Ace: Growing A Statically-Typed Language 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 type system, Ace is organized around an extensible bidirectional type system that inverts control over typechecking and translation to type-level functions according to a protocol that cannot lead to ambiguities and type safety issues when type systems are combined. We give a simplified calculus that describes the type theoretic 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. We also show how to construct safe, natural and extensible foreign function interfaces using a novel form of "staged" type propagation from Python into Ace.

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 [9, 28, 29].

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* [42] 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 [6] 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 [27, 28].

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 [12] 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, avoiding the "chicken-and-egg problem" and

occupying what we intend to convey by example as a "sweet spot" in the design space (contribution 1, Sec. 2).

Ace, together with a core lambda calculus, serves also as a vehicle for introducing a novel extension mechanism we call *active typechecking and translation*. We sidestep the expression problem entirely by leaving the forms of expressions fixed, sharing them with Python. Instead, abstraction providers extend the language by introducing new type constructors and operators using type-level functions. We show how, by repurposing the conventional syntactic forms found in languages like Python, 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).

Ace supports a form of *ad hoc* polymorphism (based on singleton types) whereby the "dynamic type" of a Python value can determine a static type "just-in-time" when Python code calls a generic Ace function. This represents a form of implicit *staging* and we show how this can significantly streamline interactive workflows (like those in scientific computing) that alternate between Python for orchestration and exploratory tasks (e.g. plotting) and a statically-typed target language via a foreign-function interface (FFI) for performance- and correctness-critical portions. We discuss a complete implementation of the OpenCL type system (a language for low-level data parallel programming on GPUs), show several higher-level extensions to it, and discuss deep bindings between Ace, the OpenCL run-time system and Python's standard numerics package, `numpy`, all implemented within a library (contribution 3, Sec. 5).

In Sec. 6, we compare Ace to related work on language extensibility, term rewriting and staged compilation. We conclude in Sec. 7 by highlighting the key features needed by a host language to support active typechecking and translation, and discussing limitations and potential future work. The appendix contains details omitted for concision as well as several more examples, including functional datatypes, flexible complex numbers and a previously little-explored type system for tracking string invariants using regular expressions statically (which we show being used in Listing 1, amongst several other examples).

2. Language Design and Usage

Listing 1 is an example of an Ace *compilation script*. As promised, compilation scripts are written directly in Python. Ace requires no modifications to the language 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

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!"
```

libraries. Note that all of the type systems that we discuss are implemented entirely as libraries.

2.1 Types

Types are constructed programmatically during execution of the compilation script. This stands in contrast to many contemporary statically-typed languages, where types (e.g. datatypes, classes, structs) can only be declared. Put another way, Ace supports *type-level computation* and Python is its type-level language (discussed further in Sec. 4 and Sec. 6). In our example, we see several types being constructed:

1. On line 7, 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. Here, `record` and `decimal` are *indexed type constructors*. We will see how they are implemented in Sec. 3, but note that syntactically, `record` borrows Python's syntax for array slices (e.g. `a[min:max]`). Note 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 8-11, 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 ap-

pendix specifies the typechecking rules, based on [14]. 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, numbers and strings.

3. On line 12, we construct a simple *prototypic object type* [23]. 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 many fores, here to allow us to describe multiple transfers differing only in amount.

what is the right term for this?

2.2 Functions

Functions implementing run-time behavior (*typed functions*) are distinguished by the presence of a decorator specifying their *base semantics* (here `py` from `examples.py`; we will discuss this further below). 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 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 typed functions [2].

On lines 14-18, we see the function `log_transfer`. As in Python, it begins 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, shortly.

when?

Line 17 borrows 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 21-30, we define the function `__toplevel__` (the base will consider this a special name for the purposes of compilation, discussed in the Sec. 2.4). After printing a run-time greeting, we construct a *value* of type `C` on lines 24-26. 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 consider it instead an *ascribed introductory form* of that type. We see another such form used with Python's tuple syntax on lines 27-28 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 24-28 are also introductory, but do not have an ascription. This is because the definition of the type `C` completely determines 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) [24]. Literal forms are only analytic.

If a 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 the variables used as keys are not 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 evaluate statically to a type constructor. 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 are synthetic, the type constructor can determine an appropriate index based on the types of the subexpressions. If we wanted a matrix of `f32`s, then we would have to write either `[1, 2] (matrix[f32])` or `[1(f32), 2(f32)] (matrix)`, or use a base with a different default ascription. We will see how bidirectional typechecking of introductory forms works in greater technical detail in Secs. 3 and 4, but note here that it is one of the key mechanisms for expressiveness given our constraint that we cannot add entirely new forms to the language (and indeed, doing so would begin to be quite crowded even if we could add distinct forms for matrices, vectors, lists, tuples, etc.).

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

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

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

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

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 from within Python. We will begin with the former, then return to the latter in Sec. 5.

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. Evaluate the provided compilation script (`listing1.py`)
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 in Sec. 5. Here our target is the default target associated with the base `examples.py.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:

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
```

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 chosen 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 "run-time" 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] Nathan?
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).

2.5 Type Errors

Listing 4 shows an example of code containing several type errors. Indeed, lines 9-13 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). Static type checking allows us to find these errors during compilation. Listing

[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.

5 shows the result of attempting to compile this code. The compilation script completes (so that functions can refer to each other 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 as shown.

3. Active Typechecking and Translation

In compilers and other tools for monolithic languages, there is typically a *syntax-directed* protocol governing type checking, translation, pretty-printing and so on. For example, one might declare functional datatypes that capture all types and term forms, and then proceed by exhaustive case analysis over terms to implement this functionality. The external visitor pattern for class-based object systems captures essentially the same protocol. In both cases, the problem is that the types and terms are defined in one place, so providers cannot extend the language modularly.

In extensible languages the focus is typically on enabling the addition of both new types and term forms. Adding new term forms requires then adding cases for all possible functions over terms (e.g. typing, translation, code highlighting, style checking and so on). This is difficult. Indeed, this is the canonical example of the widely-studied *expression problem* [41]. Although there are some approaches that work for certain purposes, retaining the ability to reason compositionally about metatheoretic issues (e.g. type safety) and avoiding ambiguities when extensions are combined can become difficult (see Sec. 6).

