Programmable Semantic Fragments

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

Abstract

This paper introduces `typy`, a bidirectionally typed language embedded reflectively into Python. Rather than defining a monolithic semantics, `typy` delegates semantic control over each term drawn from Python's fixed syntax to a contextually relevant *semantic fragment*. This fragment programmatically 1) typechecks the term, and 2) assigns dynamic meaning to the term by computing its translation to Python.

We argue that this design is *expressive* with examples of fragments that express the semantics of 1) functional record types; 2) a variation on JavaScript's prototypic object system; 3) labeled sum types (with nested pattern matching *a la* ML); and 4) foreign function interfaces to Python and OpenCL.

We further argue that this design is *compositionally well-behaved*. It sidesteps the problems of grammar composition and the expression problem by allowing different fragments to deterministically share a fixed concrete and abstract syntax. Moreover, programs are semantically stable under fragment composition (unlike systems where the semantics is a "bag of inference rules.")

1. Introduction

When choosing a programming language, developers prize library availability [5, 21]. In response, new languages tend either to extend a language that has an established ecosystem of libraries like Java, JavaScript or Python, or to define a safe and low-cost foreign function interface (FFI) to such a language. For example, TypeScript [4] and Flow [1] extend JavaScript with different object systems, and PureScript [2] is a functional language similar to Haskell that interfaces cleanly with JavaScript.

These designs satisfyingly address the problem of *backwards compatibility*, but as new languages like these continue to proliferate (with the help of language workbenches and other tools [12]), developers face the more daunting problem of *lateral compatibility*, i.e. of interfacing with libraries that are written in sibling languages. Defining an FFI between every pair of sibling languages is not a scalable solution, but absent a direct FFI, clients can interface only indirectly with these libraries through the code generated by a compiler that targets the established language. This is unnatural (the syntactic and semantic conveniences of the sibling language are unavailable), unsafe (the type system of the sibling language is not enforced) and anti-modular (if the compiler changes, client programs can break.) The *language-oriented* approach [34] seems, then, to be difficult to reconcile with the best practices of "programming in the large" [8].

This situation calls for a more compositional approach: rather than packaging, for example, new object systems or variations on functional type structures into separate "monolithic" languages, we seek a single "extensible" language where library providers can express semantic structures like these as composable *semantic fragments*. This *fragment-oriented* approach sidesteps the lateral compatibility problem – clients of a library that uses, for example, functional records, in its public interface, can simply import the record fragment themselves (i.e. the record fragment can be listed as a dependency, in the usual manner.) Of course, designing an expressive semantic fragment system equipped with useful composition principles presents several challenges.

First, language designers typically define concrete forms specific to the primitive structures that they introduce, but if we allow fragments to define new concrete forms in a decentralized manner (following the approach of Sugar* [11] and similar systems), then different fragments could define conflicting forms. For example, consider the following family of forms:

$$\texttt{\{ label}_1\texttt{: expr}_1\texttt{, ..., label}_n\texttt{: expr}_n\texttt{ \}}$$

One fragment might take these as the introductory forms for functional records, while another fragment might take these as the introductory forms for JavaScript-style objects. These forms might also conflict with those for Python-style dictionaries (which use values, rather than labels, as keys.) Such syntactic conflicts inhibit composition.

At the level of abstract syntax, we encounter the classic *expression problem* [25, 33]: if library providers can define new term constructors in a decentralized manner, then it becomes difficult to define functions that proceed by exhaustive case analysis over term constructors, like pretty-printers.

At the level of the semantics, we must maintain essential metatheoretic properties, like type safety. Moreover, clients

should be able to assume that importing a new fragment will not change the meaning of an existing program, nor allow a program to take on more than one meaning.[1] This implies that we cannot simply operationalize the semantics as a "bag of rules" that fragments contribute to unconstrained.

cite shen?

In this paper, we design a semantic fragment system that addresses the problems of concrete and abstract syntax very simply: fragment providers cannot extend the concrete nor the abstract syntax of the language. Instead, the semantics allows fragments to "share" syntactic forms by delegating control over each term to a contextually relevant fragment definition. For example, our semantics delegates control over terms of curly-brace delimited form (discussed above) to the fragment that defines the type that the term is being checked against. As such, these forms can serve as introductory forms for records, objects and dictionaries alike. The delegation protocol is deterministic and the choice it makes is stable under fragment composition, so conflicts cannot arise.

What does it mean for a fragment to be "delegated control" over a term? In our system, the delegated fragment is tasked first with programmatically typechecking the term, following a *bidirectional* protocol, i.e. one that supports local type inference by distinguishing situations where a type needs to be synthesized from situations where the type is known [24]. (Scala is another notable language that has a bidirectional type system [22].) If the fragment decides that the term is ill-typed, it generates an error message. If typechecking succeeds, the fragment is then tasked with assigning dynamic meaning to the term by computing a translation to a fixed *internal language* (IL). The IL is analagous to an intermediate language like one might find in the first stage of a compiler (here shifted into the semantics of the language.) Defining the dynamics by translation (following the approach taken in the type theoretic definition of Standard ML [15]) means that we need only establish type safety for the IL.

The general fragment delegation protocol just outlined is the primary conceptual contribution of this paper. To make matters concrete, we must choose a particular syntax and IL. We choose to repurpose Python's syntax and use Python itself as the IL. There are two main reasons for this choice: 1) Python supports both dynamic reflection and code generation, so we can implement our system itself as a Python library, called `typy`; and 2) Python's syntax, cleverly repurposed, is rich enough to express a wide variety of interesting constructs cleanly.We introduce `typy` with simple examples in Sec. 2 and then give several more sophisticated examples in Sec. 3.

Python known to be type safe – POPL paper?

We summarize related work in Sec. 4 and conclude with a discussion of present limitations and future work in Sec. 5.

[1] We assume throughout that simple naming conflicts are handled by some external coordination mechanism, e.g. a package repository.

Listing 1 Types and values in `typy`.

```
1  from typy import component
2  from typy.std import record, string_in, dyn
3
4  @component
5  def Listing1():
6    Account [: type] = record[
7      name        : string_in[r'.+'],
8      account_num : string_in[r'\d{2}-\d{8}'],
9      memo        : dyn
10   ]
11
12   test_acct [: Account] = {
13     name:        "Harry Q. Bovik",
14     account_num: "00-12345678",
15     memo:        { }
16   }
```

2. `typy`

Listing 1 shows an example of a well-typed `typy` program that first imports several fragments, then defines a top-level component, `Listing1`, that exports a record type, `Account`, and a value of that type, `test_acct`.

2.1 Dynamic Embedding

`typy` is dynamically embedded into Python, meaning that Listing 1 is simply a standard Python script at the top level. `typy` supports Python 2.6+, though in later examples, we use syntactic conveniences not introduced until Python 3.0+. Appendix A discusses the minor changes necessary to port these examples to Python 2.6+.

Package Management On Line 2, we use Python's `import` mechanism to import three fragments from `typy.std`, the `typy` standard library. This library receives no special treatment from `typy`'s semantics – it comes bundled with `typy` merely for convenience.

Fragments `typy` fragments are Python classes that extend `typy.Fragment`. These classes are never instantiated – instead, `typy` interacts with them exclusively through class methods (i.e. methods on the class object.) Listing 2 shows a portion of the `record` fragment, discussed in Sec. 2.2.

Top-Level Components On Lines 4-16 of Listing 1, we define a top-level `typy` component by decorating a Python function value with `component`, a decorator defined by `typy`. This decorator discards the decorated function value after extracting its abstract syntax tree (AST) and *static environment*, i.e. its closure and the globals dictionary, using the reflection mechanisms exposed by the `ast` and `inspect` packages in Python's standard library.[2] It then processes the syntactic forms in the body of the function definition according to an alternative semantics. In particular, `component` repurposes Python's assignment and array slicing forms to allow for the definition of type and value members. We will discuss

[2] The reader may need to refer to documentation for the `ast` package, available at https://docs.python.org/3.6/library/ast.html, to fully understand some examples in the remainder of this section.

