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 4.5).

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. 4.5 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 4.5, 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 log_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.

4. @λ: Foundations of Active Typechecking and Translation

In this section, we will describe the foundations of *active typechecking and translation* by constructing a core lambda calculus, @λ. A *program* in @λ, ρ, consists of a series of fragment definitions, ϕ, for use by an external term, e. An example of an @λ program, ρ_{example}, that uses fragment definitions $\phi_{\texttt{nat}}$ and $\phi_{\texttt{lprod}}$, defining primitive natural numbers and labeled products (i.e. ordered records), is shown in Fig. 1. In Ace, fragment definitions are packaged separately; we do not include this in the core calculus for simplicity.

In this example, we see the introduction and elimination forms for natural numbers, functions and labeled tuples all being used. However, in the core syntax, shown (from the perspective of fragment clients) in Fig. **??**, there are only two external introductory forms, $\lambda x.e$ and $\mathsf{intro}[\sigma](\overline{e})$, and one external elimination form, $e{\cdot}\mathsf{elim}[\sigma](\overline{e})$, where $\overline{e}$ is shorthand for zero or more semicolon-separated external terms, called the *arguments* of the introduction or elimination operation. The forms used in Fig. 1 are *derived forms*, defined in terms of these core forms in Fig. **??**.

Both fragments and external terms can contain *static terms*, σ, which are evaluated statically to *static values*, $\hat{\sigma}$. The static language is itself a simply-typed lambda calculus. We call the types of static terms *kinds*, κ, by convention. Static variables are written in bold font, $\mathbf{x}$, to emphasize that they are distinct from variables bound by external terms, written x. For example, in the example in Figure 1, $\mathbf{nat}$ is a static variable bound by the fragment $\phi_{\texttt{nat}}$ to the natural number type. Types have kind Ty and are introduced by naming a *type constructor*, written in small-caps, e.g. NAT and LPROD, and providing an *index*, which is a static term of the *index kind* of the type constructor. For example, NAT is indexed by Unit (because there is only one natural number type), so $\mathbf{nat}$ is defined to be $\text{NAT}[()]$, and LPROD is indexed by $\mathsf{list}[\mathsf{L} \times \mathsf{Ty}]$, classifying static lists pairing *labels* with types. Labels are static values of kind L, written abstractly using the metavariables containing ℓ in our examples.

Before typechecking the external term in a program, the program is kind checked and its static terms are normalized to static values. We write static values using the hatted metavariable $\hat{\sigma}$. Other syntactic metavariables in our system also have hatted forms if they can contain static terms. The

$$
\begin{array}{l}
\mathsf{using}\ \phi_{\mathtt{nat}}; \phi_{\mathtt{lprod}} \\
\mathsf{let}\ \mathit{one} = \mathbf{s}\langle\mathbf{z}\langle\rangle\rangle\ \mathsf{in} \\
\mathsf{let}\ \mathit{plus} = (\lambda x.\lambda y.\mathbf{natrec}\ x\ \langle y; \lambda p.\lambda r.\mathbf{s}\langle r\rangle\rangle) : \mathbf{nat} \rightharpoonup \mathbf{nat} \rightharpoonup \mathbf{nat}\ \mathsf{in} \\
\mathsf{let}\ y = \{\ell_1 = \mathit{one}, \ell_2 = \mathit{plus}\ \mathit{one}\ \mathit{one}\} : \textsc{lprod}[(\ell_1, \mathbf{nat}) :: (\ell_2, \mathbf{nat}) :: []]\ \mathsf{in} \\
y.\ell_2
\end{array}
$$

Figure 1. A program that uses the natural number fragment, $\phi_{\mathtt{nat}}$, defined in Figure 3, written with the syntactic sugar defined in Figure 5.

$$
\begin{array}{l}
\mathsf{using}\ \hat{\phi}_{\mathtt{nat}}; \hat{\phi}_{\mathtt{lprod}} \\
\mathsf{let}\ \mathit{one} = \mathsf{intro}[\mathsf{in}[\ell_{\mathrm{s}}](())](\mathsf{intro}[\mathsf{in}[\ell_{\mathrm{z}}](())](())) : \textsc{nat}[()]\ \mathsf{in} \\
\mathsf{let}\ \mathit{plus} = (\lambda x.\lambda y.x \cdot \mathsf{elim}[()](y; \lambda p.\lambda r.\mathsf{intro}[\mathsf{in}[\ell_{\mathrm{s}}]()](r)) : \textsc{parr}[(\textsc{nat}[()], \textsc{parr}[(\textsc{nat}[()], \textsc{nat}[()])])]\ \mathsf{in} \\
\mathsf{let}\ y = \mathsf{intro}[\ell_1 :: \ell_2 :: []](\mathit{one}; \mathit{plus} \cdot \mathsf{elim}[()](\mathit{one}) \cdot \mathsf{elim}[()](\mathit{one})) : \textsc{lprod}[(\ell_1, \mathbf{nat}) :: (\ell_2, \mathbf{nat}) :: []]\ \mathsf{in} \\
y \cdot \mathsf{elim}[\ell_2]()
\end{array}
$$

Figure 2. An equivalent program written in core @λ without syntactic sugar and with type-level terms normalized.

