Growing a Static Type System Inside a Python

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

Abstract

Programmers prefer using constructs available as libraries for the programming language they already use over those requiring a switch to a new language. But library providers have not been able to modularly extend the type systems of these languages, so some of the most powerful constructs have remained obscure. Our aims in this paper are to 1) introduce a mechanism that allows library providers to define new type system fragments modularly; 2) show how it can itself be implemented as a library, `typy`, for a widely used language, Python; 3) validate our approach with examples of non-trivial type system fragments that are built into other languages but that can be implemented as libraries using this mechanism, including functional constructs, an object system, a type system for secure string sanitation and a typed foreign function interface to OpenCL; and 4) formally specify the mechanism with a minimal typed lambda calculus, $@\lambda$. The key idea is to associate semantic logic with type constructors, rather than syntactic forms.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Extensible Languages

Keywords type systems; extensible languages; metaprogramming

1. Introduction

Asking programmers to adopt 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].

Yet researchers and domain experts continue to design new languages and language dialects, justifying this practice by arguing that it would be impossible to isomorphically express the constructs they need as libraries for existing languages. This is certainly a convincing argument when the constructs in question specify a non-trivial static semantics. Unfortunately, the flexibility afforded taking such a *language-oriented approach* [24] to new ideas comes at the cost of modularity: working with a component written in a different language typically requires programming against the low-level details of a particular implementation of that language, rather than its semantics, via a foreign-function interface (FFI) (e.g. to C, or JVM bytecode). This is brittle (the implementation may change), unnatural and unsafe. Mitigating the loss of static safety by adding run-time checks at language boundaries has a performance cost.

MODULARITY'15, March 16–19, 2015, Fort Collins, CO, USA

These problems are relevant even to dialects that introduce only a few new constructs. A study comparing a Java dialect, Habanero-Java (HJ), with a library, `java.util.concurrent`, found that the language-based abstractions in HJ were indeed 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 constructs they might prefer because they are only available in specialized dialects they cannot adopt [14, 15].

Language extension mechanisms have long promised to reduce these problems by giving providers the ability to define syntactic and semantic constructs more granularly, so that clients can import them as needed. Unfortunately, the mechanisms available today have several problems. First, they are themselves often housed in an obscure language dialect and so face the classic "chicken-and-egg" problem: languages like SugarJ [5] or Wyvern [16] (which permit syntax extensions over a fixed semantics) must overcome the same sorts of barriers to adoption as a language like HJ. Second, they give providers too much control over the language, again at the cost of modularity, because providers can define conflicting constructs. The conflicts, if detected at all, are detected too late: when a client attempts to use the conflicting constructs in the same program.

In this paper, we take steps toward addressing both of these problems by introducing a mechanism that allows library providers to define new type system fragments modularly, and showing how it can be itself be implemented as a library, `typy`, for an existing and widely-used language, Python (using its quasiquotation and reflection facilities). This constrains us: we do not have the power to change Python's concrete and abstract syntax. But in fact, we find that this constraint can be freeing: questions of syntactic conflict never arise, and tools that operate by exhaustive case analysis over the syntax (e.g. syntax highlighters, editor plugins, pretty-printers, style checkers, documentation generators) continue to work.

Library providers introduce type system fragments by defining new type constructors (*tycons*, for brevity). For example, a library provider might introduce record types, as are built in to functional languages like ML and Haskell, by defining a new tycon, `record`. A different library provider may introduce a statically-typed variant of an object system featuring prototypic inheritance (alá Javascript) by defining the tycon `proto`. Types are constructed by applying a tycon to a statically valued *type index*, written `tycon[tyidx]`. For example, we can construct a record type with two rows like this:

```
record[{
  'label1' : dyn[()],
  'label2' : string[()]
}]
```

The index is a dictionary mapping row labels to their corresponding types. Here, `dyn` (which we discuss later) and `string` are trivially indexed tycons, i.e. the only valid index is `()`. For concision, trivial indices can simply be omitted (we showed them here only to emphasize the uniform manner in which one should think about type construction). A prototypic object type having an analagous field structure (and for simplicity, assuming the default prototype) is constructed like this:

```
proto[{ # (fields of a prototypic object are mutable)
  'label1': dyn,    # equivalent to dyn[()]
  'label2': string # equivalent to string[()]
}]
```

