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 need to resolve link-time ambiguities.

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, the Ace compiler inverts control over type assignment and translation to type-level functions according to a type-directed protocol that does not lead to ambiguities when type systems are combined. We show that this protocol is flexible enough to admit library-based implementations of types drawn from functional, object-oriented, parallel, low-level and domain-specific languages. We also show how this permits the construction of safe, natural and extensible foreign function interfaces using a novel form of "staged" type propagation from Python into Ace. We also give a simplified calculus that describes how a type-level language can be used to enable type system extensions and demonstrates how type safety can be maintained by lifting a technique from the typed compilation literature into the language.

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 [8, 24, 25].

Unfortunately, researchers and domain experts (*providers*) designing 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* [**?**] 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 [5] 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 [23, 24].

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 [**?**] must overcome the same extrinsic issues as a language like HJ. Second, giving providers too much control over the language can lead to intrinsic safety issues, ambiguities and conflicts between extensions, as we will discuss. Too little control, on the other hand, leaves it difficult to implement interesting abstractions.

This paper describes the design and implementation of Ace, an internally extensible, statically typed programming language designed considering both these extrinsic and in-

trinsic factors. To address the "chicken-and-egg" problem, Ace is implemented within Python, entirely as a library [1, 27]. The top-level of an Ace file can be thought of as a *compilation script* written in Python and, as we will discuss, Python serves as Ace's type-level language. Ace and Python share a common syntax and package system, so users of Ace can leverage established tools and infrastructure directly.

Functions marked by a decorator as under the control of Ace are *statically typechecked*. This makes it possible to rule out many potential issues ahead of execution and thus avoid the need for costly dynamic checks. However, there is *no particular type system built into Ace*. Instead, a type is any compile-time object implementing the `ace.Type` interface. 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, which we call *active type assignment*, 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 (extensions have exclusive control over non-overlapping sets of expressions by construction). Base cases, e.g. variables and literals, 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 [**?**]) into the language.

Ace can be used from the shell, producing source files that can be further compiled or executed by external means. Ace functions targeting a language with Python bindings (e.g. Python itself, as well as the others just described) can also be compiled and invoked interactively from Python scripts. Ace supports a form of *ad hoc polymorphism* (based on singleton types) whereby types propagate into generic functions. This occurs both between Ace functions and when calling Ace functions from Python. In the latter case, the "dynamic type" of a Python value determines a static type "just-in-time" when a call occurs. This represents a form of implicit *staging* and significantly streamlines interactive workflows (common in, for example, scientific computing) that alternate between Python for orchestration and productivity-oriented tasks (e.g. exploratory plotting) and an external language via a foreign-function interface (FFI) for performance- and correctness-critical portions.

The remainder of the paper is organized as follows: in Sec. 2, we describe the basic structure and usage of Ace with an example that simultaneously uses dynamically typed terms (which can be thought of as having the static type `dyn`) alongside statically typed immutable records and a simple prototypic object system, all implemented orthogonally as active types. In Sec. 3, we detail how active type assignment and translation delegate control, at compile-time, to active types and bases. In Sec. 4, we reduce this mechanism to a core lambda calculus, λ_{Ace}, clarify the relationship to type-level computation and show how type safety is maintained. In Sec. 5, we discuss generic functions, FFIs and staging via a complete implementation of the OpenCL type system (and extensions to it) as a library. In Sec. 7, we compare Ace to related work. We conclude in Sec. 8 by summarizing our contributions, highlighting the key features needed by a host language to support these mechanisms, and discussing limitations and potential future work. The appendix contains details omitted from the main body and additional examples.

2. Language Design and Usage

Listing 1 shows 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 package system is Python's package system, so Python's build tools (e.g. `pip`) and package repostories (e.g. `PyPI`) are directly available for distributing libraries that use Ace. 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). 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.

