Safely Extending Typed Programming Systems From Within (Thesis Proposal)

Cyrus Omar
Computer Science Department
Carnegie Mellon University
comar@cs.cmu.edu

Abstract

We propose a thesis defending the following statement:

> Typed programming systems can be made more expressive by incorporating *extension mechanisms* that allow library providers to introduce new syntax, type system fragments and editor services. The code that extensions generate can be validated to preserve the system's metatheory, and to guarantee that extensions can be reasoned about separately and composed arbitrarily.

1 Motivation

> *The recent development of programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved problem.* – John Reynolds, 1970 [28]

Achieving both simplicity and generality in the design of a programming language and its associated tools (collectively, a *programming system*) requires organizing around a small number of mechanisms that support the expression of many others as libraries. Much progress has been made in the decades since Reynolds' statement above, but across all language lineages, new language and tool *dialects* remain common vehicles for new ideas, both in research and practice. This suggests that library-based mechanisms are still not general enough to encompass all desirable modes of expression.

We will consider the broad class of situations where providers need to introduce new concrete syntax, type system fragments, or editor services into a system to realize an idea. Equipping the system with *extension mechanisms* that decentralize control over these features would decrease the need for dialects, but this must done with care, because giving away too much control could also weaken the metatheory of the system. For example, were extensions allowed to arbitrarily modify the type system of the language, there would be immediate questions about type safety. Moreover, extension mechanisms must come with *modular reasoning principles* for it to be practical for library clients to rely upon them. Having to resolve conflicts between extensions or reestablish correctness properties of a program whenever a new extension is imported is unacceptably complex.

In this thesis, we will design extension mechanisms that handle these issues, permitting library-based expression of features that today would need to be built into a system by its central designers. More specifically, we show how to delegate control over certain aspects of the concrete syntax, external type system and editor services to *static functions*, i.e. user-defined functions written in a *static language*. By layering these mechanisms over a fixed typed internal language, constraining the static language appropriately and validating the code it generates, we will retain key metatheoretic properties and arrive at powerful modular reasoning principles.

1.1 Motivating Examples

To make the use cases we seek to address more concrete, let us begin with a few simple examples that we will return to throughout this work.

1.1.1 Example 1: Regular Expressions

Regular expressions are commonly used to capture patterns in strings (e.g. motifs in DNA sequences) [33]. Programmers who work with regular expressions would benefit from features like these:

1. **Concrete syntax for pattern literals**. An ideal syntax would permit programmers to express patterns in the concise, conventional manner. For example, consider a bioinformatics application: the BisI restriction enzyme cuts DNA when it sees the motif `GCNGC`, where *N* is any base. We would want to express such patterns with syntax like this, using curly braces to splice one pattern into another and parentheses to delimit captured groups:

   ```
   let N : Pattern = <A|T|G|C>
   let BisI : Pattern = <GC({N})GC>
   ```

   Malformed patterns would ideally result in intelligible *compile-time* parse errors.

2. A **static semantics** that ensures that key invariants related to regular expressions are statically maintained. For example,

   (a) At minimum, we would want that only patterns, or properly escaped strings, are spliced into a pattern literal like that above, to avoid splicing errors and injection attacks [4, 6]. For example,

   ```
   let fourChars : string = '\d\d'
   let twoDigits : Pattern = <\d\d>
   let p : Pattern = <{fourChars}-{twoDigits}>
   ```

   should either be ill-typed because `fourChars` is not a pattern, or be equivalent to the following:

   ```
   let p : Pattern = <\\d\\d-\d\d>
   ```

   It should *not* treat the string incorrectly as a pattern:

   ```
   let p : Pattern = <\d\d-\d\d> (* injection attack vector! *)
   ```

   (b) A more advanced semantics might ensure that out-of-bounds backreferences to a captured group do not occur [30]. For example, we would want `BisI`, above, to have the more precise type `Pattern[1]`, i.e. a pattern with one captured group.

3. A range of **editor services** that support syntax highlighting for patterns, retrieval of relevant documentation, interactive testing and pattern extraction from example strings have been shown to be helpful for programmers when working with complex regular expressions [24]. An example of some of these is shown in Figure 1: the editor service shown helps programmers generate correct code, here via the Java standard library's implementation of regular expressions (discussed below).

No system today provides built-in support for all of the features enumerated above in their strongest form, so library providers must leverage general-purpose mechanisms. The most common strategy, exemplified by Java, is to ask clients to introduce patterns as strings, deferring their parsing and compilation to run-time. This provides only a partial approximation to feature 1 due to clashes between string escape sequences and pattern escape sequences, as can be seen in the generated code at the bottom of Figure 1 (double backslashes are needed). Parsing errors due to these clashes have been observed

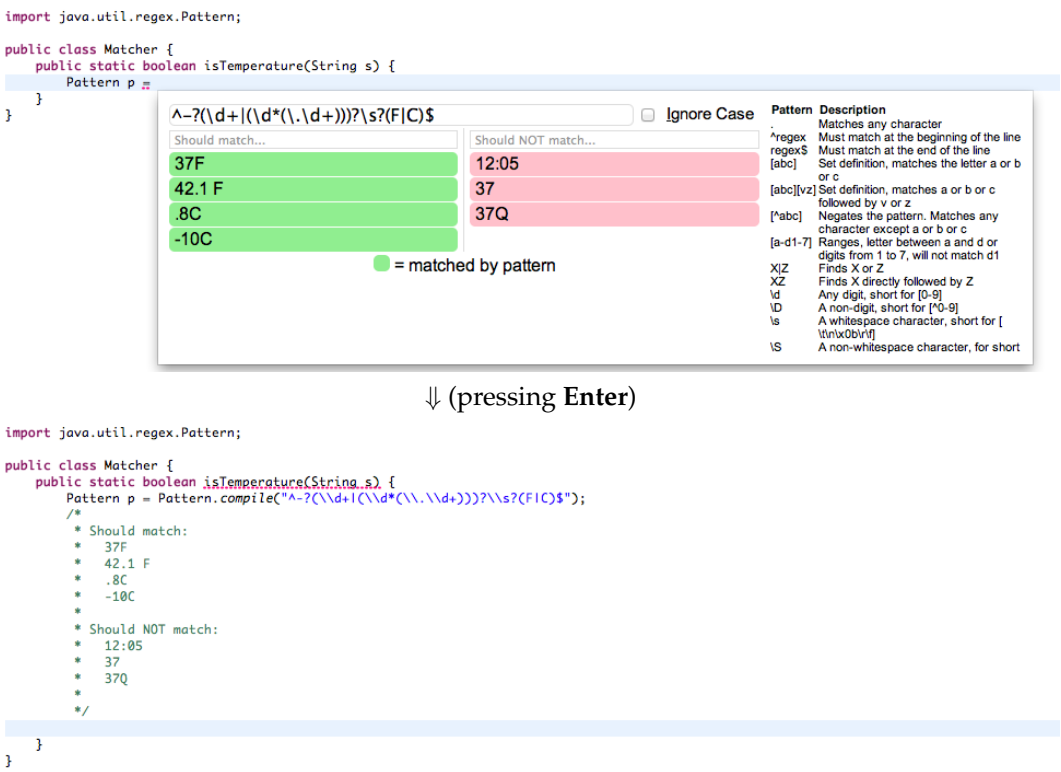

Figure 1: An example of type-specific editor services, here shown for Java.

to be difficult for even experienced programmers to diagnose [25]. Moreover, the static guarantees cannot be provided, leading to run-time exceptions (common even in well-tested code using regular expressions [30]), and mistakes that are not caught even at run-time, some of which can lead to security vulnerabilities due to injection attacks [4]. It also introduces performance overhead due to run-time parsing and compilation of patterns, and redundant well-formedness checks.

A more semantically justifiable approach is to dispense with string representations entirely and directly work with an encoding of regular expression patterns using general-purpose constructs like recursive sums and products (e.g. as exposed by datatypes in a functional language, shown in Sec. 3, or a class hierarchy in an object-oriented language), or using an abstract data type. This provides feature 2a while sacrificing feature 1 (such an encoding affords only an approximation of the verbose abstract syntax of patterns, not their concrete syntax). Advanced invariants like those described in features 2b are not directly tracked by such encodings and require solutions beyond those commonly found in simply-typed languages (a feature like GADTs might suffice in this case [36]).

None of these approaches address the important issue of tool support (feature 3). A number of tools are available online that provide services ranging from simple syntax highlighting to interactive testing and explanations of complex patterns [2] and pattern extraction from examples (e.g. [3]). We have found that programmers benefit substantially from their use [25]. Unfortunately, evidence suggests that tools that must be accessed externally are difficult to discover and are thus used infrequently [21, 8, 25]. We have found that they are also less usable than editor-integrated tools because they require the programmer to switch contexts and cannot make use of the code context [25]. For these reasons, we see editor services as within the broad scope of an abstraction specification.

