Extensible Gradual Typing Inside a Python

Cyrus Omar Jonathan Aldrich
Carnegie Mellon University
{comar, aldrich}@cs.cmu.edu

Abstract

Programmers prefer using abstractions available as libraries for the language they already use over those requiring a switch to a new language. But library providers cannot typically extend the type systems of these languages directly, so powerful statically typed abstractions remain obscure. This suggests that a language is needed that is backwards compatible with the libraries and tools of a widely adopted language, but that has an internally extensible type system, organized around a mechanism that guarantees that any combination of extensions can be imported safely.

Our aim in this paper is to 1) introduce such a mechanism; and 2) show how it can be implemented as a library for an existing widely used language, here Python. We describe two artifacts: `@lang`, an extensible statically typed language implemented as a library inside Python, and @λ, a minimal core calculus. Both `@lang` and @λ have a fixed syntax but determine the semantics of each term by delegating control, according to a simple but powerful bidirectionally typed elaboration semantics, to methods associated with a relevant *active type constructor*, defined in a library. This process is stable even when additional extensions are imported, so we avoid ambiguity by construction. We describe several examples of interesting type system fragments as libraries, including functional constructs, a type system for secure string sanitation and a foreign function interface to OpenCL.

1. Introduction

Asking programmers to import a library is easier than asking them to adopt a new programming language. Indeed, recent empirical studies underscore the difficulties of driving new languages into adoption, finding that extrinsic factors like compatibility with large existing code bases, library availability, familiarity and tool support are at least as important as intrinsic features of the language [3, 15].

Unfortunately, researchers and domain experts designing new abstractions sometimes find it difficult to soundly, completely and naturally implement them as libraries, particularly those that rely on specialized support from a static type system. In these situations, abstraction providers generally develop a new language dialect, increasingly with the aid of tools like language workbenches [6]. Unfortunately, this *language-oriented approach* [25] is problematic at language boundaries: working with components written in several foreign languages often requires programming against a low-level internal implementation, rather than its semantics, via a foreign-function interface (FFI). This is brittle, unnatural and unsafe and can cause performance issues due to necessary checks and conversions at language boundaries. As more languages are used together, more FFIs must be developed and traversed.

These issues can prevent even dialects introducing only a few new constructs from being adopted. A study comparing a Java dialect, Habanero-Java (HJ), with a comparable library, `java.util.concurrent`, for example, found that the language-based abstractions in HJ were easier to use and provided useful static guarantees [2]. Nevertheless, it concluded that the library-based abstractions remained more practical outside the classroom because using HJ, as a distinct dialect of Java with its own type system, would be problematic in settings where some developers had not adopted it, but needed to interface with code that had. It would also be difficult to use it in combination with other abstractions also implemented as dialects of Java. Moreover, its tool support is more limited. This appears to be a widespread problem: programmers and development teams often cannot use the typed abstractions they prefer because they are only available bundled with specialized languages they cannot adopt [14, 15].

Extensible languages have promised to reduce the problems associated with distinct dialects by giving abstraction providers more direct control over a language's syntax and semantics and supporting the composition of these language fragments within or alongside libraries. Unfortunately, the mechanisms available today have several problems. First, they are themselves often available only within a dialect of an existing language and so face the classic "chicken-and-egg" problem: a language like SugarJ [5] or Wyvern [16] (which permit only syntax extensions over a fixed semantics) must overcome the same extrinsic barriers as a language like HJ. Second, giving abstraction providers too much control over the language can permit the definition of conflicting extensions, a problem that is detected "too late": when extensions are combined by abstraction clients, who are not necessarily capable of understanding and fixing the conflict.

Our focus in this paper is to enable the introduction of new static type system fragments as safely composable libraries for an existing, widely-used language. The most immediately difficult problems related to composing type system fragments arise when attempting to enable the decentralized introduction of new concrete syntax or corresponding term constructors to support the operations of this new type system. Syntactic ambiguities are difficult to modularly preclude (though some techniques exist). Moreover, many tools operate by exhaustive case analysis over the concrete or abstract syntax of the terms of the language they work with (e.g. syntax highlighters, editor modes, pretty-printers, style checkers, documentation generators). Indeed, enabling the decentralized introduction of new term constructors is the standard example used when describing the more general problem of enabling the extension of a sum type with new cases as well as new functions over its cases [18], prompting Wadler to dub it the *expression problem* [24].

Approach Fewer tools operate by exhaustive case analysis over type constructors. In a conventionally formulated programming language, however, it is difficult to introduce a new type system fragment without adding new concrete syntax or term constructors to accompany the new type constructors. In this paper, we aim to show that this need not be the case. We introduce a method that allows library providers to introduce new type system fragments by defining only new type constructors, in libraries, while leaving the concrete syntax, the term constructors and the target language of compilation fixed. The key idea is that we determine the semantics of each term not based directly on the identity of its term constructor (the usual "syntax-directed" approach), but rather by delegating to a statically evaluated function associated with a relevant type constructor. The term constructor's identity only serves to determine how the delegate is assigned. We call user-defined type constructors equipped with static functions and distributed in libraries *active type constructors*.

As a simple example, consider the concrete term `e.r` (where `e` is an expression and `r` is a static label). A fragment defining statically typed pairs might want this to refer to projection of the right component of the pair, while a fragment defining functional record types might want it project out the field labeled `r` and an object system might introduce a more complex semantics involving dynamic dispatch. Were we to naively "open up" the syntax and semantics of the language, these fragments would conflict.

Instead, we assume that the syntax is fixed and includes this commonly used form, mapping it to abstract term attr[“r”](e) (i.e. the term constructor attr, *indexed* by a static representation of the label and taking one *argument*, e). Our semantics delegate responsibility for this term to a static function associated with the type constructor of the type recursively synthesized by the "target" of the operation, e. Tuples, records, objects and variants thereof can all be supported over a common syntax. As we will detail, terms for which there is no clear "target" from which to determine a delegate (e.g. number or list literals) instead either use the type they are being analyzed against (consistent with how extensible parsing is done in Wyvern [16]) or, if not available, a "default" delegate determined by the type constructor governing the function in which they appear. Symmetric binary terms (e.g. addition) use a third protocol, which we will discuss. There is always exactly one active type constructor delegated responsibility over a term and this delegate is stable with respect to additional extension imports, so semantic ambiguities cannot arise by construction.

Specific Aims Our specific aims in this paper are to introduce active type constructors and demonstrate that they are expressive, practically realizable within existing languages and theoretically well-founded. Our target audience is typed language fragment providers (researchers or domain experts) and language designers interested in extensibility mechanisms. We describe two artifacts in pursuit of these aims:

1. We build our mechanism as a library inside Python, `atlang`, using its quasiquotation and reflection facilities.[1] Python is also an interesting starting point because it can be thought of as beginning with a static type system providing just one trivially indexed type constructor, `dyn` [8]. Thus, our mechanism represents a modularly extensible gradual type system. Taken together, `@lang` is almost completely backwards compatible with the Python ecosystem with remarkably little effort: we inherit a large number of existing tools for working with Python files, Python's substantial packaging infrastructure and access to all existing Python libraries. We introduce `@lang` in Sec. 2.

Listing 1 [`listing1.py`] An `@lang` compilation script.

```
1 from atlib import record, decimal, dyn, string,
2   string_in, proto, fn, printf
3
4 print "Hello, compile-time world!"
5
6 Details = record[
7   'amount' : decimal[2],
8   'memo' : dyn]
9 Account = record[
10   'name' : string,
11   'account_num' : string_in[r'\d{2}-\d{8}']]
12 Transfer = proto[Details, Account]
13
14 @fn[Transfer, ...]
15 def log_transfer(t):
16   """Logs a transfer to the console."""
17   printf("Transferring %s to %s.",
18     t.amount.to_string, t.name)
19
20 @fn[...]
21 def __main__():
22   account = {name: "Annie Ace",
23     account_num: "00-00000001"} [:Account]
24   log_transfer(({amount: 5.50, memo: None}, account))
25   log_transfer(({amount: 15.00, memo: "Rent payment"},
26     account))
27
28 print "Goodbye, compile-time world!"
```

We continue in Sec. 3 by briefly describing some more sophisticated examples, including statically typed functional datatypes with nested pattern matching and a complete and statically safe FFI to OpenCL for GPU programming.

2. In Sec. 4, we describe active type constructors in their essential form, developing a minimal calculus called $@\lambda$. The syntax provides only variables, variable binding constructs, type ascription and two generic term constructors. Perhaps surprisingly, we show how a conventional syntax like Python's can be recovered by a purely syntactic desugaring to these term constructors. The semantics for $@\lambda$ take the form of a *bidirectionally typed elaboration semantics* where the available type constructors are tracked by a *tycon context*. We use a simple lambda calculus as both our static and internal language to emphasize the fundamental character of the approach. For more theoretically minded readers or those with no familiarity with Python, this section can be read first.

In Sec. 5, we describe how active type constructors relate to several threads of related work. We conclude in Sec. 6 by summarizing the key features needed by a host language to practically support active type constructors, and delineating present limitations and promising directions for future work.

2. Active Type Constructors in `@lang`

Listing 1 shows an example of a well-typed `@lang` program for which strong correctness guarantees have been statically maintained using several type constructors defined in libraries. As suggested by line 4, the top level of every `@lang` file can be seen as a *compilation script* written directly in Python. `@lang` requires no modifications to the language (version 2.6+ or 3.0+) or features specific to the primary implementation, CPython (so `@lang` supports alternative implementations like Jython and PyPy). This choice pays off immediately on the first line: `@lang`'s import mechanism is Python's import mechanism, so Python's build tools (e.g. `pip`) and package repostories (e.g. `PyPI`) are directly available for distributing `@lang` libraries, including those defining type constructors. Here, `atlib` is the "standard library" but does not benefit from special support in `atlang` itself.