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

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 [acec] _listing1.py successfully generated.
4 $ python _listing1.py
5 Hello, run-time world!
6 Transferring 5.50 to Annie Ace.
7 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 print_check(t1)
12 print_check(t2)
```

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 the language of a regular expression (written, to avoid needing to escape backslashes, using Python's *raw string literals*). Our appendix specifies the typechecking rules, based on [**?**]. We include it to emphasize that we aim to support specialized type systems, not just general purpose constructs like records and numbers.

3. Prototypic inheritance is a feature of some dynamically-typed languages (most notably, Self [**?**] and Javascript). On line 12, we consider a simple statically-typed variant. 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. In our example, we will perform multiple transfers differing only in amount.

what is the right term for this?

2.2 Functions

Functions implementing run-time behavior are distinguished by 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 support a variety of static semantics.

On lines 14-18, we see the function `log_transfer`. As with a Python function, we write a documentation string like the one shown specifying, informally, the behavior of the function. Before moving into the body, however, we also write a *type signature* (line 17) stating simply that the type of `t` is `Transfer`, the prototypic object type described above. As a result, we can assume 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.[1] In version 3.0 of Python, syntax for annotating arguments with arbitrary metadata using `:` was introduced, intended for use by future tools (further suggesting that users of dynamically typed languages see potential in a static typing discipline) [**?**]. Ace supports both notations when the compilation script is run using Python 3.*x*, but we use the more universally available notation here to satisfy our extrinsic goals: the Python 2.x series remains the most widely used by a large margin as of this writing (we will consider cross-compilation in Sec. 5, however).

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

[1] If the right-hand side of every annotation is not a type, it will be an error. In the rare situation where there are no type annotations and the first statement is a dictionary literal, users can use Python's `pass` keyword to skip the line.

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 syntax for a dictionary literal. We must specify which 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 and keywords like `True` and `None`) and the argument is a type, we consider it instead an *ascribed introductory form* of that type (we will see precisely how it is checked shortly). We see the same form used with tuple literals on lines 27-28 to introduce terms of type `Transfer`.

The string, number and dictionary forms inside the outermost forms on lines 24-28 are also introductory forms, but we do not include an ascription because the definition of `C` determines which types they should be analyzed against. As we will discuss shortly, Ace has a *bidirectional type system* distinguishing locations where a type must be *synthesized* from an expression (e.g. at the top-level) and locations where the expression must be *analyzed* against a known type [**?**]. Literal forms can only be analyzed against a type. For example, if they are used in a synthetic context, the base determines a "default ascription". If we had not specified an ascription on line 26, the literal would need to synthesize a type, so the base, `py`, would analyze it against the Python dictionary type, which would cause a type error because the variables used as keys are not bound.

An ascription can also be a type constructor, rather than a type. For example, we might write `[1, 2] (matrix)` rather than `[1, 2] (matrix[int])` assuming the default ascription for integer literals is `int`. The type constructor synthesizes an index based on the types synthesized by the subexpressions. If we wanted a matrix of `float`s, then we would have to write either `[1, 2] (matrix[float])` or `[1(float), 2(float)] (matrix)`. We will see how this works in greater technical detail shortly.

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 first describe the former, then return to interactive compilation in Sec. 5 with a different example. Listing 2 shows how to use the `acec` compiler to typecheck and translate `listing1.py`, producing a file `_listing1.py` which can then be executed by the standard Python interpreter. Note that the **`print`** statement at the top level of the compilation script was indeed evaluated during compilation. These two steps can be combined by running `ace listing1.py` (the intermediate file is not generated in this case unless requested).

The `acec` compiler (a simple Python script) operates in two stages:

1. Executes the provided compilation script (`listing1.py`)
2. For any top-level bindings that are Ace functions in the final environment (with dynamic type `ace.ConcreteFn`, as we will discuss), it initiates the active type-checking and translation process described in Sec. 3. If no type errors are discovered, the translations are extracted and emitted as files (their extensions are determined by the targets in use, discussed in Sec. 5; here our target is the default target associated with the base `examples.py.py`). If a type error is discovered, no file is emitted and the error message is displayed on the console.

In our example, there are no type errors, so the file `_listing1.py` is generated (Listing 3). Notice that:

1. The base placed the translation of the body of the function `__toplevel__` at the top level of the resulting file.
2. Records translated to tuples (or, if there was only a single field, its value was passed around unadorned). The field names were needed only statically.
3. Terms of type `string_in[r"..."]` translated to strings. Checks for well-formedness were rendered unnecessary statically by the type system.[2] Nathan?
4. Decimals are represented as pairs of integers. Conversion to a string happens via a helper function defined in a "run-time" package imported with an internal name, `__ace_0`, to avoid naming conflicts.
5. Prototypic objects are represented as pairs consisting of the fore and the prototype. Dispatch to the appropriate record based on the field name was static (line 5).

The file `_listing1.py` 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. As a result, execution is essentially as fast as it could be given the target language chosen (Python). We will target an even faster language, `OpenCL`, in Sec. 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. More alarmingly, if this code were written in a dynamically-typed language, they would only be found if the code was run on the first day of the month. Static type checking allows us to find these errors during compilation, without needing to execute the code. Listing 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 Ace functions are typechecked. The typechecker raises an exception at the first error, and `acec` prints it.

[2] 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.

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