1.1.2 Example 2: Regular Strings

The examples above all dealt with how regular expression patterns are represented. However, programmers also want to reason about which regular language a string value is in [14]. For example, we might want to ensure that the arguments to a function called `connect` corresponding to a database username and password are only alphanumeric strings. We might include this information directly in the type of `connect`, i.e.

```
type alphanumeric = rstring[<[A-Za-z0-9]+>]
val connect : (alphanumeric * alphanumeric) -> DBConnection
```

This example uses a specialized *type constructor*, `rstring`, indexed by a statically known regular expression. The operations on such *regular strings* would need a static semantics that track the regular language a string is in. For example, a regular string introduced by concatenating two other alphanumeric strings should also be an alphanumeric string.

Ideally, we would be able to use standard string literal syntax with regular strings, e.g.

```
let connection = connect("admin", "password")
```

Regular strings could go beyond simply refining standard string types. For example, they might define operators specific to regular strings like *captured group projection*:

```
let example : rstring[<(\d\d\d)-(\d\d\d\d)>] = "555-5555"
let group0 (* : rstring[<\d\d\d>] *) = example#0
```

Ideally, it would be possible to define this operator such that its cost is $\mathcal{O}(1)$.

1.1.3 Example 3: Labeled Products with Functional Update Operators

The simplest semantics for product types is to define only nullary and binary products [15]. However, in practice, many variations on product types are built in to various dialects of ML and other languages: n-ary tuples, labeled tuples, records (identified up to reordering), and records with width and depth coercions [9] and functional update operators [17]. The Haskell wiki notes that "No, extensible records [i.e. records with a functional update operator] are not implemented in GHC. The problem is that the record design space is large, and seems to lack local optima. [...] As a result, nothing much happens." [1]

We would ideally like to avoid needing the language designer to decide *a priori* on just one point in this large design space. Instead, they should simply provide some minimal way of defining products, perhaps only nullary and binary products, while making it possible for these other variations on products to be defined as libraries and used together without conflict. For example, we would like to be able to define labeled product types, which are like record types but maintain a row ordering. An example of a labeled product types classifying conference papers might be:

```
type Paper = lprod[{
  title : rstring[<.+>],
   conf : rstring[<[A-Z]+ \d\d\d\d>]
}]
```

Because of the row ordering, it should be possible to introduce values of this type with or without explicit labels, e.g.

```
fun make_paper(t : rstring[<.+>]) : Paper = {title=t, conf="EXMPL 2015"}
```

should be equivalent to

```
fun make_paper(t : rstring[<.+>]) : Paper = (t, "EXMPL 2015")
```

Note above that we aim to be able to re-use the standard record and tuple syntax.

We should then be able to project out rows by providing a label (or a numeric index, not shown):

```
let test_paper = make_paper "Test Paper"
test_paper#conf
```

We might also be wish to support functional update. For example, an operation that dropped a row might be used like this:

```
let title_only (* : lprod[{title : rstring[<.+>]}] *) = test_paper.drop[conf]
```

An operation that added a row, or updated the value of an existing row, might be used like this:

```
fun with_author(p : Paper, a : rstring[<.+>]) = p.ext(author=a)
```

1.2 Language-External Approaches

In situations, like these, where a library-based approach is not satisfying, providers must today take a *language-external approach*, either by specifying a new system dialect (implementing it using a *compiler generator* [7], *language workbench* [12], *DSL framework* [13], or simply by forking an existing codebase), or by using an extension mechanism for a particular compiler[1], editor or other tool. For example, a researcher interested in providing the regular expression related features just described (let us refer to these collectively as R) might design a new system with built-in support for them, perhaps basing it on an existing system containing some general-purpose features (G). A different researcher developing a new language-integrated parallel programming abstraction (P) might take the same approach. A third researcher, developing a type system for reasoning about units of measure (Q) might again do the same. This results in a collection of distinct programming systems, as diagrammed in Figure 2a.

Although often justified as "proofs of concept", this is a rather weak notion because no abstraction will be used entirely in isolation and such a construction gives us no rigorously justifiable reason to believe that it can safely and naturally coexist with others specified or implemented in the same way. That is, one must either use the system containing features G+R, G+P or G+Q. There is no system containing G, R, P and Q in other combinations, and merging the systems containing each separately can be non-trivial because there can be serious interference and safety issues, as we will discuss at length throughout this thesis.

Recent evidence indicates that this is one of the major barriers preventing research from being driven into practice. For example, developers prefer high-level language-integrated parallel programming abstractions that provide stronger semantic guarantees [10], but library-based approximations are far more widely adopted because "parallel programming languages" privilege only a few abstractions at the language level. In contrast, it is widely acknowledged that different abstractions are more appropriate in different situations [32]. Moreover, parallel programming is rarely the only relevant concern outside of a classroom or research setting. Support for regular expressions as above would be simultaneously desirable for processing large amounts of genomic data.

1.3 Language-Integrated Approaches

We argue that, due to these problems, taking a language-external approach to realizing a new feature should be considered harmful and avoided whenever possible. The goal of the research being proposed here is to design *language-integrated extension mechanisms* that give providers the ability to define, within libraries, new features that, like those above, that have previously required central planning, so that language-external approaches are less frequently necessary, as illustrated in Figure 2b. Such libraries have been called *active libraries* [34] in that they more actively influence the syntax and semantics of the system.

We must proceed with caution, however: critical issues having to do with safety must be overcome before language-integrated extension mechanisms can be introduced into a

[1]Compilers that modify, or allow modification of, the semantics of their base language, rather than simply permitting semantics-preserving optimizations, should be considered a pernicious means for creating new language dialects. That is, some programs that purport to be written in C, Haskell or Standard ML are actually written in compiler-specific dialects of these languages.

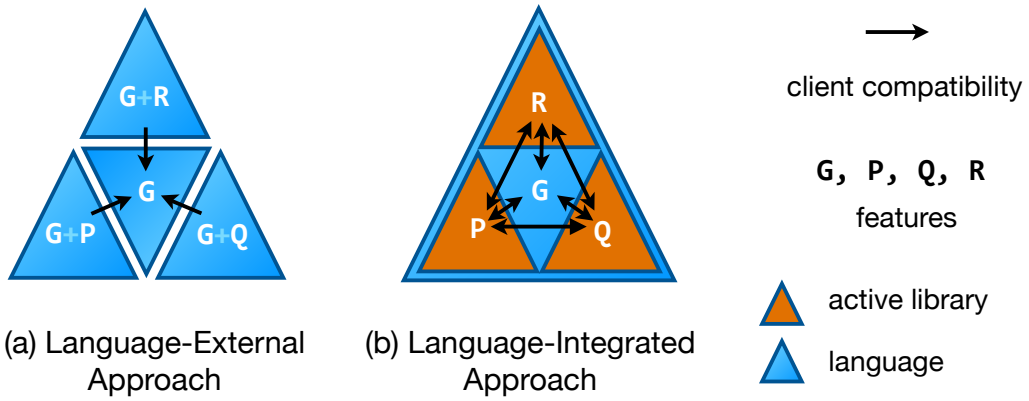

Figure 2: (a) When taking a language-external approach, new features are packaged together into separate languages and tools. (b) When taking a language-integrated approach, there is one extensible host language and the compile-time and edit-time logic governing new constructs is expressed within "active libraries".

system. If too much control over these core features of the system is given to developers, the system may become quite unreliable. Type safety, for example, may not hold if the static and dynamic semantics of the language can be modified or extended arbitrarily from within libraries. Furthermore, even if extensions can be shown not to cause such problems in isolation, there may still be conflicts between extensions that could weaken their semantics, leading to subtle problems that only appear when two extensions are used together, thwarting any attempts to reason modularly about particular programs and the system as a whole. These issues have plagued previous attempts to design language-integrated extensibility mechanisms.

I can include a more detailed related work section if requested.

2 Proposed Contributions

This thesis will introduce several language-integrated extensibility mechanisms that decentralize control over a different feature of the system relevant to examples like above.

- In Sec. 3, we will develop mechanisms that support an extensible concrete syntax for languages with a fixed underlying semantics.
- In Sec. 4, we will then develop an extensible semantics. In particular, for languages specified as a typed translation semantics, we will make it possible to define new base type constructors and their associated operators, like those described in Sec. 1.1.2 and 1.1.3, modularly.
- In Sec. 5, we will then describe a mechanism called *active code completion* that supports specialized editor services, made available via the code completion menu in type-aware editors like Eclipse. The screenshot in Figure 1 is taken from this work.