$$
\begin{array}{lcl}
\phi_{\mathtt{nat}} & := & \mathsf{tycon}\ \textsc{nat}\ \mathsf{of}\ \mathsf{Unit}\ \{\delta_{\mathtt{nat}}\} \\
 & & \mathsf{def}\ \mathbf{nat} : \mathsf{Ty} = \textsc{nat}[()]; \\
 & & \mathsf{def}\ \mathbf{z} : \Pi[\{\ell_{\text{idx}} \hookrightarrow \kappa_{\text{nat-intro-idx}}, \ell_{\text{ty}} \hookrightarrow \mathsf{Ty}\}] = (\ell_{\text{idx}} = \mathsf{in}[\ell_{\mathrm{z}}](()), \ell_{\text{ty}} = \mathbf{nat}); \\
 & & \mathsf{def}\ \mathbf{s} : \Pi[\{\ell_{\text{idx}} \hookrightarrow \kappa_{\text{nat-intro-idx}}, \ell_{\text{ty}} \hookrightarrow \mathsf{Ty}\}] = (\ell_{\text{idx}} = \mathsf{in}[\ell_{\mathrm{s}}](()), \ell_{\text{ty}} = \mathbf{nat}); \\
 & & \mathsf{def}\ \mathbf{natrec} : \kappa_{\text{nat-elim-idx}} = () \\
\kappa_{\text{nat-intro-idx}} & := & \Sigma[\{l_z \hookrightarrow \mathsf{Unit}, l_s \hookrightarrow \mathsf{Unit}\}] \\
\kappa_{\text{nat-elim-idx}} & := & \mathsf{Unit}
\end{array}
$$

Figure 3. The natural number fragment, including definitions used by the assisted intro and elim desugarings, defined and shown being used above.

desugared, statically normalized version of the example in Figure 1, $\hat{\rho}_{\text{example}}$, is shown in Figure **??**.

4.1 External Terms

a program consists of first *kind checking* its static terms, then normalizing its static terms, then typechecking the external term and, simultaneously, *translating* it to a term, ι, in the *typed internal language*. The dynamic behavior of an external term is determined entirely by its translation to this language. Internal types are written τ. The internal language is only exposed to fragment providers, not to clients (i.e. normal developers).

The key judgements are the *bidirectional active typechecking and translation judgements*, relating an external term, e, to a type, τ, and an internal term, ι, called its *translation*, under *typing context* Γ and *constructor context* Φ.

$$\Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota \qquad \text{and} \qquad \Gamma \vdash_\Phi e \Leftarrow \tau \rightsquigarrow \iota$$

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 introduced in the "imported" fragments. 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 [13]. There, as in Wyvern, external terms were given meaning by elaboration from the EL to an IL having the same type system. Here the two languages have different type systems, so we call it a translation rather than an elaboration (arranging the judgement form slightly differently to emphasize this distinction, cf. above).

In @λ, the internal language (IL) provides partial functions (via the generic fixpoint operator of Plotkin's PCF), simple product types and integers for the sake of our example (and as a nod toward practicality on contemporary machines). 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. In Sec. **??**, we will see how the internal language can itself be made user-definable. 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]).

The external language has a fixed syntax with six forms: variables, let-bindings, lambda terms, type ascription and generalized introductory and elimination forms. As we will see, the generalized introductory form is given meaning by the type it is analyzed against (similar to the protocol for TSLs in Wyvern), and the elimination form is given meaning by the type of the external term being eliminated (e). This represents an internalization into the language of Gentzen's

	initial	**statically normal**	
programs	$\rho ::=$	$\hat{\rho} ::=$	
	$\mathsf{using}\ \phi\ \mathsf{in}\ e$	$\mathsf{using}\ \hat{\phi}\ \mathsf{in}\ \hat{e}$	
fragment decls.	$\phi ::=$	$\hat{\phi} ::=$	
tycon decls.	$\mathsf{tycon}\ \textsc{tycon} : \Theta = \theta$	$\mathsf{tycon}\ \textsc{tycon} : \Theta = \hat{\theta}$	
static binding	$\mathsf{def}\ \mathbf{x} : \kappa = \sigma$		
	$\phi; \phi$	$\hat{\phi}; \hat{\phi}$	
static terms	$\sigma ::=$	$\hat{\sigma} ::=$	**kinds** $\kappa ::=$
static values	$\hat{\sigma}$		
static variables	$\mathbf{x}$		
types	$\textsc{tycon}[\sigma]$	$\textsc{tycon}[\hat{\sigma}]$	Ty
static products	$()$	$()$	Unit
	(σ, σ)	$(\hat{\sigma}, \hat{\sigma})$	$\kappa \times \kappa$
static sums	$\mathsf{in}[\ell](\sigma)$	$\mathsf{in}[\ell](\hat{\sigma})$	$\Sigma[\{\overline{\ell \hookrightarrow \kappa}\}]$
static lists	$[]_\kappa$	$[]_\kappa$	$\mathsf{list}[\kappa]$
	$\sigma :: \sigma$	$\hat{\sigma} :: \hat{\sigma}$	
static labels	ℓ	ℓ	L
external terms	$e ::=$	$\hat{e} ::=$	
variables	x	x	
binding	$\mathsf{let}\ x = e\ \mathsf{in}\ e$	$\mathsf{let}\ x = \hat{e}\ \mathsf{in}\ \hat{e}$	
fixpoints	$\mathsf{fix}\ x{:}\sigma\ \mathsf{is}\ e$	$\mathsf{fix}\ x{:}\hat{\sigma}\ \mathsf{is}\ \hat{e}$	
ascription	$e : \sigma$	$\hat{e} : \hat{\sigma}$	
analytic lambda	$\lambda x.e$	$\lambda x.\hat{e}$	
analytic intro	$\mathsf{intro}[\sigma](\overline{e})$	$\mathsf{intro}[\hat{\sigma}](\overline{\hat{e}})$	
synthetic elim	$e \cdot \mathsf{elim}[\sigma](\overline{e})$	$\hat{e} \cdot \mathsf{elim}[\hat{\sigma}](\overline{\hat{e}})$	

