Programmable Semantic Fragments

The Design and Implementation of typy

Cyrus Omar Jonathan Aldrich

Carnegie Mellon University
Pittsburgh, PA, USA
{comar, aldrich}@cs.cmu.edu

Abstract

This paper introduces typy, a statically typed programming language embedded by reflection into Python. typy features a *fragmentary semantics*, i.e. it delegates semantic control over each term, drawn from Python's fixed concrete and abstract syntax, to some contextually relevant user-defined *semantic fragment*. The delegated fragment is tasked with programmatically 1) typechecking the term (following a bidirectional protocol); and 2) assigning dynamic meaning to the term by translation to Python.

We argue that this design is *expressive* with examples of fragments that express the static and dynamic semantics of 1) functional records; 2) labeled sums (with nested pattern matching *a la* ML); 3) a variation on JavaScript's prototypal object system; and 4) typed foreign interfaces to Python and OpenCL. These semantic structures are, or would need to be, defined primitively in conventionally organized languages.

We further argue that this design is *compositionally well-behaved*. It avoids the expression problem and the problems of grammar composition because the syntax is fixed. Moreover, programs are semantically stable under fragment composition (i.e. defining new fragments will not change the meaning of existing program components.)

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

Keywords metaprogramming; bidirectional typechecking; pattern matching; foreign function interfaces; active libraries

1. Introduction

As programming languages proliferate, programmers face the daunting problem of *lateral compatibility*, i.e. of interfacing with libraries written in sibling languages. For example, there are useful libraries written in TypeScript [11], Flow [1] and PureScript [3], but these libraries are not directly accessible across language boundaries because these languages are all syntactically and semantically incompatible with one another. (The first two define different object systems, and PureScript is a functional language similar to Haskell and ML.)

One common workaround is to interface indirectly with libraries written in sibling languages through the code generated by a compiler that targets a more established language for which a *foreign interface (FI)* is available. For example, all of the languages above have compilers that target JavaScript and they are all capable of interfacing with JavaScript. Unfortunately, this approach is unnatural (the syntactic and semantic conveniences of the sibling language are unavailable) and unsafe (the type system of the sibling language is not enforced, and the internal representations of the compiler are exposed.) This problem can, at best, be mitigated by inserting dynamic checks at language boundaries [44]. It appears then that this language-oriented approach [70] is difficult to reconcile with the best practices of "programming in the large" [22].

In this paper, we propose a more compositional *fragment-oriented* approach to the problem of expressing new semantic structures. In particular, we introduce a single "extensible" statically typed language, typy, that allows library providers to define new semantic structures as *semantic fragments*. Library clients can import these fragments in any combination.[1] For example, we will consider a fragment that expresses the static and dynamic semantics of functional records, and another fragment that expresses the static and dynamic semantics of a prototypal object system.

This fragment-oriented approach diminishes the need for standalone languages – clients of a library that requires the use of, for example, functional records at its public interface can simply import the record fragment themselves, even if they otherwise prefer using an object system. Moreover, when interacting with libraries in a foreign language *is* necessary, the fragment system helps address the lateral compatibility problem by allowing library providers to implement a natural, type-safe foreign interface as a library. For example, we will define a type-safe foreign interface to OpenCL (a low-level language for working with GPUs, similar to CUDA [36].)

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

Although this vision has long been appealing, designing an extensible language equipped with useful composition principles presents several well-known challenges.

First, consider that language designers typically have the ability to define concrete forms specific to the semantic structures that they introduce, but if we give fragment providers the same ability (following, e.g., Sugar* [25]), then different fragments could define conflicting forms. For example, consider the following family of forms:

$$\{\ \mathtt{label}_1\mathtt{:}\ \mathtt{expr}_1,\ \ldots,\ \mathtt{label}_n\mathtt{:}\ \mathtt{expr}_n\ \}$$

One fragment might take these as the introductory forms for functional records, while another fragment might take these as the introductory forms for TypeScript-style objects. These forms might also conflict with those for Python-style dictionaries. Such syntactic conflicts inhibit composition.

We also encounter the classic *expression problem* [55, 69]: if fragment providers can define new term constructors in a decentralized manner, then it is difficult to define functions that proceed by exhaustive case analysis, e.g. pretty-printers.

Finally, we must maintain essential semantic properties, like type safety (in the sense of Milner [46].) Moreover, clients should be able to assume that importing a new fragment for use in one portion of a program will not change the meaning of other portions of the program, nor allow the program to take on ambiguous meaning. This implies that we cannot simply operationalize the semantics as a "bag of rules" that fragment providers freely extend.

The `typy` semantic fragment system addresses the problems of concrete and abstract syntax quite simply: fragment providers are not given the ability to extend `typy`'s concrete or abstract syntax (which is borrowed unchanged from Python.) Instead, the system allows fragments to "share" syntactic forms by delegating semantic control over each term to some contextually relevant fragment definition. For example, `typy` delegates control over terms of curly-brace delimited form (above) to the fragment that defines the type that the term is being checked against. This fragment is responsible for 1) typechecking the term; and 2) assigning dynamic meaning to the term by translation to a target language (which we take to be Python.) As such, curly-brace delimited forms can serve as introductory forms for records, objects and dictionaries.

The `typy` fragment system also addresses the semantic problems just discussed. By defining the dynamics by translation, the problem of maintaining type safety reduces ultimately to the problem of type safety for the target language. Moreover, the delegation protocol is deterministic, so ambiguities cannot arise. It is also stable under fragment composition, so defining a new fragment cannot change the meaning of an existing program component.

The remainder of the paper is organized as follows. Sec. 2 introduces `typy`'s fragment system with simple examples. Sec. 3 then describes more sophisticated examples. Sec. 4 summarizes related work. Sec. 5 concludes with a discussion of present limitations and future work.