[1] The name was chosen as an initialism for "actively typed language" and because Python marks function decorators, which we use extensively for this purpose, using the @ symbol.

The language that the compilation script is written in, Python, is dynamically typed, but functions governed by `atlang` (those annotated with `@fn` in this example) will be statically typed (Section 2.1) and then translated to dynamically typed Python code (Section 2.2), both under the control of the type constructors of the types used within them. These steps can occur either during a standalone compilation phase (Section 2.3) or just-in-time upon their first invocation (Section 2.4).

Types Types are constructed programmatically in the compilation script by applying a type constructor to an index: `tycon[idx]`. This is in contrast to many statically-typed languages where types (e.g. datatypes, classes) are not constructed programmatically, but can only be declared "literally". In this example, we see several useful types being constructed:

1. On line 6, we construct a functional record type with two (immutable) fields, assigning it to the static identifier `Details`. The field `amount` has type `decimal[2]`, which classifies decimal numbers having two decimal places, and `memo` has type `dyn`, classifying dynamic Python values. Syntactically, the index for `record` is provided using borrowed Python syntax for array slices (e.g. `a[start:end]`) to approximate conventional notation for type annotations. The field names are simply static strings (and could be computed rather than written literally, e.g. `'a' + 'mount'`, to support features like *type providers* [20]).
2. On line 9, we construct another record type. The field `name` has type `string` and the field `account_num` has a type classifying strings guaranteed statically to be in the regular language specified by the regular expression pattern provided as the index (written here as a *raw string literal* [1]). This fragment, based on recent work, statically tracks the language an immutable string is in, providing operations like concatenation, demonstrating the expressive power of our approach [7].
3. On line 12, we construct a simple *prototypic object type* [12], `Transfer`, classifying values consisting of a *fore* of type `Details` and a *prototype* of type `Account`. If a field cannot be found in the fore, the type system will delegate (statically) to the type of the prototype. This makes it easy to share the values of the fields of a common prototype amongst many fores, here to allow us to describe multiple transfers to the same account.

Active Type Constructors Type constructors are implemented in `@lang` libraries as Python classes inheriting from `atlang.Type` (classes here serving as Python's form of open sums). We show portions of `atlib.fn` in Listings 2 and 3, `atlib.record` in Listing 4 and `atlib.string_in` in Listing 6, which we will discuss as we go on. The types in our example are instances of these classes.

Incomplete Types Incomplete types arise from the application of a type constructor to an index containing an ellipsis (a valid array slice form in Python). The static terms `fn[Transfer, ...]` and `fn[...]` seen on lines 14 and 20 are incomplete types. We will discuss how the elided portions of an index (e.g. the return types, which need not be provided) are inferred below.

Incomplete types are not instances of `atlang.Type` but rather instances of `atlang.IncompleteType`. An incomplete type is defined by a type constructor (here `fn`) and an incomplete index (here, `(Transfer, Ellipsis)`). The language controls instantiation by overloading the subscript operator on the `atlang.Type` class object (using Python's *metaclass* mechanism [1]).

Function Definitions To be typechecked by `@lang` and define run-time behavior, function definitions must have an *ascription*, provided by *decorating* them with either a type or, here, an incomplete type, e.g. on lines 14 and 20 (see [1] for a discussion of

Listing 2 A portion of the type constructor `atlib.fn`.

```
1  class fn(atlang.Type):
2    def __init__(self, idx):
3      # called by atlang.Type via the [] operator
4      atlang.Type.__init__(self,
5        self._check_and_norm(idx))
6
7    @classmethod
8    def syn_idx_FunctionDef(cls, ctx, inc_idx, node):
9      arg_types = cls._check_and_norm_inc(inc_idx)
10     if not hasattr(ctx, 'assn_ctx'):
11       ctx.assn_ctx = { }
12     ctx.assn_ctx.update(zip(node.args, arg_types))
13     for stmt in node.body: ctx.check(stmt)
14     last_stmt = node.body[-1]
15     if isinstance(last_stmt, ast.Expr):
16       rty = last_stmt.value.ty
17     else: rty = unit
18     return (arg_types, rty) # fully specified index
19
20   def ana_FunctionDef(self, ctx, node):
21     if not hasattr(ctx, 'assn_ctx'):
22       ctx.assn_ctx = { } # top-level functions
23     ctx.assn_ctx[node.name] = self # recursion
24     ctx.assn_ctx.update(zip(node.args, self.idx[0]))
25     # all but last
26     for stmt in node.body[0:-1]: ctx.check(stmt)
27     last_stmt = node.body[-1]
28     if isinstance(last_stmt, ast.Expr):
29       ctx.ana(last_stmt.value, self.idx[1])
30     elif self.idx[1] == unit or self.idx[1] == dyn:
31       ctx.check(last_stmt)
32     else: raise atlang.TypeError("...", last_stmt)
```

Python decorators; we again use Python's metaclasses and operator overloading to support the use of a class as a decorator).

Ascribed function definitions do not share Python's semantics and indeed the underlying Python function is discarded immediately after its abstract syntax and static environment (its closure and the global dictionary of the Python module it was defined in) have been extracted by the compiler using Python's built-in reflection capabilities and `inspect` and `ast` packages [1]. The `ast` package defines Python's term constructors.

2.1 Active Typechecking

We will now trace through how the function `log_transfer` in Listing 1 is typechecked by `atlang`, via delegation to the type constructors `fn` and `record`, both library constructs.

Functions The `@lang` compiler begins by delegating to the type constructor of the ascription on the function in one of two ways, depending on whether the ascription is a complete type.

If the ascription is an incomplete type, as in our example, the compiler invokes the class method `syn_idx_FunctionDef`, seen on line 8, with the *compiler context* (an object that provides hooks into the compiler and a storage location for accumulating information during typechecking), the incomplete index and the syntax tree of the function definition. Here, this method

1. Checks and normalizes the incomplete function type index (here, `_check_and_norm_inc` checks that the argument types are indeed types and turns empty and single arguments into a 0- or 1-tuples for uniformity; not shown)
2. Adds the argument names and types to a field of the context that tracks the types of local assignments (initializing it first if it is a top-level function, as in our example).
3. Asks the compiler, via the method `ctx.check`, to typecheck each statement in the body (discussed below)
4. If the term constructor of the last statement is `ast.Expr` (a top-level expression), then the type of this expression is the return type, otherwise it is `atlib.unit`.

Listing 3 Some forms in the body of a function delegate to the type constructor of the function they are defined within (via class methods during typechecking).

```
1 #class fn(atlang.Type): (cont'd)
2   @classmethod
3   def check_Assign_Name(cls, ctx, stmt):
4     x, e = stmt.target.id, stmt.value # see ast
5     if x in ctx.assn_ctx: ctx.ana(e, ctx.assn_ctx[x])
6     else: ctx.assn_ctx[x] = ctx.syn(e)
7
8   @classmethod
9   def syn_Name(cls, ctx, e):
10    try: return ctx.assn_ctx[e.id]
11    except KeyError:
12      try: return self._syn_lift(ctx.static_env[e.id])
13      except KeyError: raise atlang.TypeError("...", e)
14
15  @classmethod
16  def syn_default_asc_Str(cls, ctx, e): return dyn
```

5. A fully specified index is generated, from which a type will be synthesized for the function as a whole, here `fn[Transfer, atlib.unit]`. Note that this choice of using the last expression as the implicit return value (and considering any statement-level term constructor other than `Expr` to have a trivial value) is made by `atlib`, not by the language itself.

Were the ascription a complete type like `fn[Transfer, dyn]`, the compiler would delegate to `fn` by calling an instance method on it, `ana_FunctionDef`, seen on line 20, rather than a class method. This method proceeds similarly, but does not need to perform the first step and last steps (and does not need to be a class method) because the index was already determined when the type was constructed, so it can be accessed via `self.idx`. If the final statement is an expression, it is analyzed against the provided return type (see below). Otherwise, only `unit` or `dyn` are valid return types (again, a choice made by `atlib`).

Statements The `ctx.check` method mentioned above is defined by the compiler. As is the pattern, it simply delegates to an active type constructor based on the term constructor of the statement being checked. Most statement-level term constructors other than `Expr` simply delegate to the type constructor of the function they are defined within by calling class methods of the form `check_TermCon`, where `TermCon` is a term constructor or combination of term constructors in a few cases where a finer distinction than what Python itself made was necessary.

For example, the class method `check_Assign_Name`, seen in Listing 3, is called for statements of the form `name = expr`, as on line 22 of Listing 1. In this examples, the assignables context (the `assn_ctx` field of the compiler context) is consulted to determine whether the name being assigned to already has a type due to a prior assignment, in which case the expression is analyzed against that type using `ctx.ana`. If not, the expression must synthesize a type, using `ctx.syn`, and a binding is added.

Bidirectional Typechecking The methods `ctx.ana` and `ctx.syn` are also defined by the compiler and can be called by type constructors to request type analysis (when the expected type is known) and synthesis (when the type is an "output"), respectively, for an expression. Once again, the compiler delegates to a type constructor by a protocol that depends on the term constructor, invoking methods of the form `ana_TermCon` or `syn_TermCon`. This represents a form of bidirectional type system (sometimes also called a *local type inference system*) [17], and the standard subsumption principle applies: if analysis is requested and an `ana_TermCon` method is not defined but a `syn_TermCon` method is, then synthesis proceeds and then the synthesized type is checked for equality against the type provided for analysis. Type equality is defined by checking that the two type constructors are identical and that their indices are equal. Two types with different type constructors are never equal, to ensure that the burden of ensuring that typing respects type equality is local to a single type constructor. No form of subtyping or implicit coercion is supported for the purposes of this paper (though we have designed a mechanism that similarly localizes reasoning to a single type constructor, we do not describe it here.)