Figure 4. Syntax from the perspective of fragment clients (i.e. normal developers)

$\sigma\langle e_1; \ldots; e_n\rangle$	$:=$	$\mathsf{intro}[\mathsf{fst}(\sigma)](e_1; \ldots; e_n) : \mathsf{snd}(\sigma)$	assisted intro
$\{\ell_1 = e_1, \ldots, \ell_n = e_n\}$	$:=$	$\mathsf{intro}[\ell_1 :: \ldots :: \ell_n :: []_\mathsf{L}](e_1; \ldots; e_n)$	labeled seq.
$(e_1, ..., e_n)$	$:=$	$\mathsf{intro}[()](e_1; \ldots; e_n)$	unlabeled seq.
$e\ e_1$	$:=$	$e \cdot \mathsf{elim}[()](e_1)$	application
$\sigma\ e\ \langle e_1; \ldots; e_n\rangle$	$:=$	$e \cdot \mathsf{elim}[\sigma](e_1; \ldots; e_n)$	assisted elim
$e.\ell$	$:=$	$e \cdot \mathsf{elim}[\ell]()$	projection
$\mathsf{case}\ e\ \{\ell_1\langle x_1\rangle \Rightarrow e_1 \mid \ldots \mid \ell_n\langle x_n\rangle \Rightarrow e_n\}$	$:=$	$e \cdot \mathsf{elim}[\ell_1 :: \ldots :: \ell_n :: []_\mathsf{L}](\lambda x_1.e_1; \ldots; \lambda x_n.e_n)$	case analysis
$\sigma_1 \rightharpoonup \sigma_2$	$:=$	$\textsc{parr}[(\sigma_1, \sigma_2)]$	arrow types

Figure 5. Desugaring from conventional concrete syntax to core forms. The number and string literal forms assume type-level numbers, n, and strings, s (details not shown).

what to cite for this?

inversion principle [**?**]. Fig. 5 shows how to recover more conventional introductory and elimination forms by a purely syntactic desugaring.

4.2 Types and Type-Level Computation

@λ 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*, which are type-level values of kind Ty. As in Sec. 3, types are formed by applying a *type constructor* to an *index*. User-defined type constructors are declared in fragment definitions using tycon. Each constructor in the program must have a unique name, written e.g. NAT or LPROD. A type constructor must also declare an *index kind*, κ_{tyidx}. A type is introduced by applying a type constructor to an index of this kind, written $\textsc{tycon}[\tau_{\text{tyidx}}]$. For example, the type of natural numbers is indexed trivially (i.e. by kind Unit), so it is written $\textsc{nat}[()]$.

To permit the embedding of interesting type systems, the type-level language includes several kinds other than Ty. We lift several functional data structures to the type level: here, only unit (Unit), binary products ($\kappa_1 \times \kappa_2$), binary sums ($\kappa_1 + \kappa_2$) and lists ($\mathsf{list}[\kappa]$), in addition to labels (introduced as ℓ, possibly with a subscript, having kind L). The type constructor NAT is indexed trivially because there is only one natural number type, but LPROD would be indexed by a list of pairs of labels and types. The type constructor ARROW is included in the initial constructor context, Φ_0, and has index kind Ty $\times$ Ty. As with the internal language, in