Given that a programmer might be using both of these libraries in the same program, how should our system determine the semantics of an expression like `e.label1`, where `e` is some subexpression? In a monolithic language, the usual approach is to define the semantics in a *syntax-directed* manner: the language specifies, in one place, all the logic relevant to each form. Our approach, in contrast, is *tycon-directed*: the language delegates to a *statically valued function* defined by some relevant tycon, called the *delegate*. The language specifies only the *delegation protocol* that assigns a delegate to each form. In this example, the delegate is the tycon of the type recursively synthesized by `e`. We will see how tycons like `record` and `proto` statically compute a type and a translation (to Python code) for such a term when they are delegated control by this protocol in Sec. 2. For now, it suffices to point out that the code involved is almost exactly what one would write if implementing a language that built in record types directly.

This example was of a form for which there is a clear "target" subexpression that can be examined recursively to determine the tycon to delegate control to. For other forms, it is perhaps less clear. For example, the curly-brace-delimited form in the following expression could be given a record type, a prototypic object type or be a dynamically tagged Python dictionary, in which case it would have type `dyn`:

```
f({label1: "val1", label2: "val2"})
```

We resolve this ambiguity by leveraging the *local type inference* mechanism that `typy` uses [17]. That is, for literal forms in Python's grammar, the tycon of the type that the term is being analyzed against is delegated control. Here, if `f` is a function taking an argument of the record type defined above, then `record` is delegated control over the curly-brace-delimited form. This can be understood as a variant of the mechanism that Wyvern introduced for resolving ambiguous syntax extensions [16], here applied in a setting where there is a fixed syntax but an extensible semantics.

Symmetric binary terms (e.g. addition) use a third protocol, which we will discuss.

There is always a single tycon delegated responsibility over a term and this delegate is stable with respect to additional library imports, so semantic ambiguities cannot arise by construction (unlike systems that treat the type system as a "bag of rules").

Specific Aims Our specific aims in this paper are to introduce this approach, which we call *active typechecking and translation*, 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. 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 @λ. 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 @λ 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.

[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.

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!"
```

2. Active Type Constructors in `@lang`

Listing 1 shows an example of a well-typed `@lang` program for which stronger correctness guarantees than Python can statically provide have been established using several type system fragments 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 its main 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 defining type system fragments. Here, atlib is the "standard library" but does not benefit from special support in atlang itself.

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, always taking the form of a type constructor applied to a type index: tycon[idx]. Unlike many statically-typed languages, types are constructed programmatically, rather than declared "literally" (e.g. classes in Java, datatypes in ML). In this example, we see several useful types being constructed:

1. On line 6, we construct a functional record type with two (immutable) fields, naming it (by assigning it to a static identifier) Details. The field amount has type decimal[2], which classifies decimal numbers having two decimal places, and memo has type dyn, which comes already constructed, classifying dynamic Python values. For convenience, the index for a record type can be 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* [19]).
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 regular language an immutable string is a member of through operations like concatenation and substitution, which is useful for ensuring well-formedness and preventing injection attacks [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 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) will be inferred below.

Incomplete types are not instances of atlang.Type but rather instances of atlang.IncompleteType. An incomplete type is

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) # complete 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)
```

defined by a type constructor (here fn) and an incomplete index (here, (Transfer, Ellipsis)). The language controls whether the form tycon[idx] results in a complte or incomplete type 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 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 and we will refer to these throughout the paper.

2.1 Active Typechecking

We will now trace through how the functions log_transfer and __main__ in Listing 1 are typechecked by atlang, via delegation to the type constructors just mentioned.

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 complete.