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 types, which we will allow to affect the semantics of or reinterpret existing forms (e.g. literals, as discussed above, but also nearly every other form, as we will now discuss) in a controlled manner. More specifically, the key features that distinguish active typechecking and translation are:

- a *dispatch protocol* that delegates responsibility for typechecking and translation to:
  - a type or type constructor extracted from a subterm of a compound expression
  - a per-function base semantics (or simply *base*) for base forms for which this is not possible (statements, variables and unascribed literals)
- 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

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

```
1  from listing1 import py, A
2  @py
3  def example(a):
4    {a : A}
5    x = a.amount
6    x = x + x
7    return {amount: x} (A)
```

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

3.1 Base Decorators

Function decorators in Python (line 2) are syntactic sugar for calling the decorator (`py`) with the decorated function (lines 3-5) as an argument, then replacing the binding (`example`) with the return value of this call. 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 `_aceify` function (not shown) operates as follows:

show in appendix?

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 form its static environment.
3. The argument annotations are processed. If provided in the body, as in our example, it 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 argument annotations are used, this has already been done for us.

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 Typechecking Typed Functions and Statements

When the Ace compiler begins typechecking a `TypedFn` (when asked to by `acec`, or when it is invoked interactively as we will show in Sec. 5), 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 5-7 of Listing 8, where the Python base adds an attribute tracking local bindings (initially only the arguments) and the return type to be synthesized (initially not known, indicated using `None`).

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

```
1  class TypedFn(object):
2    def __init__(self, base, ast, static_env, arg_sig):
3      # ...
4    # ...
5  class Base(object):
6    def __call__(self, f):
7      (ast, static_env, arg_sig) = _aceify(f)
8      return TypedFn(self, ast, static_env, arg_sig)
9    # ...
```

3. For each statement in the body (ignoring the documentation string and type annotation, if present), the compiler calls a method of the base, `check_`F, where F is the form of the statement (taken directly from Python's `ast` package [2]). These methods are responsible for checking that statement is well-typed (statements do not themselves have a type) and implementing any necessary effects on the context.

   The assignment statements on lines 5 and 6 of Listing 6 trigger the method `check_Assign` shown on lines 6-14 of Listing 8 and the return statement on line 7 triggers the method `check_Return` on lines 18-22. Both work similarly: if this is the first assignment to a particular identifier, or the first return statement seen, then the value must synthesize a type in the current context. Otherwise, it is analyzed against the previously synthesized type.

4. The base synthesizes a type for the function as a whole via the `syn_ty_FunctionDef_outer` method. Here, an `arrow` type is synthesized (lines 26-29) based on the argument signature and the synthesized return type.

if time, add destructuring assignment to appendix

3.3 Bidirectionally Typechecking Expressions

The novel protocol for typechecking expressions that we will now describe (and formalize in Sec. 4) is what enables us to extend the language with new semantics modularly while sidestepping the expression problem. Let us look more closely at how lines 6.5-6.7[2] are typechecked.

As stated above, line 6.5 is an assignment statement, so the method `check_Assign` on lines 8.6-8.14 is called. Because the locals dictionary in `ctx` does not yet know what type `x` should be, the base asks to synthesize a type for the expression being bound, `a.amount`, by calling the `ctx.syn_ty` method.

Active Type Synthesis This method is defined on lines 9.5-9.26.

3.4 Translation

The decorator, documentation string and argument annotations, if present, are stripped away (having already been processed).

[2] 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 8 A portion of the base used in our examples thus far, defined in the `examples.py` package.

```
1  class PyBase(ace.Base):
2    def init_ctx(self, ctx):
3      ctx.locals = dict(ctx.fn.arg_sig)
4      ctx.return_t = None
5
6    def check_Assign(self, ctx, s):
7      if len(s.targets) > 1:
8        raise ace.TypeError("...not supported...", s)
9      x, e = s.targets[0].id, s.value # see ast module
10     if x in ctx.locals:
11       ctx.ana_ty(e, ctx.locals[x])
12     else:
13       ty = ctx.syn_ty(e)
14       ctx.locals[x] = ty
15
16   def trans_Assign(self, ctx, target, s):
17     return target.direct_translation(ctx, s)
18
19   def check_Return(self, ctx, s):
20     if ctx.return_t == None:
21       ctx.return_t = ctx.syn_ty(s.value)
22     else:
23       ctx.ana_ty(s.value, ctx.return_t)
24
25   def trans_Return(self, ctx, target, s):
26     return target.direct_translation(ctx, s)
27
28   def syn_ty_FunctionDef_outer(self, ctx, f):
29     if ctx.return_t == None:
30       ctx.return_t = unit
31     return arrow[ctx.fn.arg_sig, ctx.return_t]
32
33   def trans_FunctionDef_outer(self, ctx, target, f):
34     if f.name == "__toplevel__":
35       return target.Suite(ctx.trans(f.body))
36     else:
37       return target.direct_translation(f, ctx)
38
39   def syn_ty_Name(self, ctx, e):
40     x = e.id
41     if x in ctx.locals:
42       return ctx.locals[x]
43     elif x in ctx.fn.static_env: # (Sec. 5)
44       return self.syn_ty_lifted(x)
45     else:
46       raise ace.TypeError("...var not bound...", e)
47
48   def trans_Name(self, ctx, target, e):
49     if e.id in ctx.locals:
50       return target.direct_translation(ctx, e)
51     else: # (Sec. 5)
52       return self.trans_lifted(ctx, target, e)
53   # ...
54   default_target = PyTarget()
55   default_Dict_ascription = dyn
56   # ...
57 py = PyBase()
```

Once typechecking a method is complete, the compiler enters the translation phase, where terms in the target language are generated from Ace terms. Terms in the target language are generated by calling methods of the *active target* governing the function being compiled. The translation phase operates similarly to typechecking, using the dispatch protocol to invoke methods named `trans_`X. These methods have access to the context and node just as during typechecking, as well as the active target (named `target` here).