	initial		**statically normal**	
tycon definitions	$\delta ::=$		$\hat{\delta} ::=$	
	$\mathsf{iana}\ \sigma; \mathsf{esyn}\ \sigma; \mathsf{rep}\ \sigma$		$\mathsf{iana}\ \hat{\sigma}; \mathsf{esyn}\ \hat{\sigma}; \mathsf{rep}\ \hat{\sigma}$	
(static language, cont.)	$\sigma ::= \ldots$		$\hat{\sigma} ::= \ldots$	$\kappa ::= \ldots$
type elim	$\mathsf{case}\ \sigma\ \mathsf{of}\ \textsc{tycon}\langle \mathbf{x} \rangle \Rightarrow \sigma\ \mathsf{ow}\ \sigma$			
static product elim	$\sigma.\mathsf{prl}$			
	$\sigma.\mathsf{prr}$			
static sum elim	$\mathsf{case}\ \sigma\ \{\overline{l\langle \mathbf{x} \rangle \Rightarrow \sigma}\}$			
static list elim	$\mathsf{listrec}(\sigma; \sigma; \mathbf{h}, \mathbf{t}, \mathbf{r}.\sigma)$			
decidable equality	$\mathsf{if}\ \hat{\sigma} \equiv_\kappa \hat{\sigma}\ \mathsf{then}\ \hat{\sigma}\ \mathsf{else}\ \hat{\sigma}$			
type error	tyerr			
static functions	$\lambda \mathbf{x}{:}\kappa.\sigma$		$\lambda \mathbf{x}{:}\kappa.\sigma$	$\kappa \to \kappa$
	$\sigma\ \sigma$			
term translations	$\triangleright(\overleftarrow{\iota})$		$\triangleright(\iota)$	ITm
type translations	$\blacktriangleright(\overleftarrow{\tau})$		$\blacktriangleright(\tau)$	ITy
	$\mathsf{repof}(\sigma)$			
arguments	(introduced only by typechecker)			Arg
	$\mathsf{ana}(\sigma; \sigma; \mathbf{x}.\sigma)$			
	$\mathsf{syn}(\sigma; \mathbf{x}, \mathbf{t}.\sigma)$			
internal language	$\overleftarrow{\iota} ::=$	$\overleftarrow{\tau} ::=$	$\iota ::=$	$\tau ::=$
splice	$\lhd(\sigma)$	$\blacktriangleleft(\sigma)$		
variables	x		x	
fixpoints	$\mathsf{fix}\ x{:}\overleftarrow{\tau}\ \mathsf{is}\ \overleftarrow{\iota}$		$\mathsf{fix}\ x{:}\tau\ \mathsf{is}\ \iota$	
functions	$\lambda x{:}\overleftarrow{\tau}.\overleftarrow{\iota}$	$\overleftarrow{\tau} \rightharpoonup \overleftarrow{\tau}$	$\lambda x{:}\tau.\iota$	$\tau \rightharpoonup \tau$
	$\overleftarrow{\iota}\ \overleftarrow{\iota}$		$\iota\ \iota$	
products	$()$	unit	$()$	unit
	$(\overleftarrow{\iota}, \overleftarrow{\iota})$	$\overleftarrow{\tau} \times \overleftarrow{\tau}$	(ι, ι)	$\tau \times \tau$
	$\overleftarrow{\iota}.\mathsf{prl}$		$\overleftarrow{\iota}.\mathsf{prl}$	
	$\overleftarrow{\iota}.\mathsf{prr}$		$\overleftarrow{\iota}.\mathsf{prr}$	
sums	$\mathsf{in}[\ell](\overleftarrow{\iota})$	$\Sigma[\{\overline{\ell \hookrightarrow \overleftarrow{\tau}}\}]$	$\mathsf{in}[\ell](\iota)$	$\Sigma[\{\overline{\ell \hookrightarrow \tau}\}]$
	$\mathsf{case}\ \overleftarrow{\iota}\ \{\overline{\ell(x) \Rightarrow \overleftarrow{\iota}}\}$		$\mathsf{case}\ \iota\ \{\overline{\ell(x) \Rightarrow \iota}\}$	
integers	$\mathsf{int}[z]$	int	$\mathsf{int}[z]$	int
	$\overleftarrow{\iota} \pm \overleftarrow{\iota}$		$\iota \pm \iota$	
	$\mathsf{if0}(\overleftarrow{\iota}; \overleftarrow{\iota}; \overleftarrow{\iota}; \overleftarrow{\iota})$		$\mathsf{if0}(\iota; \iota; \iota; \iota)$	

Figure 6. Syntax from the perspective of fragment providers also includes an internal language and static language.

external typing context	$\Gamma ::= \emptyset \mid \Gamma, x \Rightarrow \textsc{tycon}[\hat{\sigma}]$	
	initial	**statically normal**
static language, cont.	$\sigma ::= \ldots$	$\hat{\sigma} ::= \ldots$
argument intro	$\mathsf{arg}(\Gamma; \hat{e})$	$\mathsf{arg}(\Gamma; \hat{e})$

Figure 7. Syntax from the perspective of the semantics / compiler.

practice, one could include a richer programming language and retain the spirit of the calculus, as long as it does not introduce general recursion at the type level. For example, our desugarings add support for number literals and string literals, which require adding numbers and strings to the type-level language (and providing a means for lifting them from the type-level language to the internal language, as we will discuss).

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 datatype [17], Ty, at the type-level. Like open datatypes, there is no notion of exhaustiveness so the default case is required for totality.

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

context	syntax	well-formedness
fragment kind context	$\mathcal{F} ::= \mathcal{F}_0 \mid \mathcal{F}, \text{TYCON}[\kappa]\ \{\mathsf{intro}[\kappa]; \mathsf{elim}[\kappa]\}$	$\vdash \mathcal{F}$
kinding context	$\Delta ::= \Delta \mid \Delta, \mathbf{x} : \kappa$	
fragment context	$\Phi ::= \Phi_0 \mid \Phi, \text{TYCON}\ \{\hat{\delta}\}$	$\vdash \Phi \sim \mathcal{F}$
external typing context	$\Gamma ::= \emptyset \mid \Gamma, x \Rightarrow \hat{\sigma}$	$\vdash_\Phi \Gamma$
internal typing context	$\Omega ::= \emptyset \mid \Omega, x : \tau$	