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

1. Checks and normalizes the incomplete function type index. _check_and_norm_inc (not shown) checks that the argument

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 ctx.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
```

types are indeed types and turns empty and single arguments into a 0- or 1-tuples for uniformity.

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` (i.e. a top-level expression), then the type of this expression is the return type, otherwise it is `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.
5. A complete index is generated, from which a type will be synthesized for the function, here `fn[Transfer, atlib.unit]` because uses of the `printf` operator, also defined in `atlib`, have type `atlib.unit` (i.e. `printf` is used only for its effect).

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), rather than asked to synthesize a type. Functions that don't end in expressions can only have return types `unit` or `dyn` (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.

Expressions 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.)

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 and give it an appropriate type. This cannot always be done automatically (because it is a form of serialization). Instead, any Python value with a `atlang_syn_Lift` method returning an `atlang` type can provide explicit support for this operation. A few other constructs also have built in support. For the purposes of this section, this includes other typed `@lang` functions (which synthesize their own type), untyped Python functions and classes (which synthesize types `pyfun` and `pyclass`, and thus can only be called with values of type `dyn`), modules (which are given a *singleton type* – a type indexed by the module reference itself – from which other values that can be lifted can be accessed as attributes, see below), and immutable Python data structures like numbers, strings and tuples.

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`.

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)
```

For example, the value of the field `account_num` delegates to `string_in` via the `ana_Str` method, shown in Listing 6.

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 define a type constructor that allows for different defaults using block-scoped "pragmas" in the static language via 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!"}
```

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    def ana_Str(self, ctx, e):
6      _check_rx(self.idx, e.s)
7
8    @classmethod
9    def syn_idx_Str(cls, ctx, e):
10     return _rx_of_str(e.s) # most specific rx
11
12   # treats string as string_in[".*"]
13   handles_Add_with = set([string])
14   @classmethod
15   def syn_BinOp_Add(cls, ctx, e):
16     rlang_left = self._rlang_from_ty(ctx.syn(e.left))
17     rlang_right = self._rlang_from_ty(ctx.syn(e.right))
18     return string_in[self._concatenate_langs(
19       rlang_left, rlang_right)]
```

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

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

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 14 of Listing 6.

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 13 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.

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.

[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 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 ctx.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 ast.Tuple((f, ctx.trans(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.
```

Examples of translation methods for the `fn` and `record` type constructors are shown in Listing 8. The output of translation for 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).

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.

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

```
import atlib.runtime as _at_i0_

def log_transfer(t):
  _at_i0_.print("Transferring %s to %s." %
    (_at_i0_.dec_to_str(t[0][0], 2), t[1][0]))

def __main__():
  account = ("Annie Ace", "00-00000001")
  log_transfer((((5, 50), None), account))
  log_transfer((((15, 0), "Rent payment"), account))
if __name__ == "__main__": __main__()
```

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

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

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 were all performed statically in this example.
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).

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. we have true static typechecking).

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

```
1  from atlang import ty
2  from atlib import case, 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 # using Python 3 type annotation syntax
12 def treesum(tree : Tree(dyn)) -> dyn:
13   case(tree) [
14     Empty: 0,
15     Leaf(x): x,
16     Node((l, r)): treesum(l) + treesum(r)
17   ]
18
19 @fn
20 def __main__():
21   treesum(Tree.Node(Tree.Leaf(5), Tree.Leaf(5)))
```

Listing 14 The translation of Listing 13. Case types are implemented as fast tagged values.

```
1  def treesum(tree):
2    return (0 if tree[0] == 0 else
3      tree[1] if tree[0] == 1 else
4      treesum(tree[1][0]) + treesum(tree[1][1]))
5
6  def __main__():
7    treesum((2, ((1, 5), (1, 5))))
```

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.

If a function captures any static values that do not support lifting, then that function can only be used via interactive invocation (there is no way to separate compile-time from run-time without lifting captured values). Such values when captured in an interactively invoked setting by default synthesize type `dyn`, though they can provide a `atlang_syn_Capture` method that synthesizes a more specific type. We will see an example of this and a related use of phaseless capture in our discussion of `atlib.opencl` below.

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 at a workshop [7]. Here, we showcase two more powerful examples that demonstrate the expressive power of the 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 sum types), tuples, records and other types. For example, tree structures are well-modeled using recursive labeled sums, as we show in Listing 13. Here, `Tree` is a *case type*, which is what we call labeled sums in `atlib`. `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

Listing 15 The implementation of the case analysis operator uses intermediate type constructors that contain only typing logic but no translation logic. It also defines its own "second-order" extensibility mechanism.

