Ace: Growing a Statically-Typed Language Inside a Python

Abstract

Evidence suggests that programmers are reluctant to adopt new programming languages to gain access to new abstractions, even when they agree that these abstractions would be valuable to them. This suggests a need for languages that are *compatible* with existing languages, tools and infrastructure and *internally extensible*, so that adopting a new primitive abstraction requires only importing a library in the usual way.

In this paper, we introduce Ace, a language compatible with tools and infrastructure developed for Python, one of the most widely-adopted dynamically-typed languages today. While Python, like other similar languages, was designed for simple scripting tasks, Ace is designed for more complex situations where static typechecking and programmatic control over compilation may be beneficial. Unlike most statically-typed languages, however, Ace's type system and semantics can be extended from within by novel mechanisms that avoid key interference issues faced by previous mechanisms. We show that these can be used to implement a range of statically-typed functional, object-oriented, parallel, low-level and domain-specific abstractions, as well as safe interoperability layers with existing languages, as orthogonal libraries.

1. Introduction

Asking programmers to import a new library is far more practical than asking them to adopt a new programming language. Indeed, recent empirical studies underscore the difficulties of driving new languages into adoption, finding that extrinsic factors like compatibility with existing codebases and libraries, team familiarity and tool support are at least as important as intrinsic factors [24**? ?**]. As a result, many developers cannot use abstractions they might prefer because these abstractions are only available in languages they cannot adopt [**? ?**]. This issue was perhaps most succinctly expressed by a participant in a recent study by Basili et al. [1] who stated "I hate MPI, I hate C++. [But] if I had to choose again, I would probably choose the same."

Unfortunately, researchers and domain experts who design and develop potentially useful new abstractions can find it difficult to implement them in terms of the general-purpose abstraction mechanisms available in mainstream languages. This is particularly salient for abstractions that require support from the typechecker or compiler, such as those focused on correctness and performance, as well as those that introduce more concise or natural notations for existing abstractions. For example, a recent controlled study comparing a new language, Habanero-Java (HJ), with a comparable library, `java.util.concurrent`, found that the language-based approach was more concise, correct and easy-to-use, but concluded that the library-based approach was more practical outside the classroom because HJ introduced new constructs and keywords into Java, requiring the adoption of a new toolchain, which could lead to compatibility issues with plain Java code [7].

Internally-extensible languages promise to reduce the need for new standalone languages by giving abstraction providers more direct control over a base language's syntax and semantics from within libraries. By using such a language, programmers gain the ability to granularly import the primitive abstractions best suited to each part of their application or library. The research community thus gains the ability to more easily develop, deploy and evaluate new abstractions in the context of existing codebases, narrowing one of the gaps between research and practice [1].

Unfortunately, internally-extensible languages available today have several problems. First, an extension mechanism itself may require modifying a base language with constructs for defining, importing and using extensions. The extension mechanism is not itself a library, so it faces many of the same extrinsic issues as other new languages like HJ, leading to a "bootstrapping" problem. The extension mechanisms available today also have several intrinsic problems related to safety and expressiveness that require technical solutions before it would be appropriate to widely rely on them. We evaluate related work in Section **??**.

This paper describes the design and implementation of Ace, an internally-extensible language designed considering both extrinsic and intrinsic criteria. To solve the bootstrapping problem, Ace is implemented entirely as a library within the popular Python programming language. Ace and Python share a common syntax and package system, allowing Ace to leverage its well-established tools and infrastructure directly. Python serves as the compile-time metalanguage for Ace, but Ace functions themselves do not operate according to Python's fixed dynamically-typed semantics (cf. [**? ?**]). Instead, Ace has a statically-typed semantics that can be extended by users from within libraries.

More specifically, each Ace function can be annotated with a base semantics that determines the meaning of simple expressions like literals and certain statements. The semantics of the remaining expressions and statements are governed by logic associated with the type of a designated subexpression. We call the user-defined base semantics *active bases* and the types in Ace *active types*, borrowing terminology from *active libraries* ([32], see Sec. **??**). Both are objects that can be defined and manipulated at compile-time using Python. An important consequence of this mechanism is that it permits *compositional* reasoning – active bases and active types govern only specific non-overlapping portions of a program. As a result, clients are able to import any combination of extensions with the confidence that link-time ambiguities cannot occur (unlike many previous approaches, as we discuss in Sec. **??**).

The *target* of compilation is also user-defined. We will show examples of Ace targeting Python as well as OpenCL and CUDA, lower-level languages often used to program graphics hardware. An active base or type can support multiple *active targets*, which mediate translation of Ace code to code in a target language. Ace functions targeting a language with Python bindings can be called directly from Python scripts, with compilation occurring implicitly. For some data structures, types can propagate from Python into Ace. We show how this can be used to streamline the kinds of interactive workflows that Python is often used for. Ace can also be used non-interactively from the shell, producing source files that can be further compiled and executed by external means.