Listing 4 A portion of the `atlib.record` type constructor.

```
1 class record(atlang.Type):
2   def __init__(self, idx):
3     # Sig is an unordered mapping from fields to types
4     atlang.Type.__init__(self, Sig.from_slices(idx))
5
6   @classmethod
7   def syn_idx_Dict(self, ctx, partial_idx, e):
8     if partial_idx != Ellipsis:
9       raise atlang.TypeError("...bad index...", e)
10    idx = []
11    for f, v in zip(e.keys, e.values):
12      if isinstance(f, ast.Name):
13        idx.append(slice(f.id, ctx.syn(v)))
14      else: raise atlang.TypeError("...", f)
15    return idx
16
17  def ana_Dict(self, ctx, e):
18    for f, v in zip(e.keys, e.values):
19      if isinstance(f, ast.Name):
20        if f.id in self.idx.fields:
21          ctx.ana(v, self.idx[f.id])
22        else: raise atlang.TypeError("...", f)
23      else: raise atlang.TypeError("...", f)
24    if len(self.idx.fields) != len(e.keys):
25      raise atlang.TypeError("...missing field...", e)
26
27  def syn_Attribute(self, ctx, e):
28    if e.attr in self.idx: return self.idx[e.attr]
29    else: raise atlang.TypeError("...", e)
```

Names If the term constructor of an expression is `ast.Name`, as when referring to a bound variable or an assignable location, then the type constructor governing the function the term appears in is delegated to via the class method `syn_Name` (names, when used as expressions, must synthesize a type). We see the definition of this method for `fn` starting on line 9 in Listing 3. A name synthesizes a type if it is either in the assignables context or if it is in the static environment. In the former case, the type that was synthesized when the assignment occurred (by `syn_Assign_Name`) is returned. In the latter case, we must lift the static value to run-time. For the purposes of this paper, we support only other typed `@lang` functions, Python functions and classes (which are given type `dyn`, and can only be called with values of type `dyn`) and modules (which are given a *singleton type* – a type indexed by the module reference itself – from which @lang functions and Python functions and classes can be accessed as attributes in the usual way, see below).

Literal Expressions and Ascriptions Python designates literal forms for dictionaries, tuples, lists, strings, numbers and lambda functions. In `@lang`, the type constructor delegated control over terms arising from any of these forms is a function of how the typechecker encounters the term.

If the term appears in an analytic position, the type constructor of the type it is being analyzed against is delegated control. We see this on line 22 of Listing 1: the dictionary literal form appears in an analytic context, here because an explicit type ascription, `[:Account]`, was provided. Note that the ascription again repurposes Python's array slice syntax (the start is, conveniently, optional). The compiler invokes the `ana_Dict` method on the type the literal is being analyzed against, which is defined by its type constructor, here `atlib.record`, shown on line 17 of Listing 4. This method analyzes each field value in the literal against the type of the field, extracted from the type index (an unordered mapping from field names to their types, so that type equality is up to reordering). The various literal forms inside the outermost form thus do not need an ascription because they appear in positions where the type they will be analyzed against is provided by `record`. For example, the value of the field `account_num` delegates to `string_in` via the `ana_Str` method (not shown).

An ascription directly on a literal can also be an incomplete type. For example, we can write `[x, y] [:matrix[...]]` or more concisely `[x, y] [:matrix]` instead of `[x, y] [:matrix[i32]]` when we know `x` and `y` synthesize the type `i32`, because the type constructor can use this information to synthesize the appropriate index. The decorator `@fn[Transfer, ...]`, discussed previously, can be seen as a partial type ascription on a statement-level function literal.[2] Lambda expressions support the same ascriptions:

```
(lambda x, y: x + y) [:fn[(i32, i32), ...]]
```

Records support partial type ascription as well, e.g.:

```
{name: "Deepak Dynamic", age: "27"} [:record]
```

The compiler delegates to the type constructor by a class method in these cases (just as we saw above with `fn`). In this case, the class method `syn_idx_Dict`, shown on line 6 of Listing 4, would be called. Because no record index was provided, an index must be synthesized from the literal itself, and thus the values are not analyzed against a type, but must each synthesize a type.

This brings us to the situation where a literal appears in a synthetic position, as the two string literals above do. In such a situation, the compiler delegates responsibility to the type constructor of the function that the literal appears in, asking it to provide a "default ascription" by calling the class method `syn_default_asc_Str`, shown on line 15 of Listing 3. In this case, we simply return `dyn`, so the type of the expression above has type `record['name': dyn, 'age': dyn]`. A different function type constructor might make more precise choices. Indeed, a benefit of using Python as our static language is that it is relatively straightforward to provide a type constructor that allows us to provide different defaults, and control choices like the return semantics, as block-scoped "pragmas" in the static language using Python's `with` statement [1] (details omitted), e.g.

Listing 5 Block-scoped settings can be seen by type constructors.

```
with fn.default_asc(Num=i64, Str=string, Dict=record):
  @fn[...] # : fn[(), record["a": i64, "b": string]]
  def test(): {a: 1, b: "no ascriptions needed!"}
```

Targeted Expressions Expression forms having exactly one sub-expression, like `-e` (term constructor `UnaryOp_USub`) or `e.attr` (term constructor `Attribute`), or where there may be multiple sub-expressions but the left-most one is the only one required, like `e[slices]` (term constructor `Subscript`) or `e(args)` (term constructor `Call`), are called *targeted expressions*. For these, the compiler first synthesizes a type for the left-most sub-expression, then calls either the `ana_TermCon` or `syn_TermCon` methods on that type. For example, we see type synthesis for the field projection operation on records defined starting on line 27 of Listing 4.

Binary Expressions The major remaining expression forms are the binary operations, e.g. `e + e` or `e < e`. These are notionally symmetric, so it would not be appropriate to simply give the left-most one precedence. Instead, we begin by attempting to synthesize a type for both subexpressions. If both synthesize a type with the same type constructor, or only one synthesizes a type, that type constructor is delegated responsibility via a class method, e.g. `syn_BinOp_Add` on line 7 of Listing 6.

[2] In Python 3.0+, syntax for annotating function definitions directly with type-like annotations was introduced, so the entire index can be synthesized.

Listing 6 Binary operations in `atlib.string_in`.

```
1 class string_in(atlang.Type):
2   def __init__(self, idx):
3     atlang.Type.__init__(self, self._rlang_of_rx(idx))
4
5   # treats string as string_in[".*"]
6   handles_Add_with = set([string])
7   @classmethod
8   def syn_BinOp_Add(cls, ctx, e):
9     rlang_left = self._rlang_from_ty(ctx.syn(e.left))
10    rlang_right = self._rlang_from_ty(ctx.syn(e.right))
11    return string_in[self._concatenate_langs(
12      rlang_left, rlang_right)]
```

Listing 7 For each type constructor definition and binary operator, `atlang` runs a modular handle set check to preclude ambiguity.

```
def check_tycon(tycon):
  for other_tycon in tycon.handles_Add_with:
    if tycon in other_tycon.handles_Add_with:
      raise atlang.AmbiguousTyconError("...",
        tycon, other_tycon) # (other binops analagous)
```

If both synthesize a type but with different type constructors (e.g. we want to concatenate a `string` and a `string_in[r]`), then we consult the *handle sets* associated as a class attribute with each type constructor, e.g. `handles_Add_with` on line 6 of Listing 6. This is a set of other type constructors that the type constructor defining the handle set may potentially support binary operations with. When a type constructor is defined, the language checks that if `tycon2` appears in `tycon1`'s handler set, then `tycon1` does *not* appear in `tycon2`'s handler set. This is a very simple modular analysis (rather than one that can only be performed at "link-time"), shown in Listing 7, that ensures that the type constructor delegated control over each binary expression is deterministic and unambiguous without arbitrarily preferring one subexpression position over another. This check is performed by using a metaclass hook (technically, this can be disabled; we assume that clients are not importing potentially adversarial extensions in this work, though lifting this assumption raises quite interesting questions that we hope will be addressed by future work).

2.2 Active Translation

Once typechecking is complete, the compiler enters the translation phase. This phase follows the same delegation protocol as the typechecking phase. Each `check_/syn_/ana_TermCon` method has a corresponding `trans_TermCon` method. These are all now instance methods, because all indices have been fully determined.

Examples of translation methods for the `fn` and `record` type constructors are shown in Listing 8. The output of translation on our example is discussed in the next subsection. Translation methods have access to the context and node, as during typechecking, and must return a translation, which for our purposes, is simply another Python syntax tree (in practice, we offer some additional conveniences beyond `ast`, such as fresh variable generation and lifting of imports and statements inside expressions to appropriate positions, in the module `astx` distributed with the language). Translation methods can assume that the term is well-typed and the typechecking phase saves the type that was delegated control, along with the type that was assigned, as attributes of each term processed by the compiler. Note that not all terms need to have been processed by the compiler if they were otherwise reinterpreted by a type constructor given control over a parent term (e.g. the field names in a record literal are never treated as expressions, while they would be for a dictionary literal).