In each case, we will show that the system retains important metatheoretic properties and that extensions cannot violate one another's autonomy, in ways that we will make more precise as we go on. To justify the expressiveness of each approach, we will give a number of examples of non-trivial features that are, or would need to be, built into conventional systems, but that can be expressed within libraries using the mechanisms we introduce. To help us gather a broad, unbiased collection of examples and demonstrate the scope and applicability of our approaches in practice, we will also conduct small empirical studies when appropriate (though the primary contributions of this work are technical).

```
1  casetype Pattern (* case types are like ML datatypes *)
2    Empty
3    Str of string
4    Seq of Pattern * Pattern
5    Or of Pattern * Pattern
6    Star of Pattern
7    metadata = new (* new simply introduces a Wyvern object; objects are record-like *)
8      val parser : Parser = new (* rows, called fields, are defined with val... *)
9        def parse(ps : ParseStream) : Exp (* and methods with def *)
10         (* ... here, there would be code for a pattern syntax parser
11            generating Wyvern abstract syntax, encoded by the type Exp.
12            It will be called statically to elaborate Pattern literals (see text) ... *)
```

Figure 3: A Wyvern case type classifying regular expression patterns, with a TSL.

3 Extensible Syntax

We begin by considering extensible **concrete syntax** for languages with a fixed semantics. We will work in context of a simplified variant of a language we are developing called Wyvern, but will emphasize the generality of the approach. The examples given in Sec. 1.1 as well as in this section use Wyvern's syntax.

We observe that many syntax extensions are motivated by the desire to introduce alternative introductory forms (a.k.a. *literal forms*) for a particular type or parameterized type constructor (we will consider the former a degenerate case of the latter for simplicity). For example, regular expression pattern literals as described in Sec. 1.1.1 are introductory syntax for the `Pattern` type, defined in Wyvern as shown on lines 1-6 of Figure 3. Similarly, list literals, e.g. `[1, 2, 3]`, which are built in to most languages, are introductory syntax for the `list` type constructor, which is usually included in the standard library.

Both of these examples of specialized syntax would, if built into a language, be defined by *elaboration*, i.e. by a rewriting from the concrete syntax to an equivalent term that uses general-purpose syntax. For example, `[1, 2, 3]` might elaborate to `Cons(1, Cons(2, Cons(3, Nil)))` and the example from bullet point 2(a) in Sec. 1.1.1 might elaborate to the following:

```
let p : Pattern = Seq(Str(fourChars), Seq(Str("-"), twoDigits))
```

We would like to be able to define such elaborations in libraries, but the main problem is that allowing library providers to arbitrarily modify the base syntax of a language could lead to ambiguities when extensions are used together.

Contribution 3.1 (Tycon-Specific Languages)**.** Our first contribution will be to describe the design of a mechanism that allows literal syntax, defined in terms of an elaboration like this, to be associated directly with a user-defined type constructor declaration (for generative parameterized type constructors, i.e. those identified by name, like datatypes in ML, or classes in Java). We call these *tycon-specific languages (TSLs)*. To avoid syntactic ambiguities, the mechanism operates by shifting the parsing of literal bodies into the type system. The syntax associated with a tycon is used when a literal form appears where a term of a type it constructs is expected. For example, the parser elided on lines 10-12 of Figure 3 is invoked statically when parsing the examples in Sec. 1.1.1.

Contribution 3.2 (Layout Delimited Literal Forms)**.** Our mechanism relies on a fixed set of delimiters that separate the host language from extensions. For example, we can use curly braces, square brackets, angle brackets and quotation marks equivalently (the choice is merely stylistic). These are sufficient for simple examples like lists and regular expressions, but for specialized syntax that could contain arbitrary characters, e.g. syntax for a type encoding the structure of an HTML document, we could face the same issue discussed in Sec. 1.1.1: interference between the delimiter used and the body of the delimited form. To solve this problem, we introduce a *layout-delimited literal form*, shown in Figure 4. We formalize this layout-delimited syntax using an Adams' grammar [5].

```
let imageBase : URL = <images.example.com>
let bgImage : URL = <%imageBase%/background.png>
def resultsFor(searchQuery, page)
  serve(~) (* serve : HTML -> Unit *)
     >html
     >head
       >title Search Results
       >style ~
          body { background-image: url(%bgImage%) }
         #search { background-color: %darken(`#aabbcc`, 10pct)% }
       >body
       >h1 Results for < Text(searchQuery)
       >div[id="search"]
         Search again: < SearchBox("Go!")
       < (* fmt_results : DB * SQLQuery * Nat * Nat -> HTML *)
          fmt_results(db, ~, 10, page)
            SELECT * FROM products WHERE {searchQuery} in title
```

Figure 4: Wyvern Example with Multiple TSLs. The tilde indicates a forward referenced literal. Layout determines where it ends. Each type with a TSL is associated with a color for the purposes of our exposition. Black is the base language.

Contribution 3.3 (Formal Semantics of TSLs)**.** We formalize the static semantics, including the literal parsing logic, of TSLs in Wyvern by combining a bidirectional type system [27] with an elaboration semantics (like the Harper-Stone semantics for Standard ML [16]). By distinguishing locations where an expression synthesizes a type from locations where an expression is being analyzed against a known type, we can precisely state where literal forms can and cannot appear and how parsing is delegated to a TSL. The key judgements are of the form:

$$\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau \qquad \text{and} \qquad \Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau$$

External terms, e, elaborate to *internal terms*, i (which do not contain specialized syntax) simultaneously with typechecking. The first judgement captures situations where type analysis is occurring, the second type synthesis. The typing context, Γ, is standard, and the named type context, Θ, associates named types with their declarations and metadata.

Contribution 3.4 (Formal Semantics of Hygiene)**.** A naïve rewriting strategy would be *unhygienic* – it could allow for the inadvertent capture and shadowing of variables around a TSL literal. We show a novel mechanism that ensures hygiene by requiring that the generated AST is closed except for subtrees derived from portions of the user's parse stream that are interpreted as spliced Wyvern expressions or identifiers. We formalize this by splitting the context used to check the type of elaborations. In particular, we separate variables introduced by the desugaring from variables that are in the surrounding context.

Contribution 3.5 (Empirical Support for Applicability of TSLs and TSMs)**.** To examine how broadly applicable the technique is, we have conducted a simple corpus analysis, finding that statically parseable strings are used ubiquitously in existing Java code.

Contribution 3.6 (Typed Syntax Macros)**.** For types that are not identified nominally, i.e. arrow types, or types that already have a TSL but would benefit from alternative syntax, or types in a library that one cannot modify, we will also design and formalize the semantics of a mechanism for explicitly invoked syntax extensions based on the same underlying techniques. For example, rather than using the alternative HTML syntax defined by the TSL above, we might want to use standard HTML syntax. We could define this as follows:

```
syntax standardHTML => HTML = (* ... parser for standard HTML ... *)
```

It could then be used as a prefix on the same set of delimited forms, e.g.

```
standardHTML ~
  <html>...</html>
```

We call explicitly invoked syntax extensions like this *typed syntax macros (TSMs)*. TSMs can also be used as the sole extension mechanism in a language that does not use bidirectional typechecking (e.g. in ML-like languages that use non-local type inference).

Contribution 3.7 (Syntax for Abstract Types (Design))**.** For abstract types defined in ML-style modules, we also need some way to associate syntax with them. We can use TSMs for any particular such type, but it would be better to be able to define syntax for all implementations of an abstract type. For example, if we defined sets via a signature (i.e. module type) SET:

```
signature SET = sig
  type 'a t (* t is an abstract type constructor parameterized by type 'a *)
  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val contains : 'a -> 'a t -> bool
  ...
```

then we would want to be able to define syntax for all structures (i.e. modules) satisfying the SET signature:

```
syntax set => (s : SET).t = (* ... parser parameterized by structure s ... *)
```

We would invoke this syntax macro by providing it with a particular structure that satisfies the SET signature, e.g. `set[ListSet] {1, 2, 3}`. This would suffice for a language like ML. Indeed, one could define datatypes in ML themselves as abstract types, further simplifying the language.

In a bidirectionally typed language like Wyvern, we could go further by associating such syntax with any particular abstract type, e.g. when defining a structure:

```
structure ListSet : SET = struct
  type 'a t with syntax set = 'a list
  ...
```

or when defining a functor abstracting over structures:

```
functor MyStructure(S : SET)
  with type S.t with syntax set = struct
  ...
```

We could then use the syntax defined for these types like a TSL, without explicitly invoking `set[ListSet]`. For example:

```
val x : ListSet.t = {1, 2, 3}
```

3.1 Timeline

Contribution 3.1 through 3.5 were published at ECOOP 2014 [23]. The only piece that was not described there was support for parameterized type constructors, which we plan to formalize in Spring 2015. Note that this was joint work with several group members, but I was the lead author and led the conceptual aspects of the paper.

Contribution 3.6 is in submission to the ACM Symposium on Applied Computing (notification is Nov. 30). This was joint work with Chenglong Wang, an undergraduate student I worked closely with in Summer 2014.

I have discussed aspects of Contribution 3.7 with Prof. Harper. It will be completed in Spring 2015 (possibly for submission to ICFP in March, or as a part of a journal version of the work above).