Listing 1 Types and values in `typy`.

```
1  from typy import component
2  from typy.std import record, string_in, py
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        : py
10   ]
11
12   test_acct [: Account] = {
13     name:        "Harry Q. Bovik",
14     account_num: "00-12345678",
15     memo:        { }
16   }
```

2. Semantic Fragments in typy

Listing 1 gives 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 (see Sec. 5 for a discussion.)

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 that we will detail 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 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 type member definitions, described in Sec. 2.2, and value member definitions, described in Sec. 2.3. The return value of the decorator

[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 paper.

Listing 2 Record type validation.

```
import ast, typy
class record(typy.Fragment):
  @classmethod
  def init_idx(cls, ctx, idx_ast):
    if isinstance(idx_ast, ast.Slice):
      # Python special cases single slices
      # we don't want that
      idx_ast = ast.ExtSlice(dims=[idx_ast])
    if isinstance(idx_ast, ast.ExtSlice):
      idx_value = dict() # returned below
      for dim in idx_ast.dims:
        if (isinstance(dim, ast.Slice) and
            dim.step is None and
            dim.upper is not None and
            isinstance(dim.lower, ast.Name)):
          lbl = dim.lower.id
          if lbl in idx_value:
            raise typy.TypeFormationError(
              "Duplicate label.", dim)
          ty = ctx.as_type(dim.upper)
          idx_value[lbl] = ty
        else: raise typy.TypeFormationError(
          "Invalid field spec.", dim)
      return idx_value
    else: raise typy.TypeFormationError(
      "Invalid record spec.", idx_ast)
```

is an instance of `typy.Component` that tracks 1) the identities of type members; and 2) the types and evaluated translations of value members. (Luckily, Python chose the "generic" **`def`** keyword – had it chosen, e.g. `fun`, this might be less clean, because components are not functions.)

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* first developed for the ML module system [18, 32], which elegantly handles the details of type synonynms, type members and parameterized types (we will define a parameterized type in Listing 7.) 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 follows:

```
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 in canonical form.

To establish that a type in canonical form is valid, i.e. of kind **`type`**, `typy` calls the fragment class method `init_idx`. This method receives the *context* and the AST of the index and 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.

For example, the `record.init_idx` class method shown in Listing 2 validates record types by checking that 1) the index consists of a sequence of field specifications of the form `name : ty_expr`, where `name` is a Python name; 2) no names are duplicated; and 3) `ty_expr` is a valid type expression, as determined by calling a method of the context object, `ctx.as_type` (Line 20.) This method turns the given Python AST into a type expression, i.e. an instance of (a class that inherits from) `typy.TyExpr`, and checks that it is of kind **`type`**. The returned index value is a Python dictionary mapping the field names to the corresponding instances of `typy.TyExpr`.

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.

A type annotation is not always 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* [16, 53]. Scala is another notable language that has a bidirectional type system, albeit of different design [49]. Our system is based on the system developed by Dunfield and Krishnaswami [23]. Again, we will not repeat standard details here – our interest in the remainder of this section will be only in how `typy` delegates control to some contextually relevant fragment according to the `typy` *delegation protocol*.

2.3.1 Literal Forms

`typy` delegates control over the typechecking and translation of terms 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, so `typy` first calls the `record.ana_Dict` class

Listing 3 Typing and translation of literal forms.

```
# 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):
    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 sorted(idx.keys()))
```

method, shown in Listing 3. This method receives the context, the index value computed by `record.init_idx` and the AST of the literal term. It must return (trivially) if type analysis 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 index value, and then asks `typy` to analyze the value against the corresponding type by calling `ctx.ana`. Finally, it makes sure that all of the components specified in the index value appear in the literal.

The three subexpressions 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 `py.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 [29].[3] The `py` fragment allows dynamic Python values to appear inside `typy` programs, so `{ }` is analyzed as an empty Python dictionary. Supplemental details about Python interoperability are given in Appendix B.

If typechecking is successful, `typy` delegates to the same fragment to generate a translation, i.e. another Python AST. For example, `typy` calls the `record.trans_Dict` method shown in Listing 3, which translates records to Python tuples (the field names are needed only statically.) The field ordering is alphabetical. This method asks `typy` to determine translations for the field values by calling `ctx.trans`, which again follows the delegation protocol (`typy` stores the types determined during typechecking as attributes of the AST nodes.)

[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.

```
1  from typy import component
2  from typy.std import fn
3  from listing1 import Listing1
4
5  @component
6  def Listing4():
7    @fn
8    def hello(account : Listing1.Account):
9      """Computes a string greeting."""
10     name = account.name
11     "Hello, " + name
12
13   print(hello(Listing1.test_acct))
```

2.3.2 Definition Forms

Listing 4 shows an example of another component, `Listing4`, that defines a 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.

`typy` delegates control over the typechecking and translation of **`def`**inition forms that appear inside components, or in other synthetic positions, to the fragment that appears on the form as the first decorator.

Here, the `fn` fragment is the first (and only) 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 initialize 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, but observe on Lines 8-9 of Listing 5 that `fn` calls `ctx.check` on each statement in the function body (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 below.

We chose to take the value of the final expression in the function body as its return value, following the usual convention in functional languages (an alternative function fragment could instead use Python-style **`return`** statements.) The synthesized function type is constructed programmatically on Lines 17-18. The index value is a pair of the argument types (extracted from the type annotations, not shown) 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.

For definition forms decorated by a type expression rather than a fragment, or those in other analytic positions, `typy` treats the function definition as a literal form. We will see examples of these in Sec. 3.

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     ctx.push_bindings({}) # new bindings layer
7     # check each statement in remaining body
8     for stmt in tree.proper_body:
9       ctx.check(stmt)
10    # synthesize return type from last stmt
11    last_stmt = tree.proper_body[-1]
12    if isinstance(last_stmt, ast.Expr):
13      rty = ctx.syn(last_stmt.value)
14    else: rty = unit
15    ctx.pop_bindings() # pop local bindings
16    # generate fn type
17    return typy.CanonicalType(
18      fn, (arg_types, rty))
19
20  @classmethod
21  def check_Assign(cls, ctx, stmt):
22    # (details of _process_assn elided)
23    pat, ann, e = _process_assn(stmt)
24    if ann is None:
25      ty = ctx.syn(e)
26    else:
27      ty = ctx.as_type(ann)
28      ctx.ana(e, ty)
29    bindings = ctx.ana_pat(pat, ty)
30    ctx.add_bindings(bindings)
31
32  @classmethod
33  def check_Expr(cls, ctx, stmt):
34    ctx.syn(stmt.value)
35
36  # trans_FunctionDef, trans_Assign & trans_Expr
37  # are elided
```

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`.

