL"}\rVert; nil[\mu(s.\mathsf{str} \times \mathsf{list}[s])]))))$$

We write $\lVert ... \rVert$ for the embedding of the indicated string into the IL, and abbreviate the definitions of str, $\mathsf{list}[s]$ and the helper functions for working with lists as above. We refer to the translation as ι_{venue}.

The logic in Figure 8 simply checks that the argument list is empty using the simple helper function ***arity0***, which raises an error if the list is non-empty (rules (*k-raise*) and (*n-raise*); in practice, the provider would here provide an explicit error message to the client). It then calls the helper function ***rmatch*** on the term index to make sure it matches the regular expression provided as the type index, also extracting its groups statically. Finally, it constructs the translation above by calling simple helper functions for lowering static strings and lists into the IL using quotations. We do not show the details; this is, by design, exactly the code one would write in a compiler written in a simply-typed functional language for the branch of the case analysis in the translation logic corresponding to the regular string literal term constructor. Here, we have pulled it into the tycon definition, pushed the case analysis into the semantics, and do not require a separate term constructor by relying on bidirectional typechecking to permit us to share syntax amongst many types.

Arguments Let us now return to the important issue of how the semantics handles arguments by discussing how the literal opcon definition for LPROD in Figure 9 is called by (*ana-lit*) to derive the following translation (assuming the definition of ***list1***):

$$\begin{array}{l}\sigma_{\text{tmidx}} := \textbf{\textit{list1}}[\mathsf{Lbl}]\ \lfloor\mathsf{venue}\rfloor \\ \sigma_{\text{tyidx}} := \textbf{\textit{list1}}[\mathsf{Lbl} \times \mathsf{Ty}]\ (\lfloor\mathsf{venue}\rfloor, \mathsf{ty}[\text{RSTR}](\sigma_{\text{rsidx}})) \\ \emptyset \vdash_{\Phi_{\text{ex}}} \mathsf{lit}[\sigma_{\text{tmidx}}](e_{\text{venue}}) \Leftarrow \mathsf{ty}[\text{LPROD}](\sigma_{\text{tyidx}}) \rightsquigarrow \mathsf{pair}(\mathsf{triv}; \iota_{\text{venue}})\end{array}$$

Note that we did not here exploit the opportunity to optimize the representation of labeled products of length 1. Our interest here is not in this particular decision, but to ensure that such decisions can be freely explored and have only local implications, like changes in the representation of an abstract type in an ML-like module system.

To call the opcon definition, the semantics needs to construct the *argument interface*, σ_{args}, a list of type $\mathsf{List}[\kappa_{\text{arg}}]$. The kind κ_{arg} is defined in Figure 7: it classifies a pair of static functions that serve as hooks into the semantics, permitting the opcon definition to request synthesis or analysis, respectively, for the argument these functions abstract. The judgement $\overline{e} \twoheadrightarrow \sigma^+ \twoheadrightarrow \mathcal{G}^+; n^+$, specified in Figure 12, constructs a list of such function pairs by wrapping the static operators $\mathsf{syn}[n]$ and $\mathsf{ana}[n](\sigma)$. The kinding rules (*k-syn*) and (*k-ana*) (and the signatures in Figure 13) imply that $\emptyset\ \emptyset \vdash^n_{\Xi} \sigma_{\text{args}} : \mathsf{List}[\kappa_{\text{arg}}]$, where n is the number of arguments. All user-defined static terms, as we have seen, are always checked with $n = 0$, ensuring that these operators cannot arise directly in well-kinded opcon definitions, avoiding the possibility of "out of bounds" problems. Passing the argument interface into opcon definitions, which were checked with $n = 0$, does not cause kind safety problems due to a simple lemma, provable inductively:

Lemma 1. *If* $\Delta\ \Gamma \vdash^n_{\Xi} \sigma : \kappa$ *and* $n' > n$ *then* $\Delta\ \Gamma \vdash^{n'}_{\Xi} \sigma : \kappa$.

The arguments are placed in an *argument store*, $\mathcal{G}$. Initially, $\mathcal{G}_0$ is simply a numbered sequence of the arguments themselves, here

$\mathcal{G}_0 := \cdot, 0 \hookrightarrow e_{\text{venue}}$. The normalization semantics for $\mathsf{syn}[n]$ and $\mathsf{ana}[n](\sigma)$ can have an effect on the argument store.

The literal opcon constructor for LPROD in Figure 9 begins by ensuring that the type the literal is being analyzed against might actually be inhabited by ensuring that there are no duplicate labels in the type index.[2] It then recurses simultaneously over the term index, type index and argument list, checking corresponding labels for equality and requesting analysis of each argument against the corresponding type (the helper function ***ana*** is in Figure 13; ***listrec3*** raises an error if the inputs are of different length). The rule (*n-ana*) shows how the argument is extracted from the store and analyzed in the typing context and tycon context provided (by rule (*ana-lit*) here) in the *operation context*, $\mathcal{C}$. If it succeeds, the argument store is updated with the type provided for analysis, the translation and, crucially, a fresh internal variable, x, standing for its translation. The abstract representation of the type provided for analysis is also produced using the whitelist in $\mathcal{C}$, which contains only the tycon of the type the literal is being analyzed against (and the function tycon, which is never held abstract so functions can be used for operations with binding structure, e.g. **case**, below). We write $\mathcal{G} \otimes n \hookrightarrow \ldots$ to indicate that the previous value in the store for n is removed. Because RSTR is not in $\boxdot$, we will thus have:

$\sigma_{\text{rep}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{tyidx}}\ \sigma_{\text{args}} \parallel \emptyset\ \mathcal{G}_0 \Downarrow_{\rightharpoonup,\text{LPROD};\emptyset;\Phi_{\text{ex}}} \rhd(\mathsf{pair}(\mathsf{triv}; x)) \parallel \mathcal{D}\ \mathcal{G}$
$\mathcal{D} := \emptyset, \mathsf{ty}[\text{RSTR}](\sigma_{\text{rsidx}}) \leftrightsquigarrow \alpha$
$\mathcal{G} := \cdot, 0 \hookrightarrow e_{\text{venue}} : \mathsf{ty}[\text{RSTR}](\sigma_{\text{rsidx}}) \rightsquigarrow \iota_{\text{venue}}/x : \alpha$

Abstract Translation Checking We must now check that this *abstract translation* is *representationally consistent*. The next two premises in (*ana-lit*) generate the abstract representation of the type provided for analysis, as well as its corresponding substitution, δ, and internal type variable context, Δ, as was shown previously. The judgement $\mathcal{G} \rightsquigarrow \gamma^+ : \Gamma^+$ generates a term substitution and internal typing context based on $\mathcal{G}$ similarly (Figure 12). Here, $\mathcal{G} \rightsquigarrow (\emptyset, \iota_{\text{venue}}/x) : (\emptyset, x : \alpha)$ is generated. Finally, the semantics checks the *abstract translation* above against the abstract representation using these contexts, e.g. here $\emptyset, \alpha\ \emptyset, x : \alpha \vdash \mathsf{pair}(\mathsf{triv}; x) : 1 \times \alpha$. It is easy to see that this holds. Only after performing this check are the substitutions δ and γ applied. We will show an example where this check fails even when the check would have succeeded if we had applied the substitutions first below and return to the powerful metatheoretic implications of this check in Sec. 3.