4 Extensible Semantics

Wyvern has an extensible concrete syntax but a fixed static semantics based around recursive sum and product types. These are of course simple and highly general but there remain situations where providers may wish to extend the type system of a language directly by introducing new type and operator constructors. Examples of language dialects motivated by a need for this level of control abound in the research literature. For example, to implement the features in Sec. 1.1.2, new logic must be added to the type system to statically track information related to the regular language a string is in, which might involve complex decision procedures for language inclusion [14]. Labeled products as described in Sec. 1.1.3 similarly cannot be defined as libraries, because the projection and functional update operators would not have function type.

We seek to enable the introduction of new type and operator constructors in a manner that makes it possible to reason about their metatheory once, in a "closed world" setting, then rely on a more general principle to be sure that this reasoning is conserved in the "open world". For example, once a language is extended with regular string types as described in Sec. 1.1.2 (which Nathan Fulton and I specified in [14]), all terms having a regular string type like $\textsc{rstr}\langle\texttt{/.+/}\rangle$ (our LaTeX notation for type constructor application to a type index) should continue to behave as non-empty strings no matter which other extensions are in use. The extension defining labeled products should not be able to "sneak in" a value of this regular string type that does not obey the invariants established by the extension that defined them.

Contribution 4.1 (Modular Type Constructors (Formal Semantics))**.** We propose taking foundational steps toward this goal by constructing a simple calculus, @λ.

4.1 Overview of @λ

Internal Language At the heart of our semantics is a typed internal language supporting type abstraction (i.e. universal quantification over types) [29]. We use $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$, Figure 5, as representative of a typical intermediate language for a typed language. We assume an internal statics specified by judgements for type assignment $\Delta\ \Gamma \vdash \iota : \tau^+$, type formation $\Delta \vdash \tau$ and typing context formation $\Delta \vdash \Gamma$, and an internal dynamics specified as a structural operational semantics with a stepping judgement $\iota \mapsto \iota^+$ and a value judgement $\iota\ \texttt{val}$.[2] Both the static and dynamic semantics of the IL can be found in any standard textbook covering typed lambda calculi (e.g. [15] or [26]).

External Language Programs "execute" as internal terms, but programmers interface with @λ by writing *external terms*, e. The abstract syntax of external terms is shown in Figure 6 and we introduce various concrete desugarings as we go on. The semantics are specified as a *bidirectionally typed translation semantics*, i.e. the key judgements have the following form, pronounced "Under typing context Υ and tycon context Φ, e (synthesizes / analyzes against) type σ and has translation ι":

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

We distinguish situations where the type is an "output" from those where it must be provided as an "'input" using such a bidirectional approach, also known as *local type inference* [27], for two reasons. The first is to justify the practicality of our approach: local type inference is increasingly being used in contemporary languages (e.g. Scala [22]) because it eliminates the need for type annotations in many places and provides high quality error messages. Secondly, it will give us a clean way to reuse the abstract introductory form, $\mathsf{intro}[\sigma](\overline{e})$, and its associated desugarings, at many types.

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

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

Figure 5: Syntax of $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$, our internal language (IL). Metavariable x ranges over term variables and α and t (distinguished only for stylistic reasons) over type variables.

external terms
$e \quad ::= \quad x \mid \lambda x.e \mid \lambda x{:}\sigma.e \mid e(e) \mid \mathsf{fix}(x.e) \mid e : \sigma$
$\quad\quad \mid \quad \mathsf{intro}[\sigma](\overline{e}) \mid \mathsf{targop}[\mathbf{op}; \sigma](e; \overline{e})$
argument lists $\overline{e} ::= \cdot \mid \overline{e}; e$
external typing contexts $\Upsilon ::= \emptyset \mid \Upsilon, x : \sigma$

Figure 6: Abstract syntax of the external language (EL).

Main Example For example, consider the following types:

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

The *regular string types* σ_{title} and σ_{conf} classify values that behave as strings known to be in a specified regular language [14], i.e. σ_{title} classifies non-empty strings and σ_{conf} classifies strings having the format of a typical conference name. The *labeled product type* σ_{paper} then describes a conference paper by defining two *rows*, each having one of the regular string types just described. Regular string types are defined by tycon context Φ_{rstr} and labeled products by Φ_{lprod}, both introduced in Sec. 4.2.

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

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

We will detail the semantics in Sec. 4.3, but to briefly summarize: because of the type ascription σ_{paper} in e_{ex}, semantic control over e_{paper} will be delegated to the *intro opcon definition* of LPROD. It will decide to analyze the the row value of `title` against σ_{title} and the row value of `conf`, a string literal, against σ_{conf}, causing control to pass similarly to RSTR. Satisfied that the term is well-typed, these will generate translations. Thus, we will be able to derive:

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

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

kinds

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

static terms

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

translational internal types and terms

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

kinding contexts $\boldsymbol{\Gamma} ::= \emptyset \mid \boldsymbol{\Gamma}, \boldsymbol{x} :: \kappa$
kind formation contexts $\boldsymbol{\Delta} ::= \emptyset \mid \boldsymbol{\Delta}, \boldsymbol{\alpha} \mid \boldsymbol{\Delta}, \boldsymbol{k}$
argument environments $\mathcal{A} ::= \overline{e}; \Upsilon; \Phi$

Figure 7: Syntax of the static language (SL). Metavariables $\boldsymbol{x}$ ranges over static term variables, $\boldsymbol{\alpha}$ and $\boldsymbol{k}$ over kind variables and n over natural numbers.

Static Language As suggested above, the main novelty of @λ is that the types and term and type translations do not arise from a fixed specification. Rather, they are *statically computed* by tycon definitions using a *static language* (SL). The SL is itself a typed lambda calculus where *kinds*, κ, serve as the "types" of *static terms*, σ. Its syntax is shown in Figure 7. The portion of the SL covered by the first row of kinds and static terms, some of which are elided, forms a standard functional language consisting of total functions, polymorphic and inductive kinds, and products and sums [15]. It can be seen as a total subset of ML.

Only three new kinds are needed for the SL to serve its role: 1) Ty, classifying types (Sec. 4.2); 2) ITy, classifying *quoted translational internal types*, used to compute type translations in Sec. 4.2.4; and (3) ITm, classifying *quoted translational internal terms*, used to compute term translations in Sec. 4.3.1. The forms $\mathsf{ana}[n](\sigma)$ and $\mathsf{syn}[n]$ will allow tycon definitions to request analysis or synthesis of operator arguments, linking the SL with the EL in Sec. 4.3.2.

The kinding judgement takes the form $\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa^+$, where $\boldsymbol{\Delta}$ and $\boldsymbol{\Gamma}$ are analagous to Δ and Γ and analagous judgements $\boldsymbol{\Delta} \vdash \kappa$ and $\boldsymbol{\Delta} \vdash \boldsymbol{\Gamma}$ are defined. The natural number n is simply a bound used to prevent "out of bounds" references to arguments. The computational behavior of static terms (i.e. the *static dynamics*) is defined by a stepping judgement $\sigma \mapsto_{\mathcal{A}} \sigma^+$, a value judgement $\sigma\ \mathtt{val}_{\mathcal{A}}$ and an *error raised* judgement $\sigma\ \mathtt{err}_{\mathcal{A}}$. $\mathcal{A}$ ranges over *argument environments*, which we return to in Sec. 4.3.2. The multi-step judgement $\sigma \mapsto^*_{\mathcal{A}} \sigma^+$ is the reflexive, transitive closure of the stepping judgement and the normalization judgement $\sigma \Downarrow_{\mathcal{A}} \sigma'$ is derivable iff $\sigma \mapsto^*_{\mathcal{A}} \sigma'$ and $\sigma'\ \mathtt{val}_{\mathcal{A}}$.

4.2 Types

Types are static values of kind Ty, i.e. we write $\sigma\ \mathtt{type}_\Phi$ iff $\sigma\ \mathtt{val}_{\mathcal{A}}$ and $\emptyset\ \emptyset \vdash^n_\Phi \sigma :: \mathsf{Ty}$. All types are of the form $c\langle\sigma\rangle$, where c is a *tycon* and σ is the *type index*. Three kinding rules govern this form, shown in Figure 8, one for each of the three forms for tycons given in Figure 9. The dynamics are simple and tycon-independent: the index is eagerly normalized and errors propagate.[3]

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

Extension Types For types constructed by an *extension tycon*, written TC, rule (k-ty-ext) checks for a *tycon definition* for TC in the tycon context, Φ. The syntax of tycon contexts

[3]The full rules are provided in a detailed supplement, which is available upon request.

$$\text{K-TY-PARR}\quad \frac{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \mathsf{Ty} \times \mathsf{Ty}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \rightharpoonup\langle\sigma\rangle :: \mathsf{Ty}} \qquad \text{K-TY-EXT}\quad \frac{\mathsf{tycon}\ \mathrm{TC}\ \{\theta\} \sim \mathsf{tcsig}[\kappa_{\mathrm{tyidx}}]\ \{\chi\} \in \Phi \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa_{\mathrm{tyidx}}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathrm{TC}\langle\sigma\rangle :: \mathsf{Ty}}$$

$$\text{K-TY-OTHER}\quad \frac{\emptyset \vdash \kappa\ \mathsf{eq} \qquad \boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \sigma :: \kappa \times \mathsf{ITy}}{\boldsymbol{\Delta}\ \boldsymbol{\Gamma} \vdash^n_\Phi \mathsf{other}[m;\kappa]\langle\sigma\rangle :: \mathsf{Ty}}$$

Figure 8: Kinding rules for types, which take the form $c\langle\sigma\rangle$ where c is a tycon and σ is the type index.

is shown in Figure 9 and our main examples are in Figure 10. For now, the only relevant detail is that each tycon defines a *tycon signature*, ψ, which in turn defines the index kind used in the second premise of (k-ty-ext).

