Ace: Active Typechecking and Translation Inside a Python

Abstract

Programmers are justifiably reluctant to adopt new language dialects to access stronger type systems. This suggests a need for a language that is *compatible* with existing libraries, tools and infrastructure and that has an *internally extensible type system*, so that adopting and combining type systems requires only importing libraries in the usual way, without the possibility of link-time ambiguities or safety issues.

We introduce Ace, an extensible statically typed language embedded within and compatible with Python, a widely-adopted dynamically typed language. Python serves as Ace's type-level language. Rather than building in a particular set of type constructors, Ace introduces a novel extension mechanism, *active typechecking and translation*, organized around a bidirectional type system that inverts control over typechecking and translation to user-defined type constructors according to a protocol that cannot cause ambiguities and type safety issues at link-time. In addition to describing a full-scale language design, we give a simplified calculus that describes the foundations of this mechanism and show that, as implemented in Ace, it is flexible enough to admit practical library-based implementations of types drawn from functional, object-oriented, parallel and domain-specific languages.

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

1. Introduction

Asking programmers to import a new library is simpler than asking them to adopt a new programming language. Indeed, recent empirical studies underscore the difficulties of driving languages into adoption, finding that extrinsic factors like compatibility with large existing code bases, library support, team familiarity and tool support are at least as important as intrinsic features of the language [6, 20, 21].

Unfortunately, researchers and domain experts aiming to provide potentially useful new abstractions can sometimes find it difficult to implement them as libraries, particularly when they require strengthening a language's type system. In these situations, abstraction providers often develop a new language or language dialect. Unfortunately, this *language-oriented approach* [35] does not scale to large applications: using components written in languages with different type systems can be awkward and lead to safety problems and performance overhead at interface boundaries.

This is a problem even when a dialect introduces only a small number of new constructs. For example, a recent study [4] comparing a Java dialect, Habanero-Java (HJ), with a comparable library, `java.util.concurrent`, found that the language-based abstractions in HJ were easier to use and provided useful static guarantees. Nevertheless, it concluded that the library-based abstractions remained more practical outside the classroom because HJ, as a distinct dialect of Java with its own type system, would be difficult to use in settings where some developers had not adopted it, but needed to interface with code that had adopted it. It would also be difficult to use it in combination with other abstractions also implemented as dialects of Java. Moreover, its tool support is more limited. This suggests that today, programmers and development teams cannot use the abstractions they might prefer because they are only available bundled with languages they cannot adopt [19, 20].

Internally extensible languages promise to reduce the need for new dialects by giving abstraction providers more direct control over a language's syntax and static semantics from within libraries. Unfortunately, the mechanisms available today have several problems. First, they are themselves often available only within a dialect of an existing language and thus face a "chicken-and-egg" problem: a language like SugarJ [9] must overcome the same extrinsic issues as a language like HJ. Second, giving abstraction providers too much control over the language (by introducing an over-zealous "solution" to the *expression problem*) can introduce type safety issues and ambiguities that only become apparent when extensions are combined, as we will discuss. Too little

control, on the other hand, leaves it difficult to implement real-world abstractions.

In this paper, we describe an extensible statically-typed language, Ace, that is implemented entirely as a library within Python and flexibly repurposes its syntax to enable, for example, type annotations. This avoids the "chicken-and-egg problem" and occupies what we consider a "sweet spot" in a design space that includes the extrinsic criteria described above. We justify this by showing a variety of non-trivial type system fragments that can be embedded as composable libraries within Ace (contribution 1, Sec. 2).

Ace, together with a core lambda calculus, serves also as a vehicle for a more general contribution: a novel underlying extension mechanism, which we call *active typechecking and translation*. As suggested by the design of Ace, it sidestep the expression problem by leaving the forms of expressions fixed. Instead, abstraction providers extend the language by introducing new type constructors and operators using type-level functions. We show how, by structuring the language as a bidirectional type system and integrating techniques from the typed compilation literature directly into the language, we can maintain expressiveness while guaranteeing that link-time ambiguities cannot occur and type safety is maintained (contribution 2, Secs. 3 and 4).

In Sec. 5, we place Ace in the context of related work on language extensibility, term rewriting, staged compilation, bidirectional typechecking, type-level computation and typed compilation. We conclude in Sec. 6 by summarizing the key features needed by a host language to support active typechecking and translation, and discussing present limitations and potential future work.

2. Language Design and Usage

Listing 1 shows that the top-level of every Ace file is a *compilation script* written directly in Python. Ace requires no modifications to the language (version 2.6+ or 3.0+) or features specific to the primary implementation, CPython (so Ace supports alternative implementations like Jython and PyPy). This choice pays immediate dividends on the first five lines: Ace's import mechanism is Python's import mechanism, so Python's build tools (e.g. `pip`) and package repostories (e.g. `PyPI`) are directly available for distributing Ace-based libraries, including those defining type systems.

2.1 Types

Types are constructed programmatically during execution of the compilation script. This stands in contrast to many contemporary statically-typed languages, where types (e.g. datatypes, classes, structs) can only be declared. Put another way, Ace supports *type-level computation* and Python is its type-level language (discussed further in Sec. 4 and Sec. 5). In our example, we see several types being constructed:

1. On line 9, we construct a functional record type with a single (immutable) field named `amount` with type `decimal[2]`, which classifies decimal numbers with two decimal places. `record` and `decimal` are *indexed type constructors*. We will provide more details in Sec. 3, but note that syntactically, `record` overloads subscripting and borrows Python's syntax for array slices (e.g. `a[min:max]`) to approximate conventional functional notation for type annotations. Note also that the field name could equivalently have been written `'a' + 'mount'`, again emphasizing that types and indices are constructed programatically by a Python script.
2. On lines 10-13, we construct another record type. The field `name` has type `string`, defined in `examples.py`, while the fields `account_num` and `routing_num` have more interesting types, classifying strings guaranteed statically to be in a regular language specified statically by a regular expression pattern (written, to avoid needing to escape backslashes, using Python's *raw string literals*). The appendix will give the typechecking rules, based on [11]. To our knowledge, this type system has not previously been implemented. We include it to emphasize that we aim to support specialized type systems, not just general purpose constructs like records and strings.
3. On line 14, we construct a simple *prototypic object type* [16]. The type `Transfer` classifies terms consisting of a *fore* of type `A` and a *prototype* of type `C`. If a field cannot be found in the fore, the type system will delegate (here, statically) to the prototype. This makes it easy to share the values of the fields of a common prototype amongst

Listing 1 [`listing1.py`] An Ace compilation script.

```
1  from examples.py import py, string
2  from examples.fp import record
3  from examples.oo import proto
4  from examples.num import decimal
5  from examples.regex import string_in
6
7  print "Hello, compile-time world!"
8
9  A = record['amount' : decimal[2]]
10 C = record[
11   'name' : string,
12   'account_num' : string_in[r'\d{10}'],
13   'routing_num' : string_in[r'\d{2}-\d{4}/\d{4}']]
14 Transfer = proto[A, C]
15
16 @py
17 def log_transfer(t):
18   """Logs a transfer to the console."""
19   {t : Transfer}
20   print "Transferring %s to %s." % (
21     [string](t.amount), t.name)
22
23 @py
24 def __toplevel__():
25   print "Hello, run-time world!"
26   common = {name: "Annie Ace",
27             account_num: "0000000001",
28             routing_num: "00-0000/0001"} (C)
29   t1 = ({amount: 5.50}, common) (Transfer)
30   t2 = ({amount: 15.00}, common) (Transfer)
31   log_transfer(t1)
32   log_transfer(t2)
33
34 print "Goodbye, compile-time world!"
```

many fores, here to allow us to describe multiple transfers differing only in amount.

2.2 Typed Functions

Typed functions implement the run-time behavior and are distinguished by the presence of a decorator specifying their *base semantics* (here `py` from `examples.py`; Sec. 3). These functions are, however, still written using Python's syntax, a choice that is again valuable for extrinsic reasons: users of Ace can use a variety of tools designed to work with Python source code without modification, including code highlighters (like the one used in generating this paper), editor plugins, style checkers and documentation generators. We will see several examples of how, with a bit of cleverness, Python's syntax can be repurposed within these functions to support a variety of static and dynamic semantics that differ from Python's. Ace leverages the `inspect` and `ast` modules in the Python standard library to extract abstract syntax trees for functions annotated in this way [2].

On lines 16-21, we see the function `log_transfer`. We follow Python conventions by starting with a documentation string that informally specifies the behavior of the function. Before moving into the body, however, we also write a *type signature* (line 17) stating that the type of `t` is `Transfer`, the prototypic object type described above. As a result, we can assume on line 19 that `t.amount` and `t.name` have types `decimal[2]` and `string`, respectively. We will return to the details of `print` and the form `[string](t.amount)`, which performs an explicit conversion, in Sec. 3.

Line 19 repurposes Python's syntax for dictionary literals to approximate conventional notation for type annotations. In version 3.0 of Python, syntax for annotating arguments with arbitrary values directly was introduced [1]:

```
def log_transfer(t : Transfer):
  print ...
```

These annotations were initially intended to serve only as documentation (suggesting that users of dynamically typed languages see potential in a typing discipline, if not a static one), but they were also made available as metadata for use by unspecified future libraries. Ace supports both notations when the compilation script is run using Python 3.*x*, but we use the more universally available notation here in view of our extrinsic goals: the Python 2.x series remains the most widely used by a large margin as of this writing.