kinding	static normalization		program compilation	
$\rho\ \mathtt{ok}$	$\rho \curlyvee \hat{\rho}$	$\rho\ \mathtt{tyerr}$	$\hat{\rho} \rightsquigarrow \iota$	
$\Delta \vdash_\mathcal{F} \phi \sim \mathcal{F}$	$\phi \curlyvee_\Phi \hat{\phi}$	$\phi\ \mathtt{tyerr}$	$\hat{\phi} \sim \Phi$	
$\Delta \vdash_\mathcal{F} \delta \sim \{\kappa; \kappa\}$	$\delta \curlyvee_\Phi \hat{\delta}$	$\delta\ \mathtt{tyerr}$	**active typechecking and translation**	
$\Delta \vdash_\mathcal{F} \sigma : \kappa$	$\sigma \curlyvee_\Phi \hat{\sigma}$	$\sigma\ \mathtt{tyerr}$	$\Gamma \vdash_\Phi \hat{e} \Leftarrow \hat{\sigma} \rightsquigarrow \iota$	$\Gamma \vdash_\Phi \hat{e} \Rightarrow \hat{\sigma} \rightsquigarrow \iota$
$\Delta \vdash_\mathcal{F} e\ \mathtt{ok}$	$e \curlyvee_\Phi \hat{e}$	$e\ \mathtt{tyerr}$	$\vdash_\Phi \hat{\sigma} \rightsquigarrow \tau$	$\vdash_\Phi \Gamma \rightsquigarrow \Omega$
$\Delta \vdash_\mathcal{F} \overline{\iota}\ \mathtt{ok}$	$\overline{\iota} \curlyvee_\Phi \iota$	$\overline{\iota}\ \mathtt{tyerr}$	**internal statics**	**internal dynamics**
$\Delta \vdash_\mathcal{F} \overline{\tau}\ \mathtt{ok}$	$\overline{\tau} \curlyvee_\Phi \tau$	$\overline{\tau}\ \mathtt{tyerr}$	$\Omega \vdash \iota : \tau$	$\iota \mapsto \iota$ $\quad \iota\ \mathtt{val}$

Figure 8. Summary of judgements

$$
\begin{array}{lll}
\delta_{\mathtt{nat}} & := & \mathsf{iana}\ \lambda\mathbf{opidx}{:}\mathsf{Unit} + \mathsf{Unit}.\lambda\mathbf{tyidx}{:}\mathsf{Unit}.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{Arg}].\mathsf{case}\ \mathbf{opidx} \\
& & \quad \mathsf{of}\ \mathsf{inl}(\_) \Rightarrow \mathbf{arity0}\ \mathbf{args}\ \triangleright(0) \\
& & \quad \mid \mathsf{inr}(\_) \Rightarrow \mathbf{arity1}\ \mathbf{args}\ \lambda\mathbf{a}{:}\mathsf{Arg}.\mathsf{ana}(\mathbf{a}; \text{NAT}[()]; \mathbf{x}.\triangleright(\triangleleft(\mathbf{x}) + 1)) \\
& & \mathsf{esyn}\ \lambda\mathbf{opidx}{:}\mathsf{Unit}.\lambda\mathbf{tyidx}{:}\mathsf{Unit}.\lambda\mathbf{x}{:}\mathsf{ITm}.\lambda\mathbf{args}{:}\mathsf{list}[\mathsf{Arg}].\mathbf{arity2}\ \mathbf{args}\ \lambda\mathbf{a1}{:}\mathsf{Arg}.\lambda\mathbf{a2}{:}\mathsf{Arg}. \\
& & \quad \mathsf{syn}(\mathbf{a1}; \mathbf{t1}, \mathbf{x1}. \\
& & \quad\quad \mathsf{let}\ \mathbf{t2} : \mathsf{Ty} = \text{ARROW}[(\text{NAT}[()], \text{ARROW}[(\mathbf{t1}, \mathbf{t1})])]\ \mathsf{in} \\
& & \quad\quad \mathsf{ana}(\mathbf{a2}; \mathbf{t2}; \mathbf{x2}.(\mathbf{t1}, \\
& & \quad\quad\quad \triangleright((\mathsf{fix}\ f{:}\mathsf{int} \rightharpoonup \blacktriangleleft(\mathsf{repof}(\mathbf{t2}))\ \mathsf{is}\ \lambda x{:}\mathsf{int}. \\
& & \quad\quad\quad\quad \mathsf{if0}(x; \triangleleft(\mathbf{x1}); \triangleleft(\mathbf{x2})\ (x-1)\ (f\ (x-1))) \\
& & \quad\quad\quad) \triangleleft(\mathbf{x}))))) \\
& & \mathsf{rep}\ \lambda\mathbf{tyidx}{:}\mathsf{Unit}.\blacktriangleright(\mathsf{int})
\end{array}
$$

Figure 9. The definition of the nat type constronstructor.

be wrapped in a type-level function to emulate a first-class or uncurried version of a type constructor for convenience.

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).

4.3 Bidirectional Active Typechecking and Translation

The rules for bidirectional active typechecking and translation are shown in Fig. 10.