```
1  from listing1 import A, C, log_transfer
2  from datetime import date
3
4  @py
5  def pay_oopsie(a):
6    {a : A}
7    print "Checking date..."
8    if date.today().day == 1:
9      common = {nome: "Oopsie Daisy",
10               account_num: None,
11               routing_num: "0-0000-0001"} (C)
12     log_transfer((common, a))
13     a.amount += 1000
14
15 print 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
```

3. Active Type Synthesis and Translation

To be revised from here on down

The core of Ace consists of about 1500 lines of Python code implementing its primary concepts: generic functions, concrete functions, active types, active bases and active targets. The latter three comprise Ace's extension mechanism. Extensions provide semantics to, and govern the compilation of, Ace functions, rather than logic in Ace's core.

3.1 Active Types

Active types are the primary means for extending Ace with new abstractions. An active type, as mentioned previously, is an instance of a class implementing the `ace.ActiveType` interface. Listing 6 shows an example of such a class: the `clx.Cplx` class used in Listing 7, which implements the logic of complex numbers. The constructor takes as a parameter any numeric type in `clx` (line 6.2).

3.1.1 Dispatch Protocol

In a compiler for a monolithic language, there would be a *syntax-directed* protocol governing typechecking and translation. In a compiler written in a functional language, for example, one would declare datatypes that captured all forms of types and expressions, and the typechecker would perform exhaustive case analysis over the expression forms. That is, all the semantics are implemented in one place. The visitor pattern typically used in object-oriented languages implements essentially the same protocol. This does not work for an extensible language because new cases and logic need to be added *modularly*, in a safely composable manner.

Instead of taking a syntax-directed approach, the Ace compiler's typechecking and translation phases take a *type-directed approach*. When encountering a compound term (e.g. `e[e1]`), the compiler defers control over typechecking

Listing 6 [in `examples/clx.py`] The active type family `Ptr` implements the semantics of OpenCL pointer types.