Hygiene Note that the variables in Υ are not made available in Γ when performing this check, guaranteeing *hygienic translation*: the translation must not have any free variables other than those generated implicitly via the argument store (which applying the substitution will eliminate) so inadvertent capture cannot occur. We assume also that substitution application is *capture-avoiding* (i.e. the binding structure of terms is maintained, as can be implemented straightforwardly using a locally nameless representation). Thus, terms can be reasoned about without conditions on the context that translation logic may place them under, i.e. *compositionally*.

Targeted Operations On line 5 of Figure 2, we invoke the *labeled product extension opcon*, **with**, on v to create a new labeled product with additional fields, `title` and `doi`, with types consistent with those shown in the return type of *paper* above. The `doi` field's value is computed by invoking the *regular string concatenation opcon*, **concat**. A number of other similar *targeted operations* are seen elsewhere. These all desugar to the same abstract form, $\mathsf{targ}[\mathbf{op}; \sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e})$, where metavariable **op** ranges over opcon names, σ_{tmidx} again captures all statically known portions of the operation (e.g. the field names on line 5), e_{targ} is the *target* expression, and $\overline{e}$ contains all other external subterms.

[2] Implementing a $\mathsf{Map}[\kappa]$ kind that ensures this by construction, while remaining an equality kind, requires some simple additions to the SL, e.g. abstract equality kinds modeled also on SML, which we omit for concision and to demonstrate this more generally applicable technique.

$$\boxed{\vdash_\Phi \Upsilon} \qquad \boxed{\overline{e} \looparrowright \sigma^+ \looparrowright \mathcal{G}^+; n^+} \qquad \boxed{\mathcal{G} \rightsquigarrow \gamma^+ : \Gamma^+}$$

$$\frac{}{\vdash_\Phi \cdot}\ (\textit{ectx-emp}) \qquad \frac{\vdash \Phi : \Xi \qquad \emptyset\ \emptyset \vdash^0_\Xi \sigma : \mathsf{Ty} \qquad \sigma \Downarrow \sigma}{\vdash_\Phi \Upsilon, x \Rightarrow \sigma}\ (\textit{ectx-ext})$$

$$\frac{}{\cdot \looparrowright \sigma_{\text{args}} \looparrowright \cdot; 0}\ (\textit{mkargs-z}) \quad \textbf{\textit{nil}}[\kappa_{\text{arg}}] \Downarrow \sigma_{\text{args}}$$

$$\frac{\overline{e} \looparrowright \sigma_{\text{args}} \looparrowright \mathcal{G}; n \qquad \textbf{\textit{push}}[\kappa_{\text{arg}}]\ \sigma_{\text{args}}\ ((\lambda\_{:}1.\mathsf{syn}[n]), (\lambda \boldsymbol{t}{:}\mathsf{Ty}.\mathsf{ana}[n](\boldsymbol{t}))) \Downarrow \sigma'_{\text{args}}}{\overline{e}, e \looparrowright \sigma_{\text{args}} \looparrowright \mathcal{G}, n \hookrightarrow e; n+1}\ (\textit{mkargs-s})$$

$$\frac{}{\cdot \rightsquigarrow \emptyset : \emptyset}\ (\textit{G-emp}) \qquad \frac{\mathcal{G} \rightsquigarrow \Gamma}{\mathcal{G}, n \hookrightarrow e \rightsquigarrow \Gamma}\ (\textit{G-ignored})$$

$$\frac{\mathcal{G} \rightsquigarrow \gamma : \Gamma}{(\mathcal{G}, n \hookrightarrow e : \sigma \rightsquigarrow \iota/x : \tau) \rightsquigarrow (\gamma, \iota/x) : (\Gamma, x : \tau)}\ (\textit{G-ext})$$

Figure 12. Auxiliary judgements for external statics.

In our calculus, all targeted operations are synthetic. The type constructor of the type recursively synthesized for the target is delegated responsibility over the semantics of the targeted operation. It must define an *opcon term index kind* in its opcon signature and an *opcon definition* in its opcon structure matching the name provided, and satisfying the kinding conditions in rules (*ocs-targ*) and (*oc-targ*) of Figure 7. The rule (*syn-targ*) in Figure 11 shows that type synthesis and translation of targeted operations shares much in common with the logic for literals just described. The term index is first checked against the term index kind, then the opcon definition is extracted and an argument interface is constructed as before. We include the target itself as the notional first argument, though for convenience we don't require that each opcon re-synthesize the translation by passing the components in as shown. Whereas literal opcon definitions only had to decide a translation, targeted opcon definitions must decide both a type and a translation, returning a pair of kind $\mathsf{Ty} \times \mathsf{ITm}$. The abstract translation is checked against the abstract representation of the type produced, as above.

We will not describe each opcon in Figures 8 and 9 in detail. Once again, the code is essentially identical to the code one would write in a standard typechecker. The "magic" happens in the dynamics of the SL, only manifesting when there is a problem. We note the following interesting features in these examples:

- In **concat**, the definition uses the elimination form for types, $\mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma; \sigma)$, to extract the type index, compute the concatenated regular expression and construct a translation that performs the concatenation. This implies that tycons in our calculus have features of both open sum types (a default case is required) and modules. Tycon indices are not held abstract. In the calculus as given, there is rarely a reason to inspect the index of a different tycon. If we added support for direct calls between opcons (e.g. so that the regular expression could expose an operator that returned a labeled product containing the groups directly), having this ability would become more useful.
- The **coerce** operation supports converting between regular string types known statically to obey a sublanguage relation (cf. [10]). Rearrangement of group boundaries can also be performed in this way (this requires a regrouping operation in this choice of translation). We invoke it on line 7 in an analytic position, so subsumption is applied. An interesting direction for future work

is to add support for describing implicit coercions between types of the same tycon by enriching the subsumption rule.

- The **case** operation supports checked conversions. The first argument must synthesize a function type, where the index of the input type determines the regular expression that the string will dynamically be checked against. The "else branch", which is necessary here but not for a coercion, is analyzed against the return type of this function. Note that on line 9, the else branch will never be taken, but the type system "forgot" the information that could be used to optimize it away.
- Both example tycons define a **#** opcon, though with different index kinds. They are seen being used on line 8 to extract a field and then a subgroup. Like literals, types are used to distinguish the semantics despite the shared syntax. This has similarities to operator overloading, but here the static and dynamic semantics themselves are being overloaded statically.
- It is enlightening to derive the abstract translation and representation produced when typechecking $p\#\mathsf{venue}$ on line 8, recalling that p is a labeled product with three fields each of a different regular string type, with venue being the first. Assuming rs is the concrete representation of regular strings:
  $\iota_{\text{abs}} := \mathsf{letpair}(x; \_, y.\mathsf{letpair}(y; z, \_.z))$
  $\tau_{\text{abs}} := \alpha_1$
  $\delta := \emptyset, \mathsf{rs}/\alpha_1, \mathsf{rs}/\alpha_2, \mathsf{rs}/\alpha_3$
  $\Delta := \emptyset, \alpha_1, \alpha_2, \alpha_3$
  $\gamma := \emptyset, p/x$
  $\Gamma := \emptyset, x : 1 \times (\alpha_1 \times (\alpha_2 \times \alpha_3))$
- The **!rc** opcon definition shows an example where the representational consistency check would fail simply because a term of a type other than the one determined by the representation schema of RSTR was generated. The **!ri** opcon definition is more interesting, however: the translation produced matches the concrete type of RSTR, so type safety would not be violated if it were permitted and it would pass the check described above if it were performed post-substitution. But it is quite problematic for reasons of modular reasoning: it claims that the translation has a regular string type, but the string itself does not obey the *value invariant* that RSTR locally maintains: that the string in the translation is actually in the language specified by the type index. Luckily, it does not pass the abstract translation check because it is not *representationally independent* (fold cannot have type α).