```
1  class _case(object):
2    def syn_Lift(self):
3      return caseop[()]
4  case = _case()
5
6  class caseop(atlang.Type):
7    def syn_Call(self, ctx, e):
8      ctx.syn(e.args[0])
9      if len(e.args) != 0:
10       raise atlang.TypeError("...", e)
11     return caseop_applied[()]
12
13 class caseop_applied(atlang.Type):
14   def syn_Subscript(self, ctx, e):
15     scrutinee = e.value.args[0]
16     rules = _rules_from(e.slices)
17     ty = None
18     for rule in rules:
19       bindings = bindings_from_pat(rule, scrutinee.ty)
20       _push_bindings(ctx, bindings)
21       cur_type = ctx.syn(rule.branch)
22       _pop_bindings(ctx, bindings)
23       if ty is not None and cur_type != ty:
24         raise atlang.TypeError("inconsistent", rule)
25       else: ty = cur_type
26     return ty
27
28   def trans_Subscript(self, ctx, e):
29     scrutinee = e.value.args[0]
30     return scrutinee.ty.trans_pats(
31       _rules_from(e.slices))
32
33 class casetype(atlang.Type):
34   # ...
35   def type_pat_Call(self, ctx, p):
36     if p.func in self.idx:
37       return self.idx[p.func]
38
39   def trans_pats(self, ctx, pats):
40     # ... turns sequence of patterns at same level
41     # into a conditional expr...
42
43 class tuple(atlang.Type):
44   # ...
45   def type_pat_Tuple(self, ctx, p):
46     if len(p.elts) == len(self.idx):
47       return self.idx
48     else raise PatError("invalid pat length")
```

are identified by a name. Functional datatypes in ML and similar languages are also generative in this way. The `@ty` mechanism also supports parameterized types: here, `a` stands in for any other type. The type index to `casetype` is a series of cases with, optionally, associated 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 `atlib.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 by nested pattern matching. The `__main__` function constructs a tree and calls `treesum` on it. The result of compiling Listing 13 is shown in Listing 14. Case types translate to pairs consisting of a numeric tag and data, and case analysis translates to nested conditionals.

Some relevant portions of the implementation of case analysis are shown in Listing 15. The object `case` is an instance of the class `_case`, which defines the method `syn_Lift`. When the imported name `case` is encountered in Listing 13, this is called, per the discussion earlier on names in the static environment. The result is that `case` synthesizes type `caseop[()]`.

Listing 16 An example use of our typed FFI to OpenCL, demonstrating both template functions and phaseless capture.