For most statement forms, `typy` simply delegates control over validation and translation back to the fragment that was delegated control over the definition that the statement appears within. For example, 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, like the one 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.

No type annotation appears on the assignment in Listing 4, so `fn.check_Assign` asks `typy` to synthesize a type from the expression by calling `ctx.syn` (Line 25 of Listing 5.) We will describe how `typy` synthesizes a type for the expression `account.name` in Sec. 2.3.4 below.

In cases where an annotation 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 27-28 of Listing 5.)

Finally, `fn.check_Assign` checks that the pattern matches values of the type that was synthesized or provided as an annotation 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 below. The `ctx.add_bindings` method adds the bindings (here, a single binding) to the typing context.

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 distinct, rather than treating them as imperative assignments to a common stack location. This requires generating a fresh name when a name is reused (`ctx.add_bindings` does this by default.) As with the semantics of return values, a different function fragment could make a different decision in this regard by managing the context manually.

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. In Listing 4, this expression is of binary operator form – we will describe how `typy` synthesizes a type for expressions of this form in Sec. 2.3.5 below.

Other Statement Forms `typy` does not delegate to the fragment governing the enclosing definition for statements of definition form that have their own fragment or type decorator. Instead, `typy` delegates to the decorating fragment, just as at the top-level of a component definition. The fragment governing the enclosing function determines only how the translation is integrated into its own translation (through a `integrate_trans_FunctionDef` method, omitted for concision.)

`typy` also does not delegate to the decorating fragment for statements that 1) assign to an attribute, e.g. `e1.x = e2` or `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 subexpression, like `-e1` or `e1.attr`; and 3) expression forms where there may be multiple subexpressions but the left-most one is the only one that is syntactically required, like `e1(args)` (there may be no arguments.) 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`. Following the standard rule for variables, which are tracked by the context,

Listing 6 Typing and translation of targeted forms.

```
# class record(typy.Fragment):
  # ... continued from Listing 3 ...
  @classmethod
  def syn_Attribute(cls, ctx, idx, e):
    if e.attr in idx:
      return idx[e.attr]
    else:
      raise typy.TypeError("<bad label>", e)

  @classmethod
  def trans_Attribute(cls, ctx, idx, e):
    pos = _pos_of(e.attr, sorted(idx.keys()))
    return ast.Subscript(
      value=ctx.trans(e.value),
      slice=ast.Index(ast.Num(n=pos)))
```

we have that `account` synthesizes type `Listing1.Account`. This type is synonymous with a `record` type, so `typy` first calls the `record.syn_Attribute` class method given in Listing 6. This method looks up the attribute, here `name`, in the type's index value and returns the corresponding field type, here `string_in[r".+"]`, or raises a type error if it is not found.

To generate the translation for `account.name`, `typy` calls `record.trans_Attribute`, shown in Listing 6. Because record values translate to tuples, this method translates record field projection to tuple projection, using the position of the attribute within the record type's index value to determine the appropriate slice index.

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 binary operators are often commutative. Instead, `typy` defines a symmetric protocol to determine which fragment is delegated control over binary forms. First, `typy` tries 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 the *precedence sets* of each fragment to determine which fragment is delegated control. The precedence sets are Python sets listed in the `precedence` attribute of the fragment that contain other fragments that the defining fragment claims precedence over (if omitted, the precedence set is assumed empty.) `typy` checks that if one fragment claims precedence over another, then the reverse is not the case (i.e. precedence is anti-symmetric, to maintain determinism.)

Listing 7 Polymorphism, recursion and pattern matching in `typy`. An analagous OCaml file is shown in Appendix C.

```
1  from typy import component
2  from typy.std import finsum, tpl, fn
3  @component
4  def Listing7():
5      # polymorphic recursive finite sum type
6      tree(+a) [: type] = finsum[
7        Empty,
8        Node(tree(+a), tree(+a)),
9        Leaf(+a)
10     ]
11     # polymorphic recursive function over trees
12     @fn
13     def map(f : fn[+a, +b],
14             t : tree(+a)) -> tree(+b):
15       [t].match
16       with Empty: Empty
17       with Node(left, right):
18         Node(map(f, left), map(f, right))
19       with Leaf(x):
20         Leaf(f(x))
```

Precedence is not transitive. If a precedent fragment is found, it is delegated control. Otherwise, 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 Fragmentary Pattern Matching

As we saw on Line 29 of Listing 5, fragments can request that `typy` check that a given *pattern* matches values of a given type by calling `ctx.ana_pat`. In the example in Listing 4, the pattern was simply a name – name patterns match values of any type. In this section, we will consider other patterns. For example, the statement below uses a tuple pattern:

```
(x, y, z) = e
```

`typy` also supports a more general `match` construct, shown on Lines 15-20 of Listing 7. This construct, which spans several syntactic statements, is treated as a single expression statement by `typy`. The *scrutinee* is `t` and each *clause* is of the form **`with`** `pat: stmts` where `pat` is a pattern and `stmts` is the corresponding *branch*. `typy` also supports an analagous expression-level match construct, shown in Appendix D.

To typecheck a match expression, `typy` first synthesizes a type for the scrutinee. Here, the scrutinee, `t`, is a variable of type `tree(+a)`. This type is an instance of the parameterized recursive type `tree` defined on Lines 6-10 (the mechanisms involved in defining recursive and parameterized types are built into `typy` in the usual manner.) Type variables prefixed by `+`, like `+a` and `+b`, implicitly quantify over types at the function definition site (like `'a` in OCaml [40].)