Listing 2 Type validation in `typy`.

```
1  import ast, typy
2  class record(typy.Fragment):
3    @classmethod
4    def init_idx(cls, ctx, idx_ast):
5      if isinstance(idx_ast, ast.Slice):
6        # Python special cases single slices
7        # we don't want that
8        idx_ast = ast.ExtSlice(dims=[idx_ast])
9      if isinstance(idx_ast, ast.ExtSlice):
10       idx_value = dict()
11       for dim in idx_ast.dims:
12         if (isinstance(dim, ast.Slice) and
13             dim.step is None and
14             dim.upper is not None and
15             isinstance(dim.lower, ast.Name)):
16           lbl = dim.lower.id
17           if lbl in idx_value:
18             raise typy.TypeFormationError(
19               "Duplicate label.", dim)
20           ty = ctx.as_type(dim.upper)
21           idx_value[lbl] = ty
22         else: raise typy.TypeFormationError(
23           "Invalid field spec.", dim)
24       return idx_value
25     else: raise typy.TypeFormationError(
26       "Invalid record spec.", idx_ast)
```

the type member `Account` in Sec. 2.2 and the value member `test_acct` in Sec. 2.3 below. The return value of the decorator is an instance of `typy.Component` that 1) tracks the identities of the type members; 2) tracks the types and translations of the value members; and 3) mediates interactions with the top-level Python script.

sec...?

2.2 Fragmentary Type Validation

The type member definition on Lines 6-10 of Listing 1 is of the following general form:

```
name [: kind] = ty_expr
```

where `name` is a Python name, `kind` is a `typy` *kind* and `ty_expr` is a `typy` *type expression*. Kinds classify type expressions, so when `typy` encounters a definition like this, it checks that `ty_expr` is of kind `kind`.

`typy` adopts the system of *dependent singleton kinds* developed for the ML module system [7, 14], which elegantly handles the details of type synonynms, type members and parameterized types. The only major deviation from this established account of type expressions, which we will not repeat here, is that types in canonical form are expressed as:

```
fragment[idx]
```

where `fragment` is a fragment in the static environment and `idx` is some Python slice form. In other words, every `typy` type in canonical form is associated with a fragment (there are no "primitive" types defined by `typy` itself.) For convenience, programmers can write `fragment` by itself when the index is trivial, i.e. when it is of the form `()`.

For example, the type expression on Lines 6-10 of Listing 1 is a `record` type in canonical form. The index, which is in Python's *extended slice form*, specifies fields named `name`, `account_num` and `memo` and corresponding field types as shown. We discuss the field types in Sec. 2.3 – for now, it suffices to see that these are also canonical.

To establish that a type in canonical form is valid, i.e. of kind `type`, `typy` calls the fragment class method `init_idx` with the *context* and the AST of the index as arguments. This method must return a Python value called the type's *index value* if the type is valid, or raise `typy.TypeValidationError` with an error message and a reference to the location of the error within the index otherwise. The context 1) tracks contextual information, e.g. the kinds of type variables; and 2) provides methods that fragments use to interact with the kind system. In particular, the context provides a method `as_type` that turns the given Python AST into a type expression, i.e. an instance of `typy.TypeExpr`, and checks that it is of kind `type`.

For example, the `record` fragment validates record types as shown in Listing 2 by checking that the index consists of a sequence of field specifications of the form `name : ty`, where `name` is a Python name, no names are duplicated and `ty` is a valid type as determined by calling `ctx.as_type` (Line 20.) The returned index value is a Python dictionary mapping the names to the corresponding instances of `typy.TypeExpr`.

2.3 Fragmentary Bidirectional Typing and Translation

The value member definition on Lines 12-16 of Listing 1 is of the following general form:

```
name [: ty_expr] = expr
```

where `name` is a Python name, `ty_expr` is a type expression and `expr` is an expression. When `typy` encounters a definition like this, it 1) checks that `ty_expr` is of kind `type`, as described in Sec. 2.2; 2) *analyzes* `expr` against `ty_expr`; and 3) generates a *translation* for `expr`, which is another Python AST.

The type annotation is not necessary:

```
name = expr
```

In this case, `typy` attempts to *synthesize* a type from `expr` before generating a translation, rather than analyzing it against a known type.

Type systems that distinguish type analysis (where the type is known) from type synthesis (also known as *local type inference*, where the type must be determined from the expression) are called *bidirectional type systems* [24]. Our system is based on the system developed by Dunfield and Krishnaswami [9]. Again, we will not repeat standard details here – our interest will be only in how `typy` delegates control to some contextually relevant fragment according to the `typy` *delegation protocol*. This section describes how the delegation protocol works for different expression forms.

2.3.1 Literal Forms

`typy` delegates control over the typechecking and translation of expressions of literal form to the fragment defining the type that the expression is being analyzed against.

For example, the expression on Lines 12-16 of Listing 1 is of dictionary literal form. The type that this expression is being analyzed against is `Account`, which is synonymous with a `record` type. `typy` first calls the `record.ana_Dict` class

Listing 3 Introductory forms in `typy`.

```
# class record(typy.Fragment):
  # ... continued from Listing 2 ...
  @classmethod
  def ana_Dict(cls, ctx, idx, e):
    for lbl, value in zip(e.keys, e.values):
      if isinstance(lbl, ast.Name):
        if lbl.id in idx:
          ctx.ana(value, idx[lbl.id])
        else:
          raise typy.TyError("<bad label>", f)
      else:
        raise typy.TyError("<not a label>", f)
    if len(idx) != len(e.keys):
      raise typy.TyError("<label missing>", e)

  @classmethod
  def trans_Dict(self, ctx, idx, e):
    if len(idx) == 1:
      return ctx.trans(e.values[0])
    ast_dict = dict((k.id, v)
      for k, v in zip(e.keys, e.values))
    return ast.Tuple(
      (lbl, ctx.trans(ast_dict[lbl]))
      for lbl in idx.iterkeys())
```

method, shown in Figure 3. This method receives the context, the index value of the type and the AST of the literal expression as arguments and must return (trivially) if typechecking is successful or raise `typy.TyError` with an error message and a reference to the subterm where the error occurred otherwise. In this case, `record.ana_Dict` checks that each key expression is a Python name that appears in the type index, and then asks `typy` to analyze the value against the corresponding type by calling `ctx.ana`. Finally, it makes sure that all the components specified by the type index have appeared in the literal.

The three values in Listing 1 that `record.ana_Dict` asks `typy` to analyze are also of literal form – the values of `name` and `account_num` are string literals and the value of `memo` is another dictionary literal. As such, when `ctx.ana` is called, `typy` follows the same protocol just described, delegating to `string_in.ana_Str` to analyze the string literals and to `dyn.ana_Dict` to analyze the dictionary literal. The `string_in` fragment implements a regex-based constrained string system, which we described, along with its implementation in `typy`, in a workshop paper [13].[3] The `dyn` fragment allows dynamic Python values to appear inside `typy` programs, so `{ }` is analyzed as an empty Python dictionary.We omit the details of these methods for concision.

py?

cite leroy and mauny dynamics in ML

If typechecking is successful, `typy` must next generate a translation. Again, it does so by calling a class method on the fragment defining the type that the expression is being analyzed against. For example, the `record.trans_Dict` method shown in Figure 3 translates records to Python tuples (the labels do not need to appear in the translation.) This method asks `typy` to determine the translations of the field values, following the same protocol, by calling `ctx.trans`.

[3] Certain details of `typy` have changed since that paper was published, but the essential idea remains the same.

