

# 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 [5], 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.) It appears, then, that the language-oriented approach [45] is difficult to reconcile with the best practices of “programming in the large” [11].

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 the static and dynamic semantics of records. 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.<sup>1</sup>

Of course, designing an expressive semantic fragment system equipped with useful composition principles presents several well-known 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\* [13] and similar systems), then different fragments could define conflicting forms. For example, consider the following family of forms:

```
{ label1: expr1, ..., labeln: exprn }
```

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* [35, 44]: 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.

<sup>1</sup> 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 cannot 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 [33]. (Scala is another notable language that has a bidirectional type system, albeit of different design [30].) 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 analogous 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 [19]) 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”).

We introduce various aspects of `typy`’s fragment system with simple examples in Sec. 2, then describe more sophisticated case studies in Sec. 3. We summarize related work in Sec. 4 and 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. `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.<sup>2</sup> 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,

<sup>2</sup>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 [8, 18], which elegantly handles the details of type synonyms, 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  
exam-  
ple?

## 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* [33]. Our system is based on the system developed by Dunfield and Krishnaswami [12]. 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.

---

```
1 # class record(typy.Fragment):
2 # ... continued from Listing 2 ...
3 @classmethod
4 def ana_Dict(cls, ctx, idx, e):
5     for lbl, value in zip(e.keys, e.values):
6         if isinstance(lbl, ast.Name):
7             if lbl.id in idx:
8                 ctx.ana(value, idx[lbl.id])
9             else:
10                 raise typy.TypeError("<bad label>", f)
11         else:
12             raise typy.TypeError("<not a label>", f)
13 if len(idx) != len(e.keys):
14     raise typy.TypeError("<label missing>", e)
15
16 @classmethod
17 def trans_Dict(self, ctx, idx, e):
18     ast_dict = dict((k.id, v)
19                     for k, v in zip(e.keys, e.values))
20     return ast.Tuple(
21         (lbl, ctx.trans(ast_dict[lbl]))
22         for lbl in idx.iterkeys())
```

---

expression and must return (trivially) if typechecking is successful or raise `typy.TypeError` 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 [16].<sup>3</sup> 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

<sup>3</sup>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))
```

---

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.TypeError` 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 `ints` and `floats` 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 precedence 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 analogous 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 [23].)

More specifically, `tree(+a)` is a *recursive finite sum type* defined by the `finsum` fragment imported from `typy.std` [18]. 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 [20].)

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.

```

1 import ast, typy
2 class finsum(typy.Fragment):
3     # ...
4
5     @classmethod
6     def ana_pat_Call(cls, ctx, idx, pat):
7         if (isinstance(pat.func, ast.Name) and
8             pat.func.id in idx and
9             len(pat.args) == len(idx[pat.func.id])):
10            bindings, lbl = {}, pat.func.id
11            for p, ty in zip(pat.args, idx[lbl]):
12                _combine(bindings, ctx.ana_pat(p, ty))
13            return bindings
14        else:
15            raise typy.TypeError("<bad pattern>", pat)
16
17     @classmethod
18     def trans_pat_Call(cls, ctx, idx, pat,
19                       scrutinee_trans):
20        conditions = [
21            ast.Compare( # tag check
22                left=_project(scrutinee_trans, 0),
23                ops=[ast.Eq()],
24                comparators=[ast.Str(s=pat.func.id)])
25        ]
26        binding_translations = {}
27        for n, p in enumerate(pat.args):
28            arg_scrutinee = _project(scrutinee_trans,
29                                    n+1)
30            c, b = ctx.trans_pat(p, arg_scrutinee)
31            conditions.append(c)
32            binding_translations[pat.func.id] = b
33        condition = ast.BoolOp(op=ast.And(),
34                                values=conditions)
35        return (condition, binding_translations)
36
37 def _project(e, n): # helper function
38     return ast.Subscript(value=e,
39                           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 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  
exam-  
ple?

### 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

Like `typy`, Typed Racket is a statically typed language embedded into a dynamically typed language [41]. However, the type system of Typed Racket is monolithic – adding new semantic structures requires directly modifying the language.

In our recent work on *type-specific languages (TSLs)* in the Wyvern language, we used bidirectional typechecking to control aspects of parsing literal syntax in a manner quite similar to, but somewhat more syntactically flexible than, how we treat literal forms in `typy` [31, 32]. The increased flexibility comes because we had complete control over its grammar. By repurposing Python’s syntax, we benefit from an established ecosystem of tools (e.g. parsers, syntax highlighters, editor modes, etc.) Unlike `typy`, Wyvern has a monolithic semantics.

Qi (now Shen) is a typed Lisp dialect where library providers can introduce new typing rules declaratively [40]. The type system is, in effect, a “bag of inference rules.” The problem with this approach is that it permits non-determinism (i.e. more than one rule may apply to a term.) The fragment delegation protocol of `typy`, on the other hand, is deterministic and stable under fragment composition. This also has cognitive benefits, in that it is easy to determine which fragment’s documentation to consult to understand a term.

Language-external mechanisms for creating and combining language dialects, e.g. extensible compilers like Xoc [7] and various language workbenches [14], suffer from the same problem– the combined dialect may be semantically ambiguous or alter the semantics of existing code. Most of these systems also permit syntax extension, and so also suffer from the problem of syntactic ambiguity. The Copper tool is a notable exception in that it offers a modular analysis that

guarantees syntactic determinism, but this requires the use of verbose marking tokens and grammar names [38]. In contrast, `typy` allows many different fragments to share common syntactic forms without qualification.