3. Metatheory of @λ

Let us now make these intuitions more rigorous. At the outset, a disclaimer: the ideas described in the previous section, unifying several disparate threads of research into an evidently powerful core calculus, are intended to serve as the main intellectual contributions of the paper. We can only present the theorems and lemmas below semi-formally. Proving the theorems below completely rigorously given the normalization semantics we have presented is difficult, because it often requires us to reason about intermediate terms, rather than sub-terms. Rigorous proofs would be most straightforward if we had presented an equivalent structural operational semantics, but this would obscure some important aspects of our presentation. We hope to submit mechanized proofs using a more suitable specification style as an artifact, and have started, but not completed, this work using Coq as of submission.

Type Safety The first major issue that we must address is kind safety and type safety. Because the semantics of the EL relies on the kinding and normalization semantics of the SL, and the normalization semantics of the SL relies on the semantics of the EL, this is somewhat tricky. Let us define $\sigma\ \mathtt{type}_\Phi$ iff $\vdash \Phi \sim \Xi$ and $\emptyset\ \emptyset \vdash^0_\Xi \sigma : \mathsf{Ty}$ and $\sigma \Downarrow \sigma$. Then, we need:

Theorem 1 (Representational Consistency). *If* $\vdash \Phi \sim \Xi$ *and* $\vdash_\Phi \Upsilon$ *and* $\vdash_\Phi \Upsilon \rightsquigarrow \Gamma$ *then* $\emptyset \vdash \Gamma$ *and*

1. *If* $\Upsilon \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota$ *then* $\sigma\ \mathtt{type}_\Phi$ *and* $\vdash_\Phi \sigma \rightsquigarrow \tau$ *and* $\emptyset\ \Gamma \vdash \iota : \tau$.
2. *If* $\sigma\ \mathtt{type}_\Phi$ *and* $\Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota$ *then* $\vdash_\Phi \sigma \rightsquigarrow \tau$ *and* $\emptyset\ \Gamma \vdash \iota : \tau$.

We must simultaneously prove several important lemmas to show Theorem 1. We assume suitable definitions of judgements $\vdash_\Phi \mathcal{D}$, which makes sure that $\mathcal{D}$ stores types, and $\mathcal{D}\ \Upsilon \vdash^n_\Phi \mathcal{G}$, which ensure that the n facts in $\mathcal{G}$ are truly valid.

Many of the cases require ensuring that the normalization semantics is well-behaved (we discuss canonical forms below):

Lemma 2 (Static Preservation and Normalization). *If* $\vdash \Phi \sim \Xi$ *and* $\emptyset\ \emptyset \vdash^n_\Xi \sigma : \kappa$ *and* $\vdash_\Phi \mathcal{D}$ *and* $\mathcal{D}\ \Upsilon \vdash^n_\Phi \mathcal{G}$ *and* $\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \sigma' \parallel \mathcal{D}'\ \mathcal{G}'$ *then* $\emptyset\ \emptyset \vdash^0_\Xi \sigma' : \kappa$ *and* $\vdash_\Phi \mathcal{D}'$ *and* $\mathcal{D}'\ \Upsilon \vdash^n_\Phi \mathcal{G}'$ *and* $\sigma' \Downarrow \sigma'$.

We also need these lemmas about concrete representations.

Lemma 3 (Concrete Representations). *If* $\vdash \Phi \sim \Xi$ *and* $\sigma\ \mathtt{type}_\Phi$ *and* $\vdash_\Phi \sigma \rightsquigarrow \tau$ *then* $\emptyset\ \emptyset \vdash \tau$.

Lemma 4 (Ext. Typing Contexts). *If* $\vdash \Phi \sim \Xi$ *and* $\vdash_\Phi \Upsilon$ *then*

1. *If* $x \Rightarrow \sigma \in \Upsilon$ *then* $\sigma\ \mathtt{type}_\Phi$.
2. *If* $\vdash_\Phi \Upsilon \rightsquigarrow \Gamma$ *then* $\emptyset \vdash \Gamma$.

Both these and the cases for literals and targeted forms rely on standard properties of substitutions:

Lemma 5 (Substitutions). *If* $\delta : \Delta$ *and* $\Delta \vdash \gamma : \Gamma$ *and* $\Delta\ \Gamma \vdash \iota : \tau$, *then* $\emptyset \vdash [\delta]\Gamma$ *and* $\emptyset\ \emptyset \vdash [\delta][\gamma]\iota : [\delta]\tau$, *and if* $\Delta\ \emptyset \vdash \tau$ *then* $\emptyset\ \emptyset \vdash [\delta]\tau$.

We also need the assurance that normalization semantics are updating the store in a way that produces valid substitutions.

Lemma 6 (Stores). *If* $\vdash \Phi \sim \Xi$ *and* $\vdash_\Phi \Upsilon$ *and* $\vdash_\Phi \mathcal{D}$ *and* $\mathcal{D}\ \Upsilon \vdash_\Phi \mathcal{G}$ *and* $\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \sigma' \parallel \mathcal{D}'\ \mathcal{G}'$ *then* $\vdash_\Phi \mathcal{D}' \rightsquigarrow \delta : \Delta$ *and* $\delta : \Delta$ *and* $\mathcal{G}' \rightsquigarrow \gamma : \Gamma$ *and* $\Delta \vdash \gamma : \Gamma$.

Ensuring that induction is well-founded is the most complex part of these proofs. We observe that the representation schema for a tycon TC is kinded under a Ξ that does not include TC (Figure 7). Thus, it cannot request the representation of a type constructed by TC unless that type is extracted from the index. The argument store used in typechecking a term contains only subterms and the normalization semantics only attempt to typecheck terms in the argument store. For static terms kinded with $n = 0$, the argument store is left unchanged. In other circumstances, its domain is fixed.

The representational consistency theorem is, in essence, a version of the *type-preserving compilation theorem* developed for the TIL compiler for SML [32], here lifted into the language and using user-defined mappings from external types to internal types (governed by the lemmas proved above). It implies type safety (that well-typed programs cannot "go wrong") by simply setting $\Upsilon = \emptyset$ and invoking the internal type safety theorem:

Lemma 7 (Internal Type Safety). *If* $\emptyset\ \emptyset \vdash \iota : \tau$ *then either* ι val *or* $\iota \mapsto \iota'$ *such that* $\emptyset\ \emptyset \vdash \iota' : \tau$.

Unicity It is straightforward to prove that the type and translation of a term is unique if it exists. Each form that does not simply defer to a sub-term is either analytic, or synthetic, never both. This is an important property, as it justifies our "caching" behavior for $\mathcal{G}$ and ensures that typing is deterministic.