2.3 Bidirectional Typechecking of Introductory Forms

On lines 23-32, we define the function `__toplevel__` (the base will consider this a special name for the purposes of compilation, discussed in Sec. 2.4). After printing a run-time greeting, we introduce a value of type `C` on lines 26-28. Recall that `C` is a record type, so we provide the names of the fields (here, without quotes because we are no longer in the type-level language) and values of the appropriate type using the syntactic form normally used for dictionary literals. We specify which record type the literal should be analyzed against by giving a *literal ascription*, `(C)`. This again repurposes existing syntax: here function application. When the "function" is of literal form (dictionaries, tuples, lists, strings, numbers, booleans and `None`) and the argument is a type, we will treat it instead an *ascribed introductory form* of that type. We see another such form used with Python's tuple syntax on lines 29-30 to introduce terms of type `Transfer` by providing the fore and prototype.

The string, number and dictionary literal forms inside the outermost forms on lines 26-30 are also introductory, but do not have an ascription. This is because the outermost ascriptions completely determine which types they will need to have. As we will discuss in more detail shortly, Ace is built around a *bidirectional type system* that distinguishes locations where an expression must *synthesize* a type (e.g. to the right of bindings) from those where it can be *analyzed* against a known type (e.g. as an argument to a function with known type) [18]. Unascribed literal forms must ultimately be analyzed against a type.

If an unascribed literal is used in a synthetic location, the base can, however, specify a "default ascription". For example, if we had not specified the literal ascription on line 26, the base would cause the dictionary literal to be analyzed against the type `dyn`, covering dynamically classified Python values. This would still lead to a type error because keys are then treated as expressions, as in Python, rather than field names, and the identifiers shown have not been bound in the top-level context (emphasizing that "dynamically-typed languages" can still have a static semantics, even if it only involves checking that top-level variables are bound).

An ascription can also be a type constructor instead of a type. For example, we might write `[1, 2] (matrix)` instead of `[1, 2] (matrix[i32])` when using a base for which the default ascription for number literals is `i32`. Because the arguments synthesize the appropriate type, the type constructor can synthesize an appropriate index based on the types of the subexpressions. If we wanted a matrix of `dyn`s, then we would have to write either `[1, 2] (matrix[f32])`, ascribe each inner literal so that it synthesizes the `f32` type, `[1(f32), 2(f32)] (matrix)`, or construct a base with a different default ascription.

We will see how bidirectional typechecking works in greater technical detail in Secs. 3 and 4, but note here that having the semantics of a form change depending on the type it is being analyzed against is a key to achieving expressiveness given our constraint that we cannot add new forms to the language, for both extrinsic and intrinsic reasons.

2.4 External Compilation

To typecheck, translate and execute the code in Listing 1, we have two choices: do so externally at the shell, or perform compilation interactively (or implicitly) from within Python. We will begin with the former. Interactive and implicit compilation are implemented but we do lack space to detail it in

Listing 2 Compiling `listing1.py` using `acec`. Both steps can be performed at once by writing `ace listing1.py` (line 3 will not be printed with this command).

```
1  $ acec listing1.py
2  Hello, compile-time world!
3  Goodbye, compile-time world!
4  [acec] _listing1.py successfully generated.
5  $ python _listing1.py
6  Hello, run-time world!
7  Transferring 5.50 to Annie Ace.
8  Transferring 15.00 to Annie Ace.
```

Listing 3 [`_listing1.py`] The file generated in Listing 2.

```
1  import examples.num.runtime as __ace_0
2
3  def print_transfer(t):
4    print ("Transferring %s to %s." % (
5      __ace_0.decimal_to_str(t[0],2), t[1][0]))
6
7  print "Hello, run-time world!"
8  common = ("Annie Ace", "0000000001", "00-0000/0000")
9  t1 = ((5,50), common)
10 t2 = ((15,0), common)
11 log_transfer(t1)
12 log_transfer(t2)
```

the present paper, choosing here to focus on the core mechanism.

Listing 2 shows how to invoke the `acec` compiler at the shell to typecheck and translate `listing1.py`, resulting in a file named `_listing1.py`. This is then sent to the Python interpreter for execution. Note that the `print` statements at the top-level of the compilation script were evaluated during compilation only. These two steps can be combined by running `ace listing1.py` (the intermediate file is not generated unless explicitly requested in this case).

The Ace compiler is itself a Python library, and `acec` is a simple Python script that invokes it, operating in two steps:

1. It evaluates the compilation script to completion.
2. For any top-level bindings that are Ace functions in the final environment (instances of `ace.TypedFn`, as we will discuss), it initiates active type-checking and translation (Sec. 3). If no type errors are discovered, the translations are collected (obeying order dependencies) and emitted, with file extension(s) determined by the target(s) in use, discussed further in Sec. 3. Here our target is the default target associated with the base `py`, which emits Python 2.6 files. If a type error is discovered, no file is emitted and the error is displayed on the console.

In our example, there are no type errors, so the file `_listing1.py`, shown in Listing 3, is generated. This file is meant only to be executed. The invariants necessary to ensure that execution does not "go wrong" were checked statically and entities having no bearing on execution, like field names and types themselves, were erased. Notice that:

1. The base recognized the function name `__toplevel__` as special, placing the translation of its body at the top level of the file. Ace does not have any special function names itself; this is a feature of the user-defined base.
2. Records with two or more fields translated to tuples of their values (e.g. `common` on line 8). If there was only a single field, like the terms of type `A` inside `t1` and `t2`, the value was passed around unadorned.
3. Decimals translated to pairs of integers. Conversion to a string happened via a helper function defined in a "runtime" package imported with an internal name, `__ace_0`, to avoid naming conflicts.
4. Terms of type `string_in[r"..."]` translated to strings. Checks for membership in the specified regular language were performed entirely statically by the type system.[1]
5. Prototypic objects are represented as pairs consisting of the fore and the prototype. Dispatch to the appropriate record based on the field name is static (line 5).

Listing 4 [`listing4.py`] Lines 9-13 each have type errors.

```
1  from listing1 import py, A, C, log_transfer
2  from datetime import date
3  @py
4  def pay_oopsie(a):
5    {a : A}
6    print "Checking date..."
7    if date.today().day == 1:
8      common = {nome: "Oopsie Daisy",
9                account_num: None,
10               routing_num: "0-0000-0002"} (C)
11     log_transfer((common, a))
12     a.amount += 1000
13 print str(date.today().day)
```

Listing 5 Compiling `listing4.py` using `acec` catches the errors statically (compilation stops at first error).

```
1  $ acec listing4.py
2  2
3  [acec] TypeError in listing4.py (line 8, col 15):
4    [record] Invalid field name: nome
5             Expected: name, account_num, routing_num
```

2.5 Type Errors

Listing 4 shows an example of code containing several type errors. Indeed, lines 8-12 each contain a type error. If analagous code were written in Python itself, these could only be found if the code was executed on the first day of the month (and, depending on the implementation, not all of the issues would immediately result in run-time exceptions, possibly leading to quite subtle problems; for example, the unexpected use of `None`[2]). Static type checking allows us to find these errors during compilation. Listing 5 shows the result of attempting to compile this code. The compilation script completes (so that functions can refer to each other

[1] Note that there are situations (e.g. if a string is read in from the console) that would necessitate an initial run-time check, but no further checks in downstream functions would be necessary. See the appendix for details.

[2] The notation `+T` constructs a `None`-able option type; see appendix.

Listing 6 [listing6.py] The example detailed in Sec. 3.

```
1  from listing1 import py, A, record
2  @py
3  def example(a):
4    {a : A}
5    x = a.amount
6    x = x + x
7    return {
8      y: {amount: x} (A),
9      remark: "Creating an anonymous record"} (record)
```

mutually recursively), then the typed functions in the top-level environment are typechecked. The typechecker raises an exception statically at the first error, and `acec` prints it to the console as shown.

3. Active Typechecking and Translation

Enabling the addition of new forms to a language in an open tool ecosystem is difficult. Indeed, it is the canonical example of the long-studied *expression problem* [34]. Although a number of approaches have been developed, we argue that many of them are overly permissive, making it difficult to reason compositionally about metatheoretic issues (e.g. type safety) and avoid ambiguities when extensions are combined (see Sec. 5). Ace's term forms, as we have seen, are fixed by Python's grammar, so Ace sidesteps the expression problem almost entirely (and thus inherits broad tool support, as described earlier). The key insight is that leaving the forms fixed does not mean the semantics must also be fixed. Instead of taking a syntax-directed view of extensibility, we take a type-directed view: users cannot add new forms but they can add new type constructors, which we empower with more control over the semantics of existing forms (e.g. literals, as discussed above, but also nearly every other form, as we will now discuss) in a controlled manner. The key features that characterize active typechecking and translation are:

- a *dispatch protocol* that delegates responsibility for typechecking and translating a term to:
  - a type or type constructor extracted from a subterm
  - a per-function base semantics (or simply *base*) for forms for which this is not possible (variables, literals without ascriptions and most statements)
- a mechanism for allowing the user to define new types, type constructors and bases from within the language (in particular, *from within the type-level language*) rather than fixing them ahead of time

To see how this works, let us trace through how Ace processes the simple example in Listing 6.