Listing 8 Translation methods for the types defined above.

```
#class fn(atlang.Type): (cont'd)
  def trans_FunctionDef(self, ctx, node):
    x_body = [ctx.trans(stmt) for stmt in node.body]
    x_fun = astx.copy_with(node, body=x_body)
    if node.name == "__main__":
      x_invoker = ast.parse(
        'if __name__ == "__main__": __main__()')
      return ast.Suite([x_fun, x_invoker])
    else: return x_fun

  def trans_Assign_Name(self, ctx, stmt):
    return astx.copy_with(stmt,
      target=ctx.trans(stmt.target),
      value=ctx.trans(stmt.value))

  def trans_Name(self, ctx, e):
    if e.id in ctx.assn_ctx: return astx.copy(e)
    else: return self._trans_lift(
      ctx.static_env[e.id])

#class record(atlang.Type): (cont'd)
  def trans_Dict(self, ctx, e):
    if len(self.idx.fields) == 1:
      return ctx.trans(e.values[0])
    ast_dict = dict(zip(e.keys, e.values))
    return target.Tuple(ctx.trans(target, ast_dict[f])
      for f, ty in self.idx)

  def trans_Attribute(self, ctx, e):
    if len(self.idx.fields) == 1: return ctx.trans(e)
    else: return ast.Subscript(ctx.trans(e.value),
      ast.Index(ast.Num(self.idx.position_of(e.attr))))
```

Listing 9 Compiling `listing1.py` using the `@` script.

```
$ @ listing1.py
Hello, compile-time world!
Goodbye, compile-time world!
[@] _atout_listing1.py successfully generated.
$ python _atout_listing1.py
Transferring 5.50 to Annie Ace.
Transferring 15.00 to Annie Ace.
```

Listing 10 [`_atout_listing1.py`] The file generated in Listing 9.

```
1 import atlib.runtime as _at_i0_
2
3 def log_transfer(t):
4   _at_i0_.print("Transferring %s to %s." %
5     (_at_i0_.dec_to_str(t[0][0], 2), t[1][0]))
6
7 def __main__():
8   account = ("Annie Ace", "00-00000001")
9   log_transfer((((5, 50), None), account))
10  log_transfer((((15, 0), "Rent payment"), account))
11 if __name__ == "__main__": __main__()
```

2.3 Standalone Compilation

Listing 9 shows how to invoke the `@` compiler at the shell to typecheck and translate then execute `listing1.py`. The `@lang` compiler is itself a Python library and `@` is a simple Python script that invokes it in two steps:

1. It evaluates the compilation script to completion.
2. For any top-level bindings in the environment that are `@lang` functions, it initiates typehecking and translation as described above. If no type errors are discovered, the ordered set of translations are collected (obeying order dependencies) and emitted. If a type error is discovered, an error is displayed.

In our example, there are no type errors, so the file shown in Listing 10 is generated. This file is meant only to be executed, not edited or imported directly. The invariants necessary to ensure that execution does not "go wrong", assuming the extensions were implemented correctly, were checked statically and entities having no bearing on execution, like field names and types themselves, were erased. The dynamic semantics of the type constructors used in the program were implemented by translation to Python:

1. `fn` recognized the function name `__main__` as special, inserting the standard Python code to invoke it if the file is run directly.
2. Records translated to tuples (the field names were not needed).
3. Decimals translated to pairs of integers. String conversion is defined in a "runtime" package with a "fresh" name.
4. Terms of type `string_in[r"..."]` translated to strings. Checks could here be performed statically.
5. Prototypic objects translated to pairs consisting of the fore and the prototype. Dispatch to the appropriate record based on the field name was static (line 5).

Listing 11 [`listing11.py`] Lines 7-11 each have a type error.

```
1 from listing1 import fn, dyn, Account, Details,
2   log_transfer
3 import datetime
4 @fn[dyn, dyn]
5 def pay_oopsie(memo):
6   if datetime.date.today().day == 1: # @lang to Python
7     account = {nome: "Oopsie Daisy",
8       account_num: "0-00000000"} [:Account] # (format)
9     details = {amount: None, memo: memo} [:Details]
10    details.amount = 10 # (immutable)
11    log_transfer((account, details)) # (backwards)
12 print "Today is day " + str(datetime.date.today())
13 pay_oopsie("Hope this works..") # Python to @lang
14 print "All done."
```

Listing 12 Execution never proceeds into a function with a type error when using `@lang` for implicit compilation.

```
$ python listing11.py
Today is day 2
Traceback (most recent call last):
  File "listing11", line 9, in <module>
atlang.TypeError:
    File "listing11.py", line 7, col 12, in <module>:
      [record] Invalid field name: nome
```

Type Errors Listing 11 shows an example of code containing several type errors. If analagous code were written in Python itself, these could only be found if the code was executed on the first day of the month (and not all of the issues would immediately be caught as run-time exceptions). In `@lang`, these can be found without executing the code (i.e. true static typechecking).

2.4 Interactive Invocation

`@lang` functions over values of type `dyn` can be invoked directly from Python. Typechecking and translation occurs upon first invocation (subsequent calls are cached; we assume that the static environment is immutable). We see this occurring when we execute the code in Listing 11 using the Python interpreter in Listing 12.

In future work, we plan to detail how to insert dynamic checks and wrapper objects, defined by active type constructors in a manner similar to how static checks are defined here, so that values that are not of type `dyn` can be passed in and out of untyped Python code. Because these can be seen as implicit coercions to and from `dyn`, and we do not aim to introduce this feature in this paper, we omit discussion of this topic. Explicit coercions can be implemented using the mechanisms described above. For example, `string_in` provides an introductory form that checks a provided `string` or `dyn` value dynamically, raising an exception in the case of failure: `[raw_input()] : string_in[r"\d+"]`.

3. More Examples

In the previous section, we showed examples of several interesting type system fragments implemented as libraries using `atlang`, including functional record types, a prototypic object system and regular string types. A more detailed description of regular string types and their implementation using `atlang` was recently published [7]. Here, we showcase two more powerful examples that demonstrate the expressive power of system: functional datatypes with nested pattern matching, and a type safe foreign function interface to the complete OpenCL language for many-core programming on devices like GPUs.

3.1 Functional Datatypes and Nested Pattern Matching

A powerful feature of modern functional languages like ML is support for nested pattern matching over datatypes (i.e. recursive labeled disjoint sums) and tuples. For example, tree structures are well-modeled in this way, as we show in Listing 13. Here, `Tree` is a *case type*, which is what we call functional datatypes in `atlib`. The type constructor `Tree` is declared as a named type using the `@ty` annotation. This is a general mechanism in `atlang` for supporting recursive types that behave generatively, i.e. that are identified by a name, rather than a structure. Functional datatypes have the same character. The `@ty` annotation also supports parameterized types. Here, `a` must be another type. The type index is a series of cases with, optionally, corresponding types. Here, a (binary) tree can either be empty (no associated data), a leaf with an associated value of type `a`, or an internal node, which takes a pair of trees as data. We use the `tuple` type constructor to represent pairs.

The function `treesum` on lines 11-17 takes a tree containing `dyn` values and computes the sum of these values. The case operator is defined by the `casetype` type constructor, leveraging a combination of the `Attribute` and the `Subscript` term constructors (both are targeted operations, see above). The slice syntax is again re-interpretted, here s a series of rules, each of which starts with a pattern, followed by a colon, then a case. Variables in patterns can be bound in the case. Patterns can consist of case names and tuple patterns at arbitrary depth. More generally, we have done initial designs for an extensible pattern language, again delegating to type constructors to determine whether the pattern is valid and which variables it binds, but here we restrict ourselves to these two type constructors.

By combining the functional behavior of the `fn` type constructor, which dispenses with "return" statements, and creatively repurposing existing syntax, we are thus able to implement an essentially idiomatic statically-typed functional language, entirely as a library for Python. The constructs we defined can be composed arbitrarily with, e.g. the record types or regular string types in the previous section, without any concern regarding syntactic or semantic conflict, because `casetype` only controls the semantics of the `case` operation on values that are of such a type.

3.2 A Low-Level Foreign Function Interface to OpenCL

Python is a common language in scientific computing and data analysis domains. The performance of large-scale analyses can often be a bottleneck, so programmers often wish to write and interact with code written in a low-level language. OpenCL is designed precisely for this workflow, exposing a C-based language that can compile to a variety of specialized hardware (e.g. GPUs) via a standard API. The `pyopencl` package exposes this API to Python code and integrates it with the popular `numpy`numeric array package [11]. To compile an OpenCL function, however, users must write it as a string. This is neither idiomatic nor safe, because it defers typechecking to invocation-time.

In Listing 14, we show usage of the `atlib.opencl` package, which implements the entirety of the OpenCL language (which includes essentially the entirety of C99, plus some extensions to work with multiple memory spaces). Although the details are beyond the scope of this paper, we note that the mapping onto Python syntax was straightforward, particularly given the flexibility of analytic numeric literal forms, as described above.

After initializing a device, line 4 sends a `numpy` array to the device, assigning `buffer` its handle. Line 5 extracts its type, which is based on the dynamic element type of the `numpy`

Listing 13 An example of case types and nested pattern matching.

```
1  from atlang import ty
2  from atlib import casetype, tuple, fn
3
4  @ty
5  def Tree(a): casetype[
6    "Empty",
7    "Leaf" : a,
8    "Node": tuple[Tree(a), Tree(a)]
9  ]
10
11 @fn
12 def treesum(tree : Tree(dyn)) -> dyn:
13   tree.case[
14     Empty: 0,
15     Leaf(x): x,
16     Node((l, r)): treesum(l) + treesum(r)
17   ]
```

Listing 14 An example of a type system for OpenCL.

```
1  import atlib.opencl as opencl
2
3  dev = opencl.init_device(0)
4  buffer = dev.send(numpy.zeros((80000,)))
5  buffer_t = buffer.ty # == opencl.global_ptr[elem_t]
6  elem_t = buffer_t.elem_t # == opencl.double
7
8  @opencl.fn
9  def map(x : buffer_t, f : opencl.fn[elem_t, elem_t]):
10   gid = get_global_id(0)
11   x[gid] = f(x[gid])
12
13 @opencl.fn
14 def mad(x : elem_t, m : elem_t, y : elem_t):
15   return m*x + y
16 @opencl.fn
17 def my_mad(x : elem_t): return mad(x, 10, 5)
18 @opencl.fn
19 def my_mad2(x : elem_t): return mad(x, 20, 5)
20
21 map(buffer, my_mad)
22 map(buffer, my_mad2)
```