More specifically, `tree(+a)` is a *recursive finite sum type* defined by the `finsum` fragment imported from `typy.std` [32]. This fragment is defined such that values of finite sum type translate to Python tuples, where the first element is a string *tag* giving one of the names in the type index and the remaining elements are the corresponding values. For example,

Listing 8 Typing and translation of patterns.

```
1 import ast, typy
2 class finsum(typy.Fragment):
3   # ...
4   @classmethod
5   def ana_pat_Call(cls, ctx, idx, pat):
6     if (isinstance(pat.func, ast.Name) and
7         pat.func.id in idx and
8         len(pat.args) == len(idx[pat.func.id])):
9       bindings, lbl = {}, pat.func.id
10      for p, ty in zip(pat.args, idx[lbl]):
11        _combine(bindings, ctx.ana_pat(p, ty))
12      return bindings
13    else:
14      raise typy.TyError("<bad pattern>", pat)
15
16  @classmethod
17  def trans_pat_Call(cls, ctx, idx, pat,
18                     scrutinee_tr):
19    conditions = [
20      ast.Compare(left=_prj(scrutinee_tr, 0),
21        ops=[ast.Eq()],
22        comparators=[ast.Str(s=pat.func.id)])
23    ]
24    binding_translations = {}
25    for n, p in enumerate(pat.args):
26      arg_scrutinee = _prj(scrutinee_tr, n+1)
27      c, b = ctx.trans_pat(p, arg_scrutinee)
28      conditions.append(c)
29      binding_translations[pat.func.id] = b
30    condition = ast.BoolOp(op=ast.And(),
31      values=conditions)
32    return (condition, binding_translations)
33
34 def _prj(e, n): # helper function
35   return ast.Subscript(value=e,
36     slice=ast.Index(ast.Num(n=n)))
```

a value `Node(e1, e2)` translates to `("Node", tr1, tr2)` where `tr1` and `tr2` are the translations of `e1` and `e2`. Names and call expressions beginning with a capitalized letter are initially treated as literal forms in `typy` (following Haskell [34].) If the delegated fragment does not define their semantics, they are then treated as targeted forms.

`typy` delegates control over patterns to the fragment that defines the scrutinee type. For example, to check the pattern `Node(left, right)` on Line 16, `typy` calls `finsum.ana_pat_Call`, shown in Listing 8. This method must either return a dictionary of *bindings*, i.e. a mapping from variables to types, which `typy` adds to the typing context when typechecking the corresponding branch expression, or raise a type error if the pattern does not match values of the scrutinee type. In this case, `finsum.ana_pat_Call` first checks to make sure that 1) the name that appears in the pattern appears in the type index (for `finsum` types, this is a mapping from names to sequences of types); and 2) that the correct number of pattern arguments have been provided. If so, it asks `typy` to check each subpattern against the corresponding type. Here, `left` and `right` are both checked against `tree(+a)`. These happen to be variable patterns, but `typy` supports arbitrarily nested patterns. The returned dictionary of bindings is constructed by combining the two dictionaries returned by these calls to `ctx.ana_pat`. The `_combine` function, not shown, also checks to make sure that the bound variables are distinct.

Match expression statements translate to Python **`if`**...**`elif`** statements. For each clause, `typy` needs a boolean *condition expression*, which determines whether that branch is taken, and for each binding introduced by that clause, `typy` needs a translation. To determine the condition and the binding translations, `typy` again delegates to the fragment defining the scrutinee type, here by calling `finsum.trans_pat_Call`, given in Listing 8. This class method is passed the context, the type index, the pattern AST and an AST representing the scrutinee (bound to a variable, to avoid duplicating effects.) The generated condition expression first checks the tag. Then, for each, subpattern, it recursively generates its conditions and binding translations by calling `ctx.trans_pat(p, arg_scrutinee)`, where `arg_scrutinee` makes the new "local scrutinee" for the subpattern be the corresponding projection out of the original scrutinee. The returned condition expression is the conjunction of the tag check and the subpattern conditions. For example, for `Node(left, right)`, the condition expression is `scrutinee[0] == 'Node'`**`and`** `True` **`and`** `True`. The two `True`s are because variables always match (these are optimized away.)

The delegated fragment also has responsibility for checking exhaustiveness, via the class method `is_exhaustive` (not shown.) It can call `ctx.is_exhaustive(pats, ty)` to check the exhaustiveness of another sequence of patterns.

2.5 Determinism and Stability

We argue that the `typy` delegation protocol is compositionally well-behaved, i.e. it exhibits *determinism* and *stability under fragment composition*. By determinism, we mean that under a given context, there is always a single fragment that can be delegated control over any type expression, statement, expression or pattern form, i.e. there can be no ambiguity. By *stability*, we mean that the delegation protocol will not make a different choice simply because a new fragment has been added to the *fragment context* (the set of fragments in the static environment.) A virtue of the design we have presented is that these properties follow essentially immediately. We contrast this with related work in Sec. 4.

Consider type validation (Sec. 2.2): the fragment delegated control over `fragment[idx]` is `fragment`. The choice is explicit in the term, so determinism and stability follow trivially.

For literal forms (Sec. 2.3.1), the fragment defining the type provided for analysis is delegated control. To establish determinism and stability, we need only establish that type normalization is deterministic and stable. Our language of type expressions is a standard deterministic lambda calculus [18] and normalization interacts with the fragment context only at canonical form, which was just discussed.

For function defintions (Sec. 2.3.2), the delegated fragment is the first decorator. Most statement forms also delegate to this fragment (Sec. 2.3.3.) The delegate is again stated explicitly, so the protocol is trivially deterministic and stable.