Listing 4 Functions, targeted forms and binary forms.

```
from typy import component
from typy.std import fn
from listing1 import Listing1

@component
def Listing4():
  @fn
  def hello(account : Listing1.Account):
    """Computes a string greeting."""
    name = account.name
    "Hello, " + name

  print(hello(Listing1.test_acct))
```

2.3.2 Function Forms

Listing 4 shows an example of another component, `Listing4`, that defines a `typy` function, `hello`, on Lines 7-11 and then applies it to print a greeting on Line 13. This listing imports the component `Listing1` defined in Listing 1 (we assume that Listing 1 is in a file called `listing1.py`.)

`typy` delegates control over the typechecking and translation of function definitions that appear inside components to the fragment that appears as the top-most decorator.

Here, the `fn` fragment appears as the top-most decorator, so `typy` begins by calling the `fn.syn_FunctionDef` class method, outlined in Listing 5. This method is passed the context and the AST of the function and must manipulate the context as desired and return the type that is to be synthesized for the function, or raise `typy.TyError` if this is not possible. We omit some of the details of this method for concision.

Observe on Lines 7-8 of Listing 5 that `fn` calls `ctx.check` on each statement in the body of the function definition (other than the docstring, following Python's conventions.) This prompts `typy` to follow its delegation protocol for each statement, described in Sec. 2.3.3.

We chose to take the value of the final expression in the body of the function as its return value, following the usual convention in functional languages (an alternative function fragment could use Python-style **`return`** statements.) The synthesized function type is constructed programmatically on Lines 15-16 by pairing the argument types (elided) with the synthesized return type.

If typechecking is successful, `typy` calls the class method `fn.trans_FunctionDef` to generate the translation of the function definition. This method, elided due to its simplicity, recursively asks `typy` to generate the translations of the statements in the body of the function definition by calling `ctx.trans` and inserts the necessary **`return`** keyword on the final statement.

2.3.3 Statement Forms

Statement forms, unlike expression forms, are not classified by types. Rather, `typy` simply checks them for validity when the governing fragment calls `ctx.check` by

For most statement forms, `typy` simply delegates control over validation back to the fragment delegated control over the function that the statement appears within. For example,

Listing 5 A portion of the `fn` fragment.

```
1  class fn(typy.Fragment):
2    @classmethod
3    def syn_FunctionDef(cls, ctx, tree):
4      # (elided) process args + their types
5      # (elided) process docstring
6      # check each statement in remaining body
7      for stmt in tree.proper_body:
8        ctx.check(stmt)
9      # synthesize return type from last stmt
10     last_stmt = tree.proper_body[-1]
11     if isinstance(last_stmt, ast.Expr):
12       rty = ctx.syn(last_stmt.value)
13     else: rty = unit
14     # generate fn type
15     return typy.CanonicalType(
16       fn, (arg_types, rty))
17
18   @classmethod
19   def check_Assign(cls, ctx, stmt):
20     # (details of _process_assn elided)
21     pat, ann, e = _process_assn(stmt)
22     if ann is None:
23       ty = ctx.syn(e)
24     else:
25       ty = ctx.as_type(ann)
26       ctx.ana(e, ty)
27     ctx.ana_pat(pat, ty)
28
29   @classmethod
30   def check_Expr(cls, ctx, stmt):
31     ctx.syn(stmt.value)
32
33   # trans_FunctionDef, trans_Assign & trans_Expr
34   # are elided
```

when `fn.syn_FunctionDef` calls `ctx.check` on the assignment statement on Line 10 of Listing 4, `typy` delegates control back to the `fn` fragment by calling `fn.check_Assign`. Similarly, `fn.check_Expr` handles expression statements, i.e. statements that consist of a single expression, like the expression statement on Line 11 of Listing 4. Let us consider these in turn.

Assignment The definition of `fn.check_Assign` given in Listing 5 begins by extracting a *pattern* and an optional *type annotation* from the left-hand side of the assignment, and an expression from the right-hand side of the assignment.

There is no type ascription in Listing 4, so `fn.check_Assign` asks `typy` to synthesize a type from the expression by calling `ctx.syn`. Here, this expression is of attribute access form – we will describe how `typy` synthesizes a type for expressions of this form in Sec. 2.3.4.

In case where an ascription is provided, `fn.check_Assign` instead asks `typy` to kind check the ascription to produce a type, then it asks `typy` to analyze the expression against that type by calling `ctx.ana` (Lines 25-26.)

Finally, `fn.check_Assign` checks that the pattern matches values of the type that was synthesized or provided as an annotated by calling `ctx.ana_pat`. Patterns of variable form, like `name` in Listing 4, match values of any type. We will see more sophisticated examples of pattern matching in Sec. 2.4. A side effect of calling `ctx.ana_pat` is that the context is updated with the bindings that the pattern introduces.

During translation, `typy` delegates to `fn.trans_Assign`. This method is again omitted because it is straightforward. The only subtlety has to do with shadowing – `fn` follows the functional convention where different bindings of the same name are entirely distinct, rather than treating them as imperative assignments to a common stack location. This requires generating fresh identifiers when shadowing occurs. As with the semantics of return values, a different function fragment could make a different decision in this regard.

Expression Statements The `fn.check_Expr` method, shown in Listing 5, handles expression statements, e.g. the statement on Line 11 of Listing 4, by simply asking `typy` to synthesize a type for the expression, which in Listing 4 is of binary operator form. We will describe how `typy` synthesizes a type for expressions of this form in Sec. 2.3.5.

Other Statement Forms `typy` does not delegate to the fragment governing the enclosing function for statements of function and class definition form that have their own fragment decorator. Instead, `typy` delegates to the decorating fragment, just as at the top-level of a component definition

`typy` also does not delegate to the decorating fragment for statements that 1) assign to an attribute, e.g. `e1.x = e2`; 2) assign to a subscript, e.g. `e1[e2] = e3`; or 3) statements with guards, e.g. **`if`**, **`for`** and **`while`**. These operate as *targeted forms*, described next.

2.3.4 Targeted Forms

Targeted forms include 1) the statement forms just mentioned; 2) expression forms having exactly one sub-expression, like `-e1` or `e1.attr`; and 3) expression forms where there may be multiple sub-expressions but the left-most one is the only one required, like `e1[slices]` or `e1(args)`. When `typy` encounters terms of targeted form, it first synthesizes a type for the target subexpression `e1`. It then delegates control over typechecking and translation to the fragment defining the type of `e1`.

For example, the expression on the right-hand side of the assignment statement on Line 10 of Listing 4 is `account.name`, so `typy` first synthesizes a type for `account`. Due to the type annotation on `hello`, we have that `account` synthesizes type `Listing1.Account` (following the rule for variables, which are tracked by the context.) This type is synonymous with a `record` type, so `typy` first calls the `record.syn_Attribute` class method given in Figure 6. This method looks up the attribute, here `name`, in the type index and returns the corresponding field type, or raises a type error if the attribute is not found in the type index. Here, the synthesized type is `string_in[r".+"]`.

To generate a translation for a targeted form, `typy` follows an analagous delegation protocol. For example, `typy` calls `record.trans_Attribute`, shown in Listing 6, to generate a translation for `account.name`. Because record values translate to tuples, this class method translates field projection to tuple

Listing 6 Targeted forms in `typy`.

```
1 # class record(typy.Fragment):
2   # ... continued from Listing 3 ...
3   @classmethod
4   def syn_Attribute(cls, ctx, idx, e):
5     if e.attr in idx:
6       return idx[e.attr]
7     else:
8       raise typy.TypeError("<bad label>", e)
9
10  @classmethod
11  def trans_Attribute(self, ctx, idx, e):
12    position = position_of(e.attr, idx)
13    return ast.Subscript(
14      value=ctx.trans(e.value),
15      slice=ast.Index(ast.Num(n=position)))
```

projection, using the position of the attribute within the index to determine where the desired field value is located.