The remainder of the paper is organized as follows: in Sec. 2, we describe the basic structure and usage of Ace with an example library that internalizes and extends the OpenCL language. Then in Sec. 3, we show how this and other libraries are implemented by detailing the extension mechanisms within Ace. To explain and demonstrate the expressiveness of these mechanisms (in particular, active types) further, we continue in Sec. 4 by showing a diverse collection of abstractions drawn from different language paradigms that can be implemented as orthogonal libraries in Ace. We include functional datatypes, objects, macros, and typesafe format strings and regular expressions. In Sec. **??**, we compare Ace to related work on language extensibility and metaprogramming. We conclude in Sec. 5 by summarizing our contributions, discussing the essential features needed by a host language to support these mechanisms, and describing their limitations and potential future work.

2. Language Design and Usage

Listing 1 shows an example of an Ace file. As promised, the top level of an Ace file is written directly in Python, requiring no modifications to the language (versions 2.6+ or 3.3+) nor features specific to CPython (so Ace supports alternative implementations like Jython and IronPython). This choice pays immediate dividends on line 1: 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 Ace libraries.

The top-level statements in an Ace file, like the `print` statement on line 10, are executed to control the compile-time behavior, rather than the run-time behavior, of the program. That is, Python serves as the *compile-time metalanguage* (and, as we will see shortly, the *type-level language*) of Ace. Functions containing run-time behavior, like `map`, are governed by a semantics that can differ from Python's (in ways that we will describe below), but they share Python's syntax. As a consequence, users of Ace immediately benefit from an ecosystem of well-developed tools that work with Python syntax, including parsers, code highlighters, editor modes, style checkers and documentation generators.

2.1 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 [14]. 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.

Listing 1 [`listing1.py`] A generic data-parallel higher-order map function targeting OpenCL.

```
1  import ace, examples.clx as clx
2
3  @ace.fn(clx.base, clx.opencl)
4  def map(input, output, f):
5      thread_idx = get_global_id()
6      output[thread_idx] = f(input[thread_idx])
7      if thread_idx == 0:
8          printf("Hello, run-time world!")
9
10 print "Hello, compile-time world!"
```

Listing 2 [`listing2.py`] Metaprogramming with Ace, showing how to construct generic functions from abstract syntax trees.

```
1  import ace, examples.clx as clx, ast, astx
2
3  _fn = ace.fn(clx.base, clx.opencl)
4
5  scale = _fn(ast.parse("""def scale(x, s):
6      return x * s"""))
7
8  negate = _fn(astx.specialize(scale.ast, "negate",
9      s=ast.parse("-1")))
```

2.2 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 [**?**]. In other words, line 3 could be replaced by inserting the following statement on line 9:

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

figure out how to center this

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).

2.3 Metaprogramming in Ace

Generic functions can be generated directly from ASTs as well, providing Ace with support for straightforward metaprogramming. Listing 2 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.

Listing 3 [listing3.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 4 Compiling listing3.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.
```

2.4 Concrete Functions and Explicit Compilation

To compile a generic function to a particular *concrete function*, a type must be provided for each argument, and typechecking and translation must then succeed. Listing 3 shows how to explicitly provide type assignments to map using the subscript operator. We attempt to do so three times in Listing 3. The first, on line 3.7, 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 3.9 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 listing3.py, the command acec can be invoked from the shell, as shown in Listing 4. The acec compiler (a simple Python script) operates in two stages:

1. Executes the provided Python file (listing3.py).
2. Extracts the translations from concrete functions and other top-level constructs (e.g. types requiring declarations, or generated imports and pragmas) in the top-level Python environment. This may produce one or more files as mediated by the active targets that were used (here, just listing3.cl, but a web framework built upon Ace might produce separate HTML, CSS and JavaScript files; see Sec. 3.3).

In this case, stage 1 results in the output on lines 4.2-4.4. The type error printed on lines 4.3-4.4 will be explained in the next section. The compiler then enters stage 2 and concludes with the message on line 4.5 to indicate that one file was generated. This file is shown in Listing 5 and can be used by any programs that consume OpenCL code (e.g. a C program that invokes the generated kernels via the OpenCL host API). We will show in Section 2.8 that for targets with Python bindings, such as OpenCL, CUDA, C, Java or Python itself, generic functions can be executed directly, without any of the explicit compilation steps in Listings 3-4.

2.5 Types

Lines 3.3-3.5 construct the types assigned to the arguments of map on lines 3.7-3.10. 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,

Listing 5 [listing3.cl] The OpenCL file generated by Listing 4.