Listing 9 Prototypal objects in `typy`.

```
1  from typy import component
2  from typy.std import proto, decimal, fn, unit
3  from listing1 import Listing1
4
5  @component
6  def Listing9():
7    Transaction [: type] = proto[
8      amount : decimal,
9      incr   : fn[Transaction, unit]
10     proto  : Listing1.Account
11   ]
12
13   @Transaction
14   def test_trans():
15     amount = 36.50
16     def incr(self): self.amount += 1
17     proto = Listing1.test_acct
18
19   test_trans.incr() # self passed automatically
20   print(test_trans.name) # prints Harry Q. Bovik
```

For targeted terms (Sec. 2.3.4), `typy` synthesizes a type for the target. For binary terms (Sec. 2.3.5), `typy` also synthesizes types for sub-terms. For determinism and stability to hold, then, we need that type synthesis, implemented by `ctx.syn`, is deterministic and stable. This is a straightforward inductive argument, with the base case being variable forms. Variables are tracked by the variable context, which assigns each variable a unique type, so determinism holds. Variables lookup is independent of the fragment context, so stability holds. For binary forms, the only remaining requirement is that the possibilities described in Sec. 2.3.5 are mutually exclusive and do not depend on the fragment context, which is apparent by inspection.

Finally, for patterns (Sec. 2.4), the fragment defining the type that the scrutinee synthesizes is delegated control. Again, determinism and stability follow from determinism and stability of type synthesis.

3. More Examples

In this section, we will further demonstrate the flexibility of `typy`'s fragment system with some more complex examples.

3.1 Prototypal Object Types

JavaScript's object system supports *prototypal inheritance* (based on a similar mechanism in the Self language [42, 64].) We have implemented a statically typed variant of this system as a fragment, `typy.std.proto`.

Listing 9 defines a *prototypal object type*, `Transaction`, that specifies fields named `amount`, `incr` and `proto`. We introduce a value of this type using the **`def`** form on Lines 13-17 (i.e. this form is treated as a literal form, per Sec. 2.3.2, so `typy` calls `proto.ana_FunctionDef`.) The inner **`def`** form, on Line 16, is governed by the `fn` fragment because `proto.ana_FunctionDef` analyzes it against a `fn` type (i.e. it also behaves as a literal.) As such, no type annotations or decorators are needed.

The fields of a prototypic object are mutable, e.g. as shown in the body of `incr` on Line 16. The delegation protocol treats an assignment of this form as a targeted form, per Sec. 2.3.3.

Listing 10 `numpy` and OpenCL in `typy`.

```
1  from typy import component
2  from typy.numpy import array, f64
3  from typy.cl import buffer, to_device, kernel
4
5  @component
6  def Listing10():
7    # (device selection code elided)
8    # make numpy array + send to device
9    x [: array[f64]] = [1, 2, 3, 4]
10   d_x = to_device(x) # device buffer
11
12   # define a typed data-parallel OpenCL kernel
13   @kernel
14   def add5(x : buffer[f64]):
15     gid = get_global_id(0) # OpenCL primitive
16     x[gid] = x[gid] + 5
17
18   # spawn one device thread per element and run
19   add5(d_x, global_size=d_x.length)
20
21   y = d_x.from_device() # retrieve from device
22   print(y.to_string()) # prints [6, 7, 8, 9]
```

On Line 19, we call the `incr` method. The `proto` fragment implicitly passes in the target of the method call as the first argument, as in Python and similar to JavaScript.

When a field is not found in the object itself, e.g. `name` on Line 20, the `proto` fragment delegates to the `proto` field. Here, the prototype is the record value `Listing1.test_acct`.

3.2 Low-Level Foreign-Function Interfaces

Python is widely used in scientific computing [50]. One reason is that Python has support for calling into low-level languages like C (e.g. via SWIG [9].) Many popular libraries, e.g. `numpy` [65], operate essentially as wrappers around low-level routines written in these languages. It is also possible to dynamically compile low-level code generated by Python code as a string. This is particularly useful when working with GPUs and other compute devices, e.g. using the PyCUDA and PyOpenCL libraries [36].

We have designed fragments that allow for statically typed access to these libraries. For example, on Line 9 of Listing 10, we create a typed `numpy` array of 64-bit floating point numbers. The `typy.numpy.array` fragment supports the use of list literal syntax to do so. As such, the cost of the type annotation is "canceled out" because we don't need to explicitly call `numpy.array` as one does in Python. For arrays in analytic position (e.g. as function arguments), this interface to `numpy` is therefore of lower syntactic cost.

On Line 10, we invoke the `to_device` operator to transfer the `numpy` array to the compute device's memory (we omit the code needed once per session to select a device.)

On Lines 13-16, we then define a typed OpenCL kernel [4]. An OpenCL kernel is simply an OpenCL function that is called in a *data parallel* manner, i.e. a large number of threads are spawned, all running the same kernel. Each kernel has access to a unique ID, called the *global ID* in OpenCL. Here, `add5` determines its global ID and then adds `5` to the

corresponding element in the input buffer. Notice that we did not need to specify a return type or a type annotation on gid, because typy is bidirectionally typed (unlike OpenCL.) The translation of the definition of add5 uses a Python encoding of OpenCL ASTs. It is equivalent to the following Python code (assuming a variable ctx, which our library tracks implicitly):

```
add5 = pyopencl.Program(ctx, """
  __kernel void add5(__global double* x) {
    size_t gid = get_global_id(0);
    x[gid] = x[gid] + 5;
  }""").build()
```

The typy code is again more concise. Moreover, type errors *in the OpenCL kernel* are detected ahead-of-time by typy. This required us to implement the entirety of the OpenCL type system using typy's fragment system, including the logic of numeric type promotion and various other subtleties inherited from C. This represents the largest case study to date of our methodology. Interestingly, we were also able to extend OpenCL with various higher-level constructs, e.g. pattern matching and sum types, essentially as described in Sec. 2. In fact, in most cases we inherit from the original fragment, overriding only the translation methods.

Line 19 invokes the add5 kernel in a data parallel fashion on the device buffer d_x. The parameter global_size determines the number of threads – here, one thread per array element. Finally, Lines 21-22 retrieve the result from the device and print out the result.