Listing 15 The underlying code generated by atlib.opencl.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

double mad(double x, double m, double y) {
  return m*x + y;
}

double my_mad(double x) {
  return mad(x, 10.0, 5.0);
}

double my_mad2(double x) {
  return mad(x, 20.0, 5.0);
}

kernel void map__1_(double x) {
  size_t gid = get_global_id(0);
  x[gid] = my_mad(x[gid]);
}

kernel void map__2_(double x) {
  size_t gid = get_global_id(0);
  x[gid] = my_mad2(x[gid]);
}
```

$$
\begin{array}{llll}
\text{programs} & \rho & ::= & \mathsf{import}[\Phi](e) \\
\text{fragments} & \Phi & ::= & \emptyset \mid \Phi, \textsc{tycon} = \{\delta\} \\
\text{tycon defs} & \delta & ::= & \mathsf{analit} = \sigma;\ \mathsf{synidxlit} = \sigma; \\
 & & & \mathsf{anatarg} = \sigma;\ \mathsf{syntarg} = \sigma \\
\text{expressions} & e & ::= & x \mid \mathsf{let}(e; x.e) \mid \mathsf{slet}[\sigma](\mathbf{x}.e) \mid \mathsf{asc}[\sigma](e) \\
 & & \mid & \lambda(x.e) \mid \mathsf{lit}[\sigma](\overline{e}) \mid \mathsf{targ}[\sigma](e; \overline{e}) \\
\text{static terms} & \sigma & ::= & \mathbf{x} \mid \lambda(\mathbf{x}.\sigma) \mid \mathsf{ap}(\sigma; \sigma) \mid \mathsf{fail} \\
 & & \mid & \mathsf{ty}[\textsc{tycon}](\sigma) \mid \mathsf{tycase}[\textsc{tycon}](\sigma; \mathbf{x}.\sigma; \sigma) \\
 & & \mid & \mathsf{incty}[\textsc{tycon}](\sigma) \\
 & & \mid & \mathsf{lbl}[\mathtt{lbl}] \mid \mathsf{lbleq}(\sigma; \sigma; \sigma; \sigma) \\
 & & \mid & \mathsf{nil} \mid \mathsf{cons}(\sigma; \sigma) \mid \mathsf{listrec}(\sigma; \sigma; \mathbf{x}.\mathbf{y}.\sigma) \\
 & & \mid & \mathsf{arg}[e] \mid \mathsf{ana}(\sigma; \sigma; \mathbf{x}.\sigma) \mid \mathsf{syn}(\sigma; \mathbf{x}.\mathbf{y}.\sigma) \\
 & & \mid & \rhd(\iota) \\
\text{internal terms} & \iota & ::= & \lhd(\sigma) \mid x \mid \lambda(x.\iota) \mid \mathsf{iap}(\iota; \iota) \\
 & & \mid & \mathsf{inil} \mid \mathsf{icons}(\iota; \iota) \mid \mathsf{ilistrec}(\iota; \iota; x.y.\iota)
\end{array}
$$

Figure 2. Abstract syntax of $@\lambda$. Metavariable TYCON ranges over type constructor names (assumed globally unique), `lbl` over static labels, x, y over expression variables and $\mathbf{x}, \mathbf{y}$ over static variables. We indicate that variables or static variables are bound within a term or static term by separating them with a dot, e.g. $\mathbf{x}.\mathbf{y}.e$, and abbreviate a sequence of zero or more expressions as $\overline{e}$.

array. The type constructor `global_ptr` defines the semantics of OpenCL's pointers to global memory (e.g. subscripting, pointer arithmetic, pointer differencing), consistent with the C99 specification. Because the compilation script is programmatic, we can extract its element type programmatically as well. Here, it is the `opencl.double` type, which supports arithmetic operations, including the full array of widening rules defined by C99 when paired with any of the other numeric OpenCL types (using handle sets, above, to prioritize control to the wider type).

Lines 8-19 define several OpenCL functions using Python syntax, but retaining OpenCL's semantics. The `map` function takes a buffer argument and a function over its elements and applies the function to each element in the buffer in a data parallel manner. Each thread handles the element corresponding to its thread ID, called a `group ID` in OpenCL and accessed by a primitive on the first line. Note that OpenCL does not itself have support for higher-order functions, however. Instead, higher order functions are essentially an extension to the foreign language being encoded using active type constructors. The OpenCL function decorator treats uses of higher order functions as if each function ever provided to it had a *singleton type*, i.e. a type inhabited only by that particular function. Each call lazily generates code for a specialized variant of that function. The result can be seen in Listing 15, which shows the OpenCL code generated (as a string sent through `pyopencl`) by this example.

Though we do not show any other extensions atop the OpenCL library here, it is straightforward to implement variants of those described above that target their translation phase to OpenCL rather than Python. In many cases, the typechecking code can be inherited directly. This is, we argue, rather compelling: a decidedly low-level language like OpenCL can be extended with low-overhead versions of sophisticated features, like a prototypic object system, modularly, via its foreign function interface from a scripting language like Python.

4. $@\lambda$: Active Type Constructors, Minimally

We now turn our attention to a type theoretic formulation of the key mechanisms described above atop a minimal abstract syntax, shown in Fig. 2. This syntax supports a *purely syntactic desugaring* from a conventional concrete syntax, shown by example in Fig. 1.

Fragment Client Perspective A program, ρ, consists of a series of fragment imports, Φ, defining active type constructors for use in an expression, e. Expression forms can be indexed by static terms, σ. The abstract syntax of e provides let binding of variables and for convenience, static values can also be bound to a static variable, $\mathbf{x}$ (distinguished in bold for clarity), using slet. Static terms have a *static dynamics* and evaluate to *static values* or fail.

Types and Ascriptions An expression can be ascribed a type or an incomplete type, both static values constructed, as in the introduction, by naming an imported type constructor, TYCON, and providing a type index, another static value. The static language also includes lists and atomic *labels* for use in compound indices.

Literal Desugaring All concrete literals (other than lambda expressions, which are built in) desugar to an abstract term of the form $\mathsf{lit}[\sigma](\overline{e})$, where σ captures all static portions of the literal (e.g. a list of the labels used as field names in the labeled product literal in Fig. 1, or the numeric label used for the natural number zero) and $\overline{e}$ is a list of sub-expressions (e.g. the field values).

Targeted Expression Desugaring All non-introductory operations go through a targeted expression form (e.g. $e(e)$, or $e \cdot \mathtt{lbl}$, or $e \cdot \mathtt{lbl}(\overline{e})$; see previous section), which all desugar to an abstract term of the form $\mathtt{targ}[\sigma](e; \overline{e})$ where σ again captures all static portions of the term (e.g. the label naming an operation, e.g. `s` or `rec` on natural numbers [8]), e is the target (e.g. the natural number being operated on, the function being called, or the record we wish to project out of) and $\overline{e}$ are all other arguments (e.g. the base and recursive cases of the recursor, or the argument being applied).

Bidirectional Active Typechecking and Translation The static semantics are specified by the *bidirectional active typechecking and translation judgements* shown in Fig. 3, which relate an expression, e, to a type, σ, and a *translation*, ι, under *typing context* Γ using imported fragments Φ. The judgement form $\Gamma \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota$ specifies *type synthesis* (the type is an "output"), whereas $\Gamma \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota$ specifies *type analysis* (the type is an "input"). This can be seen as combining a bidirectional type system (in the style of Pierce and Turner [17] and a number of subsequent formalisms and languages, e.g. Scala) with an elaboration semantics in the style of the Harper-Stone semantics for Standard ML [9]. Our language of *internal terms*, ι, includes only functions and lists for simplicity. The form $\lhd(\sigma)$ is used as an "unquote" operator, and will appear only in intermediate portions of a typing derivation, not in a translation (discussed below).

The first two rows of rules in Fig. 3 are essentially standard. **ATT-SUBSUME** specifies the subsumption principle described in the previous section: if a type can be synthesized, then the term can also be analyzed against that type. We decide type equality purely syntactically here. **ATT-VAR** specifies that variables always synthesize types and elaborate identically. The typing context, Γ, maps variables to types in essentially the conventional way [8]. The rules **ATT-ANA-LET** and **ATT-SYN-LET** first synthesize a type for the bound value, then add this binding to the context and analyze or synthesize the body of the binding. The translation is to an internal function application, in the conventional manner. **ATT-ASC** begins by normalizing the provided index and checking that it is a type (Fig. 7, top). If so, it analyzes the ascribed expression against that type. The rules **ATT-ANA-SLET** and **ATT-SYN-SLET** eagerly evaluate the provided static term to a static value, then immediately perform the substitution (demonstrating the phase separation) in the expression before analysis or synthesis proceeds.

Lambdas The rule **ATT-ANA-LAM** performs type analysis on a lambda abstraction, $\lambda(x.e)$. If it succeeds, the translation is the corresponding lambda in the internal language. The type constructor FN is included implicitly in Φ and must have a type index

import $\Phi_{\text{nat}}, \Phi_{\text{lprod}}$	import$[\Phi_{\text{nat}}, \Phi_{\text{lprod}}]($