Proof assistants, e.g. TinkerType [24], PLT Redex [15], Agda [29] and Coq [26] can be used to specify and mechanize the metatheory of languages, but these require a complete specification (this has been identified as a key challenge [4].) Techniques for composing specifications and proofs exist [9, 10, 37], relying on various algebraic methods to encode “open” term encodings (e.g. Mendler-style  $f$ -algebras [10]) but they require additional proofs at “combine-time” and provide no guarantees that *all* fragments having some modularly established property can be used together. Several authors, notably Chlipala [6], suggest proof automation as a heuristic solution to the problem of combine-time proof obligations. Our approach is more pragmatic and “implementation focused”, in that no inductive specification of the intended semantics needs to be given.

Refinement types / pluggable type systems

Object algebras?

LMS [36] supports staged translation of well-typed Scala programs to other targets and supports composable optimizations. Macro systems can also be used to define optimizations and operations with interesting dynamic semantics. In these systems, the language’s type system itself remains fixed. Our fragment system permits true type system extensions. New types are not merely aliases, nor must their rules be admissible in some base type system (e.g. `string_in` would be difficult to define in Scala, particularly as directly as we can in `typy`.)

*Operator overloading* [43] and *metaobject dispatch* [21] are protocols that translate operator invocations into function calls. The function 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

TODO: revise all of this

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

Because `typy` fragments are implemented in Python, it is difficult to rigorously prove metatheoretic properties about them (though formal specifications of a large portion of Python do exist [34].) In future work, we plan to design a fragment system where the fragment definition language is dependently typed, which would make it possible to prove interesting properties about fragments and, ideally, combine these proofs automatically. A complementary approach would be to automate the generation of fragment definitions and accompanying proofs from declarative specifications, following the approach of the Veritas system [17].

There remain several promising avenues for future work, many of which we mentioned throughout this paper. From a practical perspective, extensible implicit coercions (i.e. subtyping) using a mechanism similar to our “handle sets” mechanism for binary expressions would be useful. An extensible mechanism supporting type index polymorphism would also be of substantial utility and theoretical interest. Debuggers and other tools that rely not just on Python’s syntax but also its semantics cannot be used directly. We believe that active types can be used to control debugging and other tools, and plan to explore this in the future. We have also not yet evaluated the feasibility of implementing more advanced type systems (e.g. linear, dependent or flow-dependent type systems) and those that require a more “global” view (e.g. security-oriented types) using our framework. Type systems that require tracking contextual information are possible using our framework, but the context needs to be initialized on first use of a type that needs it. This is somewhat awkward, so we plan to include a mechanism that allows any imported tycons to initialize the context ahead of typechecking.

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

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



## 6. Implementation

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

## 7. Conclusion

This appears to be a widespread problem: programmers and development teams often cannot use the constructs they might prefer because they are only available in specialized dialects they cannot adopt [27, 28].

finish  
this

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

## References

- [1] Flow — a static type checker for javascript. <http://flowtype.org/>.
- [2] Purescript. <http://www.purescript.org/>.
- [3] The Python Language Reference. <http://docs.python.org>, 2013.
- [4] 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.
- [5] G. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In *ECOOP 2014—Object-Oriented Programming*, pages 257–281. Springer, 2014.
- [6] 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.
- [7] R. Cox, T. Bergan, A. T. Clements, M. F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS*, 2008.
- [8] K. Cray. 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.
- [9] 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.
- [10] 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.
- [11] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2:80–86, 1976.
- [12] 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.
- [13] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE ’13*, pages 3–12, 2013.
- [14] 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.
- [15] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, Cambridge, Mass., 2009.
- [16] 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.
- [17] 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.
- [18] R. Harper. *Practical Foundations for Programming Languages*. Second edition, 2015. (Working Draft, Retrieved Nov. 19, 2015).
- [19] 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.
- [20] S. L. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
- [21] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, Oct. 1986.
- [26] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
- [27] L. A. Meyerovich and A. Rabkin. How not to survey developers and repositories: experiences analyzing language adoption. In *PLATEAU*, 2012.
- [28] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *OOPSLA*, 2013.
- [29] U. Norell. Towards a practical programming language based on dependent type theory. *PhD thesis, Chalmers University of Technology*, 2007.
- [30] Odersky, Zenger, and Zenger. Colored local type inference. In *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.
- [31] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP ’14*, 2014.
- [32] C. Omar, C. Wang, and J. Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC ’15)*, 2015.

- [33] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In *PLDI '09*, pages 199–210, 2009.
- [39] 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.
- [40] M. Tarver. *Functional programming in Qi*. Free University Press, 2008.
- [41] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL*, 2008.
- [42] D. Unger and R. B. Smith. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, Dec. 1987.
- [43] 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.
- [44] P. Wadler. The expression problem. *java-genericity mailing list*, 1998.
- [45] M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

---

**Listing 9** An OCaml module analagous to the component defined in Listing 7.

---

```
1 module Listing7 =
2 struct
3   type 'a tree =
4   | Empty
5   | Node of 'a tree * 'a tree
6   | Leaf of 'a
7
8   let map (f : 'a -> b,
9           tree : 'a tree) : 'b tree =
10      match tree with
11      | Empty -> Empty
12      | Node(left, right) ->
13         Node(map(f, left), map(f, right))
14      | Leaf(x) -> Leaf(f(x))
15 end
```

---

## A. Support for Python 2.x

## B. Type Equality

Although not necessary in this example, fragments can compare type expressions for equivalence [39]. 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