Types constructed by RSTR, e.g. σ_{title} and σ_{conf}, specify regular expressions as their indices. The tycon signature of RSTR, ψ_{rstr} in Figure 10, thus specifies index kind Rx, which classifies static regular expression patterns (defined as an inductive sum kind in the usual way). We wrote the type indices in our example assuming a standard concrete syntax. Recent work has shown how to define such type-specific, or here kind-specific, syntax composably [23].

Under the composed context $\Phi_{\mathrm{rstr}}\Phi_{\mathrm{lprod}}$, we defined the labeled product type σ_{paper}. The index kind of LPROD, given by ψ_{lprod} in Figure 10, is $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}]$, where list kinds are defined in the usual way, and Lbl classifies static representations of row labels, and we again used kind-specific syntax to approximate a conventional syntax for row specifications.

Other Types Rule (k-ty-other) governs types constructed by a tycon of the form $\mathsf{other}[m;\kappa]$. These will serve only as technical devices to stand in for tycons other than those in a given tycon context in Theorem 5. The natural number m serves to ensure there are arbitrarily many of these tycons. The index pairs any value of *equality kind* κ, discussed in Sec. 4.2.3, with a value of kind ITy, discussed in Sec. 4.2.4.

4.2.1 Type Case Analysis

Types might be thought of as arising from a distinguished "open datatype" [18] defined by the tycon context. Consistent with this view, a type σ can be case analyzed using $\mathsf{tycase}[c](\sigma;\boldsymbol{x}.\sigma_1;\sigma_2)$. If the value of σ is constructed by c, its type index is bound to $\boldsymbol{x}$ and branch σ_1 is taken. For totality, a default branch, σ_2, must be provided. For example, the kinding rule for extension tycons is:

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

We will see an example of its use in an opcon definition in Sec. 4.3.4. The rule for $c{=}\rightharpoonup$ is analagous, but, importantly, no rule for $c{=}\mathsf{other}[m;\kappa]$ is defined (these types always take the default branch and their indices cannot be examined).

4.2.2 Tycon Contexts

The tycon context well-definedness judgement, $\vdash \Phi$ is given in Figure 11 (omitting the trivial rule for $\Phi = \cdot$).

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

Figure 9: Syntax of tycons. Metavariables TC, **op** and m range over extension tycon and opcon names and natural numbers, respectively. We omit leading $\cdot$ in examples.

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

$\Phi_{\text{rstr}} :=$ tycon RSTR {
  trans $= \sigma_{\text{rstr/schema}}$ in
  ana intro $= \sigma_{\text{rstr/intro}}$;
  syn **conc** $= \sigma_{\text{rstr/conc}}$;
  syn **case** $= \sigma_{\text{rstr/case}}$;
  $\cdots\} \sim \psi_{\text{rstr}}$

$\Phi_{\text{lprod}} :=$ tycon LPROD {
  trans $= \sigma_{\text{lprod/schema}}$ in (Sec 4.2.4)
  ana intro $= \sigma_{\text{lprod/intro}}$; (Sec 4.3.1)
  syn # $= \sigma_{\text{lprod/prj}}$; (Sec 4.3.4)
  syn **conc** $= \sigma_{\text{lprod/conc}}$; (Sec 4.3.4)
  $\cdots\} \sim \psi_{\text{lprod}}$

Figure 10: Example tycon signatures and definitions.

4.2.3 Type Equivalence

The first of the three checks in (tcc-ext), and the check in (k-ty-other), simplifies type equivalence: type index kinds must be *equality kinds*, i.e. those for which semantically equivalent values are syntactically equal. Equality kinds are defined by the judgement $\mathbf{\Delta} \vdash \kappa$ eq (see supplement) and are exactly analagous to equality types as in Standard ML [19]. Arrow kinds are not equality kinds.

4.2.4 Type Translations

Recall that a type, σ, defines a translation, τ. Extension tycons compute translations for the types they construct as a function of each type's index by specifying a *translation schema* in the tycon structure, θ. The translation schema must have kind $\kappa_{\text{tyidx}} \to$ ITy, checked by the third premise of (tcc-ext). Terms of kind ITy are introduced by a *quotation form*, $\blacktriangleright(\hat{\tau})$, where $\hat{\tau}$ is a *translational internal type*. Each form of internal type, τ, has a corresponding form of translational internal type. For example, regular string types have type translations abbreviated str. Abbreviating the corresponding translational internal type $\hat{\text{str}}$, we define the translation schema of RSTR as $\sigma_{\text{rstr/trans}} := \lambda \boldsymbol{tyidx}{::}\text{Rx}.\blacktriangleright(\hat{\text{str}})$.

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

$$\text{K-ITY-UNQUOTE}\ \frac{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \sigma :: \text{ITy}}{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \blacktriangleright(\blacktriangleleft(\sigma)) :: \text{ITy}} \qquad \text{K-ITY-TRANS}\ \frac{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \sigma :: \text{Ty}}{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \blacktriangleright(\text{trans}(\sigma)) :: \text{ITy}}$$

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

$$\text{K-ITY-PROD}\ \frac{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \blacktriangleright(\hat{\tau}_1) :: \text{ITy} \qquad \mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \blacktriangleright(\hat{\tau}_2) :: \text{ITy}}{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^n_\Phi \blacktriangleright(\hat{\tau}_1 \times \hat{\tau}_2) :: \text{ITy}}$$

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

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

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

Figure 11: Tycon context well-definedness.

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

4.2.5 Selective Type Translation Abstraction

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

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

We can apply type substitutions to internal types, terms and typing contexts, written $[\delta]\tau$, $[\delta]\iota$ and $[\delta]\Gamma$, respectively. For example, $[\delta_{\text{paper/abs}}]\tau_{\text{paper/abs}}$ is $\tau_{\text{paper}} := \mathsf{str} \times (\mathsf{str} \times 1)$, i.e. the actual type translation of σ_{paper}. Indeed, we can now define the type translation judgement, $\vdash_{\Phi} \sigma\ \mathtt{type} \rightsquigarrow \tau$, mentioned in Sec. 4.1. We simply determine any selectively abstracted translation, then apply the implied substitution:

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

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

$$\text{ABS-EXT-DELEGATED}\quad \frac{\text{tycon TC}\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \psi \in \Phi \qquad \sigma_{\text{schema}}(\sigma_{\text{tyidx}}) \Downarrow_{\cdot;\emptyset;\Phi} \blacktriangleright(\hat{\tau}) \quad \hat{\tau} \parallel \mathcal{D} \looparrowright^{\text{TC}}_{\Phi} \tau \parallel \mathcal{D}' \qquad \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \Delta \vdash \tau}{\mathsf{trans}(\text{TC}\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \looparrowright^{\text{TC}}_{\Phi} \tau \parallel \mathcal{D}'}$$

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

$$\text{ABS-EXT-NOT-DELEGATED-NEW}\quad \frac{c \neq \text{TC} \qquad \text{TC}\langle\sigma_{\text{tyidx}}\rangle \notin \mathrm{dom}(\mathcal{D}) \qquad \text{tycon TC}\ \{\mathsf{trans} = \sigma_{\text{schema}}\ \mathsf{in}\ \omega\} \sim \psi \in \Phi \quad \sigma_{\text{schema}}(\sigma_{\text{tyidx}}) \Downarrow_{\cdot;\emptyset;\Phi} \blacktriangleright(\hat{\tau}) \qquad \hat{\tau} \parallel \mathcal{D} \looparrowright^{c}_{\Phi} \tau \parallel \mathcal{D}' \quad \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \Delta \vdash \tau \qquad (\alpha\ \text{fresh})}{\mathsf{trans}(\text{TC}\langle\sigma_{\text{tyidx}}\rangle) \parallel \mathcal{D} \looparrowright^{c}_{\Phi} \alpha \parallel \mathcal{D}', \text{TC}\langle\sigma_{\text{tyidx}}\rangle \leftrightarrow [\delta]\tau/\alpha}$$