```
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();
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();
21   output[thread_idx] = negate__1_(input[thread_idx]);
22   if (thread_idx == 0) {
23     printf("Hello, run-time world!");
24   }
25 }
```

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. 3.2). 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 3.3 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 [14]). 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. 4, 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. 5). 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 3.4 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. 3.2). 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 5.14-5.21. 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).

The type `TF` on line 3.5 is extracted from the generic function `negate` constructed in Listing 2. Generic functions, according to Sec. 2.2, have not yet had a type assigned to them, so it may seem perplexing that we are nevertheless assigning a type to it.

This notion of types as metalanguage objects is key to the Ace compilation model and also enables other mechanisms that we will discuss in subsequent sections.

2.6 Type Propagation

The type assigned to the third argument, `f`, on both Lines 4.9 and 4.10, is `add5.ace_type`. The `ace_type` attribute of a generic function is an instance of `ace.GenericFnType`, the type of Ace generic functions. Ace generic functions are compiled to concrete functions automatically at all internal call sites. That is, when the compiler encounters the call to `f` inside `map` when compiling `map_add5_double`, it compiles a version of `add5` specialized to the `double` type (seen on Line 5.3), and similarly when compiling `map_add5_int` (on Line 5.16, automatically given a unique name to avoid conflicts). This mechanism is called *type propagation*. We did not need to use `add5.compile(double)` before compiling `map_add5_dbl` because only functions that are never called in the process of compiling other functions in a module need type information explicitly provided, supporting **ease-of-use** by increasing conciseness.

In effect, this scheme allows for a form of higher-order functional programming even when targeting languages, like OpenCL, that have no support for higher-order functions (OpenCL, unlike C99, does not support function pointers). This works because the `ace.GenericFnType` for one function, such as `add5`, is not equal to the `ace.GenericFnType` for a superficially similar function, such as `add6` (defined as one would expect). To put it in type theoretic terms, `ace.GenericFnType`s are singleton types, *uniquely inhabited* by a single generic function. A consequence of this is that they are not useful as first-class values (i.e. they cannot be written into a collection). This is often valuable, particularly in parallel programming where compile-time specialization is valuable to avoid **performance** and **ease-of-use** issues that occur when using function pointers.

Concrete functions, on the other hand, can be given a true function type (e.g. `add5` could be compiled to a concrete function with type `int` $\rightarrow$ `int`) if targeting a backend that supports them, such as C99, or by using an integer-indexed jump table in OpenCL (we have not implemented this mechanism using Ace as of the time of writing, but do not anticipate difficulties in doing so).

Type propagation via generic functions can be compared to template specialization in C++, where both the template headers (containing nested template parameters for function arguments) and specialization parameters at any call sites are inferred automatically from usage. This significantly simplifies a sophisticated feature of C++ and introduces it to OpenCL and C, which do not support templates. Other uses for C++ templates are subsumed by the metaprogramming features discussed in Section 5.

2.7 Whole-Function Type Inference

On Line 5 in the generic `map` function in Listing 1, the variable `gid` is initialized with the result of calling the OpenCL primitive `get_global_id`. The type for `gid` is never given explicitly. This is a simple case of Ace's more general *whole-function type inference*. In this case, `gid` will be inferred to have type `size_t` because that is the return type of `get_global_id` (as defined in the OpenCL specification, which the `ace.OpenCL` module follows). The result can be observed on Lines 11 and 24 in Listing 5.

Inference is not restricted within single assignments, as in the `map` example, however. Multiple assignments to the same identifier with values of differing types, or multiple return statements, can be combined if the types in each case are compatible with one another (e.g. by a subtyping relation or an implicit coercion). In Listing **??**, the `threshold_scale` function assigns different values to `y` in each branch of the conditional. In the first branch, the value `0` is an `int` literal. However, in the second branch of the loop, the type depends on the types of both arguments, `x` and `scale`. We show two choices for these types on Lines 11 and 12. Type inference correctly combines these two types according to OpenCL's C99-derived rules governing numeric types (defined by the user in the `OpenCL` module, as we will describe in Section 3). We can verify this programmatically on Lines 12 and 13. Note that this example would also work correctly if the assignments to `y` were replaced with `return` statements (in other words, the return value of a function is treated as an assignable for the purpose of type inference).

Listing 6 [`listing6.py`] A full OpenCL program using the `Ace.OpenCL` Python bindings, including data transfer to and from a device and direct invocation of a generic function, `map`, as a kernel without explicit compilation.