Theorem 2 (Unicity). *If* $\vdash \Phi \sim \Xi$ *and* $\vdash_\Phi \Upsilon$ *and* $\Upsilon \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota$ *and* $\Upsilon \vdash_\Phi e \Leftarrow \sigma' \rightsquigarrow \iota'$ *then* $\sigma = \sigma'$ *and* $\iota = \iota'$.

Note that extending the calculus with analytic targeted operations (e.g. case analysis that does not require synthetic branches) needs care to ensure that unicity is maintained, due to subsumption (i.e. analysis can proceed either via an explicit analysis rule or via a synthesis rule). If we define judgements specifying that subsumption should only proceed if an explicit analysis rule fails, this can be addressed, at the expense of some awkwardness in the specification.

Decidability A useful property is *decidability of typechecking*. Here, we have suggested a decision procedure using the mode annotations, and by ensuring that normalization cannot diverge. We discussed ensuring that requesting a representation, or requesting analysis and synthesis, will not diverge above. The equality kind condition on type indices subsumes the positivity condition that would be needed to ensure that the elim form for types does not introduce non-termination. We must, however, leave rigorously proving strong normalization of the calculus as future work.

A subtle issue arises in rule (*n-syn-fail*) and (*n-ana-fail*). In brackets, we assume a judgement defining the situation where typechecking fails, though we do not inductively define it. In practice, this would not be difficult to determine, though in the specification, it is awkward to write down these cases. If the bracketed premises are omitted, normalization becomes weak (i.e. non-deterministic), but only in that it may fail when it was possible for it to succeed. Similar issues arose in the Harper-Stone semantics for SML [15].

Stability We can now turn to the situation where the tycon context is extended. Doing so does not affect kinding or normalization.

Theorem 3 (Stable Kinding and Normalization)**.** *If* $\vdash \Phi \sim \Xi$ *and* $\Delta\ \Gamma \vdash^{n}_{\Xi} \sigma : \kappa$ *and* $\vdash \Phi, \phi \sim \Xi, \mathrm{TC}[\kappa']$ *then*

1. $\Delta\ \Gamma \vdash^{n}_{\Xi, \mathrm{TC}[\kappa']} \sigma : \kappa$.
2. *If* $\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \sigma' \parallel \mathcal{D}'\ \mathcal{G}'$ *then* $\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi,\phi} \sigma' \parallel \mathcal{D}'\ \mathcal{G}'$.

Several other similar lemmas for judgements indexed by Φ are omitted. The consequence of this is that extending Φ does not affect previously well-typed terms. Note that type systems structured as "bags of rules", where extensions can pattern match arbitrarily, would not have this property simultaneously with unicity (i.e. there would be ambiguities). Our method of delegating to a tycon, which remains stable upon extension, is the key to maintaining stability.

Theorem 4 (Stable Typing)**.** *If* $\vdash \Phi \sim \Xi$ *and* $\vdash_{\Phi} \Upsilon$ *and* $\vdash_{\Phi} \Upsilon \rightsquigarrow \Gamma$ *and* $\vdash \Phi, \phi \sim \Xi, \mathrm{TC}[\kappa']$ *then*

1. *If* $\Upsilon \vdash_{\Phi} e \Rightarrow \sigma \rightsquigarrow \iota$ *then* $\Upsilon \vdash_{\Phi,\phi} e \Rightarrow \sigma \rightsquigarrow \iota$.
2. *If* $\Upsilon \vdash_{\Phi} e \Leftarrow \sigma \rightsquigarrow \iota$ *then* $\Upsilon \vdash_{\Phi,\phi} e \Leftarrow \sigma \rightsquigarrow \iota$.

Conservativity We now turn to the issue of conservativity, i.e. ensuring that the new terms that become well-typed upon extension of the tycon context cannot violate type invariants maintained in the previous "closed world". We saw at the end of Sec. 2 an example of a hazardous situation, where the LPROD tycon attempted to introduce a term of type $\mathsf{ty}[\mathrm{RSTR}](\lfloor /\backslash \mathtt{d}+/ \rfloor)$ that did not maintain the invariant that the string in the translation was in the language of the regular expression specified in the type index. Luckily, the representation of this type was held abstract from the perspective of the opcon structure of LPROD, so the translation, which would have otherwise been representationally consistent, did not pass the abstract representation check, because it was not *representationally independent*, as enforced by our use of an internal type variable to stand for the representation.

Theorem 5 (Conservativity)**.** *If* $\vdash \Phi \sim \Xi$ *and* $\mathsf{ty}[\mathrm{TC}](\sigma_{tyidx})\ \mathtt{type}_{\Phi}$ *and we have a value invariant (*$\forall e$*, if* $\emptyset \vdash_{\Phi} e \Leftarrow \mathsf{ty}[\mathrm{TC}](\sigma_{tyidx}) \rightsquigarrow \iota$ *then* $\iota \mapsto^{*} \iota'$ *and* $\iota'\ \mathtt{val}$ *implies* $P(\iota')$*) and* $\vdash \Phi, \phi \sim \Xi, \mathrm{TC}'[\kappa']$ *then (*$\forall e$*, if* $\emptyset \vdash_{\Phi,\phi} e \Leftarrow \mathsf{ty}[\mathrm{TC}](\sigma_{tyidx}) \rightsquigarrow \iota$ *then* $\iota \mapsto^{*} \iota'$ *and* $\iota'\ \mathtt{val}$ *implies* $P(\iota')$*).*

Note that this is a stronger statement than stability, as e may not have been well-typed before the extension (e.g. the operation **!ri**). Our approach will be to introduce a technical device into the language that allows us to show, however, that there is always some other term with the same type and translation in the original closed world, precisely because type constructors can only treat one another abstractly. Adding this technical device only weakens the set of properties that can be established benignly, in that one can no longer use exhaustiveness arguments to prove that the default branch of the tycase construct is never taken inside an opcon definition.

Lemma 8 (Other)**.** *If* $\vdash \Phi \sim \Xi$ *and* $\mathsf{ty}[\mathrm{TC}](\sigma_{tyidx})\ \mathtt{type}_{\Phi}$ *and* $\Upsilon \vdash_{\Phi,\phi} e \Leftarrow \mathsf{ty}[\mathrm{TC}](\sigma_{tyidx}) \rightsquigarrow \iota$ *then there exists an* e_{other} *such that* $\Upsilon \vdash_{\Phi} e \Leftarrow \mathsf{ty}[\mathrm{TC}](\sigma_{tyidx}) \rightsquigarrow \iota$.

Proof. We proceed by induction on the typing derivation in the extended context. Most cases proceed by induction, so we omit the details. The interesting cases are for literals and targeted operations.