Listing 9 The context dispatches active typechecking and translation of expressions to the base or an appropriate type or type constructor.

```
class Context(object):
  def __init__(self, fn):
    self.fn = fn

  def syn_ty(self, e):
    if instanceof(e, ast.Name):
      owner = self.fn.base
      ty = owner.syn_ty_Name(self, e)
    elif instanceof(e, ast.Attribute):
      owner = self.syn_ty(e.value)
      ty = owner.syn_ty_Attribute(self, e)
    # ... other compound forms similar (cf appendix)
    elif isinstance(e, ast.Dict):
      return self.ana_ty(self, e,
        self.fn.base.default_Dict_ty)
    # ... other literal forms similar
    elif is_ascribed_Num(e):
      owner = get_ascription(e)
      if is_tycon(owner):
        ty = owner.syn_ty_Num(self, e)
      else:
        return owner.ana_ty_Num(self, e)
    # ... other ascribed literal forms similar
    e.owner = owner
    e.ty = ty
    return ty

  def ana_ty(self, e, ty):
    if instanceof(e, ast.Dict):
      ty.ana_ty_Dict(self, e)
    # ... other literal forms similar
    else:
      syn_ty = self.syn_ty(e)
      if ty != syn_ty:
        raise TypeError("...syn/ana mismatch...", e)
      return
    e.owner = e.ty = ty

  def trans(self, target, e):
    if instanceof(e, ast.Name):
      trans = e.owner.trans_Name(self, target, e)
    elif instanceof(e, ast.Attribute):
      trans = e.owner.trans_Attribute(self, target, e)
    # ... other forms similar
    e.trans = trans
    return trans
```

3.5 Active Types

3.5.1 Dispatch Protocol

3.5.2 Typechecking

When typechecking a compound expression or statement, the Ace compiler temporarily hands control to the object selected by the dispatch protocol by calling the method `type_`X, where X is the name of the syntactic form, taken from the Python grammar [2] (appended with a suffix in some cases).

3.5.3 Translation

3.6 Compositional Reasoning

Every statement and expression in an Ace function is governed by exactly one active type, determined by the dispatch protocol, or by the single active base associated with the Ace function. The representational consistency check ensures that compilation is successful without requiring that

Listing 10 The `record` type constructor.

```
@slices_to_sig
class record(ace.Type):
  def __init__(self, sig):
    _check_sig(sig)
    self.sig = sig

  def ana_ty_Dict(self, ctx, e):
    for f, v in zip(e.keys, e.values):
      if f.id in self.sig:
        ctx.ana_ty(v, self.sig[f.id])
      else:
        raise ace.TypeError("...bad field...", f)
    if len(self.sig) != len(e.keys):
      raise ace.TypeError("...missing field...", e)

  def trans_Dict(self, ctx, target, e):
    if len(self.sig) == 1: # assuming non-empty
      return ctx.trans(e.values[0])
    else:
      value_dict = dict(zip(e.keys, e.values))
      return target.Tuple(
        ctx.trans(target, value_dict[field])
        for field, ty in self.sig)

  def syn_ty_Attribute(self, ctx, e):
    if e.attr in self.sig:
      return self.sig[e.attr]
    else:
      raise ace.TypeError("...field not found...", e)

  def trans_Attribute(self, ctx, target, e):
    idx = idx_of(self.sig, e.attr)
    return target.Subscript(
      ctx.trans(target, e.value), target.Num(idx))
```

types that abstract over other types (e.g. container types) know about their internal implementation details. Together, this permits compositional reasoning about the semantics of Ace functions even in the presence of many extensions. This stands on contrast to many prior approaches to extensibility, where extensions could insert themselves into the semantics in conflicting ways, making the order of imports matter (see Sec. 6).

4. Theoretical Foundations

5. Generic Functions, Targets and Interactive Execution

5.1 OpenCL as an Active Library

The code in this section uses `clx`, an example library implementing the semantics of the OpenCL programming language and extending it with some additional useful types, which we will discuss shortly. Ace itself has no built-in support for OpenCL.

To briefly review, OpenCL provides a data-parallel SPMD programming model where developers define functions, called *kernels*, for execution on *compute devices* like GPUs or multi-core CPUs [15]. Each thread executes the same kernel but has access to a unique index, called its *global ID*. Kernel code is written in a variant of C99 extended with some new primitive types and operators, which we will introduce as needed in our examples below.

5.2 Generic Functions

Lines 3-4 introduce `map`, an Ace function of three arguments that is governed by the *active base* referred to by `clx.base` and targeting the *active target* referred to by `clx.opencl`. The active target determines which language the function will compile to (here, the OpenCL kernel language) and mediates code generation.

The body of this function, highlighted in grey for emphasis, does not have Python's semantics. Instead, it will be governed by the active base together with the active types used within it. No such types have been provided explicitly, however. Because our type system is extensible, the code inside could be meaningful for many different assignments of types to the arguments. We call functions awaiting types *generic functions*. Once types have been assigned, they are called *concrete functions*.

Generic functions are represented at compile-time as instances of `ace.GenericFn` and consist of an abstract syntax tree, an active base and an active target. The purpose of the *decorator* on line 3 is to replace the Python function on lines 4-8 with an Ace generic function having the same syntax tree and the provided active base and active target. Decorators in Python are simply syntactic sugar for applying the decorator function directly to the function being decorated [2]. In other words, line 3 could be replaced by inserting the following statement on line 9:

```
map = ace.fn(clx.base, clx.opencl)(map)
```

The abstract syntax tree for `map` is extracted using the Python standard library packages `inspect` (to retrieve its source code) and `ast` (to parse it into a syntax tree).

5.3 Metaprogramming in Ace

Generic functions can be generated directly from ASTs as well, providing Ace with support for straightforward metaprogramming. Listing **??** shows how to generate two more generic functions, `scale` and `negate`. The latter is derived from the former by using a library for manipulating Python syntax trees, `astx`. In particular, the `specialize` function replaces uses of the second argument of `scale` with the literal `-1` (and changes the function's name), leaving a function of one argument.

5.4 Concrete Functions and Explicit Compilation

To compile a generic function to a particular *concrete function*, a type must be provided for each argument, and type-checking and translation must then succeed. Listing 11 shows how to explicitly provide type assignments to `map` using the subscript operator (implemented using Python's operator overloading mechanism). We attempt to do so three times in Listing 11. The first, on line 11.7, fails due to a type error, which we handle so that the script can proceed. The error occurred because the ordering of the argument types was incorrect. We provide a valid ordering on line 11.9 to generate the concrete function `map_neg_f32`. We then provide a different type assignment to generate the concrete function `map_neg_ci32`. Concrete functions are instances of `ace.TypedFn`, consisting of an abstract syntax tree annotated with types and translations along with a reference to the original generic function.