The details of the various fragments just described, are, of course, somewhat involved. The takeaway lesson, however, is that as the designers of typy, we did not need to anticipate this particular mode of use. A similar approach could be taken, independently, for CUDA, C and even higher-level languages for which a foreign interface is available in Python, like Java. In contrast, monolithic languages like MLj needed to primitively build in a type safe foreign interface [10].

4. Related Work

Our recent work on *type-specific languages (TSLs)* in the Wyvern language used a bidirectionally typed protocol to delegate control over the parsing of literal forms to functions associated with type definitions [52]. This inspired our treatment of literal forms in typy. Unlike Wyvern, typy's literal forms are parsed according to Python's fixed syntax. Unlike typy, Wyvern has a monolithic semantics. Both mechanisms could exist in the same language, but presently do not.

Like typy, Typed Racket is a statically typed language embedded into a dynamic language [19, 63]. Typed Racket is not modularly extensible.

Language-external mechanisms for creating and combining language dialects, e.g. extensible compilers like Xoc [17], JastAdd [24], Polyglot [48], JaCo [71], Silver [67] and various "language workbenches" [26], do not guarantee determinism. In particular, these systems presume that new language constructs define new textual forms. These forms can conflict with one another when combined, i.e. syntactic determinism is not conserved. Copper, the syntax definition system in Silver, defines a modular analysis that guarantees syntactic determinism, but this requires verbose marking tokens and grammar names [58]. In contrast, typy allows different fragments to share common forms without qualification.

Putting syntactic determinism aside, many such systems also do not guarantee semantic determinism. This is because these systems allow extension providers to exert non-local control, e.g. by allow extension providers to define new inference rules that apply throughout the program, or by allowing extension providers to define new whole-program passes. This also incurs cognitive cost: programmers have no definitive way to identify which extension is in control of a given term. In contrast, typy's delegation protocol explicitly delegates control to a single fragment, in a stable manner.

Systems based on extensible attribute grammars (if used idiomatically), e.g. Silver [67], and algebraic methods, e.g. object algebras [51], give extension providers control over only those extensions to the abstract syntax that they have defined. However, even if somehow we needed only to extend the abstract syntax (leaving the concrete syntax alone), this is problematic: it becomes impossible to define functionality that operates by exhaustive case analysis (e.g. a pretty printer.) This is particularly problematic when a new such function is invented – this is known as the *expression problem* [55, 69]. In contrast, typy operates over a fixed abstract syntax.

TeJaS is a typed variant of JavaScript that is implemented as a collection of mutually recursive ML modules, each defining a particular feature [39]. This means that modules cannot be distributed separately. A new module can redefine constructs defined elsewhere, so stability is not guaranteed.

Proof assistants, e.g. TinkerType [41], PLT Redex [27], Agda [47] and Coq [45] can be used to inductively specify and mechanize the metatheory of languages. These tools generally require a complete specification (this has been identified as a key challenge [8].) Techniques for composing specifications and proofs exist [20, 21, 57], relying on various algebraic methods to encode "open" term encodings (e.g. Mendler-style f-algebras [21]), but these techniques require additional proofs at "combine-time". Several authors, e.g. Chlipala [15], have suggested proof automation as a heuristic solution to the problem of combine-time proof obligations. The typy fragment system does not work with inductive semantic specifications – instead, fragment providers directly implement their intended semantics in Python (see Sec. 5.)

Refinement type systems [28], pluggable type systems [7, 12, 13, 43] and gradual type system [60, 61] define additional static checks for programs written against an existing semantics. Some of these systems support fragmentary definitions of new analyses [13, 43]. typy is different in that its semantics (static and dynamic) is itself programmable. In other words, typy is not a gradual type system for Python like mypy [38] or Reticulated Python [68], but rather a distinct language that 1) repurposes Python's syntax; and 2) is defined by typed translation to Python. Defining a fragmentary

refinement system that sits atop our fragmentary semantics is an interesting avenue for future work. This might allow us to use `mypy`'s annotations as refinements of the `py` type.

Lightweight modular staging (LMS) is a Scala library that supports staged translation of well-typed Scala terms to other targets [56]. In contrast, `typy`'s type system is itself programmable. No specific type structure is built in to `typy`. As described in Sec. 3.2, fragment providers can target (and even extend) different languages via a foreign interface.

Macros implement local term rewritings [14, 33]. Our fragment system is similar in that translation methods programmatically work with and generate ASTs, but the translation target is a different language – Python – from the source language – `typy`. In fact, macros can be implemented using our fragment system, by defining a singleton type for the macro for which the call operation constructs the rewriting and then asks the context to typecheck and translate it.

Operator overloading [66] and *metaobject dispatch* [35] interpret operator invocations as method calls. The method is typically selected according to either the type or the dynamic tag of one or more operands. These protocols are similar to our delegation protocol for targeted expressions. However, our strategy is a *compile-time* protocol and gives direct control over typing and translation. An object system with operator overloading could be implemented in `typy`.

5. Discussion

In summary, `typy` is a bidirectionally typed programming language with no primitive types. Instead, it is organized around a novel *semantic fragment system* that allows library providers to implement the kinding semantics for new types, the static and dynamic semantics of their associated operations and the pattern matching semantics of their associated patterns programmatically. Library clients can import these fragments in any combination because fragments are contextually delegated control over terms in a deterministic and stable manner. Unlike other language extension systems, the syntax of the language is fixed, which we take to be a feature of our system because it eliminates a number of difficult problems related to composition.

We were able to implement `typy` itself as a Python library, using Python's standard reflection and code generation facilities. Using `typy`, we have been able to implement a variety of semantic structures that are, or would need to be, built primitively into other languages.