static let **nat** $=$ NAT[nil]	slet[ty[NAT](nil)](**nat**.
let *zero* $= \mathbb{0}$: **nat**	let(asc[**nat**](lit[lbl[$\mathbb{0}$]]($\cdot$)); *zero*.
let *one* $=$ *zero*.s	let(targ[lbl[s]](*zero*; $\cdot$); *one*.
let *plus* $= (\lambda x{:}\textbf{nat}.\lambda y{:}\textbf{nat}.$	let(asc[incty[FN](**nat**)](λ(x.asc[incty[FN](**nat**)](λ(y.
$\quad x \cdot \mathtt{rec}(y; \lambda p.\lambda r.r \cdot \mathtt{s}))$	$\quad$targ[lbl[rec]](x; λ(p.λ(r.targ[lbl[s]](r; $\cdot$)))))))); *plus*.
let *two* $=$ *plus one one*	let(targ[nil](targ[nil](*plus*; *one*); *one*); *two*.
{one=*one*; two=*two*} : LPROD[$\cdots$]	asc[incty[LPROD](nil)](lit[cons(lbl[one]; cons(lbl[two]; nil))](*one*; *two*))))))

Figure 1. A program written using conventional concrete syntax, left, syntactically desugared to the abstract syntax on the right.

$$\boxed{\Gamma \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota} \quad \boxed{\Gamma \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota} \quad \Gamma ::= \emptyset \mid \Gamma, x \Rightarrow \sigma$$

ATT-SUBSUME
$$\frac{\Gamma \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota}{\Gamma \vdash_\Phi e \Leftarrow \sigma \rightsquigarrow \iota}$$

ATT-VAR
$$\frac{x \Rightarrow \sigma \in \Gamma}{\Gamma \vdash_\Phi x \Rightarrow \sigma \rightsquigarrow x}$$

ATT-ANA-LET
$$\frac{\Gamma \vdash_\Phi e_1 \Rightarrow \sigma_1 \rightsquigarrow \iota_1 \qquad \Gamma, x \Rightarrow \sigma_1 \vdash_\Phi e_2 \Leftarrow \sigma_2 \rightsquigarrow \iota_2}{\Gamma \vdash_\Phi \mathsf{let}(e_1; x.e_2) \Leftarrow \sigma_2 \rightsquigarrow \mathsf{iap}(\lambda(x.\iota_2); \iota_1)}$$

ATT-SYN-LET
$$\frac{\Gamma \vdash_\Phi e_1 \Rightarrow \sigma_1 \rightsquigarrow \iota_1 \qquad \Gamma, x \Rightarrow \sigma_1 \vdash_\Phi e_2 \Rightarrow \sigma_2 \rightsquigarrow \iota_2}{\Gamma \vdash_\Phi \mathsf{let}(e_1; x.e_2) \Rightarrow \sigma_2 \rightsquigarrow \mathsf{iap}(\lambda(x.\iota_2); \iota_1)}$$

ATT-SYN-ASC
$$\frac{\sigma \Downarrow_{\emptyset;\emptyset} \sigma' \qquad \sigma'\ \mathtt{ty}_\Phi \qquad \Gamma \vdash_\Phi e \Leftarrow \sigma' \rightsquigarrow \iota}{\Gamma \vdash_\Phi \mathsf{asc}[\sigma](e) \Rightarrow \sigma' \rightsquigarrow \iota}$$

ATT-ANA-SLET
$$\frac{\sigma_1 \Downarrow_{\emptyset;\emptyset} \sigma_1' \qquad \Gamma \vdash_\Phi [\sigma_1'/\mathbf{x}]e \Leftarrow \sigma_2 \rightsquigarrow \iota}{\Gamma \vdash_\Phi \mathsf{slet}[\sigma_1](\mathbf{x}.e) \Leftarrow \sigma_2 \rightsquigarrow \iota}$$

ATT-SYN-SLET
$$\frac{\sigma_1 \Downarrow_{\emptyset;\emptyset} \sigma_1' \qquad \Gamma \vdash_\Phi [\sigma_1'/\mathbf{x}]e \Rightarrow \sigma_2 \rightsquigarrow \iota}{\Gamma \vdash_\Phi \mathsf{slet}[\sigma_1](\mathbf{x}.e) \Rightarrow \sigma_2 \rightsquigarrow \iota}$$

ATT-ANA-LAM
$$\frac{\Phi_{\text{fn}} \subset \Phi \qquad \sigma_1\ \mathtt{ty}_\Phi \qquad \sigma_2\ \mathtt{ty}_\Phi \qquad \Gamma, x \Rightarrow \sigma_1 \vdash_\Phi e \Leftarrow \sigma_2 \rightsquigarrow \iota}{\Gamma \vdash_\Phi \lambda(x.e) \Leftarrow \mathsf{ty}[\text{FN}](\mathsf{cons}(\sigma_1; \mathsf{cons}(\sigma_2; \mathsf{nil}))) \rightsquigarrow \lambda(x.\iota)}$$

ATT-SYN-IDX-LAM
$$\frac{\Phi_{\text{fn}} \subset \Phi \qquad \sigma_1\ \mathtt{ty}_\Phi \qquad \Gamma, x \Rightarrow \sigma_1 \vdash_\Phi e \Rightarrow \sigma_2 \rightsquigarrow \iota}{\Gamma \vdash_\Phi \mathsf{asc}[\mathsf{incty}[\text{FN}](\sigma_1)](\lambda(x.e)) \Rightarrow \mathsf{ty}[\text{FN}](\mathsf{cons}(\sigma_1; \mathsf{cons}(\sigma_2; \mathsf{nil}))) \rightsquigarrow \lambda(x.\iota)}$$

ATT-ANA-LIT
$$\frac{\vdash_\Phi \mathsf{analit}(\text{TYCON}) = \sigma_{\text{def}} \qquad \mathtt{args}(\overline{e}) = \sigma_{\text{args}} \qquad \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \Downarrow_{\Gamma;\Phi} \rhd(\iota)}{\Gamma \vdash_\Phi \mathsf{lit}[\sigma_{\text{tmidx}}](\overline{e}) \Leftarrow \mathsf{ty}[\text{TYCON}](\sigma_{\text{tyidx}}) \rightsquigarrow \iota}$$

ATT-SYN-IDX-LIT
$$\frac{\vdash_\Phi \mathsf{synidxlit}(\text{TYCON}) = \sigma_{\text{def}} \qquad \mathtt{args}(\overline{e}) = \sigma_{\text{args}} \qquad \sigma_{\text{def}}\ \sigma_{\text{incidx}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \Downarrow_{\Gamma;\Phi} \mathsf{cons}(\sigma_{\text{tyidx}}; \mathsf{cons}(\rhd(\iota); \mathsf{nil}))}{\Gamma \vdash_\Phi \mathsf{asc}[\mathsf{incty}[\text{TYCON}](\sigma_{\text{incidx}})](\mathsf{lit}[\sigma_{\text{tmidx}}](\overline{e})) \Rightarrow \mathsf{ty}[\text{TYCON}](\sigma_{\text{tyidx}}) \rightsquigarrow \iota}$$

ATT-ANA-TARG
$$\frac{\Gamma \vdash_\Phi e_{\text{targ}} \Rightarrow \mathsf{ty}[\text{TYCON}](\sigma_{\text{tyidx}}) \rightsquigarrow \iota_{\text{targ}} \qquad \vdash_\Phi \mathsf{anatarg}(\text{TYCON}) = \sigma_{\text{def}} \qquad \mathtt{args}(\overline{e}) = \sigma_{\text{args}} \qquad \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ \rhd(\iota_{\text{targ}})\ \sigma_{\text{ty}}\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \Downarrow_{\Gamma;\Phi} \rhd(\iota)}{\Gamma \vdash_\Phi \mathsf{targ}[\sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e}) \Leftarrow \sigma_{\text{ty}} \rightsquigarrow \iota}$$

ATT-SYN-TARG
$$\frac{\Gamma \vdash_\Phi e_{\text{targ}} \Rightarrow \mathsf{ty}[\text{TYCON}](\sigma_{\text{tyidx}}) \rightsquigarrow \iota_{\text{targ}} \qquad \vdash_\Phi \mathsf{syntarg}(\text{TYCON}) = \sigma_{\text{def}} \qquad \mathtt{args}(\overline{e}) = \sigma_{\text{args}} \qquad \sigma_{\text{def}}\ \sigma_{\text{tyidx}}\ \rhd(\iota_{\text{targ}})\ \sigma_{\text{tmidx}}\ \sigma_{\text{args}} \Downarrow_{\Gamma;\Phi} \mathsf{cons}(\sigma_{\text{ty}}; \mathsf{cons}(\rhd(\iota); \mathsf{nil})) \qquad \sigma_{\text{ty}}\ \mathtt{ty}_\Phi}{\Gamma \vdash_\Phi \mathsf{targ}[\sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e}) \Rightarrow \sigma_{\text{ty}} \rightsquigarrow \iota'}$$

Figure 3. Bidirectional active typechecking and translation. For concision, we use standard functional notation for static function application.

Φ_{fn} := FN = {analit = nil; synidxlit = nil; anatarg = nil;
$\quad$syntarg = λ**tyidx**.λ**ifn**.λ**tmidx**.λ**args**.
$\qquad$**isnil tmidx** (**decons1 args** λ**arg**.**decons2 tyidx** λ**inty**.λ**rty**.
$\qquad\quad$ana(**arg**; **inty**; **ia**.**pair rty** $\rhd$(iap($\lhd$(**ifn**); $\lhd$(**ia**)))))}

Figure 4. The FN fragment defines function application.

consisting of a pair of types (pairs are here just lists of length 2 for simplicity). Because both the argument type and return type are known, the body of the lambda is analyzed against the return type after extending the context with the argument type. This is thus the usual type analysis rule for functions in a bidirectional setting.

The rule **ATT-SYN-IDX-LAM** covers the case where a lambda abstraction has an incomplete type ascription providing only the argument type. This corresponds to the concrete syntax seen in the definition of *plus* in Fig. 1. Here, the return type must be synthesized by the body.

These two rules can be compared to the rules in Listing 2. The main difference is that in @λ, the language itself manages variables and contexts, rather than the type constructor. This is largely for simplicity, though it does limit us in that we cannot define function type constructors that require alternative or additional contexts. Addressing this in the theory is one avenue for future work.

Function application can be defined directly as a targeted expression, as seen in the example, which we will discuss below.

Fragment Provider Perspective Fragment providers define type constructors by defining "methods", δ, that control analysis and synthesis literals and targeted expressions. These are static functions invoked by the final four rules in Fig. 3, which we will describe next. The type constructors FN, NAT and LPROD are shown in Figs. 4-6, respectively. They use helper functions for working with labels (**lbleq**), lists (**isnil**, **decons1**, **decons2**, **decons3**, **zipexact2**, **zipexact3**, and **pair**), lists of pairs interpreted as finite mappings (**lookup** and **posof**) and translating a static representation of a number to an internal representation of that number (**itermofn**). We also assume an internal function *nth* that retrieves the *n*th element of a list. All of these are standard or straightforward and omitted for concision. Failure cases for the static helper functions evaluate to fail. Derivation of the typing judgement does not continue if a fail occurs (corresponding to an `atlang.TypeError` propagated directly to the compiler).

Φ_{nat} := NAT = {analit = λ**tyidx**.λ**tmidx**.λ**args**.
$\qquad$**isnil tyidx** (**lbleq tmidx lbl**[$\mathbb{0}$] (**isnil args** $\rhd$(**inil**)));
$\quad$synidxlit = λ**incidx**.λ**tmidx**.λ**args**.
$\qquad$**isnil incidx** (**lbleq tmidx lbl**[$\mathbb{0}$] (**isnil args** (**pair** nil $\rhd$(**inil**))));
$\quad$anatarg = λ**tyidx**.λ**i1**.λ**ty**.λ**tmidx**.λ**args**.
$\qquad$**lbleq tmidx** lbl[rec] (**decons2 args** λ**arg1**.λ**arg2**.
$\qquad\quad$ana(**arg1**; **ty**; **i2**.ana(**arg2**; **fn2ty** ty[NAT](nil) **ty ty**; **i3**.
$\qquad\qquad\rhd$(ilistrec($\lhd$(**i1**); $\lhd$(**i2**); x, y.iap(iap($\lhd$(**i3**); x); y))))));
$\quad$syntarg = λ**tyidx**.λ**i1**.λ**tmidx**.λ**args**.
$\qquad$**lbleq tmidx** lbl[s] (**isnil args** (
$\qquad\quad$**pair** ty[NAT](**nil**) $\rhd$(icons(inil; $\lhd$(**i1**)))))}