2.3.5 Binary Forms

Python's grammar also defines a number of binary operator forms, e.g. `e1 + e2`. One approach for handling these forms would be to privilege the leftmost argument, `e1`, and treat these forms as targeted forms. This approach is unsatisfying because these binary operators are often commutative. For this reason, `typy` instead uses a more sophisticated protocol to determine which fragment is delegated control over binary forms. First, `typy` attempts to synthesize a type for both arguments. If neither argument synthesizes a type, a type error is raised.

If only one of the two arguments synthesizes a type, then the fragment defining that type is delegated control. For example, the binary operator on Line 11 of Listing 4 consists of a string literal on the left (which does not synthesize a type, per Sec. 2.3.1) and a variable, `name`, of type `string_in[r".+"]` on the right, so `string_in` is delegated control over this form.

If both arguments synthesize a type and both types are defined by the same fragment, then that fragment is delegated control. If each type is defined by a different fragment, then `typy` refers to a *precedence tree* to determine which fragment is delegated control. The precedence tree is generated by examining attributes of each fragment named `precedence`. These must be a sets containing other fragments that the defining fragment claims precedence over (if omitted, the precedence set is assumed empty.) `typy` checks that there are no cycles in the graph induced by these precedence sets. If no precedence can be determined, then a type error is raised.

For example, if we would like to be able to add `int`s and `float`s and these are defined by separate fragments, then we can put the necessary logic in either fragment and then place the other fragment in its precendence set.

2.4 Delegated Pattern Matching

3. More Examples

The examples in the previous section are all found in existing languages. In this section, we give examples of novel semantic structures.

3.1 Prototypic Objects

3.2 Foreign Function Interfaces

4. Related Work

Early work proposing the inclusion of compile-time logic in libraries led to the phrase *active libraries* [32]. 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 [26], 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 `typy`, 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 [31] and *metaobject dispatch* [16] 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 [6] or language workbenches [12]) 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 [33] in this way.

Typed Racket and other typed LISPs also add a type system to a fixed syntax atop a dynamically typed core [29]. However, these treat the semantics as a "bag of rules", so adding new rules can cause ambiguities. Our work (par-

ticularly @λ, 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 [14].

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 [19]. 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 [23]. Wyvern is its own language dialect and does not have an extensible type system, supporting only desugarings like SugarJ [10]. The Wyvern formalism guarantees hygiene, using an approach that is also likely applicable in the setting of this paper.

TODO: talk about Python/PHP interop work by that editor guy

TODO: talk about Delaware's work

TODO: look at prior reviews

TODO: Sylvia Grewe's recent OOPSLA(?) work

5. 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 `typy`: 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 `typy` 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.

6. Implementation

We are actively developing the `typy` library and its surrounding ecosystem as a free open source project. The `typy` website can be found at `http://www.cs.cmu.edu/~comar/typy/`.

7. Conclusion

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 [20, 21].

finish this

Second, existing mechanisms are often themselves tied to an obscure language dialect or compiler infrastructure, and so face the classic "chicken-and-egg" problem: they face the same barriers to adoption as other languages.

In this paper, we develop

This design constrains us: we must stay within the confines of Python's existing syntax. In fact, this constraint is freeing: questions about syntactic ambiguity never arise, the syntax is familiar and tools that require an exhaustive syntax specification (e.g. syntax highlighters and documentation generators) work without modification. In other words, this design largely sidesteps the *expression problem* [33].

We show that you do not need an extensible syntax to support an extensible statically-typed semantics. The trick is 1) to organize the semantics around a protocol that delegates control over typechecking and translation of each term to logic associated with some contextually-relevant *type constructor* (*tycon*, for brevity); and 2) to give the ability to define new tycons and the logic associated with them to library providers. We call this protocol **active typechecking and translation** (AT+T).

For example, using `typy`, we will show how a library provider can introduce record types, like those that are built in to languages like ML and Haskell, by defining a tycon called `record`. A different library provider, inspired to give a static semantics to a Javascript-style prototypic object system, can define the active tycon `proto`. We will discuss several other examples throughout the paper.

Types are constructed by applying a tycon to a *type index* using the subscript operator: `tycon[idx]`. For example, we can construct a record type by applying `record` to a mapping from row labels, represented as strings, to types:

```
R = record[
  'label1' : dyn,
# we discuss dyn and string later
  'label2' : string
]
```

Because the language of types, type indices and tycons is Python itself, we are able to use *array slices* to emulate conventional syntax for row specifications, and assignment to define a type synonym, `R`.

The following prototypic object type has an analagous field structure (and for simplicity, the default prototype):

```
P = proto[
  'label1' : dyn,
  'label2' : string
]
```

The first question that one should ask when shown a new type is "how do I introduce a value of this type?" Values of `typy` types are introduced inside *typed functions*, which are those placed under the control of `typy` by the placement of a decorator. Putting the details of typed functions aside for a moment, the form in Python's syntax most similar to the conventional introductory form for records in ML/Haskell is the dictionary literal form, so we would like to be able to introduce values of type `R` inside a typed function using syntax like this:

```
{'label1': val1, 'label2': val2}
```

But this is also the form that one would hope to use to introduce a value of the prototypic object type `P` above. Moreover, we still need to be able to construct Python dictionaries, of type `dyn`.

Inspired directly by our recent work on *type-specific languages* in Wyvern [23], we resolve this sort of ambiguity using *local type inference* [24]. When AT+T encounters an expression of a literal form, it delegates semantic control over it to the tycon of the type that it is being analyzed against. For example, when the expression above appears as an argument to a function `f`, then the argument type of `f` determines its meaning:

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

In positions where there is no expected type, like at the top level of a typed function, the programmer must provide an explicit type ascription. We co-opt Python's array slice syntax for type ascriptions:

```
myRecord = {'label1': val1, 'label2': val2} [: R]
```

Ascriptions can be placed at binding sites as well:

```
myObj [: P] = {'label1': val1, 'label2': val2}
```

The second question one should ask when presented with a new type is "what can I do with a value of this type?" For records, the fundamental operation is row projection. Similarly, for prototypic objects, it is field projection. In both cases, the most natural form in Python's syntax for projecting out a row/field labeled `label1` is `e.label1`, where `e` is the record/object, so we must again come up with some way to resolve the ambiguity. Once again, our solution is to take a tycon-directed approach: AT+T delegates static control to the tycon of the type of `e`.

We will see exactly how tycons like `record` and `proto` statically control typechecking and translation (to standard Python code) of expressions that they are delegated control over in Sec. 8. For now, it suffices to state that the code involved is essentially identical to what one would write if asked to implement a compiler for a dialect that built in the construct in question directly. The novelty of this paper comes not in the constructs themselves, but in how AT+T decentralizes control over the semantics. The locus of control is no longer the language definition but instead individual type constructors, defined in libraries.

Notably, the semantic flexibility of AT+T does not come at the risk of ambiguity because there is always exactly one tycon delegated control over each expression and this delegate remains stable no matter which other tycons have been defined. This stands in contrast to systems that treat the type system as a "bag of rules", e.g. Qi/Shen (a statically typed Lisp dialect) [28].

Organization The remainder of the paper is organized as follows:

- We provide more details on AT+T as realized in `typy` by tracing through a complete program in Sec. 8, covering `typy`'s notion of phase separation, function types,

the delegation protocol, the compilation model, interoperability with Python and various conveniences, including *incomplete ascriptions*.

- We outline more advanced examples of type system fragments that can be implemented as libraries using `typy` in Sec. 9: functional datatypes, as in ML and Haskell, and a typechecked foreign function interface to a low level language, OpenCL.
- We give a minimal specification of AT+T using a typed lambda calculus, @λ, in Sec. 4. The theoretically minded reader may skip ahead to this section first.
- We compare active typing to various related work in Sec. 5.
- We conclude after discussing some limitations and directions for future work in Sec. 6. We also discuss the features necessary to implement active typing within other languages.