1. The rule ATT-FLIP is standard in bidirectional type systems (without subtyping): if a term synthesizes a type, it can be analyzed against that type; the translation is unaffected by this.
2. The rule ATT-VAR says that variables synthesize the type they have in the typing context and translate to variables in the internal language.
3. The rule ATT-ASC says that a term ascribed by a type synthesizes that type if the term can be analyzed against that type, after it has been normalized. The normalization judgement for type-level terms is written $\tau \Downarrow_\Phi \tau'$. The kinding rules (not shown here) guarantee that normalization of type-level terms cannot go wrong (we will refine what precisely this means later).
4. The rule ATT-LET-SYN first synthesizes a type for the bound value, then adds this binding to the context and synthesizes a type for the term the binding is scoped over. The translation is to a function application, in the conventional manner.

$\boxed{\Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota}$ $\boxed{\Gamma \vdash_\Phi e \Leftarrow \tau \rightsquigarrow \iota}$

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

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

ATT-ASC
$$\frac{\tau \Downarrow_\Phi \tau' \qquad \Gamma \vdash_\Phi e \Leftarrow \tau' \rightsquigarrow \iota}{\Gamma \vdash_\Phi e : \tau \Rightarrow \tau' \rightsquigarrow \iota}$$

ATT-LET-SYN
$$\frac{\Gamma \vdash_\Phi e_1 \Rightarrow \tau_1 \rightsquigarrow \iota_1 \qquad \Gamma, x \Rightarrow \tau_1 \vdash_\Phi e_2 \Rightarrow \tau_2 \rightsquigarrow \iota_2 \qquad \mathsf{repof}(\tau_1) \Downarrow_\Phi \blacktriangleright(\sigma_1)}{\Gamma \vdash_\Phi \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Rightarrow \tau_2 \rightsquigarrow (\lambda x{:}\sigma_1.\iota_2)\ \iota_1}$$

ATT-LAM-ANA
$$\frac{\Gamma, x \Rightarrow \tau_1 \vdash_\Phi e \Leftarrow \tau_2 \rightsquigarrow \iota \qquad \mathsf{repof}(\tau_1) \Downarrow_\Phi \blacktriangleright(\sigma_1)}{\Gamma \vdash_\Phi \lambda x.e \Leftarrow \textsc{arrow}[(\tau_1, \tau_2)] \rightsquigarrow \lambda x{:}\sigma_1.\iota}$$

ATT-INTRO-ANA
$$\frac{\vdash_\Phi \mathsf{iana}(\textsc{tycon}) = \tau_{\text{def}} \qquad \tau_{\text{def}}\ \tau_{\text{opidx}}\ \tau_{\text{tyidx}}\ ((\Gamma; e_1)? :: \ldots :: (\Gamma; e_n)? :: []) \Downarrow_\Phi \triangleright(\iota) \qquad \mathsf{repof}(\textsc{tycon}[\tau_{\text{tyidx}}]) \Downarrow_\Phi \blacktriangleright(\sigma) \qquad \Gamma \vdash_\Phi \iota : \sigma}{\Gamma \vdash_\Phi \mathsf{intro}[\tau_{\text{opidx}}](e_1; \ldots; e_n) \Leftarrow \textsc{tycon}[\tau_{\text{tyidx}}] \rightsquigarrow \iota}$$

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

Figure 10. The bidirectional active typechecking and translation judgements.

$\boxed{\tau \Downarrow_\Phi \tau'}$

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

SYN
$$\frac{\tau \Downarrow_\Phi (\Gamma; e)? \qquad \Gamma \vdash_\Phi e \Rightarrow \tau \rightsquigarrow \iota \qquad [\tau/\mathbf{t}_{ty}, \triangleright(\iota)/\mathbf{t}_{trans}]\tau_2 \Downarrow_\Phi \tau_2'}{\mathsf{syn}(\tau_1; \mathbf{t}_{ty}, \mathbf{t}_{trans}.\tau_2) \Downarrow_\Phi \tau_2'}$$

ANA
$$\frac{\tau_1 \Downarrow_\Phi (\Gamma; e)? \qquad \tau_2 \Downarrow_\Phi \tau_2' \qquad \Gamma \vdash_\Phi e \Leftarrow \tau_2' \rightsquigarrow \iota \qquad [\triangleright(\iota)/\mathbf{t}_{trans}]\tau_3 \Downarrow_\Phi \tau_3'}{\mathsf{ana}(\tau_1; \tau_2; \mathbf{t}_{trans}.\tau_3) \Downarrow_\Phi \tau_3'}$$

Figure 11. Normalization semantics for the type-level language. Missing rules (including error propagation rules and normalization of quoted internal terms and types) are unsurprising and will be given later.

To do so, however, we must also translate the type itself so that an appropriate type annotation for the function argument can be emitted. Every type has an internal type associated with it called its *representation type*. The type-level operator $\mathsf{repof}(\tau)$ evaluates to a *quoted internal type*, $\blacktriangleright(\sigma)$ (of kind ITy), where σ is the representation type of τ. Type constructors define the representation type for every possible index by providing a type-level function called the *representation schema*, written after the keyword rep. The normalization rule showing this is given in Fig. 11. In our example, the representation schema is simple: natural numbers translate to integers. We will see an example where this is less trivial later. Note that quoted internal types support splicing using the $\blacktriangleleft(\tau)$ form. These forms are eliminated during normalization.

5. The rule ATT-LAM says that lambda terms can only be analyzed against arrow types and translate to lambda terms in the internal language.