Listing 11 [`listing11.py`] The generic `map` function compiled to map the `negate` function over two types of input.

```
1  import listing1, listing2, ace, examples.clx as clx
2
3  T1 = clx.Ptr(clx.global_, clx.float)
4  T2 = clx.Ptr(clx.global_, clx.Cplx(clx.int))
5  TF = listing2.negate.ace_type
6
7  try: map_neg_f32 = listing1.map[[TF, T1, T1]]
8  except ace.TypeError as e: print e.full_msg
9  map_neg_f32 = listing1.map[[T1, T1, TF]]
10 map_neg_ci32 = listing1.map[[T2, T2, TF]]
```

Listing 12 Compiling `listing11.py` using the `acec` compiler.

```
1  $ acec listing3.py
2  Hello, compile-time world!
3  [ace] TypeError in listing1.py (line 6, col 28):
4        'GenericFnType(negate)' does not support [].
5  [acec] listing3.cl successfully generated.
```

Listing 13 [`listing11.cl`] The OpenCL file generated by Listing 12.

```
1  float negate__0_(float x) {
2    return x * -1;
3  }
4
5  kernel void map_neg_f32(global float* input,
6      global float* output) {
7    size_t thread_idx = get_global_id(0);
8    output[thread_idx] = negate__0_(input[thread_idx]);
9    if (thread_idx == 0) {
10     printf("Hello, run-time world!");
11   }
12 }
13
14 int2 negate__1_(int2 x) {
15   return (int2)(x.s0 * -1, x.s1);
16 }
17
18 kernel void map_neg_ci32(global int2* input,
19     global int2* output) {
20   size_t thread_idx = get_global_id(0);
21   output[thread_idx] = negate__1_(input[thread_idx]);
22   if (thread_idx == 0) {
23     printf("Hello, run-time world!");
24   }
25 }
```

To produce an output file from an Ace "compilation script" like `listing11.py`, the command `acec` can be invoked from the shell, as shown in Listing 12.

5.5 Types

Lines 11.3-11.5 construct the types assigned to the arguments of `map` on lines 11.7-11.10. In Ace, types are themselves values that can be manipulated at compile-time. This stands in contrast to other contemporary languages, where user-defined types (e.g. datatypes, classes, structs) are written declaratively at compile-time but cannot be constructed,

inspected or passed around programmatically. More specifically, types are instances of a Python class that implements the `ace.ActiveType` interface (see Sec. 3.5). As Python values, types can be assigned to variables when convenient (removing the need for facilities like `typedef` in C or `type` in Haskell). Types, like all compile-time objects derived from Ace base classes, do not have visible state and operate in a referentially transparent manner (by constructor memoization, which we do not detail here).

The type named `T1` on line 11.3 corresponds to the OpenCL type `global float*`: a pointer to a 32-bit floating point number stored in the compute device's global memory (one of four address spaces defined by OpenCL [15]). It is constructed by applying `clx.Ptr`, which is an Ace type constructor corresponding to pointer types, to a value representing the address space, `clx.global_`, and the type being pointed to. That type, `clx.float`, is in turn the Ace type corresponding to `float` in OpenCL (which, unlike C99, is always 32 bits). The `clx` library contains a full implementation of the OpenCL type system (including behaviors, like promotions, inherited from C99). Ace is *unopinionated* about issues like memory safety and the wisdom of such promotions. We will discuss how to implement, as libraries, abstractions that are higher-level than raw pointers in Sec. B, but Ace does not prevent users from choosing a low level of abstraction or "interesting" semantics if the need arises (e.g. for compatibility with existing libraries; see the discussion in Sec. 7). We also note that we are being more verbose than necessary for the sake of pedagogy. The `clx` library includes more concise shorthand for OpenCL's types: `T1` is equal to `clx.gp(clx.f32)`.

The type `T2` on line 11.4 is a pointer to a *complex integer* in global memory. It does not correspond direrctly to a type in OpenCL, because OpenCL does not include primitive support for complex numbers. Instead, it uses an active type constructor `clx.Cplx`, which includes the necessary logic for typechecking operations on complex numbers and translating them to OpenCL (Sec. 3.5). This constructor is parameterized by the numeric type that should be used for the real and imaginary parts, here `clx.int`, which corresponds to 32-bit OpenCL integers. Arithmetic operations with other complex numbers, as well as with plain numeric types (treated as if their imaginary part was zero), are supported. When targeting OpenCL, Ace expressions assigned type `clx.Cplx(clx.int)` are compiled to OpenCL expressions of type `int2`, a *vector type* of two 32-bit integers (a type that itself is not inherited from C99). This can be observed in several places on lines 13.14-13.21. This choice is merely an implementation detail that can be kept private to `clx`, however. An Ace value of type `clx.int2` (that is, an actual OpenCL vector) *cannot* be used when a `clx.Cplx(clx.int)` is expected (and attempting to do so will result in a static type error): `clx.Cplx` truly extends the type system, it is not a type alias.

The type `TF` on line 11.5 is extracted from the generic function `negate` constructed in Listing **??**. Generic functions, according to Sec. 5.2, have not yet had a type assigned to them, so it may seem perplexing that we are nevertheless assigning a type to `negate`. Although a conventional arrow type cannot be assigned to `negate`, we can give it a *singleton type*: a type that simply means "this expression is the *particular* generic function `negate`". This type could also have been explicitly written as `ace.GenericFnType(listing2.negate)`. During typechecking and translation of `map_neg_f32` and `map_neg_ci32`, the call to `f` on line **??**.6 uses the type of the provided argument to compile the generic function that inhabits the singleton type of `f` (`negate` in both of these cases) to a concrete function. This is why there are two versions of `negate` in the output in Listing 13. In other words, types *propagate* into generic functions – we didn't need to compile `negate` explicitly. This also explains the error printed on line 12.3-12.4: when this type was inadvertently assigned to the first argument `input`, the indexing operation on line **??**.6 resulted in an error. A generic function can only be *statically* indexed by a list of types to turn it into a concrete function, not *dynamically* indexed with a value of type `clx.size_t` (the return type of the OpenCL primitive function `get_global_id`).

In effect, this scheme enables higher-order functions even when targeting languages, like OpenCL, that have no support for higher-order functions (OpenCL, unlike C99, does not support function pointers). Interestingly, because they have a singleton type, they are higher-order but not first-class functions. That is, the type system would prevent you from creating a heterogeneous list of generic functions. Concrete functions, on the other hand, can be given both a singleton type and a true function type. For example, `listing2.negate[[clx.int]]` could be given type `ace.Arrow(clx.int, clx.int)`. The base determines how to convert the Ace arrow type to an arrow type in the target language (e.g. a function pointer for C99, or an integer that indexes into a jump table constructed from knowledge of available functions of the appropriate type in OpenCL).

Type assignment to generic functions is similar in some ways to template specialization in C++. In effect, both a template header and type parameters at call sites are being generated automatically by Ace. This simplifies a sophisticated feature of C++ and enables its use with other targets like OpenCL.