8. `typy`

Listing 7 shows an example of a well-typed `typy` program. We will refer to it throughout this section.

Phase Separation The top level of a `typy` program is simply a Python script (versions 2.6+ and 3.0+ are supported). The execution of this script begins the *static phase* of compilation, as suggested by line 4. The main purpose of this phase is to define the tycons and types that will be used by the *typed functions* in the program, marked in this example by the decorator `@fn`. The static phase ends once these typed functions have gone through the process of active typechecking and translation. The generated translation is what ultimately determines the run-time behavior of the program.

Package Management Using Python as `typy`'s static language pays off immediately on the first line: `typy` uses Python's import mechanism directly, so Python's package management tools (e.g. `pip`) and package repostories (e.g. `PyPI`) are directly available for distributing `typy` libraries, including those defining type system fragments. In this case, all of the constructs we will use are part of the `typy` standard library, `typy.std`, though `typy`'s semantics do not fundamentally privilege it. All existing Python libraries can be used from typed functions, as we will discuss later.

Types Types are constructed programmatically in the static phase by applying a type constructor to a type index: `tycon[idx]`. In Listing 7, we construct several types and define short synonyms for them using the standard Python assignment construct:

1. On line 6, we construct a record type, as discussed in Sec. 1, and define the synonym `Account`. The row labels are written as statically valued strings (and could be statically computed rather than written literally, e.g. `'na' + 'me'`).

Listing 7 [`listing7.py`] A `typy` program.

```
1  from typy.std import (record, string, string_in,
2    proto, decimal, dyn, fn, printf)
3
4  print "Hello, static world!"
5
6  Account = record[
7    'name' : string,
8    'account_num' : string_in[r'\d{2}-\d{8}']
9  ]
10
11 Transfer = proto[
12   'amount' : decimal[2],
13   'memo' : dyn,
14   'account' :: Account # this field is the prototype
15 ]
16
17 @fn
18 def log_transfer(t):
19   """Logs a transfer to the console."""
20   {t : Transfer}
21   printf("Transferring %s to %s.",
22     t.amount.to_string, t.name)
23
24 @fn
25 def __main__():
26   account = {
27     'name': "Annie Ace",
28     'account_num': "00-00000001"} [: Account]
29   log_transfer({'amount': 5.50, 'memo': None,
30     'account': account})
31   log_transfer({'amount': 15.00, 'memo': "Rent",
32     'account': account})
33
34 print "Goodbye, static world!"
```

2. The row labeled `'account_num'` has the *regular string type* `string_in[r'\d{2}-\d{8}']`. Values of a regular string type are guaranteed to be in the regular language specified by the type index, a regular expression written as a *raw string literal* (per Python's usual conventions). In this case, this expresses the invariant that the account number must consist of two digits followed by a dash followed by 8 digits. The semantics of this fragment are based on a core calculus we have recently specified in a workshop paper [13].
3. On line 11, we construct a prototypic object type [18]. The field `'amount'` has type `decimal[2]`, which classifies decimal numbers having two decimal places, and the field `'memo'` has type `dyn`, which classifies all dynamically typed Python values.[4] The field labeled `'account'` has type `Account`. It specifies this with a double colon, which `proto` takes to mean that the value of this field should be used as the prototype. This will affect the semantics of the field projection operation: when a field is not present in the object itself, it will defer to the prototype. This allows us to share one account value across multiple transfers,

[4] The phrase "dynamically typed" is widely regarded as as a misnomer from a type theoretic perspective; "dynamically tagged" would be more accurate. We are guests inside Python, however, so we will not wage this battle here.

Listing 8 Tycons are classes during the static phase. They have the opportunity to validate and normalize the type index.

```
1 class record(typy.Type):
2   @classmethod
3   def initialize_idx(cls, idx):
4     # turns slices into a dict for convenience
5     idx = _dict_from_slices(idx)
6     # checks that idx is a map from strings to types
7     _validate_sig(idx) # not shown
8     return idx
9
10  # ... more methods defined in later listings ...
```

while conveniently allowing clients to access its rows transparently, as if they were fields in the object itself. This can be seen as a statically typed variant of Javascript's object system (which is a simplified variant of Self's object system [30]).

Type Constructors Type constructors are classes inheriting from `typy.Type`. For example, a portion of `typy.std.record` is shown in Listing 8. During the static phase, types are instances of these classes. The usual Python instantiation protocol (involving a method named `__init__`) is not used, for reasons that we will cover below. Instead, as just described, types are instantiated using the subscript operator. This is implemented internally using Python's metaclass mechanism to overload the subscript operator on the class object [3]. The tycon can validate and normalize type indices by defining a class method called `initialize_idx`, as shown in Listing 8.

Typed Functions A *typed function* can be defined by decorating a Python function definition with a tycon, here `fn`, as shown on lines 17 and 24 of Listing 7.

Decorator syntax in Python is syntactic sugar for applying the decorator to the function being decorated [3]. Because the decorator value in this case is a tycon, which is a Python class, we must again use Python's metaclasses and operator overloading, here of the call operator, to enable this usage. This is one reason why we must use subscript notation, rather than function application notation, for type construction (another being the usefulness of array slice syntax for record-like types, which is only available inside the subscript form).

The result of this decorator transformation is an instance of `typy.Function`. The underlying Python function is discarded after `typy` extracts its abstract syntax tree and *static environment*, i.e. its closure and the globals dictionary of the Python module it was defined within, using Python's reflection capabilities and the `inspect` and `ast` packages in Python's standard library [3]. The `ast` package exposes Python's parser and abstract syntax tree. The reader is encouraged to refer to its documentation (available online at https://docs.python.org/2/library/ast.html#abstract-grammar) to fully understand the example code in the remainder of this section.

Like standard Python functions, the body of a typed function can (optionally) begin with a *docstring*. Unlike standard Python functions, the line immediately following the docstring specifies a *type signature*, written by repurposing Python's dictionary literal syntax.[5] For example, the signature on line 20 of Listing 7 specifies that the argument `t` of `log_transfer` must have type `Transfer`.

The return type of the function is not specified here, so it will be inferred, as we will describe below. For clarity and in cases where it cannot be inferred, such as when writing a recursive function, the return type can be specified explicitly by a type signature of the following form:

```
{t : Transfer} >> unit
```

Typed functions with no arguments and a return type that can be inferred, like `__main__`, can omit the type signature if desired.

As in languages like ML and Haskell, typed functions in `typy` are themselves values of a function type. This type is constructed by applying the tycon used as the decorator to a tuple consisting of the argument and return types. In this example, `log_transfer` has type `fn[Transfer, unit]`. This function type is analagous to function types in ML and Haskell, e.g. `Transfer -> unit`.[6]

8.1 Active Typechecking

Let us now trace through the process of typechecking the body of the function `__main__`, which constructs a value of type `Account` and then logs two `Transfers` by calling `log_transfer`. In Sec. 8.2 we cover the subsequent active translation process. These two processes together constitute AT+T.

Function Bodies After processing the docstring and type signature, if provided, `typy` immediately hands control over the function body to the tycon provided as the decorator, i.e. `fn` in this example. This happens by one of two methods, depending on whether the return type must be inferred.

If the return type must be inferred, as in Listing 7, the semantics invokes the class method `syn_idx_FunctionDef`, seen on line 2 of Listing 9, passing it the *semantic context* (an object that provides hooks into the semantics and a storage location for accumulating information during typechecking), the *incomplete type index* (here, the pair `(Transfer, Ellipsis)`, where `Ellipsis` stands in for the unknown return type), and the syntax tree of the function definition. This method must return the complete type index (i.e. one without an `Ellipsis`). The method in Listing 9 proceeds as follows:

1. It initializes an *assignables context*, which is a dictionary that tracks assignable variables and their types. It is stored in the semantic context. Its first entries are the function arguments and their corresponding types, extracted from the incomplete type index (by dropping the last element, written `inc_idx[:-1]`).