Our design does have its limitations. Python is a complex dynamic language, so we are not able to rigorously prove determinism and stability. Our argument is simply that these properties are essentially immediate consequences of our proposed design. The same complexity makes it difficult for fragment providers to reason about correctness (relative to, e.g., an inductive specification.) In the future, we hope to develop a dialect of `typy` using a reduced subset of Python (e.g. RPython [6] or λ_π [54]) or a simpler language still for which a complete formal definition is available. Another approach would be to design a fragment system where the fragment definition language is dependently typed. This would make it possible to prove interesting correctness properties about fragments. By imposing stronger abstraction barriers between fragments, we conjecture that it should be possible for the language to guarantee that a broad class of such properties are conserved by fragment composition, without the need for "combine-time" proofs.

It is not presently possible to define fragments using `typy` itself, but this is an interesting future direction. It would also be interesting to automate the generation of fragment definitions from inductive specifications, e.g. building on the techniques developed by the Veritas project [30].

The fragment system that we have developed here could be adapted to use a different surface syntax and internal language without major difficulty. If the internal language itself has non-trivial type structure, e.g. JVM bytecode, then fragments must define a type translation method (to complement the type validation method.) Moreover, term translations must be validated against the corresponding type translations. This correctness condition has been studied in the design of the TIL compiler for Standard ML [62].

Another aspect of translation validation that we did not consider here is *hygiene*, i.e. that the translations do not make inappropriate assumptions about the surrounding bindings, or inadvertently shadow bindings in an unexpected manner [5, 37]. A proper hygiene mechanism would require that we use an internal language with a more disciplined binding structure. For now, the context simply provides a method for generating unique identifiers.

By repurposing Python's syntax, `typy` benefits from many established Python tools. However, debuggers and other tools that rely not just on Python's syntax but also its semantics do not work directly on `typy` programs. We leave the problem of integrating fragments with tools like these as future work.

`typy` imposes a bidirectional structure on all fragments. This structure is known to be highly flexible [23], and even advanced dependently typed languages like Agda are fundamentally bidirectional [47]. That said, we have not explored the practicality of implementing advanced type systems, e.g. dependent or linear type systems, using our fragment system. It would not be straightforward to implement type systems that are not bidirectional, e.g. those that rely on non-local type inference or flow typing [1, 31].

Finally, we must acknowledge that not all fragments will be tasteful. This concern must be balanced against the possibilities of a vibrant ecosystem of competing fragments. We plan to curate a substantial standard library of high-quality fragments. This will help avoid the problem of different programmers reimplementing the same structures. With an appropriate community process, our position is that a fragmentary language like `typy` will hasten the research, development and adoption of good ideas, particularly those that are found only in obscure languages today.

Implementation At the time of submission, we have implemented the fragment system and the fragments described in this paper, though many of these fragments were developed against previous versions of `typy`, which had a somewhat different API. We plan to update these and release `typy` and its standard library as a free open source project before GPCE.

References

[1] Flow — A static type checker for JavaScript. `http://flowtype.org/`. 1, 5

[2] PEP 3107 – Function Annotations. `https://www.python.org/dev/peps/pep-3107/`. A

[3] PureScript. `http://www.purescript.org/`. 1

[4] The OpenCL Specification, Version 1.1, 2010. 3.2

[5] M. D. Adams. Towards the Essence of Hygiene. In *POPL*, 2015. 5

[6] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Symposium on Dynamic Languages*, 2007. 5

[7] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A Framework for Implementing Pluggable Type Systems. In *OOPSLA*, 2006. 4

[8] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLmark Challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, 2005. 4

[9] D. M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, 2003. 3.2

[10] N. Benton and A. Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *ICFP*, 1999. 3.2

[11] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *ECOOP*. 2014. 1

[12] G. Bracha. Pluggable Type Systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. 4

[13] F. Brown, A. Nötzli, and D. Engler. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *ASPLOS*, 2016. 4

[14] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *4th Workshop on Scala*, 2013. 4

[15] A. Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010. 4

[16] D. R. Christiansen. Bidirectional Typing Rules: A Tutorial. `http://davidchristiansen.dk/tutorials/bidirectional.pdf`, 2013. 2.3

[17] 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

[18] K. Crary. A syntactic account of singleton types via hereditary substitution. In *Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, 2009. 2.2, 2.5

[19] R. Culpepper, S. Tobin-Hochstadt, and M. Flatt. Advanced macrology and the implementation of Typed Scheme. In *Workshop on Scheme and Functional Programming*, 2007. 4

[20] B. Delaware, W. R. Cook, and D. S. Batory. Product lines of theorems. In *OOPSLA*, 2011. 4

[21] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Meta-theory à la carte. In *POPL*, 2013. 4

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

[23] J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013. 2.3, 5

[24] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, 2007. 4

[25] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE*, 2013. 1

[26] 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)*. 2013. 4

[27] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009. 4

[28] T. Freeman and F. Pfenning. Refinement Types for ML. In *PLDI*, 1991. 4

[29] N. Fulton, C. Omar, and J. Aldrich. Statically Typed String Sanitation Inside a Python. In *International Workshop on Privacy and Security in Programming (PSP)*, 2014. 2.3.1

[30] S. Grewe, S. Erdweg, P. Wittmann, and M. Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2015. 5

[31] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *ESOP*, 2011. 5

[32] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2nd edition, 2016. 2.2, 2.4, B

[33] T. P. Hart. MACRO definitions for LISP. Report A. I. MEMO 57, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, Oct. 1963. 4

[34] S. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 2.4

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

[36] 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. 1, 3.2

[37] E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Symposium on LISP and Functional Programming*, pages 151–161, Aug. 1986. 5

[38] J. Lehtosalo. mypy - Optional Static Typing for Python. `http://www.mypy-lang.org/`. Retrieved June 24, 2016. 4

[39] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: retrofitting type systems for JavaScript. In *Dynamic Languages Symposium (DLS)*, 2013. 4

[40] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.02 Documentation*

and user's manual. Institut National de Recherche en Informatique et en Automatique, September 2014. 2.4

[41] M. Y. Levin and B. C. Pierce. TinkerType: A Language for Playing with Formal Systems. *Journal of Functional Programming*, 13(2), Mar. 2003. 4