3.1 Base Decorators

Decorators in Python (line 2) are syntactic sugar: we could equivalently have omitted line 3 and inserted the top-level statement `example = py(example)` on the line after the function definition. Note, however, that `py` is not a function but an instance of a class, `examples.py.PyBase`, that inherits from `ace.Base`. It can be used as a decorator because `ace.Base` overloads the call operator, shown in Listing 7. The `_process` function (not shown) operates as follows:

1. The Python standard library is used to extract an abstract syntax tree (`AST`) from the function.
2. The closure of the function, together with the globals in the module it is defined in, are extracted to reify its static environment.
3. The argument annotations are processed. If provided in the body, as in our example, `_process` checks that the argument names are spelled correctly and evaluates the type-level expressions (e.g. `A` above) in the static environment to produce a mapping of argument names to types, called the argument signature. If Python 3's annotations were used, this is not necessary.

By the end of Listing 6, `example` is thus an instance of `ace.TypedFn`. It could have been constructed directly by calling the constructor on line 2 of Listing 7 (this would be inconvenient, but could be useful for metaprogramming).

3.2 Checking Typed Functions

When the Ace compiler begins typechecking a `TypedFn` (when asked to by `acec`, or when it is invoked interactively, not discussed in this paper), it proceeds as follows:

1. An `ace.Context` is constructed with a reference to the function (Listing 9, lines 2-3).
2. The base is asked to initialize the context. This can be seen on lines 2-4 of Listing 8, where the `PyBase` base adds an attribute tracking local bindings (initially only the arguments) and the return type, which will be synthesized but initially is not known.
3. For each statement in the body (after the documentation string and type annotation), the compiler delegates to either the base, or of a type synthesized from a subterm, by calling a method named `check_`F, where F is the form of the statement (derived directly from Python's `ast` package [2]). These methods are responsible for checking that the statement is well-typed (raising an `ace.TypeError` if not) and having any needed effect on the context.
4. The base synthesizes a type for the function as a whole via the `syn_FunctionDef_outer` method. Here, an `arrow` type is synthesized (lines 26-29) based on the argument signature and the synthesized return type.

The assignment statements on lines 6.5[3] and 6.6 are checked by the base method `check_Assign_Name` shown on lines 8.6-8.14 and the return statement on line 6.7 is checked by the base's `check_Return` on lines 8.19-8.23. Both work

[3] In this section, we will need to refer to code in Listings 6, 8, 9 and 10, so we adopt this syntactic convention to refer to line numbers for concision.

Listing 7 A portion of the `ace` core showing how a base can be used as a decorator to construct a typed function.

```
class TypedFn(object):
  def __init__(self, base, ast, static_env, arg_sig):
    # ...
  # ...
class Base(object):
  def __call__(self, f):
    (ast, static_env, arg_sig) = _process(f)
    return TypedFn(self, ast, static_env, arg_sig)
  # ...
```

similarly: if this is the first assignment to a particular name, or the first return statement seen, then the value must be able to synthesize a type in the current context. Otherwise, it is analyzed against the previously synthesized type.

3.3 Bidirectional Typechecking

Synthesis and analysis are mediated by the context via its `syn` and `ana` methods. To see how it works, let us begin at the first statement in our example, the assignment statement on line 6.5. As just stated, the method `check_Assign_Name` on lines 8.6-8.14 is called. Because the locals dictionary added to the context by the base does not yet have a binding for `x`, the base asks to synthesize a type for the value being assigned, `a.amount`, by calling `ctx.syn`.

This method is defined on lines 9.5-9.26. The relevant case in this method is the one on line 9.9, because `a.amount` is of the form `Attribute` according to Python's grammar. The context delegates to the type recursively synthesized for its value, `a`. We recurse back into `syn`, now taking the first branch for terms of the form `ast.Name`. Synthesizing a type for a name is delegated to the base by calling its `syn_Name` method. We can see its implementation for the base we are using on lines 8.37-8.44. The identifier `a` is an argument to the function, which the base included in the initial locals dictionary, so we hit a base case and the type `A` is synthesized.

We can now pop back up to line 9.11, which can now delegates synthesis of a type for `a.amount` to the type `A` via the `syn_Attribute` method. Recall that `A` was constructed using the `record` constructor in Listing 1. The definition of this type constructor is shown in Listing 10. In Ace, type constructors are classes inheriting from `ace.Type` and types are instances of such classes. Constructor indices are given through the class constructor, called `__init__` in Python. Here, `record` requires a signature, which is a mapping of field names to types (an instance of `examples.fp.Sig`, which performs well-formedness checking, not shown). The class decorator `slices_to_sig` provides the convenient notation shown used in Listing 1 (by adding a metaclass to override the class object's subscript operator, not shown). The `syn_Attribute` method is shown on lines 10.30-10.34. It simply looks in the signature for the provided field name, so `decimal[2]` is synthesized. We can now go back up to the assignment statement that triggered this chain of calls

Listing 8 A portion of the base used in our examples thus far, defined in the `examples.py` package.

```
class PyBase(ace.Base):
  def init_ctx(self, ctx):
    ctx.locals = dict(ctx.fn.arg_sig)
    ctx.return_t = None

  def check_Assign_Name(self, ctx, s):
    x, e = s.target.id, s.value
    if x in ctx.locals:
      ctx.ana(e, ctx.locals[x])
    else:
      ty = ctx.syn(e)
      ctx.locals[x] = ty

  def trans_Assign_Name(self, ctx, target, s):
    return target.direct_translation(ctx, s)

  def check_Return(self, ctx, s):
    if ctx.return_t == None:
      ctx.return_t = ctx.syn(s.value)
    else:
      ctx.ana(s.value, ctx.return_t)

  def trans_Return(self, ctx, target, s):
    return target.direct_translation(ctx, s)

  def syn_FunctionDef_outer(self, ctx, f):
    if ctx.return_t == None:
      ctx.return_t = unit
    return arrow[ctx.fn.arg_sig, ctx.return_t]

  def trans_FunctionDef_outer(self, ctx, target, f):
    if f.name == "__toplevel__":
      return target.Suite(ctx.trans(f.body))
    else:
      return target.direct_translation(f, ctx)

  def syn_Name(self, ctx, e):
    x = e.id
    if x in ctx.locals:
      return ctx.locals[x]
    elif x in ctx.fn.static_env: # (Sec. 5)
      return self.syn_lifted(x)
    else:
      raise ace.TypeError("...var not bound...", e)

  def trans_Name(self, ctx, target, e):
    if e.id in ctx.locals:
      return target.direct_translation(ctx, e)
    else: # (Sec. 5)
      return self.trans_lifted(ctx, target, e)

  default_Dict_asc = default_Str_asc = dyn
  # ...
  def init_target(ctx): return PyTarget()
py = PyBase()
```

and see on line 8.14 that a binding for `x` is added to the locals dictionary and checking of this statement succeeds.

The next statement also assigns to `x`, but this time, the base asks to analyze the value against `A` using the `ana` method of `Context`, shown on lines 9.28-9.37. Analysis differs from synthesis only for unascribed literal forms. For any other form, the context simply asks for an equal type to be synthesized (we will discuss in Sec. 6 future work on integrating subtyping, where this would be relaxed). For concision, we leave the details of the delegation protocol for binary operators to the appendix, which also shows the definition of the `decimal` type constructor. The addition of

Listing 9 The `ace.Context` class delegates typechecking and translation of expressions, depending on their form and sub-terms, to a base, a type or a type constructor.

```
1  class Context(object):
2    def __init__(self, fn):
3      self.fn = fn
4
5    def syn(self, e):
6      if instanceof(e, ast.Name):
7        delegate = self.fn.base
8        ty = delegate.syn_Name(self, e)
9      elif instanceof(e, ast.Attribute):
10       delegate = self.syn(e.value)
11       ty = delegate.syn_Attribute(self, e)
12     # ... other compound forms similar (cf appendix)
13     elif isinstance(e, ast.Str):
14       return self.ana(self, e,
15         self.fn.base.default_Str_asc)
16     # ... other unascribed literal forms similar
17     elif is_ascribed_Dict(e):
18       lit, delegate = get_lit(e)
19       if issubclass(delegate, Type): # tycon
20         ty = delegate.syn_Dict(self, lit)
21       else:
22         return self.ana_Dict(lit, delegate)
23     # ... other ascribed literal forms similar
24     e.delegate = delegate
25     e.ty = ty
26     return ty
27
28   def ana(self, e, ty):
29     if instanceof(e, ast.Dict):
30       ty.ana_Dict(self, e)
31     # ... other literal forms similar
32     else:
33       syn = self.syn(e)
34       if ty != syn:
35         raise TypeError("...syn/ana mismatch...", e)
36       return
37     e.delegate = e.ty = ty
38
39   def trans(self, target, e):
40     d = e.delegate
41     if instanceof(e, ast.Name):
42       trans = d.trans_Name(self, target, e)
43     elif instanceof(e, ast.Attribute):
44       trans = d.trans_Attribute(self, target, e)
45     # ... other forms similar
46     e.trans = trans
47     return trans
```