6. The rule ATT-INTRO-ANA says that generalized introductory forms can only be analyzed against a type. That type constructor is consulted to extract its *introductory operator definition*, written (e.g. in Figure 3) after the iana keyword as a type-level function. This function is given the provided operator index, the type's type index and a list encapsulating the operator's *arguments*. An argument is reified as a type-level term of the form $(\Gamma, e)?$ and has kind Arg. Note that only the compiler should construct such a term (in practice, this could be enforced purely syntactically, so we do not enforce this judgmentally here). The operator definition is responsible for producing a translation, or evaluating to error if this is not possible. A translation is simply a quoted internal term, written $\triangleright(\iota)$, with kind ITm. Like quoted internal types, quoted internal terms support an unquote form, written $\triangleleft(\tau)$, which is normalized away (in a capture-avoiding manner, not shown).

   Natural numbers have two introductory forms, each indexed trivially. Because there is only one generalized introductory form, we instead index the operator by a simple sum kind with two trivial cases, $\mathsf{Unit} + \mathsf{Unit}$. The first case corresponds to the zero operator and the second to the successor. In the introductory operator definition, we see in the first case that the translation 0 is produced after checking that no arguments were provided (using a simple helper function, $\mathbf{arity0}$, not shown). In the second case, we first extract the single argument (using the helper function $\mathbf{arity1}$, not shown). We then analyze it against the type $\textsc{nat}[()]$ using one of the elimination forms for arguments, $\mathsf{ana}(\tau_1; \tau_2; \mathbf{t}_{trans}.\tau_3)$. If it succeeds, we construct the appropriate translation based on the translation of the argument. The normalization rule for successful analysis, ANA, is shown in Figure 11.

   Note that we do not show here the error propagation rules. We need two additional judgements for this: the judgement $\Gamma \vdash_\Phi e$ `error` says that e has a type error, and

the judgement τ $\mathtt{err}_\Phi$ says that normalizing τ produced an error term. Appropriate error propagation rules need to be written down.

If analysis of the introductory form succeeds, the translation is checked for *representational consistency*: that the produced translation has the type indicated by the type constructor's representation schema. This is expressed by essentially a standard typechecking judgement for the internal language, written $\Gamma \vdash_\Phi \iota : \sigma$ (not shown). Note that the typing context will need to be translated to a corresponding typing context for the internal language. We will discuss this further below.

Notice that the assisted introductory form in Figure 5 allows us to define type-level variables $\mathbf{z}$ and $\mathbf{s}$ that simultaneously provide an operator index and type ascription, simulating introductory forms that synthesize a type without requiring support for such in the semantics (we will discuss a more expressive variant of the system in Sec. **??** where ascriptions can also be type constructors, not just types, permitting a form of synthetic introductory form).

Note also that lambda is technically the introductory form for the ARROW type constructor, but because it requires manipulating the context, it must be built in to $@\lambda$. We do not currently support type system fragments that require adding or manipulating existing contexts.

7. The rule ATT-ELIM-SYN says that elimination forms synthesize a type. This is done again by consulting a type-level function, called the *eliminatory operator definition*, associated with a type constructor, here the type constructor of the type recursively synthesized for the primary operand, e. This definition is invoked with the type index, the operator index, the translation of the primary operand and a list of arguments. Because elimination forms synthesize a type, this definition must produce both a type and a translation. Representational consistency is then checked.

   In our example, the elimination form for natural numbers is the recursor (shown in Fig. **??**). Our definition of it first checks that exactly two arguments were provided (not including the primary operand), then synthesizes a type and extracts a translation from the first argument using the form $\mathsf{syn}(\tau_1; \mathbf{t}_{ty}, \mathbf{t}_{trans}.\tau_2)$ (Fig. 11). The second argument is analyzed against an appropriate arrow type to support variable binding. If successful, the type $\mathbf{t1}$ (the type of the base case) is synthesized and a translation implementing the dynamic semantics of the recursor by a fixpoint computation is produced.

   The assisted elimination form shown in Fig. 5 provides a means to avoid providing an operator index explicitly. Here, we simply define $\mathbf{natrec}$ as the trivial value to avoid needing to write it explicitly (and for clarity).

   Note that the elimination form for the ARROW type constructor can be defined in this manner (because it does not require binding variables), so application is simply syntactic sugar, rather than a built-in operator in the external language.

4.4 Representational Consistency Implies Type Safety

A key invariant that our operator definitions maintain is that well-typed external terms of type $\textsc{Nat}[()]$ always translate to well-typed internal terms of internal type int, the representation type of $\textsc{Nat}[()]$. Verifying this for the zero case is simple. For the translation produced by the successor case to be of internal type int requires that $\triangleleft(\mathbf{x})$ be of internal type int. Because it is the result of analyzing an argument against $\textsc{Nat}[()]$, and the only other introductory form is the zero case, this holds inductively. We will discuss the recursor later, but it also maintains this invariant inductively.