5.6 Implicit Compilation and Interactive Execution

A common workflow for *professional end-user programmers* (e.g. scientists and engineers) is to use a simple scripting language for orchestration, small-scale data analysis and visualization and call into a low-level language for performance-critical sections. Python is both designed for this style of use and widely adopted for such tasks [29, 32]. Developers can call into native functions using Python's foreign function interface (FFI), for example. A more re-

By extracting the types from the arguments, we can call the generic function `map` without first requiring an explicit type assignment, like we needed when using `acec` above. In other words, dynamic types and other metadata can propagate from Python data structures into an Ace generic function as static type information, in the same manner as it propagated *between* generic functions in the previous section. In both cases, typechecking and translation of `map` happens the first time a particular type assignment is encountered and cached for subsequent use. When called from Python, the generated OpenCL source code is compiled for the device we selected using the OpenCL retargetable compilation infrastructure, and cached for subsequent calls.

The same program written using the OpenCL C API directly is an order of magnitude longer and significantly more difficult to comprehend. OpenCL does not support higher-order functions nor is there any way to write `map` in a type-generic or shape-generic manner. If we instead use the `pyopencl` library and apply the techniques described in [22], the program is still twice as large and less readable than this code. Both the `map` and `negate` functions must be explicitly specialized with the appropriate types using string manipulation techniques, and custom shapes and orders can be awkward to handle. Higher order functions are still not available, and must also be simulated by string manipulation. That approach also does not permit the use any of the language extensions that Ace enables (beyond the useful type for `numpy` arrays just described; see Sec. B for more interesting possibilities).

6. Related Work

Some previous work has considered embedding statically-typed languages within other languages, including dynamically-typed languages. For example, Terra embeds a low-level statically-typed language within Lua [11]. Our primary example showed how to support low-level GPU programming from a high-level language. Other languages have supported similar workflows, e.g. Rust [17]. These have not been fundamentally extensible and suffer from the "bootstrapping" problem described in Sec. 1.

Libraries containing compile-time logic have previously been called *active libraries* [39]. A number of projects, such as Blitz++, have taken advantage of C++ template metaprogramming to implement domain-specific optimizations [38]. Others have chosen custom metalanguages (e.g. Xroma [39], mbeddr [40]). 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) in the compile-time metalanguage.

Generic functions are a novel strategy for *function polymorphism* – defining functions that operate over more than a single type. In Ace, generic functions are implicitly polymorphic and can be called with arguments of *any type that supports the operations used by the function*. This is related to structural polymorphism [25]. Structural types make explicit the structure required of an argument, unlike generic functions, which are only given singleton types because the structure may depend on the semantics of an active type. Structural typing can be compared to the more *ad hoc* approach taken by dynamically-typed languages like Python itself, sometimes called "duck typing". It is also comparable to the C++ template system, as discussed previously.

Operator overloading [37] and *metaobject dispatch* [21] are run-time protocols that translate operator invocations into function calls. The function is typically selected according to the 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.

There are several other language-internal extension mechanisms that have been described in the literature. These differ from our mechanism in one of the following ways, summarized in Fig. 1:

- Our mechanism is itself implemented as a library, rather than as a language extension. Only typed LISPs are similar.
- Our mechanism reuses an existing language's syntax, rather than offering syntax extension support. As we saw in, for example, Listing 16, this can still be reasonably natural, and allows the language to benefit from a variety of existing tools.
- Our mechanism permits true type system extensions. New types are not merely aliases, nor must their rules be admissible in some base type system.
- As described in Sec. 3.6, our mechanism permits compositional reasoning. Techniques that allow global extensions to the syntax or semantics (e.g. SugarJ or rule injection systems like Xroma, Typed Racket (and other typed LISPs) and mbeddr) allow extensions so much control that they can interfere.
- Our mechanism allows users to control the target language of compilation from within libraries. Many other mechanisms are based fundamentally on term-to-term rewriting.

When the mechanisms available in an existing language prove insufficient, researchers and domain experts often design a new language. A number of tools have been developed to assist with this task, including compiler generators, language workbenches and domain-specific language frameworks (cf [13]). Extensible compilers can be considered a form of language framework as well due to the interoperability issues that relying on a compiler-specific extensions can introduce. It is difficult or impossible for these language-

Approach	Examples	Library	Extensible Syntax	Extensible Type System	Extensions Compositional	Alternative Targets
Active Types	Ace	●	○	●	●	●
Desugaring	SugarJ [12]	○	●	○	○	○
Rule Injection	Racket [36], Xroma [39], mbeddr [40]	[36]	[40]	●	○	○
Static macros / Metaprogramming	Scala [4], MorphJ [18], OJ [35] MetaML [33], Template Haskell	○	○	○	●	○
Cross-Compilation	Delite [7]	●	○	○	○	●

Figure 1. Comparison to related approaches to language-internal extensibility.

external approaches to achieve interoperability, because languages are not aware of each other's semantics and merging languages is not guaranteed to be sound.

7. Discussion

1. We address the "chicken-and-egg" problem by designing Ace, an extensible language embedded within Python [2, 31] entirely as a library. The top-level of an Ace file is a *compilation script* written in Python and Python serves as Ace's *type-level language*, as we will discuss. Ace and Python thus share a common syntax and package system, so users of Ace can directly leverage established tools, infrastructure and coding standards. Functions marked by a decorator as under the control of Ace are, unlike Python functions, *statically* typechecked. This makes it possible to rule out many potential issues ahead of execution, and thus avoid costly dynamic checks.
2. There is *no particular type system built into Ace*. Instead, a type is any object implementing the `ace.Type` interface constructed by the compilation script. We call these *active types*. When statically assigning a type to a compound form, e.g. `e.x`, the Ace compiler delegates to the type of a subexpression, e.g. to the type of `e`, to determine how to assign a type to the expression as a whole. As we will show, this type-directed protocol, *active typing*, permits the expression of a rich variety of type systems as libraries. Unlike syntax-directed protocols, there cannot be ambiguities when separately defined type sytems are combined (types control non-overlapping sets of expressions by construction). Base cases, e.g. variables and certain statement forms, are handled on a per-function basis by a *base semantics*, also a compile-time object implementing an interface, `ace.Base`.