For $e = \mathsf{lit}[\sigma_{\mathrm{tmidx}}](\overline{e})$, we construct $e_{\mathrm{other}} = \mathsf{lit}[\sigma'_{\mathrm{tmidx}}](\overline{e}')$:

- We construct σ'_{tmidx} from the value of σ_{tmidx} by replacing every sub-term of the form $\mathsf{ty}[\mathrm{TC}'](\sigma''_{\mathrm{tmidx}})$, where TC' is the new tycon defined by ϕ, with $\mathsf{otherty}[m;\tau]$, where $\vdash_{\Phi,\phi} \mathsf{ty}[\mathrm{TC}'](\sigma''_{\mathrm{tmidx}}) \rightsquigarrow \tau$ and choosing an m uniquely determined by $\sigma''_{\mathrm{tmidx}}$ (needed below). The kinding rules then imply that $\emptyset\ \emptyset \vdash^{0}_{\Xi} \sigma'_{\mathrm{tmidx}} : \kappa_{\mathrm{tmidx}}$, where κ_{tmidx} is the literal index kind of TC, because no other kinding rules for values depend on Ξ. This is why opcon indices were restricted to equality kinds.
- We construct $\overline{e}'$ based on the final argument store, $\mathcal{G}$. In particular, for every entry in $\mathcal{G}$ of the form $n \hookrightarrow e : \mathsf{ty}[\mathrm{TC}'](\sigma''_{\mathrm{tmidx}}) \rightsquigarrow \iota/x : \alpha$, we append an argument of the form $\mathsf{asc}[\mathsf{otherty}[m;\tau]](\mathsf{other}[\iota])$ to $\overline{e}'$, where m is determined by the same mapping as above and τ is determined via the substitution induced by $\mathcal{D}$.
  The kinding and normalization rules for $\mathsf{otherty}[m;\tau]$ imply that its representation is always held abstract, and the tycase form always takes the "else" branch. This is precisely the same behavior that $\mathsf{ty}[\mathrm{TC}'](\sigma_{\mathrm{tyidx}})$ has, because $\mathrm{TC}' \neq \mathrm{TC}$ and the opcon definition was not kinded knowing of TC'. This implies that, when starting with $\overline{e}' \hookrightarrow \sigma'_{\mathrm{args}} \hookrightarrow \mathcal{G}'_0; n$, the derivation of $e_{\mathrm{other}} = \mathsf{lit}[\sigma'_{\mathrm{tmidx}}](\overline{e}')$ will follow the same path, producing the same abstract translation, and the final argument store will differ only at the entries we modified: the nth entry will necessarily be of the form $n \hookrightarrow \mathsf{other}[\iota] : \mathsf{otherty}[m;\tau] \rightsquigarrow \iota/x : \alpha$ where α maps to $\mathsf{otherty}[m;\tau]$ in the abstract rep store. As a result, the final substitution γ maps every x to its corresponding ι, as in the derivation using extended context (in essence, we made the same translation arise from a different type for arguments that had a type constructed by the new tycon).

The argument for targeted operations follows an analagous route. Though a bit circuitous in detail, the essence of the proof is simple: because opcon definitions can only extract indices from tycons that were already in Φ, and because the representations and translations are held abstract by the semantics, we can simply introduce a generic family of "other types" and corresponding constants, each specifying its translation directly. These are distinct from the known types, so they cannot violate their invariants, but permit reasoning generically about future extensions. □

4. Related Work and Discussion

Macros If an isomorphic embedding can be established but it is too complex, language-integrated static term rewriting ("macro")

systems, like Template Haskell [31] and Scala's static macros [6], can help by making it possible to generate "boilerplate code" automatically, moving the complexity of computing the embedding of a term into a metaprogram, but the static semantics are fixed. If each metaprogram is invoked like a function, inner macros are expanded before outer macros, and the mechanism enforces abstraction barriers by ensuring that the rewriting logic is hygienic, cannot modify surrounding code, and does not depend on shared state, then compositional reasoning is possible: the meaning of a term depends only on its subterms, not on the specific position it appears in a program.

Desugaring Sometimes, dialects address issues of complexity by introducing new syntax. Indeed, most dialects do build in syntax for a few privileged abstractions (e.g. list literals are nearly ubiquitous, monad comprehensions support a key feature of Haskell). To make the situation less asymmetric, systems that introduce new "desugarings" atop a base language have been developed. For example, dialects of languages built using Sugar* [11] allow syntax extensions to be packaged separately and combined using language-integrated declarations. If the desugaring logic obeys the same constraints described above for macros, semantic reasoning can be performed compositionally (and recent work has shown how reasoning about type correctness can be automated [19]).

These systems do not guarantee that the composition of unambiguous grammars will remain unambiguous (and realistic examples of ambiguous combinations are not uncommon, e.g. XML and HTML). Recent work has made progress in addressing this problem by restricting how syntax extensions can interface with the host syntax. Schwerdfeger and Van Wyk require a globally unique start token and describe checkable conditions pertaining to the follow sets of host language non-terminals to guarantee that extensions can be composed unambiguously [30]. This suffices for modularly adding new keyword-prefixed forms, but literal forms are awkward to define in this way. Omar et al. describe a language-integrated technique specifically for this purpose, using a technique with parallels to the one we are building on [25]. Literal parsing logic is directly associated with user-defined types, forming *type-specific languages* (TSLs), and local type inference controls invocation of TSL parsing logic, guaranteeing that composition is unambiguous. The mechanism guarantees hygiene and inner terms cannot be inspected, so fully modular reasoning is possible.

Optimization and Extensible Compilers When an embedding that preserves the static semantics and is sufficiently simple exists, but a different embedding would better preserve a desired cost semantics, term rewriting techniques can also be used to perform specialized "optimizations", thus achieving an isomorphic embedding. Care must be taken, however, to ensure that the optimized value is not manipulated directly if the rewriting does not preserve the static semantics (i.e. the rewritten term has a different, less constrained type). Type abstraction can be used directly to achieve this property. *Lightweight modular staging*, for example, uses Scala's support for abstract type members in traits to hide the optimized representations from the program [28]. As long as optimizations are modularly known to be meaning-preserving (to be sure, a non-trivial proof obligation that benefits from mechanization, as in recent work on extensible compiler optimization [33]), they can be composed arbitrarily.

Type Refinement The techniques above require that an embedding that preserves the static semantics of a desired fragment already exists. When new static distinctions need to be made for values of an existing type, but new primitive operations are not needed, one solution is to develop a system of *type refinements* based on the fragment of interest [13]. For example, one might refine the type of integers to distinguish negative integers. Most proposals for "pluggable type systems" describe such type refinement systems [5], which require only additional annotations supplied as comments or metadata (e.g. JavaCOP [3]).

Language Frameworks The most general situation is when exposing a fragment requires defining new types as well as new operators, where the static and dynamic semantics governing these new operators make non-trivial use of statically valued information. We saw labeled product types added, which require a new type constructor and new operator constructors. The regular string type's group projection operator had a specialized cost semantics, so a simple refinement would not be able to approximate it. Scala's type system does not track the necessary invariants, so LMS-like optimizations would also not be appropriate.

A variety of *language frameworks* have been developed to make it easier to define new languages, in some cases together with a compiler and other tooling for these languages. Various terms have been used in the literature, including *compiler generators* and *language workbenches*; see [12]. hese tools can decrease the effort needed to define a new language, but they either do not support forming languages from separately defined fragments or provide few modular reasoning principles that make it possible to reason separately about these fragments. Every combination of fragments is a new dialect, and must be reasoned about monolithically. Let us briefly review difficulties that arise.

If the abstract syntax needs to be extended to support a new fragment, problems also arise. In monolithic settings, terms can be implemented using finite recursive sums (i.e. term constructors are often implemented as ML-style datatype constructors), but this does not permit extension, so open sum types or products of functions (i.e. objects [2]) must instead be used. This can present issues when one wishes to modularly define new functionality that should exhaustively cover all terms in the language (e.g. pretty-printers for expressions). Reynolds first identified this problem [27] and Wadler named it the *expression problem*. In this work, we sidestep the problems of syntax, instead leaving it fixed and relying on a bidirectional type system delegating to a relevant tycon.