This invariant would not hold if, for example, we allowed the type and translation:

$$(\textsc{Nat}[()], \triangleright((0, ())))$$

In this case, there would be two different internal types, int and $\mathsf{int} \times \mathsf{unit}$, associated with a single external type, $\textsc{Nat}[()]$. This would make it impossible to reason *compositionally* about the translation of an external term of type $\textsc{Nat}[()]$, so our implementation of the successor would produce ill-typed translations in some cases but not others. Similarly, we wouldn't be able to write functions over all natural numbers because there would not be a well-typed translation to give to such a function. This violates type safety: there are now well-typed external terms, according to the active typechecking and translation judgement, for which evaluating the corresponding translation would "go wrong".

To reason compositionally about the semantics of well-typed external terms when they are given meaning by translation to a typed internal language, the system must maintain the following property: for every type, τ, there must exist an internal type, σ, called its *representation type*, such that the translation of every external term of type τ has internal type σ. This principle of *representational consistency* arises essentially as a strengthening of the inductive hypothesis necessary to prove that all well-typed external terms translate to well-typed internal terms because λ and successor are defined compositionally. It is closely related to the concept of *type-preserving compilation* developed by Morrisett et al. for the TIL compiler for SML [26], here lifted into the language. Our judgements check this extension correctness property directly by typechecking each translation produced by a user extension (the translations of, for example, lambda terms will be inductively representationally consistent, so no additional check is needed).

Translational internal terms can contain translational internal types. We see this in the definition of the recursor on natural numbers. The type assignment is the arbitrary type,

$\mathbf{t2}$. We cannot know what the representation type of $\mathbf{t2}$ is, so we refer to it abstractly using $\mathsf{repof}(\mathbf{t2})$. Luckily, the proof of representational consistency of this operator is parametric over the representation type of $\mathbf{t2}$.

4.5 Metatheory

4.5.1 Representational Consistency

The representational consistency lemmas says that active typechecking and translation of an external term, if it succeeds (producing a type, $\hat{\sigma}$ and a translation ι), always produces a well-typed translation. More specifically, the translation's type is the representation type of $\hat{\sigma}$.

Theorem 1 (Representational Consistency)**.** *Given well-kinded contexts and an external term that is both well-kinded and well-typed:*

1. $\vdash \Phi$
2. $\vdash_\Phi \Gamma$
3. $\vdash_\Phi e$
4. *Either:*
   a. $\Gamma \vdash_\Phi e \Leftarrow \hat{\sigma} \leadsto \iota$
   b. $\Gamma \vdash_\Phi e \Rightarrow \hat{\sigma} \leadsto \iota$

there exists an internal typing context, Ω, and an internal type, τ, such that $\vdash_\Phi \Gamma \leadsto \Omega$ *and* $\mathsf{rep}(\hat{\sigma}) \Downarrow_\Phi \blacktriangleright(\tau)$ *and* $\Omega \vdash \iota : \tau$.

Proof. The proof proceeds by induction on the typing derivation (4a or 4b). Because the representational consistency is explicitly checked in the rules for intro and elim forms, these cases follow directly. The remaining cases are standard and follow inductively, given standard lemmas about valid contexts and a lemma about type safety of the static language. This is a fixed, standard functional language with only a few simple additions (TODO detail this). □

4.5.2 Type Safety

Representational safety implies that well-typed external terms produce well-typed internal terms. If the internal language is type safe, then we know that well-typed external terms do not "go wrong". The internal language is simply PCF with sums and products, and a base type of integers, so this is a completely standard result [12].

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. 12:

- 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 frameworks (cf. [10]). Extensible compilers like Xoc [7] can be considered a form of language framework as well. It is diffi-

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 12. Comparison to related approaches to language extensibility.

cult 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].

TODO: mention this http://blog.codeclimate.com/blog/2014/05/06/gradual-type-checking-for-ruby/

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/2.7/reference/introduction.html`, 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. Kastner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*, pages 391–406, Portland, Oregon, USA, Oct. ACM Press.

[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 *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 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.*, 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, 1999.

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

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

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'))
```

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.

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

```
import examples.fp
__ace_base__ = fp

@datatype
def tree(a): [
  Empty,
  Node : (tree[a], tree[a]),
  Leaf : a
]

def pair_trees(x, y):
  {x, y : tree[a]]}
  with (x, y).cases:
    if Empty, Empty:
      Empty [:m,:p]
    elif Node(x, y), Node(y):
      Node((x, y))
    elif Leaf(x), Leaf(y):
      Leaf((x, y))
    else:
      raise




@fp.datatype
def x(a):
  {'B': fp.unit,
   'R': y}

@fp.datatype
def y(a):
  {'B': fp.unit,
   'R': x}

@cl.struct
def hello(): {
  a : A,
  b : B
}

@cl.struct
def hello(): [
  a : A,
  b : B
]


   'Node': tuple(tree(a), tree(a))}
@fp.datatype()

@py
def depth_gt_2(x):
  {x : tree[dyn]}
  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)
```

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

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

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

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

$$\text{K-TY-I}\ \frac{\textsc{tycon}\{\kappa_{\text{idx}}; -; -\} \in \Phi \qquad \Delta \vdash_\Phi \tau_{\text{idx}} : \kappa_{\text{idx}}}{\Delta \vdash_\Phi \textsc{tycon}[\tau_{\text{idx}}] : \mathsf{Ty}}$$

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

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

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

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

(omitted forms are trivially recursive)

Figure 13. Kinding for type-level terms. The kinding context Δ maps type variables to kinds.