[5] Python 3 introduced syntax for function type ascriptions, but we do not use it because 1) Python 3 is not yet widely adopted, for the reasons discussed in Sec. 1; and 2) it is awkward when one wishes to introduce parametric polymorphism (an issue we will not discuss at depth in this paper).

[6] As of this writing, we have not decided whether to privilege `typy.std.fn` with a monopoly over comparable infix syntax, e.g. `Transfer >> unit`.

Listing 9 A portion of the type constructor `typy.std.fn`.

```
1  class fn(typy.Type):
2    @classmethod
3    def syn_idx_FunctionDef(cls, ctx, inc_idx, node):
4      if not hasattr(ctx, 'assn_ctx'):
5        ctx.assn_ctx = { } # init. assignables context
6      # add function arguments to assn_ctx
7      ctx.assn_ctx.update(zip(node.args, inc_idx[:-1]))
8      # check each statement in body (post-type sig.)
9      for stmt in node.proper_body: ctx.check(stmt)
10     # synthesize return type from last stmt
11     last_stmt = node.proper_body[-1]
12     if isinstance(last_stmt, ast.Expr):
13       rty = last_stmt.value.ty
14     else: rty = unit
15     # generate complete idx
16     return inc_idx[:-1] + (rty,)
17
18   def ana_FunctionDef(self, ctx, node):
19     arg_tys, rty = self.idx[0:-1], self.idx[-1]
20     if not hasattr(ctx, 'assn_ctx'):
21       ctx.assn_ctx = { } # init. assignables context
22     # add fn name (for recursion) + args to assn_ctx
23     ctx.assn_ctx[node.name] = self
24     ctx.assn_ctx.update(zip(node.args, arg_tys))
25     # check all but last statement in body
26     for stmt in node.proper_body[0:-1]: ctx.check(stmt)
27     last_stmt = node.proper_body[-1]
28     # for last statement, if expr, analyze against rty
29     if isinstance(last_stmt, ast.Expr):
30       ctx.ana(last_stmt.value, rty)
31     # otherwise, rty must be unit or dyn
32     elif rty == unit or rty == dyn:
33       ctx.check(last_stmt)
34     else: raise typy.TypeError("<msg>", last_stmt)
35
36   @classmethod
37   def check_Assign_Name(cls, ctx, stmt):
38     x, e = stmt.target.id, stmt.value # cf. ast docs
39     if x in ctx.assn_ctx: ctx.ana(e, ctx.assn_ctx[x])
40     else: ctx.assn_ctx[x] = ctx.syn(e)
41
42   @classmethod
43   def check_Expr(cls, ctx, stmt):
44     ctx.syn(stmt.value)
45
46   @classmethod
47   def syn_Name(cls, ctx, e):
48     try: return ctx.assn_ctx[e.id]
49     except KeyError:
50       try: return ctx.syn_Lift(ctx.static_env[e.id])
51       except KeyError: raise typy.TypeError("...", e)
52
53   @classmethod
54   def syn_default_asc_Str(cls, ctx, e): return string
```

2. It asks `typy`, by invoking the method `ctx.check`, to typecheck each statement in the body (discussed below).
3. If the abstract form of the final statement is `ast.Expr` (i.e. a top-level expression), then its type is used as the return type (typechecking of expressions is also discussed below). Otherwise it is `typy.std.unit`. Note that this choice of using the last expression as the implicit return value (and considering any statement of a form other than `Expr` to have a trivial value) is made by `fn`, not by `typy`'s semantics.[7]
4. Having determined the return type, the complete type index is generated and returned (by dropping the `Ellipsis` and concatenating a 1-tuple of the return type). `typy` will then use this to construct the function's type as described above.

Had the type signature specified a return type explicitly, then the type index would not need to be inferred. In this case, `typy` constructs the complete type first, i.e. `fn[Transfer, unit]`, then invokes its instance method `ana_FunctionDef`, seen on line 18 of Listing 4. This method proceeds essentially as above, but has access to the complete index as `self.idx`. Thus, if the last statement in the function is an expression, it is analyzed against the provided return type, rather than asked to synthesize a type (see below).

Statements The `ctx.check` method called on each statement in the function body by the logic in Listing 9 is provided by `typy`. As is the pattern, this method delegates to a tycon determined based on the abstract form of the statement being checked. Most statements simply delegate back to the tycon of the function they appear within via a class method named `check_AbsForm`, where `AbsForm` is the name of the statement's abstract form as defined by Python's abstract syntax (or a combination of abstract form names in a few cases where finer distinctions were necessary).

For example, the class method `check_Assign_Name`, seen in Listing 4, is invoked for statements of the form `name = expr`, as on line 26 of Listing 7. In this example, the assignables context (the `assn_ctx` field of the semantic context) is consulted to determine whether the name that is 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 to `assn_ctx`.

The class method `check_Expr`, defined on line 42 of Listing 4, specifies that top-level expressions must synthesize a type.

There are a number of other statement forms in Python's syntax, but `typy` treats them all similarly, so we omit a detailed discussion of the delegation protocol (it is given in the `typy` documentation).

Expressions The methods `ctx.ana` and `ctx.syn` are also provided by `typy`. As we just saw, `ctx.ana` is invoked by tycons to request type analysis when the expected type of an expression is known. The expression and the expected type are provided as arguments. When the type of an expression must be locally inferred, `ctx.syn` is invoked, taking only the expression as an argument and returning the synthesized type. In both cases, ill-typed expressions raise `typy.TypeError`, which is equipped with an error message and a reference to the location where the error originated.

[7] The actual implementation of `fn` supports return statements as well, but we omit this logic here for simplicity.

Like `ctx.check`, these `ctx.ana` and `ctx.syn` methods do very little heavy lifting themselves. Instead, they again delegate control to a tycon by invoking a method with a name based on the abstract form of the expression, either `ana_AbsForm` or `syn_AbsForm`. We cover how the delegate is chosen for various forms below.

Because we distinguish type analysis from type synthesis, the process of active typechecking can be understood as a novel form of *bidirectional typechecking* [24]. The usual *subsumption principle* applies as follows: if analysis is requested and a `syn_AbsForm` method is defined, 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 equal and that their indices are equal, according to Python's usual protocol. Two types with different tycons are never equal, to keep the burden of ensuring that typing respects type equality local to a single tycon.

Let us now discuss how a delegated tycon is chosen for various expression forms in Python's syntax.

Literal Forms Python's syntax includes literal forms for strings, numbers, tuples, lists, dictionaries and lambda functions. As we discussed in Sec. 1, we would like to be able to use these forms for values of types other than `dyn`.

To support this, when `typy` encounters an expression of a literal form in an analytic position, the tycon of the type it is being analyzed against is delegated control. For example, on line 26 of Listing 7, the dictionary literal form appears in an analytic position here because an *explicit type ascription*, written `[: Account]`, was provided. Note that this type ascription form repurposes Python's array slice syntax. Array slices can still be performed, because this form is only treated as an ascription when it contains a valid type.

Because `Account` is a synonym for a record type, `typy` invokes the `ana_Dict` method shown on line 4 of Listing 10. This method analyzes each row value in the literal expression against the row type specified in the type index. Recall that the type index is an unordered mapping from field names to their types, so that type equality is up to reordering. Various error conditions are shown (in practice, the error messages would be more verbose than shown).

The string literal forms inside the outermost record form on line 26 of Listing 7 do not need an ascription, because `record` requests analysis against the appropriate row type. For example, the value of the field `account_num` delegates to the tycon `string_in` via the `ana_Str` method, shown in Listing 12. This method, consistent with the static semantics we have specified in a recent workshop paper, simply statically checks the string against the regular expression provided as the type index [13].