The dynamic behavior of an Ace function is determined by translation to a *target language*. We begin by targeting Python, then discuss targeting OpenCL and CUDA, lower-level languages used to program many-core processors (e.g. GPUs). Active types and bases control translation by the same type-directed protocol that governs type assignment. We call this phase *active translation*. A *target* mediates this process and is also a user-defined compile-time object implementing an interface, `ace.Target`. When targeting a typed language, care must be taken to ensure that well-typed Ace expressions have well-typed translations, so we integrate a technique for compositionally reasoning about compiler correctness (developed for the TIL compiler for Standard ML [34]) into the language.

TODO

Static type systems are powerful tools for programming language design and implementation. By tracking the type of a value statically, a typechecker can verify the absence of many kinds of errors over all inputs. This simplifies and increases the performance of the run-time system, as errors need not be detected dynamically using tag checks and other kinds of assertions.

It is legitimate to ask, however, why dynamically-typed languages are so widely-used in multiple domains. Although slow and difficult to reason about, these languages generally excel at satisfying the criteria of **ease-of-use**. More specifically, Cordy identified the principle of *conciseness* as elimination of redundancy and the availability of reasonable defaults [10]. Statically-typed languages, particularly those that professional end-users are exposed to, are often verbose, requiring explicit and often redundant type annotations on each function and variable declaration, separate header files, explicit template headers and instantiation and other sorts of annotations. Dynamically-typed languages, on the other hand, avoid most of this overhead by relying on support from the run-time system. Ace was first conceived to explore the question: *does conciseness require run-time mechanisms, or can one develop a statically-typed language with the same usability profile?*

Rather than designing a new syntax, or modifying the syntax of an existing language, we chose to utilize, *without modification*, the syntax of an existing language, Python. This choice was not arbitrary, but rather a key means by which Ace satisfies extrinsic design criteria related to familiarity and tool support. Researchers often dismiss the importance of syntax. By repurposing a well-developed syntax, they no longer need to worry about the "trivial" task of implementing it.

Most programming languages are *monolithic* – a collection of primitives are given first-class treatment by the language implementation, and users can only creatively combine them to implement abstractions of their design.

Although highly-expressive general-purpose mechanisms have been developed (such as object systems or functional datatypes), these may not suffice when researchers or domain experts wish to evolve aspects of the type system, exert control over the representation of data, introduce specialized run-time mechanisms, if defining an abstraction in terms of existing mechanisms is unnatural or verbose, or if custom error messages are useful. In these situations, it would be desirable to have the ability to modularly extend existing systems with new compile-time logic and be assured that such extensions will never interfere with one another when used in the same program.

Python does not truly prevent extensions from interfering with one another because it lacks, e.g., data hiding mechanisms. One type could change the implementation of another by replacing its methods, for example. But these kinds of conflicts do not occur "innocently", as conflicts between two libraries in SugarJ that use similar syntax might.

Ace has some limitations at the moment. 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. Debugging type errors can also be difficult when there are many nested generic functions that were implicitly given type assignments. Type debuggers like `scalad` could help make this easier [30]. 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.

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.

This paper is presented as a language design paper. The theoretical foundations of this work lie in type-level computation. We have developed a simplified, type-theoretic formalism of active typechecking and translation, where we prove the safety properties we have outlined here, as well as several additional ones that are only possible to achieve using a typed metalanguage (currently in submission at ESOP 2014). The work described here extends that mechanism in several directions: Ace supports a rich syntax, makes syntax trees available to extensions, supports a pluggable backend and per-function base semantics. It is implemented in an existing language and supports a wider range of practical extensions.

The mechanisms described here can be implemented within any language that offers some form of quotation of function ASTs and a reasonably flexible collection of syntactic forms and basic reflection facilities. It is particularly useful when the language supports interoperability layers with a number of target languages.

References

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

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

[3] D. Bonachea. Gasnet specification, v1. *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.

[4] 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.

[5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 47–56. ACM, 2011.

[6] 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.

[7] 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, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 35–46. ACM, 2011.

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[9] 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.

[10] J. Cordy. Hints on the design of user interface language features: lessons from the design of turing. In *Languages for developing user interfaces*, pages 329–340. AK Peters, Ltd., 1992.

[11] 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.

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

[13] 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.

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

[15] K. O. W. Group et al. The opencl specification, version 1.1, 2010. *Document Revision*, 44.

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

[17] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis. Gpu programming in rust: Implementing high-level abstractions in a systems-level language. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 315–324, 2013.

[18] 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.

[19] L. V. Kale and G. Zheng. Charm++ and ampi: Adaptive runtime strategies via migratable objects. *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282, 2009.

[20] A. Kennedy. Types for units-of-measure: Theory and practice. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2009.

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

[22] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 2011.

[23] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 21:11 of *ACM Sigplan Notices*, pages 214–223, Oct. 1986.

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

[25] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. *Programming Languages and Systems*, pages 95–111, 2009.

[26] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 499–504. ACM, 2011.

[27] 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.

[28] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, OOPSLA '13, pages 1–18, New York, NY, USA, 2013. ACM.

[29] 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.

[30] H. Plociniczak. Scalad: an interactive type-level debugger. In *Proceedings of the 4th Workshop on Scala*, page 8. ACM, 2013.

[31] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the full monty. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, OOPSLA '13, pages 217–232, New York, NY, USA, 2013. ACM.

[32] M. F. Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

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

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

[35] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *1st OOPSLA Workshop on Reflection and Software Engineering*, volume 1826 of *LNCS*, pages 117–133. Springer Verlag, 2000.

[36] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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

A. Complex Numbers

For example, if `c` is a complex number, then `c.ni` and `c.i` are its non-imaginary and imaginary components, respectively. These expressions are of the form `Attribute`, so the typechecker calls `type_Attribute` (line 15.7). This method receives the compilation context, `context`, and the abstract syntax tree of the expression, `node` and must return a type assignment for the node, or raise an `ace.TypeError` if there is an error. In this case, a type assignment is possible if the attribute name is either `"ni"` or `"i"`, and an error is raised otherwise (lines 15.8-15.10). We note that error messages are an important and sometimes overlooked facet of ease-of-use [26]. A common frustration with using general-purpose abstraction mechanisms to encode an abstraction is that they can produce verbose and cryptic error messages that reflect the implementation details instead of the semantics. Ace supports custom error messages.

Complex numbers also support binary arithmetic operations partnered with both other complex numbers and with non-complex numbers, treating them as if their imaginary component is zero. The typechecking rules for this logic is implemented on lines 15.17-15.29. Arithmetic operations are usually symmetric, so the dispatch protocol checks the types of both subexpressions for support. To ensure that the semantics remain deterministic in the case that both types support the binary operation, Ace asks the left first (via `type_BinOp_left`), asking the right (via `type_BinOp_right`) only if the left indicates an error. In either position, our implementation begins by recursively assigning a type to the other operand in the current context via the `context.type` method (line 15.24). If supported, it applies the C99 rules for arithmetic operations to determine the resulting type (via `c99_binop_t`, not shown).