The translations of "other" types are given directly in their indices (in Sec. 4.4, we will replace some extension types with other types, and this is how we preserve their translations):

$$\frac{\hat{\tau} \parallel \mathcal{D} \twoheadrightarrow_{\Phi}^{\mathsf{other}[m;\kappa]} \tau \parallel \mathcal{D}' \qquad \mathcal{D}' \rightsquigarrow \delta : \Delta \qquad \Delta \vdash \tau}{\mathsf{trans}(\mathsf{other}[m;\kappa]\langle(\sigma, \blacktriangleright(\hat{\tau})\rangle)) \parallel \mathcal{D} \twoheadrightarrow_{\Phi}^{\mathsf{other}[m;\kappa]} \tau \parallel \mathcal{D}'} \text{ ABS-OTHER-DELEGATED}$$

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

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

$$\frac{\mathsf{trans}(\sigma_1) \parallel \mathcal{D} \twoheadrightarrow_{\Phi}^{c} \tau_1 \parallel \mathcal{D}' \qquad \mathsf{trans}(\sigma_2) \parallel \mathcal{D}' \twoheadrightarrow_{\Phi}^{c} \tau_2 \parallel \mathcal{D}''}{\mathsf{trans}(\rightharpoonup\langle(\sigma_1, \sigma_2)\rangle) \parallel \mathcal{D} \twoheadrightarrow_{\Phi}^{c} \tau_1 \rightharpoonup \tau_2 \parallel \mathcal{D}''} \text{ ABS-PARR}$$

4.3 External Terms

Now that we have established how types are constructed and how type translations are computed, we are ready to give the semantics for external terms, shown in Figure 12.

Because we are defining a bidirectional type system, the rule (subsume) is needed to allow synthetic terms to be analyzed against an equivalent type. Per Sec. 4.2.3, equivalent types must be syntactically identical at normal form, and we consider analysis only if σ $\mathtt{type}_{\Phi}$, so the rule is straightforward. To use an analytic term in a synthetic position, the programmer must provide a type ascription, written $e : \sigma$. The ascription is kind checked and normalized to a type before being used for analysis by rule (ascribe).

Rules (syn-var) states that variables synthesize types, as is standard. Functions operate in the standard manner given our definitions of types and type translations (used to generate annotations in the IL). We use Plotkin's fixpoint operator for general recursion (cf. [15]), and define it only analytically with rule (ana-fix). We also define an analytic form of lambda without a type ascription to emphasize that bidirectional typing allows you to omit type ascriptions in analytic positions.

4.3.1 Generalized Intro Operations

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

Before discussing rules (ana-intro) and (ana-intro-other), we note that we can recover a variety of standard concrete introductory forms by desugaring. For example, the string literal form, `"s"`, desugars to $\mathsf{intro}[\texttt{"s"}_{\text{SL}}](\cdot)$, i.e. the term index is the corresponding static value of kind Str and there are no arguments. Similarly, we define a generalized labeled collection form, $\{\mathtt{lbl}_1 = e_1, \ldots, \mathtt{lbl}_n = e_n\}$, that desugars to $\mathsf{intro}[[\mathtt{lbl}_1, \ldots, \mathtt{lbl}_n]](e_1; \ldots; e_n)$, i.e. a static list constructed from the row labels is the term index and the corresponding row values are the arguments. Additional desugarings are discussed in the supplement. The literal form in e_{paper}, from Sec. 4.1, for example, desugars to $e_{\text{paper}} := \mathsf{intro}[\mathtt{[title, conf]}](\mathit{title}; \mathsf{intro}[\texttt{"EXMPL 2015"}_{\text{SL}}](\cdot))$.

Let us now derive the typing judgement in Sec. 4.1. We first apply (syn-lam), which will ask $(e_{\text{paper}} : \sigma_{\text{paper}})$ to synthesize a type in the typing context $\Upsilon_{\text{ex}} = \mathit{title} : \sigma_{\text{title}}$. Then (ascribe) will analyze e_{paper} against σ_{paper} via (ana-intro).

The first premise of (ana-intro) simply finds the tycon definition for the tycon of the type provided for analysis, in this case LPROD. The second premise extracts the *intro term index kind*, κ_{tmidx}, from the *opcon signature*, χ, it specifies. This is simply the kind of term index expected by the tycon, e.g. in Figure 10, LPROD specifies $\mathsf{List}[\mathsf{Lbl}]$, so that it can use

the labeled collection form, while RSTR specifies Str, so that it can use the string literal form. The third premise checks the provided term index against this kind.

The fourth premise extracts the *intro opcon definition* from the *opcon structure*, ω, of the tycon structure, calling it σ_{def}. This is a static function that is applied, in the seventh premise, to determine whether the term is well-typed, raising an error if not or computing a translation if so. Its kind is checked by the judgement $\vdash_\Phi \omega \sim \psi$, which was the final premise of the rule (tcc-ext) and is defined in Figure 13. Rule (ocstruct-intro) specifies that it has access to the type index, the term index and an interface to the list of arguments, discussed below, and returns a *quoted translational internal term* of kind ITm, analagous to ITy. The intro opcon definitions for RSTR and LPROD are given in Figures 14 and 15.

Though the latter will be encountered first in our example derivation, let us first consider the intro opcon definition in Figure 14 because it is more straightforward. It will be used to analyze the row value $\mathsf{intro}[\texttt{"EXMPL 2015"}_{\text{SL}}](\cdot)$ against σ_{conf}. It begins by making sure that no arguments were passed in using the helper function ***arity0*** :: $\mathsf{List}[\mathsf{Arg}] \to 1$ defined such that any non-empty list will raise an error, via the static term $\mathsf{raise}[1]$. In practice, the tycon provider would specify an error message here. Next, it checks the string provided as the term index against the regular expression given as the type index using ***rmatch*** :: $\mathsf{Rx} \to \mathsf{Str} \to 1$, which we assume is defined in the usual way and again raises an error on failure. Finally, the *translational internal string* corresponding to the static string provided as the term index is generated via the helper function ***str_of_Str*** :: $\mathsf{Str} \to \mathsf{ITm}$.

Static terms of kind ITm are introduceed by the quotation form, $\triangleright(\hat{\iota})$, where $\hat{\iota}$ is a *translational internal term*. This is analagous to the form $\blacktriangleright(\hat{\tau})$ for ITy in Sec. 4.2.4. Each form in the syntax of ι has a corresponding form in the syntax for $\hat{\iota}$ and the kinding rules and dynamics simply recurse through these in the same manner as in Sec. 4.2.4. There is also an analagous unquote form, $\triangleleft(\sigma)$. The two interesting forms of translational internal term are $\mathsf{anatrans}[n](\sigma)$ and $\mathsf{syntrans}[n]$. These stand in for the translation of argument n, the first if it arises via analysis against type σ and the second if it arises via type synthesis. Before giving the rules, let us motivate the mechanism with the intro opcon definition for LPROD, shown in Figure 15.

The first line checks that the type provided is inhabited, in this case by checking that there are no duplicate labels via the helper function ***uniqmap*** :: $\mathsf{List}[\mathsf{Lbl} \times \mathsf{Ty}] \to 1$, raising an error if there are (we briefly discuss alternative strategies in the supplement). The rest of the definition folds over the three lists provided as input: the list mapping row labels to types provided as the type index, the list of labels provided as the term index, and the list of argument interfaces, which give access to the corresponding row values. We assume a straightforward helper function, ***fold3***, that raises an error if the three lists are not of the same length. The base case is the translational empty product.

The recursive case checks, for each row, that the label provided in the term index matches the label in the type index using helper function ***lbleq*** :: $\mathsf{Lbl} \to \mathsf{Lbl} \to 1$. Then, we request type analysis of the corresponding argument, ***rowarg***, against the type in the type index, ***rowty***, by writing ***ana rowarg rowty***. Here, ***ana*** is a helper function defined in Sec. 4.3.2 below that triggers type analysis of the provided argument. If this succeeds, it evaluates to a translational internal term of the form $\triangleright(\mathsf{anatrans}[n](\sigma))$, where n is the position of ***rowarg*** in ***args*** and σ is the value of ***rowty***. If analysis fails, it raises an error. The final line constructs a nested tuple based on the row value's translation and the recursive result. Taken together, the translational internal term that will be generated for our example involving e_{paper} above is $\hat{\iota}_{\text{paper}} := (\mathsf{anatrans}[0](\sigma_{\text{title}}), (\mathsf{anatrans}[1](\sigma_{\text{conf}}), ()))$, i.e. it simply recalls that the two arguments were analyzed against σ_{title} and σ_{conf}, without yet inserting their translations directly. This will be done after *translation validation*, triggered by the final premise of (ana-intro) and described in Sec. 4.3.3.