```
1  import atlib.opencl as opencl
2  import numpy
3
4  device = opencl.init_device(0)
5  buffer = device.send(numpy.zeros((800000,))
6
7  @opencl.template_fn
8  def map(b, f):
9    gid = get_global_id(0) # data parallel thread ID
10   b[gid] = f(b[gid])
11
12 @opencl.template_fn
13 def shift(x):
14   return x + 5
15
16 @opencl.template_fn
17 def triple(x):
18   return 3 * x
19
20 map(buffer, triple)
21 map(buffer, shift)
```

The only purpose of the type constructor `caseop` is to define the `syn_Call` method, which ensures that the case operator is applied to a single well-typed scrutinee. The combined term `case(tree)` then synthesizes type `caseop_applied[()]`. Notice that neither method is paired with a `trans_` method – these types are merely "bookkeeping" for the compound case analysis form. Parts of this form are not meaningful by themselves, so translation would fail if, for example, `case(tree)` was written by itself without cases.

The `caseop_applied` tycon defines the `syn_Subscript` method, which actually processes the list of provided rules. First, the type of the scrutinee is asked to process the bindings from the pattern portion of each rule. This involves calling methods of the form `type_pat_TermCon`, which must return types for all the sub-patterns. For example, the `type_pat_Call` method of `casetype` makes sure that the case named is defined, and if so returns the associated type, taken from its type index, as the type of the inner pattern. The process repeats recursively, so that a tuple pattern might then be processed by `type_pat_Tuple`. Any type constructor can participate in this protocol by defining appropriate methods, so it is extensible. Note, however, that this is a "second-order" extensibility mechanism – nothing in `atlang` itself supports pattern matching. Instead, `case` itself defined this protocol in a library, demonstrating the power of using a flexible general-purpose language for writing extensions.

The translation phase begins at the `trans_Subscript` method of `caseop_applied`. This simply hands responsibility for translating the entire set of rules to the type of the scrutinee. This may then hand control to the types of inner patterns (analagous to the typing phase, not shown).

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, object types or regular string types in the previous section, or the typed OpenCL FFI we will describe next, without any concern regarding syntactic or semantic conflicts. This is enabled entirely by taking a type constructor oriented view toward the extensibility problem.

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 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 data parallel language that can compile to a variety of specialized hardware (e.g. GPUs) via a standard API, as well as a runtime that allows a program to manage device memory. The `pyopencl` package exposes these APIs to Python code and integrates them 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. OpenCL is also often too low-level, not supporting even features found in other low-level languages like function overloading, function templates and higher-order functions.

In Listing 16, we show usage of the `atlib.opencl` package, which implements the entirety of the OpenCL type system (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, to support the wide variety of number types in C99-based languages.

After initializing a device, line 4 sends a `numpy` array to the device, assigning `buffer` its handle, an object inheriting from `opencl.Buffer`. The type of the `numpy` array is also extracted and stored in this object.

Listing 17 The underlying code generated by `atlib.opencl` as a string passed through `pyopencl`.

```
1  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2
3  double shift(double x) {
4    return x + 5.0;
5  }
6
7  double triple(double x) {
8    return 3.0 * x;
9  }
10
11 kernel void map__0_(global double* b) {
12   size_t gid = get_global_id(0);
13   b[gid] = triple(b[gid])
14 }
15
16 kernel void map__1_(global double* b) {
17   size_t gid = get_global_id(0);
18   b[gid] = shift(b[gid])
19 }
```

Listing 18 A portion of the logic of OpenCL template functions, showing how they defer to the logic for standard OpenCL functions at each call site, rather than at the declaration site.

```
1  class template_fn(atlang.Type):
2    # ...
3    def syn_Call(self, ctx, e):
4      arg_types = [ctx.syn(arg) for arg in e.args]
5      return fn[fn.syn_idx_FunctionDef(ctx, self.idx.fn,
6        (arg_types, Ellipsis))]
7
8    def trans_Call(self, ctx, e):
9      return _wrap_in_pyopencl(
10       e.ty.trans_FunctionDef(ctx, self.idx.fn))
```

Listing 19 The `opencl.Buffer` class represents OpenCL memory objects in global device memory, inheriting from the `pyopencl.Buffer` class. These supports phaseless capture at the corresponding global pointer type.

