

# 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 prototypal 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 [10], Flow [1] and PureScript [3], but these libraries are not directly accessible across language boundaries because these languages are all syntactically and semantically incompatible with one another (the first two define different object systems, and PureScript is a functional language similar to Haskell and ML.)

One common workaround is to interface indirectly with libraries written in sibling languages through the code generated by a compiler that targets a more established language for which a *foreign interface (FI)* is available. For example, all of the languages above have compilers that target JavaScript and

they are all capable of interfacing with JavaScript. Unfortunately, this approach is unnatural (the syntactic and semantic conveniences of the sibling language are unavailable), 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 [41], it appears that the language-oriented approach [64] is difficult to reconcile with the best practices of “programming in the large” [19].

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.<sup>1</sup> 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\* [22] 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* [51, 63]: 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 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 [49]. (Scala is another notable language that has a bidirectional type system, albeit of different design [45].) 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 [30]) 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.<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 type member definitions, described in Sec. 2.2, and value member definitions, described in Sec. 2.3. The return value of the decorator

<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)

```

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 [16, 29], 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* [49]. Our system is based on the system developed by Dunfield and Krishnaswami [20]. 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 Listing 3. This method receives the context, the index value of the type and the AST of the literal 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.

---

**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 sorted(idx.keys()))
```

---

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 [26].<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 Listing 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`. The field ordering is alphabetical.

### 2.3.2 Function Forms

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

`typy` delegates control over the typechecking and translation of function definition forms that appear inside com-

<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))
```

---

ponents, or in other synthetic positions, 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 initialize 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.

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

### 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` or `e1.x += e2`; 2) assign to a subscript, e.g. `e1[e2] = e3`; or 3) statements with guards, e.g. `if`, `for` and `while`. These operate as *targeted forms*, described next.

### 2.3.4 Targeted Forms

*Targeted forms* include 1) the statement forms just mentioned; 2) expression forms having exactly one subexpression, like `-e1` or `e1.attr`; and 3) expression forms where there may be multiple subexpressions but the left-most one is the only one 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 patterns. For example, the assignment statement below uses a tuple pattern:

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

**Listing 7** Polymorphism, recursion and pattern matching in `typy`. An analogous OCaml file is shown in Appendix B.