Similary, the prototypic object literals passed to `log_transfer` on lines 29-32 of Listing 7 do not need ascriptions because they will be analyzed by the argument type of `log_transfer`, i.e. `Transfer` (we omit the details of prototypic dispatch for concision; the function application form is discussed below).

Listing 10 A portion of the `typy.std.record` type constructor.

```
1  class record(typy.Type):
2    # ... continued from Listing 2 ...
3
4    def ana_Dict(self, ctx, e):
5      for lbl, value in zip(e.keys, e.values):
6        if isinstance(lbl, ast.Str):
7          if lbl.s in self.idx.rows:
8            ctx.ana(value, self.idx[lbl.s])
9          else: raise typy.TypeError("<not found>", f)
10       else: raise typy.TypeError("<not a label>", f)
11     if len(self.idx.fields) != len(e.keys):
12       raise typy.TypeError("<row missing>", e)
13
14   @classmethod
15   def syn_idx_Dict(self, ctx, inc_idx, e):
16     if inc_idx != Ellipsis:
17       raise typy.TypeError("<bad index>", e)
18     return Signature((lbl.s, ctx.syn(value))
19       if isinstance(lbl, ast.Str)
20       else raise typy.TypeError("<bad label>", f)
21       for lbl, value in zip(e.keys, e.values))
22
23   def syn_Attribute(self, ctx, e):
24     if e.attr in self.idx.labels:
25       return self.idx[e.attr]
26     else: raise typy.TypeError("...", e)
```

Incomplete Type Ascription An ascription placed directly on a literal can also be an *incomplete type*. Incomplete types arise from the application of a tycon to a type index containing an ellipsis (a valid array slice form in Python; the value of an ellipsis is the constant `Ellipsis`). For example, we can write `(e1, e2) [: tuple[...]]`, or equivalently when the index consist solely of an ellipsis, `(e1, e2) [: tuple]`, instead of `(e1, e2) [: tuple[T1, T2]]` when `e1` and `e2` synthesize types `T1` and `T2`, respectively. The tycon is asked to synthesize the complete index via a class method named `syn_idx_AbsForm`, where `AbsForm` is the literal form.

Recall that the process for synthesizing the return type of a typed function also involved the use of an `Ellipsis` (perhaps mysterious at the time). Indeed, a typed function definition missing a return type in its type signature is treated exactly like a function literal ascribed an incomplete function type. This is more explicit when one considers lambda expression literals, e.g. we can write `(lambda x, y: x + y) [: fn[i32, i32, ...]]` when a lambda function is needed in a synthetic position (this is rare in practice; recall that in an analytic position, e.g. when calling a higher-order function, no ascription would be needed at all). More specifically, the default implementation of the `syn_idx_Lambda` calls `syn_idx_FunctionDef`. For `fn`, this method, shown in Listing 4, was discussed earlier.

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

```
{"name": "Deepak Dynamic", "job": "Example"} [: record]
```

The implementation of `syn_idx_Dict` is shown on line 14 of Listing 10. It constructs the record signature by requesting type synthesis for each row value.

This, however, puts us in a situation where the string literals appearing as row values above are in a synthetic position without an ascription. Here, `typy` delegates control to the tycon 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 53 of Listing 4. In this case, we simply return `string`, so the type of the expression above is `record['name': string, 'age': string]`.

A benefit of using Python as our static language is that it is relatively straightforward to define `fn` instead such that it allows for different defaults using block-scoped "pragmas" in the static language via Python's `with` statement [3] (details omitted), e.g.

Listing 11 Block-scoped settings for type constructors.

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

Lowercase Names When the form of an expression is `ast.Name` and the identifier begins with a lowercase character, as when referring to the argument `t` of `log_transfer` or the assignable location `account` in `__main__`, the type constructor governing the function the term appears in is delegated control via the class method `syn_Name`, i.e. such names must synthesize a type. We see the definition of this method for `fn` starting on line 47 in Listing 4. 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 `typy_syn_Lift` method returning an `typy` 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 `typy` 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.

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 23 of Listing 10.

Listing 12 Binary operations in `typy.std.string_in`.

```
1  class string_in(typy.Type):
2    def ana_Str(self, ctx, e):
3      _check_rx(e.s, self.idx)
4
5    @classmethod
6    def syn_idx_Str(cls, ctx, e):
7      return _rx_of_str(e.s) # most specific rx
8
9    # treats string as string_in[".*"]
10   handles_Add_with = set([string])
11   @classmethod
12   def syn_BinOp_Add(cls, ctx, e):
13     rlang_left = self._rlang_from_ty(ctx.syn(e.left))
14     rlang_right = self._rlang_from_ty(ctx.syn(e.right))
15     return string_in[self._concatenate_langs(
16       rlang_left, rlang_right)]
```

Listing 13 For each type constructor definition and binary operator, `typy` 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 typy.AmbiguousTyconError("...",
5         tycon, other_tycon) # (other binops analagous)
```

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 11 of Listing 12.

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 10 of Listing 12. 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 13, 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.

8.2 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 14. 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).

8.3 Standalone Compilation

Listing 15 shows how to invoke the `@` compiler at the shell to typecheck and translate then execute `listing1.py`. The `typy` 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 `typy` 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 16 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.

Listing 14 Translation methods for the types defined above.

```
#class fn(typy.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(typy.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 15 Compiling `listing7.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 16 [`_atout_listing7.py`] The file generated in Listing 15.

```
import typy.std.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__()
```

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.

Listing 17 [listing17.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 18 Execution never proceeds into a function with a type error when using `typy` for implicit compilation.

```
1 $ python listing17.py
2 Today is day 2
3 Traceback (most recent call last):
4   File "listing17", line 9, in <module>
5 typy.TypeError:
6     File "listing17.py", line 7, col 12, in <module>:
7       [record] Invalid field name: nome
```

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 17 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 `typy`, these can be found without executing the code (i.e. we have true static typechecking).

8.4 Interactive Invocation

`typy` 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 17 using the Python interpreter in Listing 18.

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 `typy_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 `typy.std.opencl` below.

9. More Examples

In the previous section, we showed examples of several interesting type system fragments implemented as libraries using `typy`, including functional record types, a prototypic object system and regular string types. A more detailed description of regular string types and their implementation using `typy` was recently published at a workshop [13]. 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.

Listing 19 An example of case types and nested pattern matching.

```
1 from typy import ty
2 from typy.std 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 20 The translation of Listing 19. 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))))
```

9.1 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 19. Here, `Tree` is a *case type*, which is what we call labeled sums in `typy.std`. `Tree` is declared as a *named type* using the `@ty` annotation. This is a general mechanism in `typy` for supporting recursive types that behave generatively, i.e. that 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,

Listing 21 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.

```
class _case(object):
  def syn_Lift(self):
    return caseop[()]
case = _case()

class caseop(typy.Type):
  def syn_Call(self, ctx, e):
    ctx.syn(e.args[0])
    if len(e.args) != 0:
      raise typy.TypeError("...", e)
    return caseop_applied[()]

class caseop_applied(typy.Type):
  def syn_Subscript(self, ctx, e):
    scrutinee = e.value.args[0]
    rules = _rules_from(e.slices)
    ty = None
    for rule in rules:
      bindings = bindings_from_pat(rule, scrutinee.ty)
      _push_bindings(ctx, bindings)
      cur_type = ctx.syn(rule.branch)
      _pop_bindings(ctx, bindings)
      if ty is not None and cur_type != ty:
        raise typy.TypeError("inconsistent", rule)
      else: ty = cur_type
    return ty

  def trans_Subscript(self, ctx, e):
    scrutinee = e.value.args[0]
    return scrutinee.ty.trans_pats(
      _rules_from(e.slices))

class casetype(typy.Type):
  # ...
  def type_pat_Call(self, ctx, p):
    if p.func in self.idx:
      return self.idx[p.func]

  def trans_pats(self, ctx, pats):
    # ... turns sequence of patterns at same level
    # into a conditional expr...

class tuple(typy.Type):
  # ...
  def type_pat_Tuple(self, ctx, p):
    if len(p.elts) == len(self.idx):
      return self.idx
    else raise PatError("invalid pat length")
```

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 `typy.std.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 19 is shown in Listing 20. 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 21. 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 19, this is called, per the discussion earlier on names in the static environment. The result is that `case` synthesizes type `caseop[()]`.