Figure 5. The NAT fragment, based on Gödel's **T** [8].

Literals The rule **ATT-ANA-LIT**, invokes the analit method of the type constructor of the type the literal is being analyzed against, asking it to return a translation (a value of the form $\rhd(\iota)$). The type

$$
\begin{array}{l}
\Phi_{\text{lprod}} := \text{LPROD} = \{\mathsf{analit} = \lambda\mathbf{tyidx}.\lambda\mathbf{tmidx}.\lambda\mathbf{args}.\\
\quad \mathsf{listrec}(\mathbf{zipexact3\ tyidx\ tmidx\ args}; \rhd(\mathsf{inil}); \mathbf{h}.\mathbf{ri}.\\
\qquad \mathbf{decons3\ h}\ \lambda\mathbf{idxitem}.\lambda\mathbf{lbl}.\lambda\mathsf{e}.\mathbf{decons2\ idxitem}\ \lambda\mathbf{lblidx}.\lambda\mathbf{tyidx}.\\
\qquad\quad \mathbf{lbleq\ lbl\ lblidx}\ (\mathsf{ana}(\mathbf{e}; \mathbf{tyidx}; \mathbf{i}.\rhd(\mathsf{icons}(\lhd(\mathbf{i}), \lhd(\mathbf{ri}))))));\\
\ \ \mathsf{synidxlit} = \lambda\mathbf{incidx}.\lambda\mathbf{tmidx}.\lambda\mathbf{args}.\\
\quad \mathsf{listrec}(\mathbf{zipexact2\ tmidx\ args}; \mathbf{pair}\ \mathsf{nil}\ \rhd(\mathsf{inil}); \mathbf{h}.\mathbf{r}.\\
\qquad \mathbf{decons2\ h}\ \lambda\mathbf{lbl}.\lambda\mathsf{e}.\mathbf{decons2\ h}\ \lambda\mathbf{ridx}.\lambda\mathbf{ri}.\mathsf{syn}(\mathbf{e}; \mathbf{ty}.\mathbf{i}.\\
\qquad\quad \mathbf{pair}\ \mathsf{cons}(\mathbf{pair\ lbl\ ty}; \mathbf{ridx})\ \rhd(\mathsf{icons}(\lhd(\mathbf{i}); \lhd(\mathbf{ri})))));\\
\ \ \mathsf{anatarg} = \mathsf{nil}; \text{(destructuring let could be implemented here)}\\
\ \ \mathsf{syntarg} = \lambda\mathbf{tyidx}.\lambda\mathbf{i}.\lambda\mathbf{lbl}.\lambda\mathbf{args}.\\
\quad \mathbf{isnil\ args\ (pair\ (lookup\ lbl\ tyidx)}\\
\qquad \rhd(\mathsf{iap}(\mathsf{iap}(\mathit{nth}; \lhd(\mathbf{i})); \lhd(\mathbf{itermofn\ (posof\ lbl\ tyidx)})))))
\end{array}
$$

Figure 6. The LPROD fragment (labeled products are like records, but the field order matters; cf. Listing 4).

$$\boxed{\sigma\ \mathtt{ty}_\Phi} \qquad \text{TY}\ \frac{\mathsf{ty}[\text{TYCON}](\sigma) \Downarrow_{\emptyset;\Phi} \mathsf{ty}[\text{TYCON}](\sigma)}{\mathsf{ty}[\text{TYCON}](\sigma)\ \mathtt{ty}_\Phi}$$

$$\boxed{\sigma \Downarrow_{\Gamma;\Phi} \sigma} \qquad \text{N-TY}\ \frac{\text{TYCON} \in \mathrm{dom}(\Phi) \qquad \sigma \Downarrow_{\Gamma;\Phi} \sigma'}{\mathsf{ty}[\text{TYCON}](\sigma) \Downarrow_{\Gamma;\Phi} \mathsf{ty}[\text{TYCON}](\sigma')}$$

$$\text{N-ARG}\ \frac{}{\mathsf{arg}[e] \Downarrow_{\Gamma;\Phi} \mathsf{arg}[e]} \qquad \text{N-ANA}\ \frac{\begin{array}{c}\sigma_1 \Downarrow_{\Gamma;\Phi} \mathsf{arg}[e] \qquad \sigma_2 \Downarrow_{\Gamma;\Phi} \sigma_{\text{ty}} \qquad \sigma_{\text{ty}}\ \mathtt{ty}_\Phi\\ \Gamma \vdash_\Phi e \Leftarrow \sigma_{\text{ty}} \rightsquigarrow \iota \qquad [\rhd(\iota)/\mathbf{x}]\sigma_3 \Downarrow_{\Gamma;\Phi} \sigma_3'\end{array}}{\mathsf{ana}(\sigma_1; \sigma_2; \mathbf{x}.\sigma_3) \Downarrow_{\Gamma;\Phi} \sigma_3'}$$

$$\text{N-SYN}\ \frac{\begin{array}{c}\sigma_1 \Downarrow_{\Gamma;\Phi} \mathsf{arg}[e] \qquad \Gamma \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota\\ [\sigma/\mathbf{x}_1, \rhd(\iota)/\mathbf{x}_2]\sigma_2 \Downarrow_{\Gamma;\Phi} \sigma_2'\end{array}}{\mathsf{syn}(\sigma_1; \mathbf{x}_1.\mathbf{x}_2.\sigma_2) \Downarrow_{\Gamma;\Phi} \sigma_2'} \qquad \text{N-QUOTE}\ \frac{\iota \curlyvee_{\Gamma;\Phi} \iota'}{\rhd(\iota) \Downarrow_{\Gamma;\Phi} \rhd(\iota')}$$

$$\boxed{\iota \curlyvee_{\Gamma;\Phi} \iota} \qquad \text{Q-UQ}\ \frac{\sigma \Downarrow_{\Gamma;\Phi} \rhd(\iota)}{\lhd(\sigma) \curlyvee_{\Gamma;\Phi} \iota} \qquad \text{Q-X}\ \frac{}{x \curlyvee_{\Gamma;\Phi} x} \qquad \text{Q-LAM}\ \frac{\iota \curlyvee_{\Gamma;\Phi} \iota'}{\lambda(x.\iota) \curlyvee_{\Gamma;\Phi} \lambda(x.\iota')}$$

Figure 7. Selected normalization rules for the static language.

and term index are provided, as well as a list of *reified arguments*: static values of the form $\mathsf{arg}[e]$. The FN type constructor does not implement this (functions are introduced only via lambdas). The NAT type constructor implements this by checking that the term index was the label corresponding to `0` and no arguments were provided. The LPROD type constructor is more interesting: it folds over each corresponding item in the type index (a pair consisting of a label and a type), the term index (a label) and the argument list, checking that the labels match and programmatically analyzing the argument against the type it should have. A reified argument σ_1 against a type σ_2, binding the translation to $\mathbf{x}$ in σ_3 if successful (and failing otherwise) with the static term $\mathsf{ana}(\sigma_1; \sigma_2; \mathbf{x}.\sigma_3)$, the semantics of which are in Fig. 7. Labeled products translate to lists by recursively composing the translations of the field values using the "unquote" form, $\lhd(\sigma)$, which is eliminated during normalization (also seen in Fig. 7). This function can be compared to the method `ana_Dict` in Listing 4.

Literals with an incomplete type ascription can synthesize a type by rule **ATT-SYN-IDX-LIT**. The synidxlit method of type constructor of the partial ascription is called with the incomplete type index, the term index and the arguments as above, and must return a pair consisting of the complete type index and the translation. Again, FN does not implement this. The NAT type constructor supports it, though it is not particularly interesting, as NAT is always indexed trivially, so it follows essentially the same logic as in analit. The LPROD type constructor is more interesting: in this case, when the incomplete type index is trivial (as in the example in Fig. 1), the list of pairs of labels and types must be synthesized from the literal itself. This is done by programatically synthesizing a type and translation for a reified argument using $\mathsf{syn}(\sigma_1; \mathbf{x}.\mathbf{y}.\sigma_2)$, also specified in Fig. 7. The type index and translation are recursively formed. This can be compared to the class method `syn_idx_Dict` in Listing 4.

Targeted Terms Targeted terms are written $\mathsf{targ}[\sigma_{\text{tmidx}}](e_{\text{targ}}; \overline{e})$. The type constructor of the type recursively synthesized by the *target*, e_{targ}, is delegated control over analysis and synthesis via the methods anatarg and syntarg, respectively, as seen in rules **ATT-ANA-TARG** and **ATT-SYN-TARG**. Both receive the type index, the translation of the target, the term index and the reified arguments. The former also receives the type being analyzed and only needs to produce a translation. The latter must produce a pair consisting of a type and a translation.

The FN type constructor defines function application by straightforwardly implementing syntarg. Because of subsumption, anatarg need not be separately defined.

The NAT type constructor defines the successor operation synthetically and the recursor operation analytically (because it has two branches that must have the same type). The latter analyzes the second argument against a function type, avoiding the need to handle binding itself. Natural numbers translate to lists, so the nat recursor can be implemented straightforwardly using the list recursor. In a practical implementation, we might translate natural numbers to integers and use a fixpoint computation instead.

The LPROD type constructor defines the projection operation synthetically, using the helper functions mentioned above to lookup the appropriate item in the type index. Note that one might also define an analytic targeted operation on labeled products corresponding to pattern matching, e.g. let $\{\mathsf{a}{=}x, \mathsf{b}{=}y\} = r$ in e. `@lang` supports this using Python's syntax for destructuring assignment, but we must omit the details.

Metatheory The formulation shown here guarantees that type synthesis actually produces a type, given well-formed contexts. The definitions are straightforward and the proof is a simple induction. We write Φ `fragment` for fragments with no duplicate tycon names and closed tycon definitions only, and $\Gamma\ \mathtt{ctx}_\Phi$ for typing contexts that only map variables to types constructed with tycons in Φ.

THEOREM 1 (Synthesis). *If* Φ `fragment` *and* $\Gamma\ \mathtt{ctx}_\Phi$ *and* $\Gamma \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota$ *then* $\sigma\ \mathtt{ty}_\Phi$.

We also have that importing additional type constructors cannot change the semantics of a previously well-typed term, assuming that naming conflicts have been resolved by some extrinsic mechanism. Indeed, the proof is an essentially trivial induction because of the way we have structured our mechanism. The type constructor delegated responsibility over a term is deterministically determined irrespective of the structure of Φ.

THEOREM 2 (Stable Extension). *If* Φ `fragment` *and* $\Gamma\ \mathtt{ctx}_\Phi$ *and* $\Gamma \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota$ *and* Φ' `fragment` *and* $dom(\Phi) \cap dom(\Phi') = \emptyset$ *then* $\Gamma \vdash_{\Phi;\Phi'} e \Rightarrow \sigma \rightsquigarrow \iota$.

Other metatheoretic guarantees about the translation cannot be provided in the formulation as given. However, inserting straightforward checks to guarantee that the translation is a closed term, and provide simple mechanisms for hygiene, would be simple, but are omitted to keep our focus on the basic structure of the calculus as a descriptive artifact.

