# Programmable Semantic Fragments

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

## Abstract

This paper introduces `typy`, a statically typed programming language embedded by reflection into Python. Rather than defining a monolithic semantics, `typy` has a *fragmentary semantics*, i.e. it delegates semantic control over each term, drawn from Python's fixed syntax, to a contextually relevant user-defined *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 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

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 [6], Flow [1] and PureScript [2], 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), unsafe (the type system of the sibling language is not enforced) and anti-modular (if the compiler representations change, client programs can break.) Although the safety problem can be ameliorated by inserting dynamic checks [34], it appears that the language-oriented approach [55] is difficult to reconcile with the best practices of "programming in the large" [15].

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" language, `typy`, that allows library providers to package a variety of semantic structures into *semantic fragments* that library clients can import in any combination. For example, rather than building functional record types into `typy`, a library provider can define a fragment that expresses their static and dynamic semantics. This approach sidesteps the lateral compatibility problem in that the clients of a library that requires record values at its public interface can simply import that fragment themselves.[1] 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 fragment providers to define new concrete forms in a decentralized manner (following the approach of Sugar* [18] 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 TypeScript-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.

We also encounter the classic *expression problem* [43, 54]: if library providers are able to 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.

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

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 for use in one portion of a program will not change the meaning of other portions of the program, nor allow a program to take on ambiguous meaning. This implies that we cannot simply operationalize the semantics as a "bag of rules" that fragments contribute to unconstrained.

The semantic fragment system that we introduce addresses the problems of concrete and abstract syntax quite simply: fragment providers are not given the ability to extend `typy`'s concrete nor abstract syntax. 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 (see 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, so ambiguities cannot arise, and stable under fragment composition, so defining a new fragment cannot change the meaning of an existing program component.

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 [41]. (Scala is another notable language that has a bidirectional type system, albeit of different design [38].) If the fragment decides that the term is ill-typed, it generates an error message. If typechecking succeeds, the fragment is tasked with assigning dynamic meaning to the term by computing a translation to a fixed *internal language* (IL). The IL is analagous to a first-stage compiler intermediate language (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 [25]) means that we need only be convinced that the IL is type safe.

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 for `typy`. We choose to repurpose Python's syntax and use Python itself as the IL (which explains why we named our language `typy`.) There are two main reasons for this choice: 1) Python supports both dynamic reflection and code generation, so we can embed our system itself into Python as a library (no standalone compiler is needed); and 2) Python's syntax, cleverly repurposed, is rich enough to express a wide variety of interesting constructs cleanly (so we inherit various tooling "for free".)

Sec. 2 introduces various aspects of `typy`'s fragment system with simple examples. Sec. 3 then describes more sophisticated case studies. Sec. 4 summarizes related work. We conclude after a discussion of present limitations and future work in Sec. 5.

**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. Fragments in `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, which is 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 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 members, described in Sec. 2.2, and value members,

[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.

```
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)
```

described in Sec. 2.3. The return value of the decorator is an instance of `typy.Component` that tracks 1) the identities of type members; and 2) the types and evaluated translations of value members.

### 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 [12, 24], 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 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.

For example, the `record.init_idx` class method given in Listing 2 validates record types by checking that 1) the index consists of a sequence of field specifications of the form `name : ty`, where `name` is a Python name; 2) no names are duplicated; and 3) `ty` is a valid type 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 `typy.TyExpr`, and checks that it is of kind `type`. The returned index value is a Python dictionary mapping the names to the corresponding instances of `typy.TyExpr`.

show example?

### 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* [41]. Our system is based on the system developed by Dunfield and Krishnaswami [16]. 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 method, shown in Figure 3. This method receives the context, the index value of the type and the AST of the literal

**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 idx.iterkeys())
```

expression 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 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 [22].[3] The `py` 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.

If typechecking is successful, `typy` delegates to the same fragment to generate a translation. For example, `typy` calls the `record.trans_Dict` method shown in Figure 3, which translates records to Python tuples (the labels do not need to appear in the translation.) This method asks `typy` to determine translations for the field values by calling `ctx.trans`.

### 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

[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))
```

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`.