Listing 10 The `examples.fp.record` type constructor.

```
1  @slices_to_sig
2  class record(ace.Type):
3    def __init__(self, sig, anon=False):
4      self.sig, self.anon = sig, anon
5
6    @classmethod
7    def syn_Dict(cls, ctx, e):
8      sig = Sig((f, ctx.syn_ty(v))
9        for f, v in zip(e.keys, e.values))
10     return cls(sig, anon=True)
11
12   def ana_Dict(self, ctx, e):
13     for f, v in zip(e.keys, e.values):
14       if f.id in self.sig:
15         ctx.ana(v, self.sig[f.id])
16       else:
17         raise ace.TypeError("...extra field...", f)
18     if len(self.sig) != len(e.keys):
19       raise ace.TypeError("...missing field...", e)
20
21   def trans_Dict(self, ctx, target, e):
22     if len(self.sig) == 1:
23       return ctx.trans(e.values[0])
24     else:
25       value_dict = dict(zip(e.keys, e.values))
26       return target.Tuple(
27         ctx.trans(target, value_dict[field])
28         for field, ty in self.sig)
29
30   def syn_Attribute(self, ctx, e):
31     if e.attr in self.sig:
32       return self.sig[e.attr]
33     else:
34       raise ace.TypeError("...field not found...", e)
35
36   def trans_Attribute(self, ctx, target, e):
37     if len(self.sig) == 1:
38       return ctx.trans(target, e)
39     else:
40       idx = idx_of(self.sig, e.attr)
41       return target.Subscript(
42         ctx.trans(target, e.value), target.Num(idx))
43
44   def __eq__(self, other):
45     if isinstance(other, record):
46       if self.anon or other.anon: return self is other
47       else: return self.sig == other.sig
48
49   def trans_type(self, target):
50     return target.dyn
```

two terms of type `decimal[2]` has type `decimal[2]`, so checking the second assignment statement also succeeds.

The final statement in `example` is a return statement. Because it is the first return statement encountered, it too requires that the returned value synthesize a type. Here it is an ascribed literal. We can see on lines 9.17-9.22 how this is handled. Because the ascription is a type constructor, `record`, rather than a type, the literal synthesizes a type by calling a *class method*, `record.syn_Dict`, shown on lines 10.6-10.10. This method constructs a record signature by synthesizing a type for each value (using comprehension syntax for concision) then returns a new instance of the class. It is marked as anonymous. Anonymous records differ only in how equality is decided, shown on lines 10.44-10.47. We want anonymous records to be equal structurally, while records declared like `A` are by default distinct even if they have the same signature (following the convention of most functional languages with record declarations).

Synthesizing a type for the value of the field `y`, also an ascribed literal, takes a different path because the ascription is a type, `A`, not a type constructor. Type ascriptions, as we have seen in the previous examples, cause the literal to analyzed. This proceeds through the `ana_Dict` method on lines 10.12-10.19, which analyzes the value of each field against the corresponding type in the signature, raising a type error if there are missing or extra fields.

Synthesizing a type for the value of the field `remark`, an unascribed literal form, follows a third path, seen on lines 9.13-9.15. Unascribed literals are treated as if they had been given the default ascription specified by the base. Our base specifies the default ascription as `dyn` on line 8.52, which accepts our literal, so synthesis succeeds. The return type of `example` is thus equal to:

```
record(Sig(('y', A), ('remark', dyn)), anon=True)
```

3.4 Active Translation

Once typechecking is complete, the compiler enters the translation phase. The base creates an instance of a class inheriting from `ace.Target` for use during this phase via the `init_target` method (line 8.54). This object provides methods for code generation and supports features like fresh variable generation, adding imports and so on. It's interface is not constrained by Ace (we will see what Ace requires of a target below) and the mechanics of code generation are orthogonal to the focus of this paper, so we will discuss it relatively abstractly. The simplest API would be string-based code generation. For the Python target we use here, we generate ASTs. The API is based directly on the `ast` library, with a few additional conveniences just mentioned.

This phase follows the same delegation protocol as the typechecking phase. Each `check_`/`syn_`/`ana_`X method has a corresponding `trans_`X method. The typechecking phase saved the entity that was delegated control, along with the type assignment, as attributes of each node, `delegate` and `ty` respectively, so that these need not be determined again. This protocol can be seen on lines 9.39-9.47. Translation methods have access to the context and node, as during typechecking, as well as the target.

Because we are targeting Python directly, most of our `trans` methods are direct translations (factored out into a helper function). The main methods of interest that are not entirely trivial are `trans_FunctionDef_outer` on lines 8.33-8-37 and the two translation methods in Listing 10, which implement the logic described in Sec. 2.4.

Each type constructor must also specify a `trans_type` method. This is important when targeting a typed language (so that the translations of type annotations can be generated, for example). As we will see in the next two sections, this is also critical to ensuring type safety. The language *checks* not only that translations having a given type are well-typed, but that they have the type specified by this method (requiring that the target provide a `is_of_type` method). Here, because we are simply targeting Python directly, the `trans_type` method on line 10.49-10.50 simply generates `target.dyn`.

When this phase is complete, each node processed by the context will have a translation, available via the `trans` attribute. In particular, each typed function has a translation. Note that some nodes are never processed by the context because they were reinterpreted by the delegate (e.g. the field names in a record literal), so they do not have translations (as expected, given our discussion above).

To support external compilation, the target must have an `emit` method that takes the compilation script's file name and a reference to a string generator (an instance of `ace.util.CG`) and emits source code. The string generator we provide can track indentation levels (to support Python code generation and make generation for other languages more readable, for the purposes of debugging). It allows non-local string generation via the concept of user-defined *locations*. Each file that needs to be generated is a location and there can also be locations within a file (e.g. the imports vs. the top-level code), specified by a target the first time it finds that a necessary location is not defined. A generated entity (e.g. an import, class definition or function definition) can only be added once at a location. We saw this in Listing 3 for the decimal-to-string conversion. The API will be discussed further in the appendix.

$$
\begin{array}{rcl}
\rho & ::= & \mathsf{tycon}\ \textsc{tycon}\ \mathsf{of}\ \kappa_{\text{idx}}\ \{TC\};\rho \mid e \\
TC & ::= & \mathsf{iana}\ \{\tau\};\mathsf{isyn}\ \{\tau\};\mathsf{esyn}\ \{\tau\};\mathsf{rep}\ \{\tau\} \\
\\
e & ::= & x \mid \lambda x{:}\tau.e \mid I \mid I : \tau \mid I :: \textsc{tycon} \mid E \\
I & ::= & \mathsf{intro}[\tau_{\text{idx}}](e_1;\ldots;e_n) \\
E & ::= & e \cdot \mathsf{elim}[\tau_{\text{idx}}](e_1;\ldots;e_n) \\
\\
\tau & ::= & \mathbf{t} \mid \lambda \mathbf{t}{:}\kappa.\tau \mid \tau_1\ \tau_2 \\
 & \mid & [] \mid \tau_1 :: \tau_2 \mid \mathsf{fold}(\tau_1;\tau_2;\mathbf{h},\mathbf{t},\mathbf{r}.\tau_3) \\
 & \mid & \ell \mid () \mid (\tau_1,\tau_2) \mid \mathsf{fst}(\tau) \mid \mathsf{snd}(\tau) \mid \cdots \\
 & \mid & \textsc{tycon}\langle\tau_{\text{idx}}\rangle \\
 & \mid & \mathsf{case}\ \tau\ \mathsf{of}\ \textsc{tycon}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 \\
 & \mid & \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 \\
 & \mid & [\![\tau_1 \rightsquigarrow \tau_2]\!] \mid \mathsf{error} \mid \mathsf{typeof}(\tau) \mid \mathsf{transof}(\tau) \\
 & \mid & \triangleright(\iota) \mid \blacktriangleright(\sigma) \mid \mathsf{repof}(\tau) \\
 & \mid & (\Gamma;e)? \mid \mathsf{syn}(\tau) \mid \mathsf{ana}(\tau;\tau') \\
\kappa & ::= & \kappa_1 \to \kappa_2 \mid \mathsf{list}[\kappa] \mid \mathsf{L} \mid \mathsf{1} \mid \kappa_1 \times \kappa_2 \mid \cdots \\
 & \mid & \mathsf{Ty} \mid \mathsf{TT} \mid \mathsf{ITm} \mid \mathsf{ITy} \mid \mathsf{Arg} \\
\\
\iota & ::= & x \mid \mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ \iota \mid \lambda x{:}\sigma.\iota \mid \iota_1\ \iota_2 \\
 & \mid & () \mid (\iota_1,\iota_2) \mid \mathsf{fst}(\iota) \mid \mathsf{snd}(\iota) \mid \cdots \mid \triangleleft(\tau) \\
\sigma & ::= & \sigma_1 \rightharpoonup \sigma_2 \mid \mathsf{1} \mid \sigma_1 \times \sigma_2 \mid \cdots \mid \blacktriangleleft(\tau)
\end{array}
$$

Figure 1. Syntax of Core λ_{Ace}. Here, x ranges over external and internal language variables, $\mathbf{t}$ ranges over type-level variables, TYCON ranges over type constructor names and ℓ ranges over labels.