5. Related Work

Early work proposing the inclusion of compile-time logic in libraries led to the phrase *active libraries* [23]. We borrow the prefix "active". Our focus on type constructors and type system extension differs substantially from previous work, which focused on term rewriting for the purpose of optimization over a fixed semantics. Several contemporary projects have the same goals, e.g. LMS [19], which allows staged translation of well-typed Scala programs to other targets and supports composing optimizations. Macro systems can also be used to define optimizations and operations with interesting dynamic semantics. In both cases, the language's type system itself remains fixed. Our mechanism permits true type system extensions. New types are not merely aliases, nor must their rules be admissible in some base type system (e.g. `string_in` would be difficult to define in Scala, particularly as directly as we can here). Note that in `@lang`, macros can be seen as a mode of use of the term constructor `Call`, as we can give the macro a singleton type (like Python modules, discussed above) that performs the desired rewriting in `ana_Call` and `syn_Call`.

Operator overloading [22] and *metaobject dispatch* [10] are run-time protocols that translate operator invocations into function calls. The function is typically selected according to the dynamic tag or value of one or more operands. These protocols share the notion of *inversion of control* with our strategy for targeted expressions. However, our strategy is a *compile-time* protocol. Note that we used Python's operator overloading and metaobject protocol for convenience in the static language.

Language-external mechanisms for creating and composing dialects (e.g. extensible compilers like Xoc [4] or language workbenches [6]) have ambiguity problems, and so they might benefit from the type constructor oriented view that we propose here as well. We argue that if ambiguities cannot occur and safety is guaranteed, there is no reason to leave the mechanism outside the language. Though we do not support syntax extension (other than via creative reuse of string literals), we argue that this is actually a benefit: we can use a variety of existing tools and avoid many facets of the expression problem [24] in this way.

Typed Racket and other typed LISPs also add a type system to a fixed syntax atop a dynamically typed core [21]. However, these treat the semantics as a "bag of rules", so adding new rules can cause ambiguities. Our work (particularly $@\lambda$, with its use of lists ubiquitously) provides a blueprint for a typed LISP oriented around type constructors, rather than rulesets. Type constructors are a natural organizational unit. For example, Harper's textbook, upon which we base our terminology and abstract syntax in this paper, identifies languages directly as a collection of type constructors [8].

Bidirectional type systems are increasingly being used in practical settings, e.g. in Scala and C#. They are useful in producing good error messages (which our mechanism shows can be customized) and help avoid the need for redundant type annotations while avoiding decidability limitations associated with whole-program type inference. Bidirectional techniques have also been used for adding refinement types to languages like ML and Twelf [13]. Refinement types can add stronger static checking over a fixed type system (analagous to formal verification), but cannot be used to add new operations directly to the language. Our use of a dynamic semantics for the static language relates to the notion of *type-level computation*, being explored in a number of languages (e.g. Haskell) for reasons other than extensibility.

Recent work we have done on the Wyvern programming language uses bidirectional typechecking to control aspects of literal syntax in a manner similar to, but somewhat more flexible than, how we treat introductory forms [16]. Wyvern is its own language dialect and does not have an extensible type system, supporting only desugarings like SugarJ [5]. The Wyvern formalism guarantees hygiene, using an approach that is also likely applicable in the setting of this paper.

6. Discussion

This work aimed to show that one can add static typechecking to a language like Python as a library, without undue syntactic overhead using techniques available in a growing number of languages: reflection, quasiquotes and a form of open sum for representing type constructors (here, Python's classes). The type system is not fixed, but flexibly extensible in a natural and direct manner, without resorting to complex encodings and, crucially, without the possibility of ambiguities arising at link-time.

Although we only touched on the details here, we have conducted a substantial case study using `@lang`: an implementation of the entirety of the OpenCL type system (a variant of C99), as well as several extensions to it, as a library. Our operations translate to Python's lower-level FFI with OpenCL but guarantee type safety at the interface between languages. A neurobiological circuit simulation framework has been built atop this library (a detailed case study is in preparation).

There remain several promising avenues for future work, many of which we mentioned throughout this paper. From a practical perspective, extensible implicit coercions (i.e. subtyping) using a mechanism similar to our "handle sets" mechanism for binary expressions would be useful. An extensible mechanism supporting type index polymorphism would also be of substantial utility and theoretical interest. Debuggers and other tools that rely not just on Python's syntax but also its semantics cannot be used directly. We believe that active types can be used to control debugging and other tools, and plan to explore this in the future. We have also not yet evaluated the feasibility of implementing more advanced type systems (e.g. linear, dependent or flow-dependent type systems) and those that require a more "global" view (e.g. security-oriented types) using our framework.

From a theoretical perspective, the next step is to introduce a static semantics for the internal language and the static language, so that we can help avoid issues of extension correctness and guarantee extension safety (by borrowing techniques from the typed compilation literature). An even more interesting goal would be to guarantee that extensions are mutually conservative: that one cannot weaken any of the guarantees of the other. We believe that by enforcing strict abstraction barriers between extensions, we can approach this goal. Bootstrapping the `@lang` compiler would be an interesting direction to explore to enable this.

Not all extensions will be useful. Indeed, some language designers worry that offering too much flexibility to users leads to abuse. This must be balanced with the possibilities made available by a vibrant ecosystem of competing statically-typed abstractions that can be developed and deployed as libraries, and thus more easily evaluated in the wild. With an appropriate community process, this could lead to faster adoption of ideas from the research community, and quicker abandonment of mistakes.

References

[1] The Python Language Reference. `http://docs.python.org`, 2013.

[2] 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.

[3] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang. An empirical study of programming language trends. *Software, IEEE*, 22(3):72–79, 2005.

[4] R. Cox, T. Bergan, A. T. Clements, M. F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS*, 2008.

[5] S. Erdweg, T. Rendel, C. Kastner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*.

[6] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The

state of the art in language workbenches. In *Software Language Engineering*. 2013.

[7] 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.

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

[9] 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.

[10] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[11] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 2011.

[12] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, Oct. 1986.

[13] W. Lovas and F. Pfenning. A bidirectional refinement type system for LF. In *Electronic Notes in Theoretical Computer Science, 196:113–128, January 2008.*, 2008.

[14] L. A. Meyerovich and A. Rabkin. How not to survey developers and repositories: experiences analyzing language adoption. In *PLATEAU*, 2012.

[15] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *OOPSLA*, 2013.

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

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

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

[19] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, June 2012.

[20] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Data Driven Functional Programming*, 2013.

[21] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL*, 2008.

[22] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language algol 68. *Acta Informatica*, 1975.

[23] 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.

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

[25] M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.