Finally, a complex number can be constructed inside an Ace function using Ace's special constructor form: `[clx.Cplx](3,4)` represents $3 + 4i$, for example. The term within the braces is evaluated at *compile-time*. Because `clx.Cplx` evaluates not to an active type, but to a class, this form is assigned a type by handing control to the class object via the *class method* `type_New`. It operates as expected, extracting the types of the two arguments to construct an appropriate complex number type (lines 15.50-15.57), raising a type error if the arguments cannot be promoted to a common type according to the rules of C99 or if two arguments were not provided.

As seen in Listing 13, we are implementing complex numbers internally using OpenCL vector types, like `int2`. Let us look first at `trans_New` on lines 15.54-15.60, where new complex numbers are translated to vector literals by invoking `target.VecLit`. This will ultimately generate the necessary OpenCL code, as a string, to complete compilation (these strings are not directly manipulated by extensions, however, to avoid problems with, e.g. precedence). For it to be possible to reason compositionally about the correctness of compilation, all complex numbers must translate to terms in the target language that have a consistent target type. The `trans_type` method of the `ace.ActiveType` associates a type in the target language, here a vector type like `int2`, with the active type. Ace supports a mode where this *representational consistency* is dynamically checked during compilation (requiring that the active target know how to assign types to terms in the target language, which can be done for our OpenCL target as of this writing).

The translation methods for attributes (line 15.12) and binary operations (line 15.31) proceed in a straightforward manner. The context provides a method, `trans`, for recursively determining the translation of subexpressions as needed. Of note is that the translation methods can assume that typechecking succeeded. For example, the implementation of `trans_Attribute` assumes that if `node.attr` is not `'ni'` then it must have been `'i'` on line 15.14, consistent with the implementation of `type_Attribute` above it. Typechecking and translation are separated into two methods to emphasize that typechecking is not target-dependent, and to allow for more advanced uses, like type refinements and hypothetical typing judgements, that we do not describe here.

B. Expressiveness

Thus far, we have focused mainly on the OpenCL target and shown examples of fairly low-level active types: those that implement OpenCL's primitives (e.g. `clx.Ptr`), extend them in simple but convenient ways (e.g. `clx.Cplx`) and those that make interactive execution across language boundaries safe and convenient (e.g. `clx.NPArray`). Ace was first conceived to answer the question: *can we build a statically-typed language with the semantics of a C but the syntax and ease-of-use of a Python?* We submit that the work described above has answered this question in the affirmative.

But Ace has proven useful for more than low-level tasks like programming a GPU with OpenCL. We now describe several interesting extensions that implement the semantics of primitives drawn from a range of different language paradigms, to justify our claim that these mechanisms are highly expressive.

B.1 Growing a Statically-Typed Python Inside an Ace

Ace comes with a target, base and type implementing Python itself: `ace.python.python`, `ace.python.base` and `ace.python.dyn`. These can be supplemented by additional active types and used as the foundation for writing actively-typed Python functions. These functions can either be compiled ahead-of-time to an untyped Python file for execution, or be immediately executed with just-in-time compilation, just like the OpenCL examples we have shown. Many of the examples in the next section support this target

Listing 15 [in `examples/clx.py`] The active type family `Ptr` implements the semantics of OpenCL pointer types.

```
class Cplx(ace.ActiveType):
  def __init__(self, t):
    if not isinstance(t, Numeric):
      raise ace.InvalidTypeError("<error message>")
    self.t = t

  def type_Attribute(self, context, node):
    if node.attr == 'ni' or node.attr == 'i':
      return self.t
    raise ace.TypeError("<error message>", node)

  def trans_Attribute(self, context, target, node):
    value_x = context.trans(node.value)
    a = 's0' if node.attr == 'ni' else 's1'
    return target.Attribute(value_x, a)

  def type_BinOp_left(self, context, node):
    return self._type_BinOp(context, node.right)

  def type_BinOp_right(self, context, node):
    return self._type_BinOp(context, node.left)

  def _type_BinOp(self, context, other):
    other_t = context.type(other)
    if isinstance(other_t, Numeric):
      return Cplx(c99_binop_t(self.t, other_t))
    elif isinstance(other_t, Cplx):
      return Cplx(c99_binop_t(self.t, other.t))
    raise ace.TypeError("<error message>", other)

  def trans_BinOp(self, context, target, node):
    r_t = context.type(node.right)
    l_x = context.trans(node.left)
    r_x = context.trans(node.right)
    make = lambda a, b: target.VecLit(
      self.trans_type(self, target), a, b)
    binop = lambda a, b: target.BinOp(
      a, node.operator, b)
    si = lambda a, i: target.Attribute(a, 's'+str(i))
    if isinstance(r_t, Numeric):
      return make(binop(si(l_x, 0), r_x), si(r_x, 1))
    elif isinstance(r_t, Cplx):
      return make(binop(si(l_x, 0), si(r_x, 0)),
        binop(si(l_x, 1), si(r_x, 1)))

  @classmethod
  def type_New(cls, context, node):
    if len(node.args) == 2:
      t0 = context.type(node.args[0])
      t1 = context.type(node.args[1])
      return cls(c99_promoted_t(t0, t1))
    raise ace.TypeError("<error message>", node)

  @classmethod
  def trans_New(cls, context, target, node):
    cplx_t = context.type(node)
    x0 = context.trans(node.args[0])
    x1 = context.trans(node.args[1])
    return target.VecLit(cplx_t.trans_type(target),
      x0, x1)

  def trans_type(self, target):
    return target.VecType(self.t.trans_type(target),2)
```

in addition to the OpenCL target we have focused on thus far.