```
1  class Buffer(pyopencl.Buffer):
2    # ...
3    def atlang_syn_Capture(self):
4      return global_ptr[
5        _np_dtypes_to_cltypes(self.dtype)]
```

On lines 7-18, we define three OpenCL template function. Let us consider the first, `map`, which takes two arguments. A template function is one that does not have specified argument types, generally because many types might be valid, as in this case. The body of `map` would be valid for any combination of types for `b` and `f` that support subscripting and calls as shown. Template functions thus defer typechecking to every invocation site, based on the types synthesized by the provided arguments. This is analagous to template functions in C++, but here we implicitly assume a distinct template parameter for each argument and implicitly instantiate these parameters at use sites. Technically, such functions have a singleton type, indexed by the function definition itself, as shown in Listing 18. Note that OpenCL itself doesn't have template functions; we are implementing them, essentially by extending the language across its FFI.

We see two applications of `map` on lines 20 and 21. Note that these are in the dynamically typed portion of the code. By default, the first argument in each case, `buffer`, would synthesize type `dyn` and typechecking would fail (at line 10, technically, because `dyn` doesn't support subscripting with the type that `gid` synthesizes). But we make use of the support for phaseless capture discussed in Sec. 2.4, which allows values to define a method, `atlang_syn_Capture`, that permits capture and, in this case, function application with values that would not otherwise support crossing into `atlang` functions with argument types other than `dyn`. In this case, `Buffer` objects synthesize a corresponding pointer type, represented by the tycon `opencl.global_ptr` and indexed by the target type of the pointer, here `opencl.double` because the initial `numpy` array was an array of `doubles` and there is a mapping from `numpy` data types to OpenCL types. The implementation is shown in Listing 19. Note that phaseless capture is distinct from lifting – lifting requires a value persist from compile-time to run-time, but OpenCL buffers are clearly ephemeral. Phaseless capture can only occur when interactive invocation is being used.

Each invocation of `map` received the same buffer, but a different function. Because these were template functions, their types were distinct. As such, two different versions of map were generated, as shown in Listing 17. Indeed, without support for function pointers in OpenCL, code generation like this is the standard way of supporting "higher-order" functions like `map`. Rather than having to do this manually, however, our type system handled this internally.

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 or case types, modularly, via its foreign function interface from a scripting language like Python.

4. @λ: 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. Static terms are analagous to top-level Python code in `atlang`, and external terms are analagous to terms inside typed functions.

programs	ρ	::=	$\mathsf{import}[\Phi](e)$
fragments	Φ	::=	$\emptyset \mid \Phi, \text{TYCON} = \{\delta\}$
tycon defs	δ	::=	$\mathsf{analit} = \sigma;\ \mathsf{synidxlit} = \sigma;$ $\mathsf{anatarg} = \sigma;\ \mathsf{syntarg} = \sigma$
expressions	e	::=	$x \mid \mathsf{let}(e; x.e) \mid \mathsf{slet}[\sigma](\mathbf{x}.e) \mid \mathsf{asc}[\sigma](e)$
		$\mid$	$\lambda(x.e) \mid \mathsf{lit}[\sigma](\overline{e}) \mid \mathsf{targ}[\sigma](e; \overline{e})$
static terms	σ	::=	$\mathbf{x} \mid \lambda(\mathbf{x}.\sigma) \mid \mathsf{ap}(\sigma; \sigma) \mid \mathsf{fail}$
		$\mid$	$\mathsf{ty}[\text{TYCON}](\sigma) \mid \mathsf{tycase}[\text{TYCON}](\sigma; \mathbf{x}.\sigma; \sigma)$
		$\mid$	$\mathsf{incty}[\text{TYCON}](\sigma)$
		$\mid$	$\mathsf{lbl}[\mathtt{lbl}] \mid \mathsf{lbleq}(\sigma; \sigma; \sigma; \sigma)$
		$\mid$	$\mathsf{nil} \mid \mathsf{cons}(\sigma; \sigma) \mid \mathsf{listrec}(\sigma; \sigma; \mathbf{x}.\mathbf{y}.\sigma)$
		$\mid$	$\mathsf{arg}[e] \mid \mathsf{ana}(\sigma; \sigma; \mathbf{x}.\sigma) \mid \mathsf{syn}(\sigma; \mathbf{x}.\mathbf{y}.\sigma)$
		$\mid$	$\triangleright(\iota)$
internal terms	ι	::=	$\triangleleft(\sigma) \mid x \mid \lambda(x.\iota) \mid \mathsf{iap}(\iota; \iota)$
		$\mid$	$\mathsf{inil} \mid \mathsf{icons}(\iota; \iota) \mid \mathsf{ilistrec}(\iota; \iota; x.y.\iota)$

Figure 2. Abstract syntax of @λ. 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 $\overline{e}$.

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 $\mathsf{targ}[\sigma](e; \overline{e})$ 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 $\triangleleft(\sigma)$ 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