4.3.2 Argument Interfaces

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

The index n on the kinding judgement is an upper bound on the argument index of terms of the form $\mathsf{ana}[n](\sigma)$ and $\mathsf{syn}[n]$, enforced in Figure 16. Thus, if $\mathsf{args}(n) = \sigma_{\mathrm{args}}$ then $\emptyset\ \emptyset \vdash^{n}_{\Phi} \sigma_{\mathrm{args}} :: \mathsf{List}[\mathsf{Arg}]$. The rule (ocstruct-intro) ruled out writing either of these forms explicitly in an opcon definition by checking against bound $n = 0$. This is to prevent out-of-bounds errors: tycon providers can only access these forms via the argument interface list, which has the correct length.

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

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

If it fails, an error is raised:

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

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

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

The kinding rules also prevent these translational forms from being well-kinded when $n = 0$ and, like $\mathsf{trans}(\sigma)$ in Sec. 4.2.4, they are retained in values of kind ITm.

4.3.3 Translation Validation

The judgement $\vdash^{c}_{\mathcal{A}} \hat{\iota} : \sigma \rightsquigarrow \iota^{+}$, defined by a single rule in Figure 17 and appearing as the final premise of (ana-intro) and the other rules described below, is pronounced "translational internal term $\hat{\iota}$ generated by an opcon associated with tycon c under argument environment $\mathcal{A}$ for an operation with type σ is valid, so translation ι is produced". For example,

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

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

The first premise generates the selectively abstracted type translation for σ given that c was the delegated tycon as described in Sec. 4.2.5. In our running example, this is $\tau_{\mathrm{paper/abs}}$, i.e. $\alpha_0 \times (\alpha_1 \times 1)$.

The judgement $\hat{\iota} \parallel \mathcal{D}\ \mathcal{G} \looparrowright^{c}_{\mathcal{A}} \iota^{+} \parallel \mathcal{D}^{+}\ \mathcal{G}^{+}$, appearing as the next premise, relates a translational internal term $\hat{\iota}$ to an internal term ι called a *selectively abstracted term translation*, because all references to the translation of an argument (having any

type) are replaced with a corresponding variable, x, which will be of the selectively abstracted type translation of the type of that argument. For our example, $\hat{\iota}_{\text{paper}} \parallel \mathcal{D}_{\text{paper/abs}}\ \emptyset \leftrightarrowtail^{\text{LPROD}}_{(title;\texttt{"EXMPL 2015"});\Upsilon_{\text{ex}};\Phi_{\text{rstr}}\Phi_{\text{lprod}}} \iota_{\text{paper/abs}} \parallel \mathcal{D}_{\text{paper/abs}}\ \mathcal{G}_{\text{paper/abs}}$ where $\iota_{\text{paper/abs}} := (x_0, (x_1, ()))$.

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

To derive this, the judgement proceeded recursively along shared forms, as in Sec. 4.2.5. The interesting rule for argument translations derived via analysis is below ($\textsf{syntrans}[n]$ is analagous; see supplement):

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

The third premise of (validate-tr) generates the type substitution and type formation contexts implied by the final type translation store as described in Sec. 4.2.5. Similarly, each term translation store $\mathcal{G}$ implies an internal term substitution, $\gamma ::= \emptyset \mid \gamma, \iota/x$, and corresponding Γ by the judgement $\mathcal{G} \rightsquigarrow \gamma : \Gamma$, appearing as the fourth premise. Here, $\gamma_{\text{paper/abs}} := title/x_0, \texttt{"EXMPL 2015"}_{\text{IL}}/x_1$ and $\Gamma_{\text{paper/abs}} := x_0 : \alpha_0, x_1 : \alpha_1$.

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

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

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

Applying the substitutions $\gamma_{\text{paper/abs}}$ and $\delta_{\text{paper/abs}}$ in the conclusion of the rule, we arrive at the actual term translation $\iota_{\text{paper}} := (title, (\texttt{"EXMPL 2015"}_{\text{IL}}, ()))$. Note that ι_{paper} has type τ_{paper} under the translation of Υ_{ex}, i.e. $\vdash \Upsilon_{\text{ex}}\ \texttt{ctx} \rightsquigarrow \Gamma_{\text{ex}}$ where $\Gamma_{\text{ex}} := title : \textsf{str}$. This relationship will always hold, and implies type safety (Sec. 4.4).

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

4.3.4 Generalized Targeted Operations

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

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

Figure 10 showed portions of the opcon signatures of RSTR and LPROD. The opcons associated with RSTR are taken directly from Fulton et al.'s specification of regular string types [14], with the exception of **case**, which generalizes case analysis as defined there to arbitrary string patterns, based on the form of desugaring we show in the supplement. The opcons associated with LPROD are also straightforward: **#** projects out the row with the provided label and **conc** concatenates two labeled products (updating common rows with the value from the right argument). Note that both RSTR and LPROD can define concatenation.

The fifth premise of (syn-targ) extracts the *targeted opcon definition* of **op** from the opcon structure, ω. Like the intro opcon definition, this is a static function that generates a translational internal term on the basis of the target tycon's type index, the term index and an argument interface list. Targeted opcon definitions additionally synthesize a type. The rule (ocstruct-targ) in Figure 13 ensures that it is well-kinded. For example, the definition of RSTR's **conc** opcon is shown in Figure 18 (others will be in the supplement).

The helper function ***arity2*** checks that two arguments, including the target argument, were provided. We then request synthesis of both arguments. We can ignore the type synthesized by the first because by definition it is a regular string type with type index ***tyidx***. We case analyze the second against RSTR, to extract its index regular expression (raising an error if it is not of regular string type). We then synthesize the resulting regular string type, using the helper function ***rxconcat*** $::$ Rx $\rightarrow$ Rx $\rightarrow$ Rx which generates the synthesized type's index by concatenating the indices of the argument's types consistent with the specification in [14]. Finally the translation is generated using helper function $sconcat$: str $\rightarrow$ str $\rightarrow$ str, the translational term for which we assume has been substituted in directly.

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

4.3.5 Operations Over Other Types

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

4.4 Metatheory

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

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

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

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

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

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

Type Safety Type safety in a typed translation semantics requires that well-typed external terms translate to well-typed internal terms. Type safety for the IL [15] then implies that evaluation cannot go wrong. To prove this, we must in fact prove a stronger theorem: that a term's translation has its type's translation under the typing context's translation (the analagous notion is *type-preserving compilation* in type-directed compilers [31]):

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

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

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

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

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

We have fixed the tycon context Φ_{rstr}, so we can essentially treat the calculus like a type-directed compiler for a calculus with only two tycons, $\rightharpoonup$ and RSTR, plus some "other" one. Such a calculus and compiler specification was given in [14], so we must simply show that the opcon definitions in RSTR adequately satisfy these specification using standard techniques for the SL, a simply-typed functional language [11]. The only "twist" is that the rule (syn-targ-other) can synthesize a regular string type paired with any translational term $\hat{\tau}$. But it will be validated against $\tau_{\text{abs}} = \alpha$ because rule (abs-ext-not-delegated-new) applies in that case. Thus, the invariants cannot be violated by direct application of parametricity in the IL (i.e. this case can always be dispatched via a "free theorem") [35].

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

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

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

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

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

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

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

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

4.5 Timeline

Contribution 4.1 is in submission to PLDI 2015. The supplemental material (approximately 70 pages) contains the majority of the details, although a bit more needs to be done to flesh out the proofs at the end.

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

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

ASCRIBE
$$\frac{\emptyset\,\emptyset \vdash^0_\Phi \sigma :: \mathsf{Ty} \qquad \sigma \Downarrow_{\cdot;\emptyset;\Phi} \sigma' \qquad \Upsilon \vdash_\Phi e \Leftarrow \sigma' \rightsquigarrow \iota}{\Upsilon \vdash_\Phi e : \sigma \Rightarrow \sigma' \rightsquigarrow \iota}$$

SYN-VAR
$$\frac{x : \sigma \in \Upsilon}{\Upsilon \vdash_\Phi x \Rightarrow \sigma \rightsquigarrow x}$$

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

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

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

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

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

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

ANA-INTRO-OTHER
$$\frac{|\overline{e}| = n \qquad \emptyset\,\emptyset \vdash^n_\Phi \rhd(\hat{\iota}) :: \mathsf{ITm} \qquad \rhd(\hat{\iota})\ \mathtt{val}_{\overline{e};\Upsilon;\Phi} \qquad \vdash^{\mathsf{other}[m;\kappa]}_{\overline{e};\Upsilon;\Phi} \hat{\iota} : \mathsf{other}[m;\kappa]\langle\sigma_{\mathrm{tyidx}}\rangle \rightsquigarrow \iota}{\Upsilon \vdash_\Phi \mathsf{intro}[\rhd(\hat{\iota})](\overline{e}) \Leftarrow \mathsf{other}[m;\kappa]\langle\sigma_{\mathrm{tyidx}}\rangle \rightsquigarrow \iota}$$

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

Figure 12: Typing

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

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

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

Figure 13: Opcon structure kinding against tycon signatures

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

Figure 14: The intro opcon definition for RSTR, $\sigma_{\text{rstr/intro}}$.