Once syntactic issues have been addressed, however, there are a host of semantic guarantees that must be established before a language can be relied upon, as we saw. Modern language frameworks guarantee few or none of these properties about the dialects they produce. More alarmingly, even when these properties have been established for two dialects (either in the metatheory or mechanically using a logical framework), and syntactic conflicts are addressed, there is no guarantee that merging the dialects together will conserve these properties. We saw examples of such failures of modularity above. If our goal is to integrate fragment composition into the language, these tools are thus of limited utility. Clients would have to take on the burden of reasoning about these basic properties for every combination of "libraries" they chose to import.

Though we have not mechanically proven these properties for our design, we have given strong evidence that these properties are maintained modularly. That is, we need only establish the semantics of the opcon structures in isolation. The most immediate direction for future work is to finishing mechanizing the metatheory for this technique, and to embed it into an existing proof assistant (using a continuation passing style to simulate the stores in our normalization semantics). Despite this, we believe that the technique as described semi-formally above is compelling. Adding support for tycon-specific typing contexts, implicit coercions and direct opcon calls can all be taken up by the community. Delving further into the question of when can two tycons with the same signature be substituted for one another, using a technique based on admissible relations, is also an avenue we wish to explore [14].

Our work has some similarities to work on maintaining type abstraction using a form of effect system [9], and on translating

$\mathit{arity0} : \mathsf{List}[\kappa_{\mathrm{arg}}] \to 1$
$\mathit{arity1} : \mathsf{List}[\kappa_{\mathrm{arg}}] \to \kappa_{\mathrm{arg}}$
$\mathit{arity2} : \mathsf{List}[\kappa_{\mathrm{arg}}] \to \kappa_{\mathrm{arg}} \times \kappa_{\mathrm{arg}}$
$\mathit{nil} : \forall(\boldsymbol{\alpha}.\mathsf{List}[\boldsymbol{\alpha}])$
$\mathit{list1} : \forall(\boldsymbol{\alpha}.\boldsymbol{\alpha} \to \mathsf{List}[\boldsymbol{\alpha}])$
$\mathit{list2} : \forall(\boldsymbol{\alpha}.\boldsymbol{\alpha} \to \boldsymbol{\alpha} \to \mathsf{List}[\boldsymbol{\alpha}])$
$\mathit{listrec} : \forall(\boldsymbol{\alpha}_1.\forall(\boldsymbol{\alpha}.\mathsf{List}[\boldsymbol{\alpha}_1] \to \boldsymbol{\alpha} \to (\boldsymbol{\alpha}_1 \to \boldsymbol{\alpha} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}))$
$\mathit{listrec2} : \forall(\boldsymbol{\alpha}_1.\forall(\boldsymbol{\alpha}_2.\forall(\boldsymbol{\alpha}.\mathsf{List}[\boldsymbol{\alpha}_1] \to \mathsf{List}[\boldsymbol{\alpha}_2] \to \boldsymbol{\alpha} \to (\boldsymbol{\alpha}_1 \to \boldsymbol{\alpha}_2 \to \boldsymbol{\alpha} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha})))$
$\mathit{listrec3} : \forall(\boldsymbol{\alpha}_1.\forall(\boldsymbol{\alpha}_2.\forall(\boldsymbol{\alpha}_3.\forall(\boldsymbol{\alpha}.\mathsf{List}[\boldsymbol{\alpha}_1] \to \mathsf{List}[\boldsymbol{\alpha}_2] \to \mathsf{List}[\boldsymbol{\alpha}_3] \to \boldsymbol{\alpha} \to (\boldsymbol{\alpha}_1 \to \boldsymbol{\alpha}_2 \to \boldsymbol{\alpha}_3 \to \boldsymbol{\alpha} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}))))$
$\mathit{syn} := \lambda \boldsymbol{a}{:}\kappa_{\mathrm{arg}}.\mathsf{letpair}\ (\mathit{synfn}, \_) = \boldsymbol{a}\ \mathsf{in}\ \mathit{synfn}\ ()$
$\mathit{ana} := \lambda \boldsymbol{a}{:}\kappa_{\mathrm{arg}}.\mathsf{letpair}\ (\_, \mathit{anafn}) = \boldsymbol{a}\ \mathsf{in}\ \mathit{anafn}$

$\mathit{str\_of\_Str} : \mathsf{Str} \to \mathsf{ITm}$
$\mathit{rstrs\_of\_Strs} : \mathsf{List}[\mathsf{Str}] \to \mathsf{ITm}$
$\mathit{rx\_of\_Rx} : \mathsf{Rx} \to \mathsf{ITm}$
$\mathit{nat\_of\_Nat} : \mathsf{Nat} \to \mathsf{ITm}$
$\mathit{prjnth} : \mathsf{Nat} \to \mathsf{ITm} \to \mathsf{ITm}$
$\mathit{append} : \forall(\boldsymbol{\alpha}.\mathsf{List}[\boldsymbol{\alpha}] \to \boldsymbol{\alpha} \to \mathsf{List}[\boldsymbol{\alpha}])$
$\mathit{rscheck} : \forall(t.\mathsf{rx} \rightharpoonup \mathsf{rs} \rightharpoonup (\mathsf{rs} \rightharpoonup t) \rightharpoonup t)$

$\mathit{uniqmap} : \mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}] \to 1$
$\mathit{lookup} : \mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}] \to \mathsf{Lbl} \to (\mathsf{Nat} \times \mathsf{Ty})$
$\mathit{rmatch} : \mathsf{Rx} \to \mathsf{Str} \to (\mathsf{Str} \times \mathsf{List}[\mathsf{Str}])$
$\mathit{rconcat} : \mathsf{Rx} \to \mathsf{Rx} \to \mathsf{Rx}$
$\mathit{rgroupn} : \mathsf{Nat} \to \mathsf{Rx} \to \mathsf{Rx}$
$\mathit{rsublang} : \mathsf{Rx} \to \mathsf{Rx} \to 1$

$\mathit{lbleq} : \mathsf{Lbl} \to \mathsf{Lbl} \to 1$
$rsconcat : \mathsf{rs} \rightharpoonup \mathsf{rs} \rightharpoonup \mathsf{rs}$
$rsgroupn : \mathsf{nat} \rightharpoonup \mathsf{rs} \rightharpoonup \mathsf{rs}$
$rsregroup : \mathsf{rx} \rightharpoonup \mathsf{rs} \rightharpoonup \mathsf{rs}$

Figure 13. Definitions and signatures of helpers used in the examples.

modules to System F [29]. Unlike modules, tycons permit the representation types to be computed, use a separate IL, and permit operations whose types cannot simply be written as functions.

References

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

[2] J. Aldrich. The power of interoperability: why objects are inevitable. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 101–116. ACM, 2013.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] K. Crary, R. Harper, and D. Dreyer. A type system for higher-order modules. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) , Portland, Oregon*, 2002.

[10] C. Doczkal, J.-O. Kaiser, and G. Smolka. A constructive theory of regular languages in coq. In G. Gonthier and M. Norrish, editors, *CPP*, volume 8307 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2013.