```
import Φ_nat, Φ_lprod                    import[Φ_nat, Φ_lprod](
static let nat = NAT[nil]                slet[ty[NAT](nil)](nat.
let zero = 𝟘 : nat                       let(asc[nat](lit[lbl[𝟘]](·)); zero.
let one = zero.s                         let(targ[lbl[s]](zero; ·); one.
let plus = (λx:nat.λy:nat.               let(asc[incty[FN](nat)](λ(x.asc[incty[FN](nat)](λ(y.
  x · rec(y; λp.λr.r · s))                 targ[lbl[rec]](x; λ(p.λ(r.targ[lbl[s]](r; ·)))))))); plus.
let two = plus one one                   let(targ[nil](targ[nil](plus; one); one); two.
{one=one; two=two} : LPROD[···]          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.

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

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

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

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

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

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

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

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

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

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

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 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.

```
Φ_fn := FN = {analit = nil; synidxlit = nil; anatarg = nil;
  syntarg = λtyidx.λifn.λtmidx.λargs.
    isnil tmidx (decons1 args λarg.decons2 tyidx λinty.λrty.
      ana(arg; inty; ia.pair rty ▷(iap(◁(ifn); ◁(ia)))))}
```

Figure 4. The FN fragment defines function application.

```
Φ_nat := NAT = {analit = λtyidx.λtmidx.λargs.
    isnil tyidx (lbleq tmidx lbl[𝟘] (isnil args ▷(inil)));
  synidxlit = λincidx.λtmidx.λargs.
    isnil incidx (lbleq tmidx lbl[𝟘] (isnil args (pair nil ▷(inil))));
  anatarg = λtyidx.λi1.λty.λtmidx.λargs.
    lbleq tmidx lbl[rec] (decons2 args λarg1.λarg2.
      ana(arg1; ty; i2.ana(arg2; fn2ty ty[NAT](nil) ty ty; i3.
        ▷(ilistrec(◁(i1); ◁(i2); x, y.iap(iap(◁(i3); x); y)))))));
  syntarg = λtyidx.λi1.λtmidx.λargs.
    lbleq tmidx lbl[s] (isnil args (
      pair ty[NAT](nil) ▷(icons(inil; ◁(i1)))))}
```

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

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.

```
Φ_lprod := LPROD = {analit = λtyidx.λtmidx.λargs.
    listrec(zipexact3 tyidx tmidx args; ▷(inil); h.ri.
      decons3 h λidxitem.λlbl.λe.decons2 idxitem λlblidx.λtyidx.
        lbleq lbl lblidx (ana(e; tyidx; i.▷(icons(◁(i), ◁(ri))))));
  synidxlit = λincidx.λtmidx.λargs.
    listrec(zipexact2 tmidx args; pair nil ▷(inil); h.r.
      decons2 h λlbl.λe.decons2 h λridx.λri.syn(e; ty.i.
        pair cons(pair lbl ty; ridx) ▷(icons(◁(i); ◁(ri)))));
  anatarg = nil; (destructuring let could be implemented here)
  syntarg = λtyidx.λi.λlbl.λargs.
    isnil args (pair (lookup lbl tyidx)
      ▷(iap(iap(nth; ◁(i)); ◁(itermofn (posof lbl tyidx)))))
```

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

$\boxed{\sigma\ \mathtt{ty}_\Phi}$

$$\textbf{TY}\ \frac{\mathtt{ty}[\mathrm{TYCON}](\sigma) \Downarrow_{\emptyset;\Phi} \mathtt{ty}[\mathrm{TYCON}](\sigma)}{\mathtt{ty}[\mathrm{TYCON}](\sigma)\ \mathtt{ty}_\Phi}$$