```
1  class Cplx(ace.ActiveType):
2    def __init__(self, t):
3      if not isinstance(t, Numeric):
4        raise ace.InvalidTypeError("<error message>")
5      self.t = t
6
7    def type_Attribute(self, context, node):
8      if node.attr == 'ni' or node.attr == 'i':
9        return self.t
10     raise ace.TypeError("<error message>", node)
11
12   def trans_Attribute(self, context, target, node):
13     value_x = context.trans(node.value)
14     a = 's0' if node.attr == 'ni' else 's1'
15     return target.Attribute(value_x, a)
16
17   def type_BinOp_left(self, context, node):
18     return self._type_BinOp(context, node.right)
19
20   def type_BinOp_right(self, context, node):
21     return self._type_BinOp(context, node.left)
22
23   def _type_BinOp(self, context, other):
24     other_t = context.type(other)
25     if isinstance(other_t, Numeric):
26       return Cplx(c99_binop_t(self.t, other_t))
27     elif isinstance(other_t, Cplx):
28       return Cplx(c99_binop_t(self.t, other.t))
29     raise ace.TypeError("<error message>", other)
30
31   def trans_BinOp(self, context, target, node):
32     r_t = context.type(node.right)
33     l_x = context.trans(node.left)
34     r_x = context.trans(node.right)
35     make = lambda a, b: target.VecLit(
36       self.trans_type(self, target), a, b)
37     binop = lambda a, b: target.BinOp(
38       a, node.operator, b)
39     si = lambda a, i: target.Attribute(a, 's'+str(i))
40     if isinstance(r_t, Numeric):
41       return make(binop(si(l_x, 0), r_x), si(r_x, 1))
42     elif isinstance(r_t, Cplx):
43       return make(binop(si(l_x, 0), si(r_x, 0)),
44         binop(si(l_x, 1), si(r_x, 1)))
45
46   @classmethod
47   def type_New(cls, context, node):
48     if len(node.args) == 2:
49       t0 = context.type(node.args[0])
50       t1 = context.type(node.args[1])
51       return cls(c99_promoted_t(t0, t1))
52     raise ace.TypeError("<error message>", node)
53
54   @classmethod
55   def trans_New(cls, context, target, node):
56     cplx_t = context.type(node)
57     x0 = context.trans(node.args[0])
58     x1 = context.trans(node.args[1])
59     return target.VecLit(cplx_t.trans_type(target),
60       x0, x1)
61
62   def trans_type(self, target):
63     return target.VecType(self.t.trans_type(target),2)
```

and translation to the active type of a designated subexpression (e.g. `e`) determined by Ace's fixed *dispatch protocol*. Below are examples of the choices made in Ace. Due to space constraints, we do not show the full protocol.

- Responsibility over **attribute access** (`e.attr`), **subscripting** (`e[e1]`) and **calls** (`e(e1, ..., en)`) and **unary operations** (e.g. `-e`) is handed to the type recursively assigned to `e`.

- Responsibility over **binary operations** (e.g. `e1 + e2`) is first handed to the type assigned to the left operand. If it indicates a type error, the type assigned to the right operand is handed responsibility, via a different method call. Note that this operates like the corresponding rule in Python's *dynamic* operator overloading mechanism; see Sec. 7 for a discussion.
- Responsibility over **constructor calls** (`[t](e1, ..., en)`), where `t` is a *compile-time Python expression* evaluating to an active type, is handed to that type. If `t` evaluates to a *family* of types, like `clx.Cplx`, the active type is first generated via a class method, as discussed below.

3.1.2 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 [1] (appended with a suffix in some cases).

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 6.7). 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 6.8-6.10). We note that error messages are an important and sometimes overlooked facet of ease-of-use [22]. 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 6.17-6.29. 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 6.24). 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 6.50-6.57), 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.

3.1.3 Translation

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).

As seen in Listing 9, we are implementing complex numbers internally using OpenCL vector types, like `int2`. Let us look first at `trans_New` on lines 6.54-6.60, 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 6.12) and binary operations (line 6.31) 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 6.14, 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.

3.2 Active Bases

Each generic function is associated with an active base, which is an object implementing the `ace.ActiveBase` interface. The active base specifies the *base semantics* of that function. It controls the semantics of statements and expressions that do not have a clear "primary" subexpression for the dispatch protocol to defer to. A base is handed control over typechecking of statements and expressions in the same way as active types: via `type_`X and `trans_`X methods. Each function can have a different base.

Literals are the most prominent form given their semantics by an active base. Our example active base, `clx.base`, assigns integer literals the type `clx.int32` while floating point literals have type `clx.double`, consistent with the semantics of OpenCL and C99. The `clx.base` object is an instance of `clx.Base` that is pre-constructed for convenience. Alternative bases can be generated via different constructor arguments. For example, `clx.Base(flit_t=clx.float)` is a base semantics where floating point literals have type `clx.float`. This is useful because some OpenCL devices do not support double-precision numbers, or impose a significant performance penalty for their use. Indeed, to even use the `double` type in OpenCL, a flag must be toggled by a `#pragma` (The OpenCL target we have implemented inserts this automatically when needed, along with some other flags, and annotations like `kernel`). Similarly, for some applications, avoiding accidental overflows is more important than performance. Infinite-precision integers can be implemented as an active type (not shown) and the base can be asked to use it for numeric literals. In all cases, this occurs by inserting a constructor call form, as described above.

The base also initializes the context and governs the semantics of variables and assignment. This can also permit it to allow for the use of "built-in" operators that were not explicitly imported. The semantics of `get_global_id` and `printf` are provided by `clx.base`. In fact, the base provides extended versions of them (the former permits multi-dimensional global IDs, the latter supports typechecking format strings).A base which does not provide them as built-ins (requiring that they be imported explicitly) can be generated by passing the `primitives=False` flag.

3.3 Active Targets and Cross-Compilation

An active target, which we describe being used above, is an instance of `ace.ActiveTarget`. It has two primary roles in Ace: 1) providing an API for code generation to a target language, providing term and type constructors for use during translation (and, optionally, a type system implementation to support representational consistency checks); 2) implementing the interoperability layer to support calling of generic or concrete function directly from Python, as described in Sec. 5.6.

An active type or base can support multiple active targets by simply examining `target` during translation (e.g. by performing capability checks using Python's `hasattr` function). Alternatively, the active target can simulate the interface of a different target but generate code differently. For example, although our implementation of OpenCL's type system as a collection of active types and an active base relies on a target that supports OpenCL terms and types, our `CUDA` and `C99` targets provide partial support for the same term and type forms, in order to support cross-compilation. We will show the latter being used in Sec. 6.

3.4 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 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. 7).

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 [13]. 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 [1]. 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 7 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 7. The first, on line 7.7, 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 7.9 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.ConcreteFn`, consisting of an abstract syntax tree annotated with types and translations along with a reference to the original generic function.

To produce an output file from an Ace "compilation script" like `listing7.py`, the command `acec` can be invoked from the shell, as shown in Listing 8.

Listing 7 [`listing7.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 8 Compiling `listing7.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 9 [`listing7.cl`] The OpenCL file generated by Listing 8.