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

Figure 15: The intro opcon definition for LPROD, $\sigma_{\text{lprod/intro}}$.

$$\text{K-ANA}\ \frac{n' < n \qquad \mathbf{\Delta}\ \mathbf{\Gamma} \vdash^{n}_{\Phi} \sigma :: \mathsf{Ty}}{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^{n}_{\Phi} \mathsf{ana}[n'](\sigma) :: \mathsf{ITm}} \qquad \text{K-SYN}\ \frac{n' < n}{\mathbf{\Delta}\ \mathbf{\Gamma} \vdash^{n}_{\Phi} \mathsf{syn}[n'] :: \mathsf{Ty} \times \mathsf{ITm}}$$

Figure 16: Kinding for the SL-EL interface.

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

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

Figure 17: Translation Validation

syn **conc** = λ***tyidx***::Rx.λ***tmidx***::1.λ***args***::List[Arg].
  letpair (***arg1***, ***arg2***) = ***arity2 args*** in
  letpair (_, ***tr1***) = ***syn arg1*** in
  letpair (***ty2***, ***tr2***) = ***syn arg2***
  tycase[RSTR](***ty2***; ***tyidx2***.
    (RSTR$\langle$***rxconcat tyidx tyidx2***$\rangle, \triangleright(sconcat \triangleleft($***tr1***$) \triangleleft($***tr2***$)))$
  ; raise[Ty $\times$ ITm])

Figure 18: A targeted opcon definition, $\sigma_{\text{rstr/conc}}$.

5 Extensible Editor Services

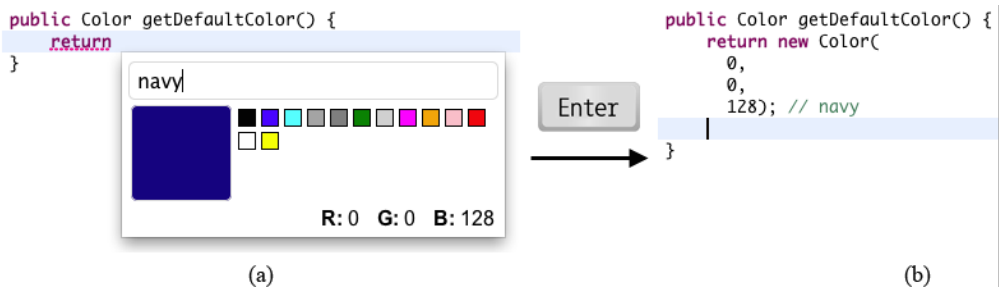

Figure 19: (a) An example code completion palette associated with the `Color` class. (b) The elaboration generated by this palette.

An abstraction might benefit from support beyond the semantics. For example, software developers today make heavy use of the code completion support found in modern source code editors [20]. Code completion typically takes the form of a floating menu containing contextually-relevant variables, assignables, fields, methods, types and other code snippets. When an item in the menu is selected, code is inserted immediately, without further input from the developer. By navigating and selecting from this menu, developers are able to avoid many common spelling and logic errors, eliminate unnecessary keystrokes and explore unfamiliar APIs without incurring the mental overhead associated with switching to an external documentation tool.

These systems are difficult to extend: a fixed strategy determines the completions that are available, so library providers cannot directly specify new domain-specific or contextually-relevant logic or provide assistance beyond that which a menu can provide.

Contribution 5.1 (Active Code Completion)**.** We will describe a technique called *active code completion* that makes developing and integrating a broad array of highly-specialized code generation tools directly into the editor, via the familiar code completion command, significantly simpler.

We discuss active code completion in the context of object construction in Java because type-aware editors for Java are better developed than those for other languages, because we wish to do empirical studies, and because Java already provides a way to associate metadata with classes. The techniques in this section apply equally well to type-aware editors for any language with a similar mechanism.

For example, consider the following Java code fragment:

```
1  public Color getDefaultColor() {
2      return _
```

If the developer invokes the code completion command at the indicated cursor position (_), the editor looks for a *palette definition* associated with the *type* that the expression being entered is being analyzed against, which in this case is `Color` (due to the return type annotation on the method). If an associated palette is found, a menu item briefly describing this palette is added to the standard code completion menu. When selected, the corresponding palette is shown, replacing the standard code completion menu. Figure 19(a) gives an example of a simple palette that may be associated with the `Color` class[4].

The developer can interact with such palettes to provide parameters and other information related to their intent, and receive immediate feedback about the effect these choices will have on the behavior of the object being constructed. When this interaction is complete, the palette produces an elaboration for insertion at the cursor, of the type (here, class) that the palette was associated with (Figure 19(b)).

[4] A video demonstrating this process is available at `http://www.cs.cmu.edu/~NatProg/graphite.html`.

This is, in essence, an enriched edit-time variant of the mechanism used in Wyvern. As discussed in Sec. 3, we associate a palette by providing metadata in the form of a class annotation. For example, we might write:

```
@GraphitePalette(url="http://cs.cmu.edu/~comar/color-palette.html")
class Color { ... }
```

Were a type-aware editor for Wyvern written, it might instead use metadata:

```
objtype Color
  ...
  metadata : HasPalette = new
    val palette_url = <http://cs.cmu.edu/~comar/color-palette.html>
```

Contribution 5.2 (Gathering of Design Criteria)**.** We sought to address the following questions before designing and implementing our active code completion system:

- What *specific* use cases exist for this form of active code completion in a professional development setting?
- What *general* criteria are common to types that would and would not benefit from an associated palette?
- What are some relevant usability and design criteria for palettes designed to address such use cases?
- What capabilities must the underlying active code completion system provide to enable these use cases and user interface designs?

To help us answer these questions, we conducted a survey of 473 professional developers. Their responses, along with information gathered from informal interviews and code corpus analyses, revealed a number of non-trivial functional requirements for palette interfaces as well as the underlying active code completion architecture. Participants also suggested a large number of use cases, demonstrating the broad applicability of this technique. We organized these into several broad categories. The primary concerns relevant to this thesis are:

- The palette mechanism should not be tied to a specific editor implementation.
- The palette mechanism should not be able to arbitrarily access the surrounding source code.
- We provide a mechanism by which users can associate palettes with types externally. This could cause conflicts with palettes that the type defines itself.

Contribution 5.3 (Design and Implementation of Graphite)**.** Next, we developed Graphite, an Eclipse plug-in that implements the active code completion architecture for the Java programming language, allowing Java library developers to associate custom palettes with their own classes. We describe several design choices that we made to satisfy the requirements discovered in our preliminary investigations and examine trade-offs.

Contribution 5.4 (Graphite Pilot Study)**.** Finally, we conducted a pilot lab study with a more complex palette, implemented using Graphite, that assists developers as they write regular expressions. The study provides specific evidence in support of the broader claim that highly-specialized tools that are integrated directly with the editing environment are useful. We conclude that active code completion systems like Graphite are useful because they make developing, deploying and discovering such tools fundamentally simpler.

5.1 Timeline

This work has been published at ICSE 2012 [25].

References

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

[2] RegExr : Create & Test Regular Expressions. http://regexr.com/.

[3] txt2re: headache relief for programmers :: regular expression generator. http://txt2re.com/.

[4] OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2013.

[5] M. D. Adams. Principled Parsing for Indentation-Sensitive Languages: Revisiting Landin's Offside Rule. In *POPL 2013*, pages 511–522, New York, NY, USA, 2013. ACM.

[6] M. Bravenboer, E. Dolstra, and E. Visser. Preventing Injection Attacks with Syntax Embeddings. In *GPCE '07*, pages 3–12, New York, NY, USA. ACM.

[7] R. Brooker, I. MacCallum, D. Morris, and J. Rohl. The compiler compiler. *Annual Review in Automatic Programming*, 3:229–275, 1963.

[8] D. Campbell and M. Miller. Designing Refactoring Tools for Developers. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, pages 9:1–9:2, New York, NY, USA, 2008. ACM.

[9] 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.

[10] V. Cavé, Z. Budimlić, and V. Sarkar. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*, 2010.

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

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

[13] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

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

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

[16] 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.

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

[20] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[21] E. Murphy-Hill and G. C. Murphy. Peer interaction effectively, yet infrequently, enables programmers to discover new tools. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 405–414, New York, NY, USA, 2011. ACM.

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

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

[24] C. Omar, Y. Yoon, T. LaToza, and B. Myers. Active code completion. In *Proc. 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*, pages 261–262, 2011.

[25] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active Code Completion. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press.

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

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

[28] J. C. Reynolds. GEDANKEN - a simple typless language based on the principle of completeness and reference concept. *Comm. A.C.M.*, 13(5), May 1970.

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

[30] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26. ACM, 2012.

[31] 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.

[32] S. Tasharofi, P. Dinges, and R. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In G. Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer Berlin Heidelberg, 2013.

[33] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[34] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proc. 1998 SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

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

[36] Xi, Chen, and Chen. Guarded recursive datatype constructors. In *POPL: 30th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003.