4. Theoretical Foundations

We will now give a core typed lambda calculus, λ_{Ace}, that captures the semantics described in the previous sections. It is intended to make precise how active typechecking and translation works and how our mechanism relates to existing work on bidirectional typechecking, type-level computation and typed compilation while abstracting away from the details of Python's syntax and imposing a stronger type-level semantics that will allow us to state metatheoretic properties of interest. We will assume a fixed base for functions (providing the standard semantics of lambda functions) and target language, which we here call the *internal language*.

The syntax of λ_{Ace} is shown in Figure 1 and an example *program* that defines a type constructor, RECORD, with a

```
tycon RECORD of list[L × Ty] {                                          (1)
  iana {λi:list[L × Ty] × list[L].λa:list[Arg].                         (2)
    fold(zip3 fst(i) snd(i) a; ▷(()); h, t, r.                          (3)
      if fst(first h) ≡L second h then                                  (4)
        let x : ITm = ana(third h; snd(first h)) in                     (5)
        fold(t; x; _, _, _. ▷ ((◁(x), ◁(r)))) else error)}              (6)
  isyn {λi:list[L].λa:list[Arg].                                        (7)
    fold(zip2 i a; ⟦RECORD⟨[]⟩ ⇝ ▷(())⟧; h, t, r.                       (8)
      let htt : TT = syn(snd(h)) in                                     (9)
      let hty : Ty = typeof(htt)                                        (10)
      let ty : Ty = RECORD⟨(fst(h), hty) :: []⟩ in                       (11)
      fold(t; ⟦ty ⇝ transof(htt)⟧; _, _, _.                             (12)
        ⟦ty ⇝ ▷((◁(transof(htt)), ◁(transof(r))))⟧))}                   (13)
  esyn {λi:L.λa:list[Arg].arity1 a λty:Ty.λx:ITm.                       (14)
    case ty of RECORD⟨sig⟩ ⇒                                             (15)
      fold(sig; error; h, t, r.                                         (16)
        if fst(h) ≡L i then fold(t; ⟦snd(h) ⇝ x⟧; _, t', _.              (17)
          fold(t'; ⟦snd(h) ⇝ ▷(fst(◁(x)))⟧; _, _, r'.                   (18)
            ⟦snd(h) ⇝ ▷(snd(◁(transof(r))))⟧))                          (19)
        else r)                                                         (20)
    ow error                                                            (21)
  rep {λi:list[L × Ty].                                                 (22)
    fold(i; ▶(1); s, j, r.fold(j; repof(snd(s)); _, _, _.               (23)
      ▶(◀(repof(snd(s))) × ◀(r))))}                                     (24)
};                                                                      (25)
let R : Ty = RECORD⟨(ℓ1, RECORD⟨[]⟩) :: []⟩ in                          (26)
let id = λx:R.{ℓ1 = x · ℓ1} :: RECORD                                   (27)
let triv = id[{ℓ1 = {}}] · ℓ1                                           (28)
```

Figure 2. The definition of the record type in λ_{Ace} using the desugarings in Figure 3 and with the addition of simple let bindings for both type and term variables (not shown). Some type-level helper functions also omitted for concision.

semantics similar to that given in Listing 10[4], using it to write an identity function and compute the value $triv$ (the empty record), is shown in Figure 2.

4.1 Overview

A *program*, ρ, consists of a series of constructor declarations followed by an external term, e. Compiling a program consists of first *kind checking* it (see below), then typechecking the external term and simultaneously translating it to a term, ι, in the typed internal language. The key judgements are the *bidirectional active typing judgements* (Fig. 3, which we describe starting in Sec. 4.3). They relate an external term, e, to a type, τ, called its *type assignment*, and an internal term, ι, called its *translation*, under *typing context* Γ and *constructor context* Φ. The first is synthesis, the second analysis.

$$\Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota \qquad \Gamma \vdash_\Phi e \Leftarrow \tau \rightsquigarrow \iota$$

The typing context, Γ, maps variables to types in essentially the conventional way ([12] contains the necessary background for this section). The constructor context, Φ, tracks user-defined type constructors.

The dynamic behavior of an external term is determined entirely by its translation to the internal language, which has a conventional operational semantics. This form of semantics can be seen as lifting into the language specification the first stage of a type-directed compiler like the TIL compiler for Standard ML [26] and has some parallels to the Harper-Stone semantics for Standard ML, where external terms were also given meaning by elaboration from the EL to an IL [13].

In λ_{Ace}, the internal language (IL) provides partial functions (via the generic fixpoint operator of Plotkin's PCF) and simple product types for the sake of our example. In practice, the internal language could be any typed language with a specification for which type safety and decidability of typechecking have been satisfyingly determined. The internal type system serves as a "floor": guarantees that must hold for terms of any type (e.g. that out-of-bounds access to memory never occurs) must be maintained by the internal type system. User-defined constructors can enforce invariants stronger than those the internal type system maintains at particular types, however. Performance is also ultimately limited by the internal language and downstream compilation stages that we do not here consider (safe compiler extension has been discussed in previous work, e.g. [27]).

4.2 Types and Type-Level Computation

λ_{Ace} supports, and makes extensive use of, simply-kinded type-level computation. Specifically, type-level terms, τ, themselves form a typed lambda calculus. The classifiers of type-level terms are called *kinds*, κ, to distinguish them from *types*. Types are type-level values of kind Ty. In Ace, the type-level language (together with the level of programs) is written in Python, which can be thought of as having a rather limited kind system (with one kind, dyn). Here, we are able to more precisely discuss the kinds of values in the type-level language.

In most languages, types are formed by applying one of a collection of *type constructors* to zero or more *indices*. In λ_{Ace}, the situation is notionally similar. User-defined type constructors can be declared at the top of a program (or lifted to the top, in practice) using tycon. Each constructor in the program must have a unique name, written e.g. RECORD.[5] A type constructor must also declare an *index kind*, κ_{idx}.

[4] Technically, this defines labeled tuples, because the order of labels matters.

[5] We assume naming conflicts can be avoided by some extrinsic mechanism.

To permit the embedding of interesting type systems, the type-level language includes several kinds other than Ty. We functional data structures to the type level: here, only unit (1), binary products ($\kappa_1 \times \kappa_2$) and lists (list$[\kappa]$), in addition to labels (introduced as ℓ, possibly with a subscript, having kind L). Our record type constructor is indexed by a list of pairs of labels and types (a signature, in essence; line 1). The type constructor ARROW is included in the initial constructor context, Φ_0 and has index kind Ty $\times$ Ty. In practice, one could include a richer functional programming language and retain the spirit of the calculus, as long as it does not introduce general recursion at the type level.

A type is introduced by applying a type constructor to an index of this kind, written $\textsc{Tycon}\langle\tau_{\text{idx}}\rangle$. For example, the type of natural numbers is indexed trivially, so it is written $\textsc{Nat}\langle()\rangle$. We see a record type, abbreviated **R**, constructed on line 26.

The kind Ty also has an elimination form,

$$\mathsf{case}\ \tau\ \mathsf{of}\ \textsc{Tycon}\langle\mathbf{x}\rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2$$

allowing the extraction of a type index by case analysis against a contextually-available type constructor. To a first approximation, one might think of type constructors as constructors of a built-in open [17], Ty, at the type-level. Like open datatypes, there is no notion of exhaustiveness so the default case is required for totality. We will see where this is used shortly.

Type constructors are not first-class; they do not themselves have arrow kind as in some kind systems (e.g. [36]; Ch. 22 of *PFPL* describes a related system [12]). The type-level language does, however, include total functions of arrow kind, written $\kappa_1 \to \kappa_2$. Type constructor application can be wrapped in a type-level function to emulate a first-class or uncurried version of a type constructor for convenience (indeed, such a wrapper could be generated automatically, though we do not do so).

Two type-level terms of kind Ty are equivalent if they apply the same constructor, identified by name, to equivalent indices. Going further, we ensure that deciding type equivalence requires only checking for syntactic equality after normalization by imposing the restriction that equivalence at a type constructor's index kind must be decidable in this way. Our treatment of equivalence in the type-level language is thus quite similar to the treatment of term-level equality using "equality types" in a language like Standard ML. Conditional branching on the basis of equality at an equality kind can be performed in the type-level language. Equivalence at arrow kind is not decidable by our criteria, so type-level functions cannot appear within type indices. This also prevents general recursion from arising at the type level. Without this restriction, a type-level function taking a type as an argument could "smuggle in" a self reference as a type index, extracting it via case analysis (continuing our analogy to open datatypes, this is closely related to the positivity condition for inductive datatypes in total functional languages like Coq).

$$\begin{array}{rcl}
\{\ell_1 = e_1, \ldots, \ell_n = e_n\} & := & \mathsf{intro}[\ell_1 :: \ldots :: \ell_n :: []](e_1; \ldots; e_n) \\
(e_1, ..., e_n) & := & \mathsf{intro}[()](e_1; \ldots; e_n) \\
\#n & := & \mathsf{intro}[n]() \\
"s" & := & \mathsf{intro}[s]() \\
e.\ell & := & e \cdot \mathsf{elim}[\ell]() \\
e[e_1; \ldots; e_n] & := & e \cdot \mathsf{elim}[()](e_1; \ldots; e_n) \\
e.\ell(e_1; ...; e_n) & := & e \cdot \mathsf{elim}[\ell](e_1; \ldots; e_n)
\end{array}$$

Figure 3. Desugaring from conventional syntax to core forms. We assume that the type-level language has numbers, n, and strings, s (not shown for concision).