```
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 }
```

5.5 Types

Lines 7.3-7.5 construct the types assigned to the arguments of `map` on lines 7.7-7.10. 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. 3.1). 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 7.3 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 [13]). 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. 6, 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. 8). 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 7.4 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. 3.1). 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 9.14-9.21. 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 7.5 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 9. In other words, types *propagate* into generic functions – we didn't need to compile `negate` explicitly. This also explains the error printed on line 8.3-8.4: 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 [25, 28]. Developers can call into native functions using Python's foreign function interface (FFI), for example. A more recent trend is to generate and compile code without leaving Python, using a Python wrapper around a compiler. For example, `weave` works with C and C++, and `pycuda` and `pyopencl` work with CUDA and OpenCL, respectively [20]. The OpenCL language was designed for this workflow, exposing a retargetable compiler and data management routines as an API, called the *host API* [13]. The `pyopencl`

Listing 10 [listing10.py] A full OpenCL program using the clx Python bindings, including data transfer to and from a device and direct invocation of a generic function, map.

```
1  import listing1, listing2, examples.clx as clx, numpy
2
3  clx.opencl.ctx = clx.Context.for_device(0, 0)
4
5  input = numpy.ones((1024,1024))
6  d_input = clx.to_device(input)
7  d_output = clx.alloc(like=input)
8
9  listing1.map(d_input, d_output, listing2.negate,
10   global_size=d_in.shape)
11
12 assert (cl.from_device(d_out) == input * -1).all()
```

library exposes this API to Python and simplifies interoperation with numpy, a popular package for manipulating contiguously-allocated numeric arrays in Python [20].

Ace supports a refinement to this workflow, as an alternative to the acec compiler described above, for targets that have wrappers like this available, including clx.opencl. Listing 10 shows an example of this workflow where the user chooses a compute device (line 10.3), constructs a numpy array (line 10.5), transfers it to the device (line 10.6), allocates an empty equal-sized buffer for the result of the computation (line 10.7), launches the generic kernel map from Listing **??** with these device arrays as well as the function negate from Listing **??** (line 10.9) choosing a number of threads equal to the number of elements in the input array (line 10.10), and transfers the result back into main memory to check that the device computed the expected result (line 10.12).

For developers experienced with the usual OpenCL or CUDA workflow, the fact that this can be accomplished in a total of 6 statements may be surprising. This simplicity is made possible by Ace's implicit tracking of types throughout the code. First, numpy keeps track of the type, shape and order of its arrays. The type of input, for example, is numpy.float64 by default, its shape is (1024,1024) and its order is row-major by default. The pyopencl library, which the clx mechanism is built upon, uses this metadata to automatically call the underlying OpenCL host API function for transferring byte arrays to the device without requiring the user to calculate the size. The clx wrapper further retains this metadata in d_input and d_output, the Python wrappers around the allocated device arrays. The Ace active type of these wrappers is an instance of clx.NPArray parameterized by this metadata. This type knows how to typecheck and translate operations, like indexing with a multi-dimensional thread index, automatically.

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 [20], 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. 6 for more interesting possibilities).

6. 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.

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

Listing 11 [datatypes_t.py] An example using statically-typed functional datatypes.