[42] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *OOPSLA*, 1986. 3.1

[43] S. Markstrum, D. Marino, M. Esquivel, T. D. Millstein, C. Andreae, and J. Noble. JavaCOP: Declarative pluggable types for Java. *ACM Trans. Program. Lang. Syst.*, 32(2), 2010. 4

[44] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3), 2009. 1

[45] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. 4

[46] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. 1

[47] U. Norell. Towards a practical programming language based on dependent type theory. *PhD thesis, Chalmers University of Technology*, 2007. 4, 5

[48] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Compiler Construction: 12th International Conference*, 2003. 4

[49] M. Odersky, M. Zenger, and C. Zenger. Colored Local Type Inference. In *POPL*, 2001. 2.3

[50] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007. 3.2

[51] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses. In *ECOOP*. Springer, 2012. 4

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

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

[54] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the full monty. In *OOPSLA*, 2013. 5

[55] 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, 4

[56] 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

[57] C. Schwaab and J. G. Siek. Modular type-safety proofs in Agda. In *Workshop on Programming Languages Meets Program Verification (PLPV)*, 2013. 4

[58] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In *PLDI '09*, pages 199–210, 2009. 4

[59] D. Scott. Lambda calculus: some models, some philosophy. *Studies in Logic and the Foundations of Mathematics*, 101:223–265, 1980. B

[60] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, 2007. 4

[61] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006. 4

[62] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *PLDI*, 1996. 5

[63] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, 2008. 4

[64] D. Unger and R. B. Smith. Self: The Power of Simplicity. In *OOPSLA*, pages 227–242, Dec. 1987. 3.1

[65] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. 3.2

[66] 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

[67] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, Jan. 2010. 4

[68] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *Dynamic Languages Symposium*, 2014. 4

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

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

[71] M. Zenger and M. Odersky. Implementing extensible compilers. In *Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2001. 4

Appendix

A. Syntactic Support for Python 2.x

Listing 11 An alternative syntax for argument and return type annotations.

```
@fn
def map(f, t):
  {f : fn[+a, +b], t : tree(+a)} > tree(+b)
  [t].match
  with Empty: Empty
  with Node(left, right):
    Node(map(f, left), map(f, right))
  with Leaf(x):
    Leaf(f(x))
```

Python 3.0 introduced syntax for argument and return type annotations on function definitions [2]. However, Python 2.6+ remains widely adopted because Python 3.0+ did not maintain backwards compatibility. As such, our implementations of `typy.std.fn` and `typy.opencl.kernel` support an alternative syntax for argument and return type annotations that supports Python 2.6+ as well as Python 3.0+. For example, Listing 11 shows the function `map` from Listing 7 written using this alternative syntax.

B. Python Interoperability

Listing 12 Lifting and pattern matching over `py`.

```
from typy import component
from typy.std import str
x = 3
@component
def TestLift():
  y = x # y has type std.py

  with let[z : str]: # block let
    [x].match
    with int(n): "n : std.int"
    with float(f): "f : std.float"
    with str(s): "s : std.string"
    with list(xs): "xs : std.list[py]"
    with tpl[2](xs): "xs : std.tpl[py, py]"
    with {'a': a}: "a : py"
    with instance[C](x): "x : py"
    with _: "otherwise"
```

The `py` fragment defines a single type, `py[()]` (which can be abbreviated `py`, per Sec. 2.2.) This type classifies Python values. Variables that refer to values in the static environment are assigned type `py` by default. (Technically, the default is the canonical type that `typy.lift_type` is set to. Importing `typy.std` sets `typy.lift_type` to `typy.CanonicalType(py, ())`.) For example, the component member `y` in Listing 12 refers to `x`, so it has type `py`.

In addition to standard operations, `py` supports pattern matching on the Python dynamic type tag or class, as shown in Listing 12. This is consistent with a view of Python as a uni-typed language with a rich dynamic tag structure [32, 59].

Most fragments in `typy.std` also support a coercion back to type `py` by accessing the "attribute" `.to_py`. For example, if `x : std.tpl[py, py]` (a pair of `py` values), then `x.to_py` is of type `py`. All arguments to Python functions of type `py` must be of type `py` (we are exploring coercions between `std.fn` and `std.py`, but have not implemented this as of this writing.)

The recommended way to interact with `typy` code from a Python script is simply to define a `typy` component. This is straightforward because `typy` is itself a library – it does not require that an entire project be compiled using a different toolchain. However, for convenience, Python code can directly access values of type `py` that appear in a `typy` component. Internally, the fragment system asks fragments that wish to expose themselves to untyped code to indicate this by inheriting from `typy.PythonInterop`. They must also implement a coercion method `coerce_python` that is called to transform the value upon first access. For `py`, this is simply the identity function. We have also experimented with allowing calls to `std.fn` values that take and return `py` values – here, `coerce_python` wraps the function with a check to ensure that the correct number of arguments have been provided.

C. OCaml Example

Listing 13 An OCaml module analagous to the component defined in Listing 7.

```
module Listing7 =
struct
  type 'a tree =
  | Empty
  | Node of 'a tree * 'a tree
  | Leaf of 'a

  let map (f : 'a -> b,
           tree : 'a tree) : 'b tree  =
    match tree with
    | Empty -> Empty
    | Node(left, right) ->
        Node(map(f, left), map(f, right))
    | Leaf(x) ->
        Leaf(f(x))
end
```

`typy`'s fragment system, together with its primitive support for recursive and polymorphic types, is powerful enough to capture standard idioms in languages like OCaml at very similar syntactic cost – compare Listing 13 to Listing 7.

D. Match Expressions

Listing 14 Expression-level pattern matching.

```
[scrutinee].match[
  pat_1: branch_n,
  # ...
  pat_n: branch_n]
```

Python's syntax distinguishes statements from expressions. In Sec. 2.4, we described a statement-level match expression. It is also sometimes useful to pattern match inside an expression, so `typy` supports an expression-level match expression, shown in Listing 14, that operates analagously to the statement-level match expression.