Every type constructor also defines a *representation*, a type-level function that associates with every type an internal *representation type* (analagous to `trans_type` above). We will return to this after introducing the external forms and operator definitions.

4.3 Core External Forms and Desugaring

The syntax for external terms (Figure 1) contains variables, λ terms, three generalized introductory forms and a single generalized elimination form. The introductory forms are either unascribed, ascribed with a type or ascribed with a type constructor, as in our discussion of Ace. These generalized forms take a single a type-level value as an index and $n \geq 0$ arguments, which are other external terms. To better motivate this choice, we can give a purely syntactic desugaring of a Python-like syntax with labels to these forms, shown in Figure 3. It is instructive to rewrite lines 27-28 of Figure 2 using these desugarings.

4.4 Operator Definitions and Representational Consistency

The expressive and metatheoretic power of the calculus arises from how the rules for the active typing judgement handle these generalized forms. Rather than fixing the specification of a finite collection of operator constructors and tasking the *compiler* with deciding a typing derivation on its basis, the specification instead delegates to a type-level function associated with a type constructor. In the core calculus, this is one of three functions, called iana, isyn and esyn in the grammar.[6] We see how this is done with the rules for the active typing judgements, given in Figure 4.

The first two rules are standard in bidirectional type systems: variables synthesize types and if a term synthesizes a type, it can be analyzed against a type (cf. Listing 9). Variables translate to variables, and if we are simply converting from synthesis to analysis, translation is not affected.

[6] This means that any particular type constructor only supports a single introduction and elimination desugaring. This is a minor inconvenience in some cases that is resolved in Ace by the use of methods.

$\boxed{\Gamma \vdash_\Phi e : \tau \rightsquigarrow \iota}$ $\quad \Gamma ::= \emptyset \mid \Gamma, x \Rightarrow \tau$

ATT-VAR
$$\frac{x \Rightarrow \tau \in \Gamma}{\Gamma \vdash_\Phi x \Rightarrow \tau \rightsquigarrow x}$$

ATT-SYN-TO-ANA
$$\frac{\Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota}{\Gamma \vdash_\Phi e \Leftarrow \tau \rightsquigarrow \iota}$$

ATT-LAM
$$\frac{\tau_1 \Downarrow_\Phi \tau_1' \qquad \mathsf{repof}(\tau_1') \Downarrow_\Phi \blacktriangleright(\sigma) \qquad \Gamma, x \Rightarrow \tau_1' \vdash_\Phi e \Rightarrow \tau_2 \rightsquigarrow \iota}{\Gamma \vdash_\Phi \lambda x{:}\tau_1.e \Rightarrow \textsc{arrow}\langle(\tau_1', \tau_2)\rangle \rightsquigarrow \lambda x{:}\sigma'.\iota}$$

ATT-I-UNASC
$$\frac{\vdash_\Phi \mathsf{iana}(\textsc{tycon}) = \tau_{\text{def}} \qquad \tau_{\text{def}}\ (\tau_{\text{idx}}, \tau_{\text{idx}}')\ ((\Gamma; e_1)? :: \ldots :: (\Gamma; e_n)? :: [])\Downarrow_\Phi \triangleright(\iota) \qquad \mathsf{repof}(\textsc{tycon}\langle\tau_{\text{idx}}'\rangle) \Downarrow_\Phi \blacktriangleright(\sigma) \qquad \Gamma \vdash_\Phi \iota : \sigma}{\Gamma \vdash_\Phi \mathsf{intro}[\tau_{\text{idx}}](e_1; \ldots; e_n) \Leftarrow \textsc{tycon}\langle\tau_{\text{idx}}'\rangle \rightsquigarrow \iota}$$

ATT-I-ASC-TY
$$\frac{\Gamma \vdash_\Phi I \Leftarrow \tau \rightsquigarrow \iota}{\Gamma \vdash_\Phi I : \tau \Rightarrow \tau \rightsquigarrow \iota}$$

ATT-I-ASC-TYCON
$$\frac{\vdash_\Phi \mathsf{isyn}(\textsc{tycon}) = \tau_{\text{def}} \qquad \tau_{\text{def}}\ \tau_{\text{idx}}\ (\Gamma; e_1)?::\ldots::(\Gamma; e_n)?::[]) \Downarrow_\Phi [\![\textsc{tycon}\langle\tau_{\text{idx}}'\rangle \rightsquigarrow \triangleright(\iota)]\!] \qquad \mathsf{repof}(\tau_1') \Downarrow_\Phi \blacktriangleright(\sigma) \qquad \Gamma \vdash_\Phi \iota : \sigma}{\Gamma \vdash_\Phi \mathsf{intro}[\tau_{\text{idx}}](e_1; \ldots; e_n)] :: \textsc{tycon} \Rightarrow \tau \rightsquigarrow \iota'}$$

ATT-ELIM
$$\frac{\Gamma \vdash_\Phi e \Rightarrow \textsc{tycon}\langle -\rangle \rightsquigarrow - \qquad \vdash_\Phi \mathsf{esyn}(\textsc{tycon}) = \tau_{\text{def}} \qquad \tau_{\text{def}}\ ((\Gamma; e)? :: (\Gamma; e_1)? :: \ldots :: (\Gamma; e_n)? :: [])\Downarrow_\Phi [\![\iota \rightsquigarrow \tau]\!] \qquad \mathsf{repof}(\tau) \Downarrow_\Phi \blacktriangleright(\sigma) \qquad \Gamma \vdash_\Phi \iota : \sigma}{\Gamma \vdash_\Phi e \cdot \mathsf{elim}[\tau_{\text{idx}}](e_1; \ldots; e_n) \Rightarrow \tau \rightsquigarrow \iota}$$

Figure 4. The active typing judgement. The normalization judgement for type-level terms ($\Downarrow$) and the representational consistency check will be in an appendix.

The rule ATT-LAM must take into account the fact that, because we support type-level computation, the type annotation on the argument may not be in normal form. Thus, we evaluate it to normal form. Note that the kinding rules (not shown) will guarantee that, because τ_1 is of kind Ty, its normal form, τ_1', is of the form $\textsc{tycon}\langle\tau_{\text{idx}}\rangle$ for some normal τ_{idx}. To generate an appropriate internal type annotation in the translation, we need to compute the representation type of τ_1'. This involves calling the representation function associated with its type constructor (rule REPOF in Figure 5). The form $\triangleright(\sigma)$ is a *quoted internal type*. Note in the syntax that there is an unquote form, $\blacktriangleleft(\tau)$, which allows us to compose internal types compositionally without needing to expose an elimination form. Indeed, it is an interesting facet of our calculus that we never need to examine syntax trees directly to implement extensions. The evaluation semantics remove quotations, so the normal form of a quoted internal type contains an internal type with no quotations.

Lambda functions are the only introductory form requiring special support in the calculus (because they need to manipulate the context; see Discussion). The next three rules

REPOF
$$\frac{\tau \Downarrow_\Phi \textsc{tycon}\langle\tau_{\text{idx}}\rangle \qquad \vdash_\Phi \mathsf{rep}(\textsc{tycon}) = \tau_{\text{rep}} \qquad \tau_{\text{rep}}\tau_{\text{idx}} \Downarrow_\Phi \blacktriangleright(\sigma)}{\mathsf{repof}(\tau) \Downarrow_\Phi \blacktriangleright(\sigma)}$$

SYN
$$\frac{\tau \Downarrow_\Phi (\Gamma; e)? \qquad \Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota}{\mathsf{syn}(\tau) \Downarrow_\Phi [\![\tau \rightsquigarrow \triangleright(\iota)]\!]}$$

ANA
$$\frac{\tau_1 \Downarrow_\Phi (\Gamma; e)? \qquad \tau_2 \Downarrow_\Phi \textsc{tycon}\langle\tau_{\text{idx}}\rangle \qquad \Gamma \vdash_\Phi e \Leftarrow \textsc{tycon}\langle\tau_{\text{idx}}\rangle \rightsquigarrow \iota}{\mathsf{ana}(\tau_1; \tau_2) \Downarrow_\Phi \triangleright(\iota)}$$

Figure 5. Evaluation semantics for the type-level language. Missing rules (including error propagation rules, which immediately cause failure of typechecking) are unsurprising and will be given in an appendix.

show how any other abstractions that we define (e.g. records, decimals, etc.) make use of a generalized introductory form.