$\boxed{\sigma \Downarrow_{\Gamma;\Phi} \sigma}$

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

$$\textbf{N-ARG}\ \frac{}{\mathtt{arg}[e] \Downarrow_{\Gamma;\Phi} \mathtt{arg}[e]}$$

$$\textbf{N-ANA}\ \frac{\sigma_1 \Downarrow_{\Gamma;\Phi} \mathtt{arg}[e] \qquad \sigma_2 \Downarrow_{\Gamma;\Phi} \sigma_{\mathrm{ty}} \qquad \sigma_{\mathrm{ty}}\ \mathtt{ty}_\Phi \qquad \Gamma \vdash_\Phi e \Leftarrow \sigma_{\mathrm{ty}} \rightsquigarrow \iota \qquad [\triangleright(\iota)/\mathbf{x}]\sigma_3 \Downarrow_{\Gamma;\Phi} \sigma_3'}{\mathtt{ana}(\sigma_1; \sigma_2; \mathbf{x}.\sigma_3) \Downarrow_{\Gamma;\Phi} \sigma_3'}$$

$$\textbf{N-SYN}\ \frac{\sigma_1 \Downarrow_{\Gamma;\Phi} \mathtt{arg}[e] \qquad \Gamma \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota \qquad [\sigma/\mathbf{x}_1, \triangleright(\iota)/\mathbf{x}_2]\sigma_2 \Downarrow_{\Gamma;\Phi} \sigma_2'}{\mathtt{syn}(\sigma_1; \mathbf{x}_1.\mathbf{x}_2.\sigma_2) \Downarrow_{\Gamma;\Phi} \sigma_2'}$$

$$\textbf{N-QUOTE}\ \frac{\iota \curlyvee_{\Gamma;\Phi} \iota'}{\triangleright(\iota) \Downarrow_{\Gamma;\Phi} \triangleright(\iota')}$$

$\boxed{\iota \curlyvee_{\Gamma;\Phi} \iota}$

$$\textbf{Q-UQ}\ \frac{\sigma \Downarrow_{\Gamma;\Phi} \triangleright(\iota)}{\triangleleft(\sigma) \curlyvee_{\Gamma;\Phi} \iota} \qquad \textbf{Q-X}\ \frac{}{x \curlyvee_{\Gamma;\Phi} x} \qquad \textbf{Q-LAM}\ \frac{\iota \curlyvee_{\Gamma;\Phi} \iota'}{\lambda(x.\iota) \curlyvee_{\Gamma;\Phi} \lambda(x.\iota')}$$

Figure 7. Selected normalization rules for the static language.

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).

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 $\triangleright(\iota)$). The type and term index are provided, as well as a list of *reified arguments*: static values of the form $\mathtt{arg}[e]$. 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 $\mathtt{ana}(\sigma_1; \sigma_2; \mathbf{x}.\sigma_3)$, 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, $\triangleleft(\sigma)$, 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 $\mathtt{syn}(\sigma_1; \mathbf{x}.\mathbf{y}.\sigma_2)$, 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 $\mathtt{targ}[\sigma_{\mathrm{tmidx}}](e_{\mathrm{targ}}; \overline{e})$. 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 $\{\mathtt{a}{=}x, \mathtt{b}{=}y\} = r$ 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 Γ $\mathtt{ctx}_\Phi$ for typing contexts that only map variables to types constructed with tycons in Φ.

THEOREM 1 (Synthesis). *If* Φ `fragment` *and* Γ $\mathtt{ctx}_\Phi$ *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* Γ $\texttt{ctx}_\Phi$ *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* [22]. 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 [18], 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 [21] 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 [23] in this way.

Typed Racket and other typed LISPs also add a type system to a fixed syntax atop a dynamically typed core [20]. 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.

In recent work on the Wyvern programming language, we also used bidirectional typechecking to control aspects of literal syntax in a manner similar to, but somewhat more syntactically 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. Type systems that require tracking contextual information are possible using our framework, but the context needs to be initialized on first use of a type that needs it. This is somewhat awkward, so we plan to include a mechanism that allows any imported tycons to initialize the context ahead of typechecking.

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 (and a manuscript is in preparation). Bootstrapping the `@lang` compiler would be an interesting direction

to explore, to avoid the awkwardness of relying on a loose dynamically typed language to implement static type systems.

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, 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.

[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] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

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