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, 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

**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     bindings = ctx.ana_pat(pat, ty)
28     ctx.push_bindings(bindings)
29
30   @classmethod
31   def check_Expr(cls, ctx, stmt):
32     ctx.syn(stmt.value)
33
34   # trans_FunctionDef, trans_Assign & trans_Expr
35   # are elided
```

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.

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 23 of Listing 5.) We will describe how `typy` synthesizes a type for the expression `account.name` in Sec. 2.3.4.

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 25-26 of Listing 5.)

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. The `ctx.push_bindings` method pushes the bindings (here, a single binding) into the typing context.

During translation, `typy` delegates to `fn.trans_Assign`. This method is again omitted because it is straightforward.

**Listing 6** Typing and translation of targeted forms.

```
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)))
```

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 a name is reused. 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. 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.

***Other Statement Forms*** `typy` does not delegate to the fragment governing the enclosing function for statements of function 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. 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`; 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 required, like `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 Listing 6. This method looks up the attribute, here `name`, in the type index and returns the corresponding field type, here `string_in[r".+"]`, or raises a type error if the attribute is not found in the type index.

To generate the translation for `account.name`, `typy` calls `record.trans_Attribute`, shown in Listing 6. Because record values translate to tuples, this class method translates field projection to tuple projection, using the position of the attribute within the index to determine the 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 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 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.) The precedent fragment is delegated control. `typy` checks that if one fragment claims precedence over another, then the reverse is not the case (i.e. precedence is anti-symmetric.) Precedence is not transitive. 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 Fragmentary Pattern Matching

As we saw on Line 27 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 more sophisticated

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

patterns. For example, the assignment statement below uses a tuple pattern:

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

`typy` also supports a more general `match` construct, shown on Lines 17-22 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 following form:

```
with pat: e
```

where `pat` is a pattern and `e` is the corresponding *branch expression*.

To typecheck a match expression statement, `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 7-11 (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 [31].)

More specifically, `tree(+a)` is a *recursive finite sum type* defined by the `finsum` fragment imported from `typy.std` [24]. Values of finite sum type are translated into 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, a value `Node(e1, e2)` translates to `("Node", tr1, tr2)` where `tr1` and `tr2` are the translations of `e1` and `e2`. Note that names and call expressions beginning with a capitalized letter are treated as literal forms in `typy` (following Haskell's datatype convention [27].)

For each clause in a match expression statement, `typy` checks that the pattern matches values of the scrutinee type by delegating to the fragment defining that type. For example, to check the pattern `Node(left, right)` on Line 18, `typy` calls

**Listing 8** Typing and translation of patterns.

```
import ast, typy
class finsum(typy.Fragment):
  # ...

  @classmethod
  def ana_pat_Call(cls, ctx, idx, pat):
    if (isinstance(pat.func, ast.Name) and
        pat.func.id in idx and
        len(pat.args) == len(idx[pat.func.id])):
      bindings, lbl = {}, pat.func.id
      for p, ty in zip(pat.args, idx[lbl]):
        _combine(bindings, ctx.ana_pat(p, ty))
      return bindings
    else:
      raise typy.TyError("<bad pattern>", pat)

  @classmethod
  def trans_pat_Call(cls, ctx, idx, pat,
                     scrutinee_trans):
    conditions = [
      ast.Compare( # tag check
        left=_project(scrutinee_trans, 0),
        ops=[ast.Eq()],
        comparators=[ast.Str(s=pat.func.id)])
    ]
    binding_translations = {}
    for n, p in enumerate(pat.args):
      arg_scrutinee = _project(scrutinee_trans,
                               n+1)
      c, b = ctx.trans_pat(p, arg_scrutinee)
      conditions.append(c)
      binding_translations[pat.func.id] = b
    condition = ast.BoolOp(op=ast.And(),
      values=conditions)
    return (condition, binding_translations)