[11] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12. ACM, 2013.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In G. Morrisett and T. Uustalu, editors, *ICFP*, pages 331–342. ACM, 2013.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

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

[30] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In M. Hind and A. Diwan, editors, *PLDI*, pages 199–210. ACM, 2009.

[31] 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.

[32] 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.

[33] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 111–121. ACM, 2010.

(constructs lifted from $\mathcal{L}\{\to \forall\, \mu_{\text{ind}}\, 1 \times +\}$ are standard, omitted [14])

$\boxed{\Delta\ \Gamma \vdash^n_\Xi \sigma : \kappa^+}$ $\boxed{\Delta\ \Gamma \vdash^n_\Xi \tau}$ $\boxed{\Delta\ \Gamma \vdash^n_\Xi \iota}$

(*k-ty*)
$$\frac{c[\kappa_{\text{tyidx}}] \in \Xi \qquad \Delta\ \Gamma \vdash^n_\Xi \sigma_{\text{tyidx}} : \kappa_{\text{tyidx}}}{\Delta\ \Gamma \vdash^n_\Xi \mathsf{ty}[c](\sigma_{\text{tyidx}}) : \mathsf{Ty}}$$

(*k-otherty*)
$$\frac{}{\Delta\ \Gamma \vdash^n_\Xi \mathsf{otherty}[m;\tau] : \mathsf{Ty}}$$

(*k-tycase*)
$$\frac{c[\kappa_{\text{tyidx}}] \in \Xi \qquad \Delta\ \Gamma \vdash^n_\Xi \sigma : \mathsf{Ty} \qquad \Delta\ \Gamma, \boldsymbol{x} : \kappa_{\text{tyidx}} \vdash^n_\Xi \sigma_1 : \kappa \qquad \Delta\ \Gamma \vdash^n_\Xi \sigma_2 : \kappa}{\Delta\ \Gamma \vdash^n_\Xi \mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) : \kappa}$$

(*k-qity*)
$$\frac{\Delta\ \Gamma \vdash^n_\Xi \tau}{\Delta\ \Gamma \vdash^n_\Xi \blacktriangleright(\tau) : \mathsf{ITy}}$$

(*k-qity-uq*)
$$\frac{\Delta\ \Gamma \vdash^n_\Xi \sigma : \mathsf{ITy}}{\Delta\ \Gamma \vdash^n_\Xi \blacktriangleleft(\sigma)}$$

(*k-rep*)
$$\frac{\Delta\ \Gamma \vdash^n_\Xi \sigma : \mathsf{Ty}}{\Delta\ \Gamma \vdash^n_\Xi \mathsf{rep}(\sigma) : \mathsf{ITy}}$$

(*k-qitm*)
$$\frac{\Delta\ \Gamma \vdash^n_\Xi \iota}{\Delta\ \Gamma \vdash^n_\Xi \triangleright(\iota) : \mathsf{ITm}}$$

(*k-qitm-uq*)
$$\frac{\Delta\ \Gamma \vdash^n_\Xi \sigma : \mathsf{ITm}}{\Delta\ \Gamma \vdash^n_\Xi \triangleleft(\sigma)}$$

(rules for remaining quoted forms generically recursive)

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

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

(*k-raise*)
$$\frac{\Delta\ \emptyset \vdash \kappa}{\Delta\ \Gamma \vdash^n_\Xi \mathsf{raise}[\kappa] : \kappa}$$

Figure 14. Kinding for static terms.

concrete rep whitelist $\boxdot ::= \rightharpoonup \mid \boxdot, \mathrm{TC}$ **abstract rep store** $\mathcal{D} ::= \emptyset \mid \mathcal{D}, \sigma \leftrightsquigarrow \alpha$ **argument store** $\mathcal{G} ::= \cdot \mid \mathcal{G}, n \hookrightarrow e \mid \mathcal{G}, n \hookrightarrow e : \sigma \rightsquigarrow \iota/x : \tau$ **operation context** $\mathcal{C} ::= \boxdot; \Upsilon; \Phi$

$\boxed{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma^+ \parallel \mathcal{D}^+\ \mathcal{G}^+}$

(constructs lifted from $\mathcal{L}\{\to \forall\, \mu_{\text{ind}}\, 1 \times +\}$ standard, stores propagate as suggested by rules below, error propagation rules omitted [14])

$\boxed{\sigma \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\mathcal{C}}}$

(*n-ty*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}'}{\mathsf{ty}[c](\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathsf{ty}[c](\sigma') \parallel \mathcal{D}'\ \mathcal{G}'}$$

(*n-otherty*)
$$\frac{}{\mathsf{otherty}[m;\tau] \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathsf{otherty}[m;\tau] \parallel \mathcal{D}\ \mathcal{G}}$$

(*n-tycase-1*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathsf{ty}[c](\sigma_{\text{tyidx}}) \parallel \mathcal{D}'\ \mathcal{G}' \qquad [\sigma_{\text{tyidx}}/\boldsymbol{x}]\sigma_1 \parallel \mathcal{D}'\ \mathcal{G}' \Downarrow_{\mathcal{C}} \sigma_1' \parallel \mathcal{D}''\ \mathcal{G}''}{\mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma_1' \parallel \mathcal{D}''\ \mathcal{G}''}$$

(*n-tycase-2*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \qquad \sigma' \neq \mathsf{ty}[c](\sigma_{\text{tyidx}}) \qquad \sigma_2 \parallel \mathcal{D}'\ \mathcal{G}' \Downarrow_{\mathcal{C}} \sigma_2' \parallel \mathcal{D}''\ \mathcal{G}''}{\mathsf{tycase}[c](\sigma; \boldsymbol{x}.\sigma_1; \sigma_2) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma_2' \parallel \mathcal{D}''\ \mathcal{G}''}$$

(*n-qity-parr*)
$$\frac{\blacktriangleright(\tau_1) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_1') \parallel \mathcal{D}'\ \mathcal{G}' \qquad \blacktriangleright(\tau_2) \parallel \mathcal{D}'\ \mathcal{G}' \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_2') \parallel \mathcal{D}''\ \mathcal{G}''}{\blacktriangleright(\tau_1 \rightharpoonup \tau_2) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_1' \rightharpoonup \tau_2') \parallel \mathcal{D}''\ \mathcal{G}''}$$

(*n-qity-alpha*)
$$\frac{}{\blacktriangleright(\alpha) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\alpha) \parallel \mathcal{D}\ \mathcal{G}}$$

(*n-qity-forall*)
$$\frac{\blacktriangleright(\tau) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\tau') \parallel \mathcal{D}'\ \mathcal{G}'}{\blacktriangleright(\forall(\alpha.\tau)) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\forall(\alpha.\tau')) \parallel \mathcal{D}'\ \mathcal{G}'}$$

(other quoted internal types analagously recursive)

(*n-qity-uq*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\tau) \parallel \mathcal{D}'\ \mathcal{G}'}{\blacktriangleright(\blacktriangleleft(\sigma)) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\tau) \parallel \mathcal{D}'\ \mathcal{G}'}$$