The rule ATT-I-UNASC shows that unascribed introductory forms can only be analyzed against a type. Given such a type, the rule extracts a definition, named iana, from the type constructor (cf. the `ana_Dict` and `trans_Dict` methods earlier). It calls this function with a pair containing the operator and type index and a list of *reified arguments*, which have kind Arg and introductory form $(\Gamma; e)?$. These are only constructed by the compiler (there would be no corresponding form in the concrete syntax). Their purpose is to allow the definition to programatically invoke synthesis and analysis as needed using the ana and syn operators. We can see the RECORD type constructor doing so on line 5 of Figure 2 to ensure that the field value provided as an argument has the same type as the corresponding label (accomplished by simultaneously folding over all three pieces of input data). The rule for performing analysis, ANA, is in Figure 5. If it succeeds, analysis returns a *translation*, which is a quoted internal term with kind ITm and introductory form $\triangleright(\iota)$. Like quoted internal types, there is an unquote form that is eliminated during evaluation. The operator definition uses this to construct a translation for the record. As before, empty records translate to units and records with a single field are unadorned. In this example, records with two or more fields translate to nested tuples.

Because we have a representation type associated with the type, we can check that the translation is *representationally consistent* (the final premise). As we will discuss, representational consistency combined with type safety of the internal language implies type safety overall (it arises as a strengthening of the inductive hypothesis needed to prove type safety, and is closely related to work on *type-preserving compilation* in the TIL compiler for Standard ML, [26]).

The next rule, ATT-I-ASC-TY states that introductory forms ascribed with a type are analyzed against that type.

Introductory forms ascribed with a type constructor can, according to the final introductory rule, ATT-I-ASC-TYCON, synthesize a type via the definition isyn. It is passed the operator index (there is no type index, since we only have a type constructor) and a list of arguments, as before. In this case, it must return not just a translation, but also a type. We use the form $[\![\tau_1 \rightsquigarrow \tau_2]\!]$ for such a pairing, which has kind TT. The type and translation can be extracted from it using the appropriately named elimination forms (indeed, as given, it is merely a pair, but we give it special syntax for clarity – it looks like the conclusion of the typing judgement – and for other reasons that will become clear in future work). To synthesize a type from a list of labels and arguments, we must be able to synthesize types from the arguments. The $\mathsf{syn}(\tau)$ operator permits this, per rule SYN, seen being used in Listing 2. We check representational consistency of the result, as before.

Finally, we show the rule for elimination forms. It operates by first synthesizing a type for the "primary" subterm, then extracting the esyn definition from its type constructor. As before, it is called with the arguments and representational consistency is checked. Note that the primary subterm is itself the first argument (though we have already synthesized a type for it, it is more uniform and clear to allow the operator to do so again, which is done in our example by the **arity1** helper function).

4.5 Metatheory

The metatheoretic properties of interest are: type safety and termination of the type-level language (guaranteed by our kind system, though the details will need to be provided in an appendix; termination is non-trivial), type safety and decidability of the internal language (it is a standard variant of PCF, so this is trivial) and representational consistency (discussed above, follows because we check it explicitly). Type safety for the language as a whole comes as a corollary of these lemmas. We plan to provide detailed proofs of these, but for now, they should be treated as conjectures and our formulation above as expository. We believe that despite this, the formalization is surprisingly elegant and concise, given its expressive power. It is a useful exercise to implement natural numbers, ala Gödel's **T**, using this calculus, assuming that the type-level and internal languages include integers and one can lift them from the former into the latter.

5. Related Work

Libraries containing compile-time logic have previously been called *active libraries* [32]. A number of projects, such as Blitz++, have taken advantage of C++ template metaprogramming to implement domain-specific optimizations [31]. Others have chosen custom metalanguages (e.g. Xroma [32], mbeddr [33]). In Ace, we replace these brittle mini-languages with a general-purpose language, Python, and significantly expand the notion of active libraries by consideration of types, base semantics and target languages as values (in this case, objects). We borrow the term "active" due to this relationship.

Operator overloading [29] and *metaobject dispatch* [15] are run-time protocols that translate operator invocations into function calls. The function is typically selected according to the dynamic type or value of one or more operands. These protocols share the notion of *inversion of control* with our strategy. However, our strategy is a *compile-time* protocol where the typechecking and translation semantics are implemented for different operators. Note that for syntactic convenience, we used Python's operator overloading and metaobject protocol substantially at the type level.

There are several other mechanisms that have been described as enabling forms of internal "extension" in the literature. These differ from our mechanism in one of the following ways, summarized in Fig. 6:

- Our mechanism is itself implemented as a library, rather than as a language dialect.
- Our mechanism reuses an existing language's syntax, rather than offering syntax extension support. As we saw, this can still be reasonably natural, and allows the language to benefit from a variety of existing tools. We avoid many facets of the expression problem [34] in this way.
- Our mechanism permits true type system extensions. New types are not merely aliases, nor must their rules be admissible in some base type system. Term rewriting and macro systems, as well as systems that involve cross-compilation do not handle type system extensions at all. LMS, for example, uses Scala's type system, focusing instead on code generation and optimization [23]. This is orthogonal to the goals we are seeking to achieve (e.g. the regular expression types we have described would be somewhat difficult to implement as libraries in Scala).
- Techniques that allow global extensions to the syntax or treat the semantics as a "bag of rules" (e.g. SugarJ or rule injection systems like Xroma, Typed Racket (and other typed LISPs) and mbeddr) allow extensions so much control that there can be ambiguities. In Ace, there *can never be more than one extension assigned as the delegate for a term*, so there are no ambiguities.
- Our mechanism allows users to control the target language of compilation from within libraries.

When the mechanisms available in an existing language prove insufficient, researchers and domain experts often design a new language. Previous work has considered adding statically-typed features to languages like Python in this way. For example, Terra embeds a low-level statically-typed language within Lua [8]. A number of tools have been developed to assist with this task, including compiler generators, language workbenches and domain-specific language frame-

Approach	Examples	Library	Extensible Syntax	Extensible Type System	Unambiguous Composition	Alternative Targets
Active Types	Ace	●	○	●	●	●
Desugaring	SugarJ [9]	○	●	○	○	○
Type-Specific Literals	Wyvern [22]	○	●	○	●	○
Rule Injection	Racket [28], Xroma [32], mbeddr [33]	[28]	[33]	●	○	○
Attribute Grammars	Silver [30], Xoc [7]	○	●	(unsafe)	[24]	○
Static macros / Metaprogramming	Scala [3], MorphJ [14], MetaML [25], Template Haskell	○	○	○	●	○
Cross-Compilation	LMS/Delite [5, 23]	●	○	○	○	●

Figure 6. Comparison to related approaches to language extensibility.

works (cf. [10]). Extensible compilers like Xoc [7] can be considered a form of language framework as well. It is difficult or impossible for these language-external approaches to achieve interoperability, because languages are not aware of each other's semantics and merging languages is not guaranteed to be unambiguous and sound. A few approaches (e.g.) have made steps toward addressing the issue of ambiguity (albeit with some cost in expressiveness), but soundness is not guaranteed. In Ace (and in the theory, assuming a suitable packaging system), using different type constructors at the same time cannot cause type safety issues because representational consistency is checked. Representational consistency permits compositional reasoning.

Bidirectional type systems have been used for adding refinement checking to languages like ML and Twelf [18]. Refinement types can add stronger static logic, but do not have control over translation and have not been shown practical in a language as widely adopted as Python. We believe this is a fruitful avenue for future work. The Wyvern programming language uses bidirectional typechecking to control aspects of parsing in a manner similar to how we treat introductory forms, but Wyvern is also a language dialect and does not have an extensible type system [22].

6. Discussion

This work aimed to show that one can add static typechecking to a language like Python as a library, without undue syntactic overhead. Moreover, the type system is not fixed, but flexibly extensible. We achieve this by using a bidirectional type system and an elaboration semantics targeting a user-defined target language, along with per-function base semantic annotations. We show that the core of this mechanism leads to a surprisingly clean theoretical formalism that combines work on type-level computation, typed compilation and bidirectional type systems to enable type system extensions over a fixed, but flexible, syntax.

Ace has several limitations still. Debuggers and other tools that rely not just on Python's syntax but also its semantics cannot be used directly, so if code generation introduces significant complexity that leads to bugs, this can be an issue. We believe that active types can be used to control debugging, 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) using Ace. In particular, it is difficult to orthogonally implement type systems that use different contexts for typing, because the base controls the context (though hacks are possible). Not all extensions will be useful. Indeed, some language designers worry that offering too much flexibility to users leads to abuse (this is, for example, widely credited as the reason why Java doesn't support operator overloading). We do not argue with this point. Instead, we argue that the potential for abuse 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 a convergence toward stable collections of curated, high-quality collections more quickly than is possible today.

The mechanisms described here can be implemented within any language that offers some form of quotation of function ASTs, capturing of function closures, and a reasonably flexible collection of syntactic forms and basic reflection facilities.

References

[1] Pep 3107 – function annotations (http://legacy.python.org/dev/peps/pep-3107/), 2010.

[2] The python language reference (http://docs.python.org/), 2013.

[3] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[4] V. Cavé, Z. Budimlić, and V. Sarkar. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*, page 9. ACM, 2010.

[5] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In C. Cascaval and P.-C. Yew, editors, *PPoPP*, pages 35–46. ACM, 2011.

[6] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang. An empirical study of programming language trends. *Software, IEEE*, 22(3):72–79, 2005.

[7] R. Cox, T. Bergan, A. T. Clements, M. F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In S. J. Eggers and J. R. Larus, editors, *ASPLOS*, pages 244–254. ACM, 2008.