B.2 Recursive Labeled Sums

Listing 16 shows an example of the use of statically-typed functional datatypes (a.k.a. recursive labeled sum types) together with the Python implementation just described. It shows two syntactic conveniences that were not mentioned

Listing 16 [`datatypes_t.py`] An example using statically-typed functional datatypes.

```
import ace, ace.python as py, examples.fp as fp

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

DT = Tree('DT', py.dyn)

@ace.fn(py.Base(trailing_return=True))[[DT]]
def depth_gt_2(x):
  x.case({
    DT.Node(DT.Node(_), _): True,
    DT.Node(_, DT.Node(_)): True,
    _: False
  })

@ace.fn(py.Base(main=True))[[]]
def __main__():
  my_lil_tree = DT.Node(DT.Empty, DT.Empty)
  my_big_tree = DT.Node(my_lil_tree, my_lil_tree)
  assert not depth_gt_2(my_lil_tree)
  assert depth_gt_2(my_big_tree)
```

Listing 17 [`datatypes.py`] The dynamically-typed Python code generated by running `acec datatypes_t.py`.

```
class DT(object):
  pass

class DT_Empty(DT):
  pass

class DT_Leaf(DT):
  def __init__(self, data):
    self.data = data

class DT_Node(DT):
  def __init__(self, data):
    self.data = data

def depth_gt_2(x):
  if isinstance(x, DT_Node):
    if isinstance(x.data[0], DT_Node):
      r = True
    elif isinstance(x.data[1], DT_Node):
      r = True
  elif True:
    r = False
  return r

if __name__ == "__main__":
  my_lil_tree = DT_Node(DT_Empty(), DT_Empty())
  my_big_tree = DT_Node(my_lil_tree, my_lil_tree)
  assert not depth_gt_2(my_lil_tree)
  assert depth_gt_2(my_big_tree)
```

previously: 1) if no target is provided to `ace.fn`, the base can provide a default target (here, `ace.python.python`); 2) a concrete function can be generated immediately by providing a type assignment after `ace.fn` using braces. Listing 16 generates Listing 17.

Lines 16.3-16.11 define a function that generates a recursive algebraic datatype representing a tree given a name and another Ace type for the data at the leaves. This type implemented by the active type family `fp.Datatype`. A name for the datatype and the case names and types are provided

programmatically on lines 16.3-16.8. To support recursive datatypes, the case names are enclosed within a lambda term that will be passed a reference to the datatype itself. These lines also show two more active type families: units (the type containing just one value), and immutable pairs. Line 16.13 calls this function with the Ace type for dynamic Python values, `py.dyn`, to generate a type, aliased `DT`. This type is implemented using class inheritance when the target is Python, as seen on lines 17.1-17.13. For C-like targets, a `union` type can be used (not shown).

The generic function `depth_gt_2` demonstrates two features. First, the base has been setup to treat the final expression in a top-level branch as the return value of the function, consistent with typical functional languages. Second, the `case` "method" on line 16.17 creatively reuses the syntax for dictionary literals to express a nested pattern matching construct. The patterns to the left of the colons are not treated as expressions (that is, a type and translation is never recursively assigned to them). Instead, the active type implements the standard algorithm for ensuring that the cases cover all possibilities and are not redundant [16]. If the final case were omitted, for example, the algorithm would statically indicate an error, just as in statically-typed functional languages like ML.

B.3 Nested Data Parallelism

Functional constructs have shown promise as the basis for a nested data-parallel parallel programming model. Many powerful parallel algorithms can be specified by composing primitives like `map` and `reduce` on persistent data structures like lists, trees and records. Data dependencies are directly encoded in these specifications, and automatic code generation techniques have shown promise in using this information to automatically execute these specifications on concurrent hardware and networks. The Copperhead programming language, for example, is based in Python as well and allows users to express functional programs over lists using common functional primitives, compiling them to CUDA code [5]. By combining active types and the metaprogramming facilities described in Sec. 2 to implement optimizations, like fusion, on the typed syntax trees available from concrete functions, these same techniques can be implemented as an Ace library as well.

B.4 Product Types

Product types (types that group heterogeneous values together) like structs, unions, tuples, records and objects can all be implemented as Ace type families parameterized by a dictionary mapping field names (indices, in the case of tuples) to types. Each family differs slightly in the semantics of the operations available. Structs support mutation, for example, while records do not. Object systems typically introduce additional logic for calling methods, binding special variables like `this`, and accessing information through an inheritance hierarchy. Listing 18 shows an example of each of these. The prototypic object system delegates to a backing record if a field is not available in the foreground. This example also demonstrates cross-compilation to C99, supported for a subset of operations in `clx`. All three abstractions are implemented as simple `structs`, as can be seen in Listing 19. This example further demonstrates the use of argument annotations to generate concrete functions (line 18.9), which is a feature only available when using Python 3.3+.

B.5 Distributed Programming

Many distributed programming abstractions can be understood as implementing object models with complex forms of dynamic dispatch. For example, in Charm++, dynamic dispatch involves message passing over a dynamically load-balanced network [19]. While Charm++ is a sophisticated system, and we do not anticipate that implementing it as an Ace extension would be easy, it is possible to do so by targeting a system like Adaptive MPI, which exposes the Charm++ runtime system [19].

A number of recent languages designed for distributed programming on clusters use a partitioned global address space model, e.g. Chapel [8]. These languages provide first-class support for accessing data transparently across a massively parallel cluster. Their type systems track information about *data locality* so that the compiler can emit more efficient code. The extension mechanism can also track this information in a manner analogous to how address space information is tracked in the OpenCL example above, and target an existing portable run-time such as GASNet [3].

B.5.1 Units of Measure

A number of domain-specific type systems can be implemented within Ace as well. For example, prior work has considered tracking units of measure (e.g. grams) statically to validate scientific code [20]. This cannot easily be implemented using many existing abstraction mechanisms because this information should only be maintained statically to avoid excessive run-time overhead associated with tagging/boxing, and the typechecking logic is reasonably complex. The Ace extension mechanism allows this information to be tracked in the type system, but not included during translation, by a mechanism similar to how address space information is tracked with pointers. Because Ace extensions can use Python, the needed logic can be implemented directly.

B.6 OOFPCLX

Listing 18 [ooclxfp.py] An example combining structs and immutable records using a prototype-based object system, cross-compiled to C99. Uses Python 3 argument annotations.

```
import examples.clx as clx, examples.fp as fp,
  examples.oo as oo

A = clx.Struct('A', {'x': clx.int})
B = fp.Record('B', {'y': clx.float})
C = oo.Prototypic('C', A, B)

@ace.fn(clx.base, clx.c99)
def test(input : C):
  return input.x + input.y
```

Listing 19 [ooclxfp.c] The C99 code generated by running `acec ooclxfp.py`.

```
typedef struct {
    int x;
} A;

typedef struct {
    float y;
} B;

typedef struct {
    A self;
    B proto;
} C;

float test(C input) {
    return C.self.x + C.proto.y;
}
```