(*n-rep-conc-tc*)
$$\frac{\begin{array}{c}\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \mathsf{ty}[\mathrm{TC}](\sigma_{\text{tyidx}}) \parallel \mathcal{D}'\ \mathcal{G}' \\ \mathsf{ty}[\mathrm{TC}](\sigma_{\text{tyidx}}) \leftrightsquigarrow \_ \notin \mathcal{D}' \qquad \mathrm{TC} \in \boxdot \qquad \mathsf{tycon}\ \mathrm{TC}\ \{\omega\} : \psi \in \Phi \qquad \mathsf{rep} = \sigma_{\text{rep}} \in \omega \\ \sigma_{\text{rep}}\ \sigma_{\text{tyidx}} \parallel \mathcal{D}' \cdot \Downarrow_{\boxdot;\emptyset;\Phi} \blacktriangleright(\tau) \parallel \mathcal{D}'' \cdot\end{array}}{\mathsf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \blacktriangleright(\tau) \parallel \mathcal{D}''\ \mathcal{G}'}$$

(*n-rep-conc-parr*)
$$\frac{\begin{array}{c}\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathsf{ty}[\rightharpoonup]((\sigma_1, \sigma_2)) \parallel \mathcal{D}'\ \mathcal{G}' \\ \mathsf{rep}(\sigma_1) \parallel \mathcal{D}' \cdot \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_1) \parallel \mathcal{D}'' \cdot \\ \mathsf{rep}(\sigma_2) \parallel \mathcal{D}'' \cdot \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_1) \parallel \mathcal{D}''' \cdot\end{array}}{\mathsf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\tau_1 \rightharpoonup \tau_2) \parallel \mathcal{D}'''\ \mathcal{G}'}$$

(*n-rep-abs*)
$$\frac{\begin{array}{c}\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \mathsf{ty}[c](\sigma_{\text{tyidx}}) \parallel \mathcal{D}'\ \mathcal{G}' \\ \mathsf{ty}[c](\sigma_{\text{tyidx}}) \leftrightsquigarrow \_ \notin \mathcal{D}' \qquad c \notin \boxdot \qquad (\alpha\ \text{fresh})\end{array}}{\mathsf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \blacktriangleright(\alpha) \parallel \mathcal{D}', \mathsf{ty}[c](\sigma_{\text{tyidx}}) \leftrightsquigarrow \alpha\ \mathcal{G}'}$$

(*n-rep-abs-other*)
$$\frac{\begin{array}{c}\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \mathsf{otherty}[m;\tau] \parallel \mathcal{D}'\ \mathcal{G}' \\ \mathsf{otherty}[m;\tau] \leftrightsquigarrow \_ \notin \mathcal{D}' \qquad (\alpha\ \text{fresh})\end{array}}{\mathsf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\alpha) \parallel \mathcal{D}', \mathsf{otherty}[m;\tau] \leftrightsquigarrow \alpha\ \mathcal{G}'}$$

(*n-rep-seen*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \qquad \sigma' \leftrightsquigarrow \alpha \in \mathcal{D}'}{\mathsf{rep}(\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\alpha) \parallel \mathcal{D}'\ \mathcal{G}'}$$

(*n-qitm-x*)
$$\frac{}{\triangleright(x) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \triangleright(x) \parallel \mathcal{D}\ \mathcal{G}}$$

(*n-qitm-lam*)
$$\frac{\blacktriangleright(\tau) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \blacktriangleright(\tau') \parallel \mathcal{D}'\ \mathcal{G}' \qquad \triangleright(\iota) \parallel \mathcal{D}'\ \mathcal{G}' \Downarrow_{\mathcal{C}} \triangleright(\iota') \parallel \mathcal{D}''\ \mathcal{G}''}{\triangleright(\lambda[\tau](x.\iota)) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \triangleright(\lambda(x.\iota')) \parallel \mathcal{D}''\ \mathcal{G}''}$$

(other quoted internal terms analagously recursive)

(*n-qitm-uq*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \triangleright(\iota) \parallel \mathcal{D}'\ \mathcal{G}'}{\triangleright(\triangleleft(\sigma)) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \triangleright(\iota) \parallel \mathcal{D}'\ \mathcal{G}'}$$

(*n-syn*)
$$\frac{\begin{array}{c}n \hookrightarrow e \in \mathcal{G} \qquad \Upsilon \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota \qquad (x\ \text{fresh}) \\ \mathsf{rep}(\sigma) \parallel \mathcal{D} \cdot \Downarrow_{\boxdot;\Upsilon;\Phi} \blacktriangleright(\tau) \parallel \mathcal{D}' \cdot\end{array}}{\mathsf{syn}[n] \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} (\sigma, \triangleright(x)) \parallel \mathcal{D}'\ \mathcal{G} \otimes n \hookrightarrow e : \sigma \rightsquigarrow \iota/x : \tau}$$

(*n-syn-seen*)
$$\frac{n \hookrightarrow e : \sigma \rightsquigarrow \iota/x : \tau \in \mathcal{G}}{\mathsf{syn}[n] \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} (\sigma, \triangleright(x)) \parallel \mathcal{D}\ \mathcal{G}}$$

(*n-syn-fail*)
$$\frac{n \hookrightarrow e \in \mathcal{G} \qquad [\Upsilon \vdash_\Phi e \not\Rightarrow]}{\mathsf{syn}[n] \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\boxdot;\Upsilon;\Phi}}$$

(*n-ana*)
$$\frac{\begin{array}{c}\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \qquad n \hookrightarrow e \in \mathcal{G}' \\ \Upsilon \vdash_\Phi e \Leftarrow \sigma' \rightsquigarrow \iota \qquad (x\ \text{fresh}) \qquad \mathsf{rep}(\sigma') \parallel \mathcal{D}' \cdot \Downarrow_{\boxdot;\Upsilon;\Phi} \blacktriangleright(\tau) \parallel \mathcal{D}'' \cdot\end{array}}{\mathsf{ana}[n](\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \triangleright(x) \parallel \mathcal{D}''\ \mathcal{G}' \otimes n \hookrightarrow e : \sigma' \rightsquigarrow \iota/x : \tau}$$

(*n-ana-seen-ok*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \qquad n \hookrightarrow e : \sigma' \rightsquigarrow \iota/x : \tau \in \mathcal{G}'}{\mathsf{ana}[n](\sigma) \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \triangleright(x) \parallel \mathcal{D}'\ \mathcal{G}'}$$

(*n-ana-seen-err*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\mathcal{C}} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \qquad n \hookrightarrow e : \sigma'' \rightsquigarrow \iota/x : \tau \in \mathcal{G}' \qquad \sigma' \neq \sigma''}{\mathsf{ana}[n](\sigma) \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\mathcal{C}}}$$

(*n-ana-fail*)
$$\frac{\sigma \parallel \mathcal{D}\ \mathcal{G} \Downarrow_{\boxdot;\Upsilon;\Phi} \sigma' \parallel \mathcal{D}'\ \mathcal{G}' \qquad n \hookrightarrow e \in \mathcal{G}' \qquad [\Upsilon \vdash_\Phi e \not\Leftarrow \sigma']}{\mathsf{ana}[n](\sigma) \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\boxdot;\Upsilon;\Phi}}$$

(*n-raise*)
$$\frac{}{\mathsf{raise}[\kappa] \parallel \mathcal{D}\ \mathcal{G}\ \mathtt{err}_{\mathcal{C}}}$$

Figure 15. Static Normalization