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 `typy` 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.

9.2 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

Listing 22 An example use of our typed FFI to OpenCL, demonstrating both template functions and phaseless capture.

```
1 import typy.std.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)
```

Listing 23 The underlying code generated by `typy.std.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 }
```

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 [17]. 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 22, we show usage of the `typy.std.opencl` package, which implements the entirety of the OpenCL type system (which includes essentially the entirety of C99, plus some

Listing 24 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(typy.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 25 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 typy_syn_Capture(self):
4     return global_ptr[
5       _np_dtypes_to_cltypes(self.dtype)]
```

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.

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 24. 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. 8.4, which allows values to define a method, `typy_syn_Capture`, that permits capture and, in this case, function application with values that would not otherwise support crossing into `typy` 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 25. 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 23. 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.

10. 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 `typy`, and external terms are analagous to terms inside typed functions.

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,

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 $\overline{e}$.

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 [14]), 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

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* = $\mathtt{0}$: **nat**	let(asc[**nat**](lit[lbl[$\mathtt{0}$]](·)); *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*.
$x \cdot \mathtt{rec}(y; \lambda p.\lambda r.r \cdot \mathtt{s})$)	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
$$\frac{\sigma \Downarrow_{\emptyset;\emptyset} \sigma' \quad \sigma'\ \mathtt{ty}_\Phi \quad \Gamma \vdash_\Phi e \Leftarrow \sigma' \rightsquigarrow \iota}{\Gamma \vdash_\Phi \mathsf{asc}[\sigma](e) \Rightarrow \sigma' \rightsquigarrow \iota}$$

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

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

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

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

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

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

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

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

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

combining a bidirectional type system (in the style of Pierce and Turner [24] 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 [15]. 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 [14]. The rules **ATT-ANA-LET** and **ATT-SYN-LET** first synthesize a type for the bound value,

```
Φfn := 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.

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.

These two rules can be compared to the rules in Listing 9. 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

```
Φnat := 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** [14].

```
Φlprod := 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 10).

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 `typy.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 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 $\mathbb{0}$ 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 **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, $\triangleleft(\sigma)$, which is eliminated during normalization (also seen

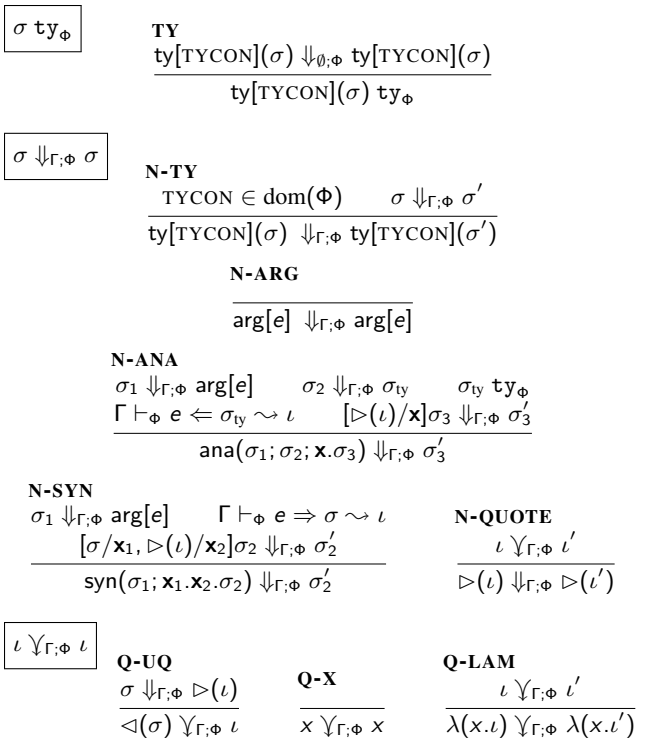

Figure 7. Selected normalization rules for the static language.

in Fig. 7). This function can be compared to the method `ana_Dict` in Listing 10.

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 10.

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. $\mathsf{let}\ \{\mathsf{a}{=}x, \mathsf{b}{=}y\} = r\ \mathsf{in}\ e$. `typy` 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 $\mathtt{ctx}_\Phi$ *and* $\Gamma \vdash_\Phi e \Rightarrow \sigma \rightsquigarrow \iota$ *and* Φ' `fragment` *and* $\mathit{dom}(\Phi) \cap \mathit{dom}(\Phi') = \emptyset$ *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.

References

[1] Flow — a static type checker for javascript. http://flowtype.org/. 1

[2] Purescript. http://www.purescript.org/. 1

[3] The Python Language Reference. http://docs.python.org, 2013. 8, 8, 8.1

[4] G. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In *ECOOP 2014–Object-Oriented Programming*, pages 257–281. Springer, 2014. 1

[5] 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. 1

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

[7] K. Crary. A syntactic account of singleton types via hereditary substitution. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP '09, McGill University, Montreal, Canada, August 2, 2009*, pages 21–29, 2009. 2.2

[8] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2:80–86, 1976. 1

[9] J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In G. Morrisett and T. Uustalu, editors, *ICFP*, pages 429–442. ACM, 2013. 2.3

[10] S. Erdweg, T. Rendel, C. Kastner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*, pages 187–188, 2011. 4

[11] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12, 2013. 1

[12] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. The state of the art in language workbenches. In *Software Language Engineering (SLE '13)*. 2013. 1, 4

[13] 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 '14)*, 2014. 2.3.1, 2, 8.1, 9

[14] R. Harper. *Practical Foundations for Programming Languages*. Second edition, 2015. (Working Draft, Retrieved Nov. 19, 2015). 2.2, 4, 10, 10, 5

[15] R. Harper and C. Stone. A Type-Theoretic Interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. 1, 10

[16] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991. 4

[17] 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. 9.2

[18] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, Oct. 1986. 3

[19] 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. 4

[20] L. A. Meyerovich and A. Rabkin. How not to survey developers and repositories: experiences analyzing language adoption. In *PLATEAU*, 2012. 7

[21] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *OOPSLA*, 2013. 1, 7

[22] Odersky, Zenger, and Zenger. Colored local type inference. In *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001. 1

[23] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP '14*, 2014. 4, 7

[24] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000. 1, 2.3, 7, 8.1, 10

[25] 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. 1

[26] 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. 4

[27] C. A. Stone and R. Harper. Deciding type equivalence in a language with singleton kinds. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) , Boston, Massachusetts*, pages 214–227, Jan. 2000. B

[28] M. Tarver. *Functional programming in Qi*. Free University Press, 2008. 7

[29] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL*, 2008. 4

[30] D. Unger and R. B. Smith. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, Dec. 1987. 3

[31] 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. 4

[32] 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. 4

[33] P. Wadler. The expression problem. *java-genericity mailing list*, 1998. 1, 4, 7

[34] M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994. 1

A. Support for Python 2.x

B. Type Equality

Although not necessary in this example, fragments can compare type expressions for equivalence [27]. Two types in canonical form are equivalent if they arise from the same fragment and have equivalent index values (as determined by Python's == operator, which can be overloaded to introduce interesting equivalences.) For example, we can freely reorder the fields in the definition of `Account` because Python dictionaries are not order-preserving.