def _project(e, n): # helper function
  return ast.Subscript(value=e,
    slice=ast.Index(ast.Num(n=n)))
```

`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)`. 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. Both the the condition expression and the binding translations can refer to the scrutinee. 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 "effective 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.

show example?

## 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

In our recent work on *type-specific languages (TSLs)* in the Wyvern language, we used a bidirectional protocol to delegate control over the parsing of literal forms to functions associated with type definitions [39, 40]. This inspired our treatment of literal forms in `typy`. Unlike Wyvern, the literal forms are parsed according to Python's fixed syntax. Unlike `typy`, Wyvern has a monolithic semantics.

Like `typy`, Typed Racket is a statically typed language embedded into a dynamic language [51]. However, Typed Racket also has a monolithic semantics – defining new semantic structures requires directly modifying the language.

Language-external mechanisms for creating and combining language dialects, e.g. extensible compilers like Xoc [11], JastAdd [17], Polyglot [37], JaCo [56], Silver [53] and various "language workbenches" [19], suffer from the possibility of ambiguity. In particular, combining different dialects can cause the syntax or semantics to become ambiguous, even if the dialects in isolation are unambiguous. The problem is fundamentally that systems like these allow extension/dialect providers to introduce new concrete and abstract syntactic forms. Our insight is that we can sidestep the difficulties inherent in doing so by instead repurposing a fixed syntax.

Proof assistants, e.g. TinkerType [32], PLT Redex [20], Agda [36] and Coq [35] 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 [5].) Techniques for composing specifications and proofs exist [13, 14, 45], relying on various algebraic methods to encode "open" term encodings (e.g. Mendler-style $f$-algebras [14]), but these techniques require additional proofs at "combine-time". Several authors, e.g.

Chlipala [10], have suggested proof automation as a heuristic solution to the problem of combine-time proof obligations. The `typy` fragment system does not operate on inductive semantic specifications – instead, fragment providers directly implement their intended semantics in Python. Python is a complex dynamic language, so it is difficult to formally verify that fragments are correct and do not explicitly interfere with one another (though a formal specification of a substantial subset of Python does exist [42].) In future work, we plan to design a fragment system where the fragment definition language is dependently typed. This would make it possible to prove interesting properties about fragments, e.g. correctness with respect to an inductive specification. A complementary approach would be to automate the generation of fragment definitions and the accompanying proofs from inductive specifications, building on the techniques developed by the Veritas project [23].

Refinement type systems [21], pluggable type systems [4, 7, 8, 33] and gradual type system [46, 47] define additional static checks for programs written against an existing semantics. Some of these systems support fragmentary definitions of new analyses [8, 33]. `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` [30]), but rather a distinct language that 1) repurposes Python's syntax; and 2) is defined by typed translation to Python. We have implemented an interface to Python using our fragment system (i.e. the `py` fragment.) Calling into `typy` from Python, however, is only possible by defining a `typy` function. Luckily, `typy` is itself embedded into Python as a library, rather than a separate compiler, so this is straightforward. 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 [44]. In contrast, `typy`'s type system is itself programmable. No specific type structure is built in to `typy`. As we discussed in Sec. 3, `typy` can be used to implement and even extend low-level foreign languages in a staged manner. We plan to further explore this approach in future work.

Macros implement local term rewritings [9, 26]. Our fragment system is similar in that translation methods programmatically generate term representations, but the translation target is a different language – Python – from the source language – `typy` (they happen to share a syntactic representation, but are governed by different semantics.) In fact, macros can be implemented using our fragment system (by defining a singleton type for the macro for which the call operation, which is a targeted form, constructs the rewriting and then asks the context to typecheck and translate it.)

*Operator overloading* [52] and *metaobject dispatch* [28] are protocols that rewrite operator invocations into method calls. The method is typically selected according to either the type or 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, described in Sec. 2.3.4. However, our strategy is a *compile-time* protocol and gives control over typing and translation. An object system with support for operator overloading could be implemented using our fragment system.

## 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 non-overlapping sets of expressions 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.

Our design does have its limitations. As discussed in the previous section, Python is a complex dynamic language, so we cannot rigorously prove interesting metatheorems about our system, nor can fragment providers rigorously prove interesting metatheorems about their fragments. This is comparable to the situation of many programming languages for which "the compiler is the specification", but it is nevertheless ultimately unsatisfactory. In the future, we hope to develop a dialect of `typy` using a reduced subset of Python (e.g. RPython [**?** ] or some variant thereof) for which a complete formal definition is available. A related line of future work is to allow new fragments to be written using `typy` itself, rather than Python.

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 form of translation validation has been studied in the design of the TIL compiler for Standard ML [50].

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 [3, 29]. A proper hygiene mechanism would require that we use an internal language with a more restricted binding

structure than Python. For now, the context simply provides the standard tool to fragment providers: a mechanism for generating unique identifiers.

By repurposing Python's syntax, `typy` is able to benefit from many well-developed Python tools. Debuggers and other tools that rely not just on Python's syntax but also its semantics do not work directly with `typy`. It is likely that there is interesting work to be done on enabling fragments to extend the workings of a tool like a debugger.

`typy` imposes a bidirectional structure on all fragments. This structure is known to be highly flexible [16], and even advanced dependently typed languages like Agda are fundamentally bidirectional [36]. That said, we have not explored the practicality of implementing more advanced type systems, e.g. dependent or linear type systems, using our fragment system. Type systems that require tracking specialized contextual information are possible using our framework, but the corresponding attribute of 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 imported fragments to initialize the context ahead of typechecking. 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].

We have experimented with various extensions of `typy` that we did not have space to describe here. In particular, we have added a notion of fragmentary implicit coercion to `typy` (which generalizes subtyping [49].) We have also experimented with a system where literal forms can be annotated with *incomplete types*, i.e. types where the type index contains an ellipsis. For example, rather than stating all of the element types in the tuple below, we simply state that it is a `tpl`. The `tpl` fragment synthesizes the complete index:

```
(x, y, z) [: tpl[...]]
```

Finally, it may be wise to acknowledge that not all fragments will be tasteful. Indeed, some language designers worry that offering too much flexibility to users leads to abuse. This concern must be balanced with the possibilities of a vibrant ecosystem of competing fragments that can be developed and deployed as libraries, and thus more easily evaluated in the wild. We plan to curate a substantial standard library of fragments, so that different programmers do not reimplement the same structures. With an appropriate community process, our position is that a fragmentary language like `typy` will lead to faster adoption of ideas from the research community, and make it easier to abandon mistakes.

## Implementation

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

## References


[1] Flow — a static type checker for javascript. http://flowtype.org/. 1, 5

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

[3] M. D. Adams. Towards the essence of hygiene. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 457–469, 2015. 5

[4] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA*, OOPSLA '06, pages 57–74, New York, NY, USA, 2006. ACM. 4

[5] 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*, number 3603 in LNCS, pages 50–65. Springer, 2005. 4

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

[7] G. Bracha. Pluggable type systems, Oct. 2004. OOPSLA Workshop on Revival of Dynamic Languages. 4

[8] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–157. ACM, 2016. 4

[9] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*, pages 3:1–3:10, 2013. 4

[10] A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (37th POPL'10)*, pages 93–106, Madrid, Spain, Jan. 2010. ACM Press. 4

[11] 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

[12] 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

[13] B. Delaware, W. R. Cook, and D. S. Batory. Product lines of theorems. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (26th OOPSLA'11)*, pages 595–608, Portland, Oregon, USA, Oct. 2011. ACM Press. 4

[14] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Meta-theory à la carte. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 207–218. ACM, 2013. 4

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

[16] 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, 5

[17] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM. 4

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

[19] 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. 4

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

[21] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. 4

[22] 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

[23] S. Grewe, S. Erdweg, P. Wittmann, and M. Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 137–150. ACM, 2015. 4

[24] R. Harper. *Practical Foundations for Programming Languages*. Second edition, 2015. (Working Draft, Retrieved Nov. 19, 2015). 2.2, 2.4

[25] 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

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

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

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

[29] 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

[30] J. Lehtosalo. mypy - optional static typing for python. http://www.mypy-lang.org/. Retrieved June 24, 2016. 4

[31] 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

[32] M. Y. Levin and B. C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), Mar. 2003. A preliminary version appeared as an invited paper at the *Logical Frameworks and Metalanguages Workshop (LFM)*, June 2000. 4

[33] 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

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

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

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

[37] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction: 12'th International Conference, CC 2003*, volume 2622, pages 138–152, New York, NY, Apr. 2003. Springer-Verlag. 4

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

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

[40] C. Omar, C. Wang, and J. Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC '15)*, 2015. 4

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

[42] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the full monty. In *ACM SIGPLAN Notices*, volume 48, pages 217–232. ACM, 2013. 4

[43] 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

[44] 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

[45] C. Schwaab and J. G. Siek. Modular type-safety proofs in agda. In M. Might, D. V. Horn, A. A. 0001, and T. Sheard, editors, *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 3–12. ACM, 2013. 4

[46] J. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007. 4

[47] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings, Scheme and Functional Programming Workshop 2006*, pages 81–92. University of Chicago TR-2006-06, 2006. 4

[48] 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

[49] V. Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93(1):172–221, 1991. 5

[50] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI '96*, 1996. 5

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

[52] 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

[53] 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

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

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

[56] M. Zenger and M. Odersky. Implementing extensible compilers. In *Proceedings of the ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages*, number LAMP-CONF-2001-004, 2001. 4

**Listing 9** 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
```

# A. Support for Python 2.x

# B. Type Equality

Although not necessary in this example, fragments can compare type expressions for equivalence [48]. 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.

# C. OCaml Examples