[8] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *PLDI*, pages 105–116, 2013.

[9] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, 2011.

[10] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *Software Language Engineering*, pages 197–217. Springer, 2013.

[11] N. Fulton. A typed lambda calculus for input sanitation. Senior thesis, Carthage College, 2013.

[12] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[13] R. Harper and C. Stone. A type-theoretic interpretation of standard ml. In *IN PROOF, LANGUAGE AND INTERACTION: ESSAYS IN HONOUR OF ROBIN MILNER*. MIT Press, 2000.

[14] S. S. Huang and Y. Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst.*, 33(2):6:1–6:44, Feb. 2011.

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

[16] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N. Meyrowitz, editor, *OOPSLA*, volume 21:11 of *ACM Sigplan Notices*, pages 214–223, Oct. 1986.

[17] A. Löh and R. Hinze. Open data types and open functions. In A. Bossi and M. J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pages 133–144. ACM, 2006.

[18] W. Lovas and F. Pfenning. A bidirectional refinement type system for lf. In *Electronic Notes in Theoretical Computer Science, 196:113–128, January 2008. [NPP07] [Pfe92] [Pfe93] [Pfe01] Aleksandar Nanevski, Frank Pfenning, and Brigitte*, 2008.

[19] L. A. Meyerovich and A. Rabkin. How not to survey developers and repositories: experiences analyzing language adoption. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, PLATEAU '12, pages 7–16, New York, NY, USA, 2012. ACM.

[20] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *OOPSLA*, OOPSLA '13, pages 1–18, New York, NY, USA, 2013. ACM.

[21] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana. A survey of scientific software development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 12. ACM, 2010.

[22] C. Omar, B. Chung, D. Kurilova, A. Potanin, and J. Aldrich. Type-directed, whitespace-delimited parsing for embedded dsls. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 8–11, New York, NY, USA, 2013. ACM.

[23] 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.

[24] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In M. Hind and A. Diwan, editors, *PLDI*, pages 199–210. ACM, 2009.

[25] T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–??, 1999.

[26] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[27] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 111–121. ACM, 2010.

[28] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM.

[29] 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*, 5:1–236, 1975.

[30] 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.

[31] T. L. Veldhuizen. Blitz++: The library that thinks it is a compiler. In *Advances in Software tools for scientific computing*, pages 57–87. Springer, 2000.

[32] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proc. 1998 SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

[33] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[34] P. Wadler. The expression problem. *java-genericity Mailing List*, 1998.

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

[36] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. Specifying properties of concurrent computations in clf. *Electronic Notes in Theoretical Computer Science*, 199:67–87, 2008.

Listing 11 Portions of the target showing how non-local code emission works.

```
class PyTarget(ace.Target):
  class PyAST(object): """Base class for ASTs"""
  target_type = ast
  class Attribute(PyAST, ast.Attribute):
    def emit(self, cg):
      cg.append(self.value.emit(cg), '.', self.attr)
  # ...
  class AnonModule(object):
    def __init__(self, name):
      self.name = name
      self.guid = Guid()

    def emit(self, cg):
      import_loc = cg['imports']
      _cache = AnonModule._idx_cache[import_loc]
      if self not in _cache:
        anon_name = '__ace_' + str(len(_cache)+1)
        imp_stmt = ('import ' + self.name + ' as ' +
          anon_name + cg.newline)
        _cache[self] = (anon_name, imp_stmt)
      else:
        anon_name, imp_stmt = _cache[self]
      imports.add_snippet(imp_stmt)
      cg.append(anon_name)

  def emit(self, script_name, cg):
    file_name = '_' + script_name + '.py'
    if file_name not in generator:
      if 'imports' not in generator[file_name]:
        generator[file_name].push_loc('imports')
      if 'main' not in format_set[file_name]:
        generator[file_name].push_loc('main')
      translation.emit((file_name, 'main'))
```

A. Source Code Emission

Portions of the `Target` class related to source code emission are shown. If compilation occurred externally, the `acec` library asks the targets used by the top-level typed functions to generate one or more files by deciding on a name and file extension on the basis of the name of the compilation script, and emitting code and *snippets*, which are simply strings corresponding to functions, type declarations, imports and other top-level entities in the target language, each inserted at a *location*. Each file being generated is a location, and locations might also be nested, depending on the language (e.g. the import block may be a separate location when generating Java, or the body of the <head> tag when generating HTML+CSS+Javascript). Snippets are only added once to a particular location (so imports are not duplicated, for example).

B. Kinding

Some of the kinding rules for type-level terms are shown.

C. Recursive Labeled Sums

Listing 12 shows how recursive labeled sums (i.e. functional datatypes) would look.

$\boxed{\Delta \vdash_\Phi \tau : \kappa}$

$$\text{K-VAR}\ \frac{}{\Delta, \mathbf{t} : \kappa \vdash_\Phi \mathbf{t} : \kappa} \qquad \text{K-ARROW-I}\ \frac{\Delta, \mathbf{t} : \kappa_1 \vdash_\Phi \tau : \kappa_2}{\Delta \vdash_\Phi \lambda \mathbf{t}{:}\kappa_1.\tau : \kappa_1 \to \kappa_2} \qquad \text{K-ARROW-E}\ \frac{\Delta \vdash_\Phi \tau_1 : \kappa_1 \to \kappa_2 \quad \Delta \vdash_\Phi \tau_2 : \kappa_1}{\Delta \vdash_\Phi \tau_1\ \tau_2 : \kappa_2}$$

(kinding for integers, labels, lists, products and sums also standard)

$$\text{K-EQ}\ \frac{\kappa\ \mathsf{eq} \quad \Delta \vdash_\Phi \tau_1 : \kappa \quad \Delta \vdash_\Phi \tau_2 : \kappa \quad \Delta \vdash_\Phi \tau_3 : \kappa' \quad \Delta \vdash_\Phi \tau_4 : \kappa'}{\Delta \vdash_\Phi \mathsf{if}\ \tau_1 \equiv_\kappa \tau_2\ \mathsf{then}\ \tau_3\ \mathsf{else}\ \tau_4 : \kappa'}$$

$$\text{K-TY-I}\ \frac{\mathrm{TYCON}\{\kappa_{\mathrm{idx}}; -; -\} \in \Phi \quad \Delta \vdash_\Phi \tau_{\mathrm{idx}} : \kappa_{\mathrm{idx}}}{\Delta \vdash_\Phi \mathrm{TYCON}\langle \tau_{\mathrm{idx}} \rangle : \mathsf{Ty}}$$

$$\text{K-TY-E}\ \frac{\Delta \vdash_\Phi \tau : \mathsf{Ty} \quad \mathrm{TYCON}\{\kappa_{\mathrm{idx}}; -; -\} \in \Phi \quad \Delta, \mathbf{x} : \kappa_{\mathrm{idx}} \vdash_\Phi \tau_1 : \kappa \quad \Delta \vdash_\Phi \tau_2 : \kappa}{\Delta \vdash_\Phi \mathsf{case}\ \tau\ \mathsf{of}\ \mathrm{TYCON}\langle \mathbf{x} \rangle \Rightarrow \tau_1\ \mathsf{ow}\ \tau_2 : \kappa} \qquad \text{K-D-I}\ \frac{\Delta \vdash_\Phi \tau_1 : \mathsf{Ty} \quad \Delta \vdash_\Phi \tau_2 : \mathsf{ITm}}{\Delta \vdash_\Phi [\![\tau \rightsquigarrow \iota]\!] : \mathsf{D}}$$

$$\text{K-D-E}\ \frac{\Delta \vdash_\Phi \tau : \mathsf{D}}{\Delta \vdash_\Phi \mathsf{typeof}(\tau) : \mathsf{Ty}} \qquad \text{K-REPTYPE-I}\ \frac{\Delta \vdash_\Phi \bar{\sigma}\ \mathtt{wk}}{\Delta \vdash_\Phi \blacktriangleright(\sigma) : \mathsf{ITy}}$$

$\boxed{\Delta \vdash_\Phi \bar{\sigma}\ \mathtt{wk}}$

$$\text{K-S-UNQUOTE}\ \frac{\Delta \vdash_\Phi \tau : \mathsf{ITy}}{\Delta \vdash_\Phi \blacktriangleleft(\tau)\ \mathtt{wk}}$$

(omitted forms are trivially recursive)

Figure 7. Kinding for type-level terms. The kinding context Δ maps type variables to kinds.

Listing 12 `[datatypes_t.py]` An example using statically-typed functional datatypes.

```
tree = lambda name, a: fp.data(name,
  lambda tree: {
    'Empty': fp.unit,
    'Leaf': a,
    'Node': fp.tuple(tree, tree)
  })

dyntree = tree(dyn)

@py
def depth_gt_2(x):
  {x : dyntree}
  return x.case({
    DT.Node(DT.Node(_), _): True,
    DT.Node(_, DT.Node(_)): True,
    _: False
  })

@py
def __toplevel__():
  my_lil_tree = dyntree.Node(dyntree.Empty, dyntree.Empty)
  my_big_tree = dyntree.Node(my_lil_tree, my_lil_tree)
  assert not depth_gt_2(my_lil_tree)
  assert depth_gt_2(my_big_tree)
```