```
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))
```

`typy` also supports a more general match construct, shown on Lines 4-9 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 [37].)

More specifically, `tree(+a)` is a *recursive finite sum type* defined by the `finsum` fragment imported from `typy.std` [29]. 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 [32].)

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 5, `typy` calls `finsum.ana_pat_Call`, shown in Listing 8. This method must either return a dictionary of *bindings*, i.e. a mapping from variables to types, which `typy` adds to the typing context when typechecking the corresponding branch expression, or

---

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

---

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

---

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 pat-

tern 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?

## 2.5 Determinism and Stability

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

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

For literal forms (Sec. 2.3.1), the fragment defining the type provided for analysis is delegated control. To establish determinism and stability, we need only establish that type normalization is deterministic and stable. Our language of type expressions is a standard deterministic lambda calculus [16] and normalization interacts with the fragment context only at canonical form, which was just discussed.

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

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

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

---

**Listing 9** Prototypal objects in `typy`.

---

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

---

Like many other “pragmatic” language designs, we are not able to formally prove these metatheorems about `typy` (because it incorporates the entirety of Python.)

### 3. More Examples

In this section, we will further demonstrate the flexibility of `typy`’s fragment system by outlining additional examples.

#### 3.1 Prototypal Object Types

JavaScript’s object system supports *prototypal inheritance* (based on a similar mechanism in the Self language [39, 59].) We have implemented a statically typed variant of this system as a fragment, `typy.std.proto`.

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

The fields of a prototypic object are mutable, e.g. as shown in the body of `incr` on Line 16. The delegation protocol treats an assignment of this form as a targeted form, so in this case, `proto` is delegated control.

On Line 19, we call `incr`. The `proto` fragment passes in the target of a method call as the first argument.

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

#### 3.2 Low-Level Foreign-Function Interfaces

Python is widely used in computational science and engineering [46]. One reason is that Python has support for calling into low-level languages like C (e.g. via SWIG [8].) Many popular libraries, e.g. `numpy` [60], operate essentially as wrappers around low-level routines written in these languages. It

---

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

---

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

---

is also possible to dynamically compile low-level code generated by Python code as a string. This is particularly useful when working with GPUs and other compute devices, e.g. using the `PyCUDA` and `PyOpenCL` libraries [34].

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

On Line 11, we then invoke the `to_device` operator to transfer the `numpy` array to the compute device’s memory (we omit the code needed once per session to select a device.)

On Lines 14–17, we then define a typed OpenCL kernel. An OpenCL kernel is simply an OpenCL function that is called in a *data parallel* manner, i.e. a large number of threads are spawned, all running the same kernel. Each kernel has access to a unique ID, called the *global ID* in OpenCL. Here, `add5` determines its global ID and then adds 5 to the corresponding element in the input buffer. Notice that unlike in OpenCL (which is an extension of a substantial subset of C), we did not need to specify a return type or a type annotation on `gid`, because `typy` is bidirectionally typed. The translation of the definition of `add5` is equivalent to the following Python code (assuming a variable `ctx`, which our library tracks implicitly):

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



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

Line 20 invokes the `add5` kernel in a data parallel fashion on the device buffer `d_x`. The parameter `global_size` determines the number of threads – here, one thread per array element. Finally, Lines 23-24 retrieve the result from the device and print out the result.

The details of the various fragments just described, are, of course, somewhat involved. The takeaway lesson, however, is that as the designers of `typy`, we did not need to anticipate this particular mode of use. Low-level array types and even the type system of a foreign language could all be realized using the fragment system described in Sec. 2. A similar approach could be taken, independently, for CUDA, C and perhaps even higher-level languages for which a foreign interface is available in Python, like Java. In contrast, monolithic languages like MLj needed to primitively build in a type safe foreign interface [9].

## 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 [47, 48]. 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 [58]. 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 [15], JastAdd [21], Polyglot [44], JaCo [65], Silver [62] and various “language workbenches” [23], 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 [38], PLT Redex [24], Agda [43] and Coq [42] 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 [7].) Techniques for composing specifications and proofs exist [17, 18, 53], relying on various algebraic methods to encode “open” term encodings (e.g. Mendler-style  $f$ -algebras [18]), but these techniques require additional proofs at “combine-time”. Several authors, e.g. Chlipala [14], 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 [50].) 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 [27].

Refinement type systems [25], pluggable type systems [6, 11, 12, 40] and gradual type system [54, 55] define additional static checks for programs written against an existing semantics. Some of these systems support fragmentary definitions of new analyses [12, 40]. `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` [36], but rather a distinct language that 1) repurposes Python's syntax; and 2) is defined by typed translation to Python. Defining a fragmentary refinement system that sits atop our fragmentary semantics is an interesting avenue for future work. This might allow us to use `mypy`'s annotations as refinements of the `py` type.

*Lightweight modular staging (LMS)* is a Scala library that supports staged translation of well-typed Scala terms to other targets [52]. 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 [13, 31]. 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* [61] and *metaobject dispatch* [33] 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 terms in a deterministic and stable manner. Unlike other language extension systems, the syntax of the language is fixed, which we take to be a feature of our system because it eliminates a number of difficult problems related to composition. We were able to implement `typy` itself as a Python library, using Python’s standard reflection and code generation facilities. Using `typy`, we have been able to implement a variety of semantic structures that are, or would need to be, built primitively into other languages.

Our design does have its limitations. Python is a complex dynamic language, so we are not able to rigorously prove determinism and stability. Our argument is simply that these properties are essentially immediate consequences of our proposed design. In the future, we hope to develop a dialect of `typy` using a reduced subset of Python (e.g. RPython [5] or some variant thereof) or a simpler language still for which a complete formal definition is available.

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 [57].

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 [4, 35]. A proper hygiene mechanism would require that we use an internal language with a more disciplined binding structure. For now, the context simply provides a method for generating unique identifiers.

By repurposing Python’s syntax, `typy` benefits from many established Python tools. However, debuggers and other tools that rely not just on Python’s syntax but also its semantics do

not work directly on `typy` programs. It is likely that there is interesting research to be done on how to integrate a fragment system with tools like these.

Type systems that require tracking specialized contextual information (e.g. ML-style exceptions [28, 29]) are possible using our framework, but a corresponding attribute of the context object needs to be initialized by the fragment the first time that fragment is delegated control. This is somewhat awkward, so we plan to explore a mechanism that allows imported fragments to initialize the context ahead of time.

`typy` imposes a bidirectional structure on all fragments. This structure is known to be highly flexible [20], and even advanced dependently typed languages like Agda are fundamentally bidirectional [43]. That said, we have not explored the practicality of implementing advanced type systems, e.g. dependent or linear type systems, using our fragment system. It would not be straightforward to implement type systems that are not bidirectional, e.g. those that rely on non-local type inference or flow typing [1].

We have experimented with various extensions of `typy` that we did not have space to detail here. In particular, we have added a notion of fragmentary implicit coercion to `typy` (which generalizes subtyping [56].) 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, the `tp1` fragment synthesizes the complete index:

```
(x, y, z) [: tp1[...]]
```

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

## Implementation

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

## References

- [1] Flow — A static type checker for JavaScript. <http://flowtype.org/>. 1, 5
- [2] PEP 3107 – Function Annotations. <https://www.python.org/dev/peps/pep-3107/>. A
- [3] PureScript. <http://www.purescript.org/>. 1
- [4] M. D. Adams. Towards the Essence of Hygiene. In *POPL*, 2015. 5
- [5] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Symposium on Dynamic Languages*, 2007. 5
- [6] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A Framework for Implementing Pluggable Type Systems. In *OOPSLA*, 2006. 4
- [7] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLmark Challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, 2005. 4
- [8] D. M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, 2003. 3.2
- [9] N. Benton and A. Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *ICFP*, 1999. 3.2
- [10] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *ECOOP*. 2014. 1
- [11] G. Bracha. Pluggable Type Systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. 4
- [12] F. Brown, A. Nötzli, and D. Engler. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *ASPLOS*, 2016. 4
- [13] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *4th Workshop on Scala*, 2013. 4
- [14] A. Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010. 4
- [15] 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
- [16] K. Cray. A syntactic account of singleton types via hereditary substitution. In *Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, 2009. 2.2, 2.5
- [17] B. Delaware, W. R. Cook, and D. S. Batory. Product lines of theorems. In *OOPSLA*, 2011. 4
- [18] 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
- [19] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2:80–86, 1976. 1
- [20] J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013. 2.3, 5
- [21] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, New York, NY, USA, 2007. 4
- [22] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12, 2013. 1
- [23] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. The state of the art in language workbenches. In *Software Language Engineering (SLE)*. 2013. 4
- [24] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009. 4
- [25] T. Freeman and F. Pfenning. Refinement Types for ML. In *PLDI*, 1991. 4
- [26] N. Fulton, C. Omar, and J. Aldrich. Statically Typed String Sanitation Inside a Python. In *International Workshop on Privacy and Security in Programming (PSP)*, 2014. 2.3.1
- [27] S. Grewe, S. Erdweg, P. Wittmann, and M. Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2015. 4
- [28] R. Harper. Programming in Standard ML. <http://www.cs.cmu.edu/~rwh/smlbook/book.pdf>, 1997. Working draft, retrieved June 21, 2015. 5
- [29] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2nd edition, 2016. 2.2, 2.4, 5
- [30] 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
- [31] T. P. Hart. MACRO definitions for LISP. Report A. I. MEMO 57, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, Oct. 1963. 4
- [32] S. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 2.4
- [33] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991. 4
- [34] 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. 3.2
- [35] 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
- [36] J. Lehtosalo. mypy - Optional Static Typing for Python. <http://www.mypy-lang.org/>. Retrieved June 24, 2016. 4
- [37] 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

- [38] M. Y. Levin and B. C. Pierce. TinkerType: A Language for Playing with Formal Systems. *Journal of Functional Programming*, 13(2), Mar. 2003. [4](#)
- [39] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *OOPSLA*, 1986. [3.1](#)
- [40] 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](#)
- [41] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3), 2009. [1](#)
- [42] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. [4](#)
- [43] U. Norell. Towards a practical programming language based on dependent type theory. *PhD thesis, Chalmers University of Technology*, 2007. [4](#), [5](#)
- [44] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Compiler Construction: 12th International Conference*, 2003. [4](#)
- [45] M. Odersky, M. Zenger, and C. Zenger. Colored Local Type Inference. In *POPL*, 2001. [1](#)
- [46] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007. [3.2](#)
- [47] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014. [4](#)
- [48] C. Omar, C. Wang, and J. Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC)*, 2015. [4](#)
- [49] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000. [1](#), [2.3](#)
- [50] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the full monty. In *OOPSLA*, 2013. [4](#)
- [51] 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](#)
- [52] 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](#)
- [53] C. Schwaab and J. G. Siek. Modular type-safety proofs in Agda. In *Workshop on Programming Languages Meets Program Verification (PLPV)*, 2013. [4](#)
- [54] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, 2007. [4](#)
- [55] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006. [4](#)
- [56] V. Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93(1):172–221, 1991. [5](#)
- [57] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *PLDI*, 1996. [5](#)
- [58] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, 2008. [4](#)
- [59] D. Unger and R. B. Smith. Self: The Power of Simplicity. In *OOPSLA*, pages 227–242, Dec. 1987. [3.1](#)
- [60] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. [3.2](#)
- [61] 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](#)
- [62] 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](#)
- [63] P. Wadler. The expression problem. *java-genericity mailing list*, 1998. [1](#)
- [64] M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994. [1](#)
- [65] M. Zenger and M. Odersky. Implementing extensible compilers. In *Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2001. [4](#)



## Appendix

### A. Syntactic Support for Python 2.x

Python 3.0 introduced syntax for argument and return type annotations on function definitions [2]. However, Python 2.6+ remains widely adopted. As such, our implementations of `typy.std.fn` and `typy.openc1.kernel` support an alternative syntax for argument and return type annotations that supports Python 2.6+ as well as Python 3.0+. For example, Listing 11 shows the function `map` from Listing 7 written using this alternative syntax.

---

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

---

```
1 @fn
2 def map(f, t):
3     {f : fn[+a, +b], t : tree(+a)} > tree(+b)
4     [t].match
5     with Empty: Empty
6     with Node(left, right):
7         Node(map(f, left), map(f, right))
8     with Leaf(x):
9         Leaf(f(x))
```

---

### B. OCaml Example

---

**Listing 12** 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
```

---