```
1  import ace, ace.python as py, examples.fp as fp
2
3  Tree = lambda name, a: fp.Datatype(name,
4    lambda tree: {
5      'Empty': fp.unit,
6      'Leaf': a,
7      'Node': fp.tuple(tree, tree)
8    })
9
10 DT = Tree('DT', py.dyn)
11
12 @ace.fn(py.Base(trailing_return=True))[[DT]]
13 def depth_gt_2(x):
14   x.case({
15     DT.Node(DT.Node(_), _): True,
16     DT.Node(_, DT.Node(_)): True,
17     _: False
18   })
19
20 @ace.fn(py.Base(main=True))[[]]
21 def __main__():
22   my_lil_tree = DT.Node(DT.Empty, DT.Empty)
23   my_big_tree = DT.Node(my_lil_tree, my_lil_tree)
24   assert not depth_gt_2(my_lil_tree)
25   assert depth_gt_2(my_big_tree)
```

Listing 12 [datatypes.py] The dynamically-typed Python code generated by running `acec datatypes_t.py`.

```
1  class DT(object):
2    pass
3
4  class DT_Empty(DT):
5    pass
6
7  class DT_Leaf(DT):
8    def __init__(self, data):
9      self.data = data
10
11 class DT_Node(DT):
12   def __init__(self, data):
13     self.data = data
14
15 def depth_gt_2(x):
16   if isinstance(x, DT_Node):
17     if isinstance(x.data[0], DT_Node):
18       r = True
19     elif isinstance(x.data[1], DT_Node):
20       r = True
21   elif True:
22     r = False
23   return r
24
25 if __name__ == "__main__":
26   my_lil_tree = DT_Node(DT_Empty(), DT_Empty())
27   my_big_tree = DT_Node(my_lil_tree, my_lil_tree)
28   assert not depth_gt_2(my_lil_tree)
29   assert depth_gt_2(my_big_tree)
```

6.2 Recursive Labeled Sums

Listing 11 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 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 11 generates Listing 12.

Lines 11.3-11.11 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 11.3-11.8. 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 11.13 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 12.1-12.13. 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 11.17 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 [14]. If the final case were omitted, for example, the algorithm would statically indicate an error, just as in statically-typed functional languages like ML.

6.3 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 [4]. 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.

6.4 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 13 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 14. This example further demonstrates the use of argument annotations to generate concrete functions (line 13.9), which is a feature only available when using Python 3.3+.

6.5 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 [17]. 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 [17].

A number of recent languages designed for distributed programming on clusters use a partitioned global address space model, e.g. Chapel [7]. 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 [2].

6.5.1 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 [18]. 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.

Listing 13 `[ooclxfp.py]` An example combining structs and immutable records using a prototype-based object system, cross-compiled to C99. Uses Python 3 argument annotations.

```
1  import examples.clx as clx, examples.fp as fp,
2    examples.oo as oo
3
4  A = clx.Struct('A', {'x': clx.int})
5  B = fp.Record('B', {'y': clx.float})
6  C = oo.Prototypic('C', A, B)
7
8  @ace.fn(clx.base, clx.c99)
9  def test(input : C):
10   return input.x + input.y
```

Listing 14 `[ooclxfp.c]` The C99 code generated by running `acec ooclxfp.py`.

```
1  typedef struct {
2      int x;
3  } A;
4
5  typedef struct {
6      float y;
7  } B;
8
9  typedef struct {
10     A self;
11     B proto;
12 } C;
13
14 float test(C input) {
15     return C.self.x + C.proto.y;
16 }
```

7. 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 [10]. 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 [15]. 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* [34]. A number of projects, such as Blitz++, have taken advantage of C++ template metaprogramming to implement domain-specific optimizations [33]. Others have chosen custom metalanguages (e.g. Xroma [34], mbeddr [35]). 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

Approach	Examples	Library	Extensible Syntax	Extensible Type System	Extensions Compositional	Alternative Targets
Active Types	Ace	●	○	●	●	●
Desugaring	SugarJ [11]	○	●	○	○	○
Rule Injection	Racket [31], Xroma [34], mbeddr [35]	[31]	[35]	●	○	○
Static macros / Metaprogramming	Scala [3], MorphJ [16], OJ [30] MetaML [29], Template Haskell	○	○	○	●	○
Cross-Compilation	Delite [6]	●	○	○	○	●

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

to structural polymorphism [21]. 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 [32] and *metaobject dispatch* [19] 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 11, 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. 3.4, 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 [12]). 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-external approaches to achieve interoperability, because languages are not aware of each other's semantics and merging languages is not guaranteed to be sound.

8. Discussion

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 [9]. 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 famil-

iarity 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 [26]. 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] The python language reference (http://docs.python.org/), 2013.

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

[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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.