```
1  import listing1, listing2, examples.clx as clx, numpy
2
3  clx.default_ctx = clx.Context.for_device(0, 0)
4
5  input = numpy.ones((1024,))
6  d_input = clx.to_device(input)
7  d_output = clx.alloc(like=input)
8
9  listing1.map(d_input, d_output, listing2.thresh_dbl,
10   global_size=d_in.shape, local_size=(128,))
11
12 assert (cl.from_device(d_out) == input * 2).all()
```

2.8 Implicit Compilation and Execution

Professional end-users today generally use dynamically-typed high-level languages like MATLAB, Python, R or Perl for tasks that are not performance-sensitive, such as small-scale data analysis and plotting [24]. For portions of their analyses where the performance overhead of dynamic type checking and automatic memory management is too high, they will typically call into code written in a statically-typed, low-level language, most commonly C or Fortran, that uses low-level parallel abstractions like pthreads and MPI [1, 5]. Unfortunately, these low-level languages and abstractions are notoriously difficult to use and automatic verification is intractable in general.

integrate this

As discussed in the Introduction, a common workflow for professional end-users is to use a high-level scripting language for orchestration, small-scale data analysis and visualization and call into a low-level language for performance-critical sections. Python is designed for this style of use [26] and is widely used by professional end-users in HPC as a high-level scripting language. It features mature support for calling into code written in low-level languages. Developers can call into native libraries using its foreign function interface (FFI), or by using a wrapper library like `pycuda` for code compiled with CUDA [20] or `weave` for C and C++.

Although C, C++ and CUDA's compilers are separate executables on the system, the OpenCL language was also designed for this workflow, in that it exposes the compiler and memory management directly as an API, called the *host API*. The `pyopencl` module exposes this API and supports basic interoperability with `numpy`, the low-level linear algebra package for Python. With both

pyopencl and pycuda, users generate OpenCL or CUDA source code as strings, compile it programmatically, then execute it using the run-time APIs that each library provides [20]. This mode of use can be considered one where Python serves as an *interactive compilation environment*.

Ace supports a refinement to this workflow, as an alternative to the acec compiler described above. In other words, acec can generate source code for kernels but it does not specify how and from what language thet will be called. This mode of use allows Python to be the calling language.

At the time of writing, this is only supported with the OpenCL backend, but other backends can implement this feature as well by satisfying a simple interface specification. The OpenCL host API wraps and is a superset of the pyopencl, adding a simpler type-aware API. Both generic functions and concrete functions can then be called like regular Python functions, with additional keyword arguments specifying the global and local size (i.e. the number of threads and how they are grouped). Device buffers can carry type information, unlike in the basic OpenCL host API, and this type information can be propagated from Python to Ace functions directly, as if the call had been within a kernel, eliminating the compile step that we have been using thus far and thus all mention of types.

An example of this for the generic map function defined in Listing 1 is shown in Listing 6, with the call itself on Lines 12-13. The first two arguments to map are OpenCL buffers, generated using a simplified wrapper to the pyopencl APIs on Lines 9-10. This wrapper associates type information with each buffer, based on the type of the numpy array, and this is used to implicitly compile map as appropriate the first time it is called for any given combination of input types. Notice also that add5 is passed in directly.

By way of comparison, the same program written using the OpenCL C API directly is an order of magnitude larger and significantly more difficult to read and understand. It does not support higher-order functions nor is there any way to write map in a type-generic way. A full implementation of the logic of map written using the pyopencl bindings and metaprogramming techniques as described in [20] is still twice as large and more difficult to comprehend than the code we have shown thus far. Not shown are several additional conveniences, such as delegated kernel sizing and In and Out constructs that can reduce the size and improve the clarity of this code further; due to a lack of space, the reader is referred to the language documentation.

3. Extensibility

3.1 Active Bases

3.2 Active Types

3.3 Active Targets

If no active target is explicitly provided, as in our example, the active base provides a default target. The default target provided by clx.base is clx.opencl, but other targets, like clx.cuda, a vendor-specific language similar to OpenCL, as well as Python itself (ace.python), can also be defined in libraries. One base can support multiple targets. We will discuss targets in Sec. **??**.

Thus far, we have been discussing the OpenCL module in our examples. This module faithfully implements all the types and operations of the OpenCL kernel language, a portable standard for writing low-level code on multi-core processors and accelerators [14]. The functions we wrote have used these primitives with the OpenCL.OpenCL backend, so the translation has been direct and no run-time overhead has been introduced. Thus, as described so far, Ace has affirmatively resolved the question it was originally conceived to address, discussed in Section 2.1.

3.4 Monolithic vs. Extensible Languages

OpenCL is not necessarily the best tool for every job in high-performance computing. Indeed, HPC is an area where designing a set of primitives that satisfy all users has been particularly challenging, and it appears unlikely that a broad consensus will emerge given the variety of architectures, applications, scales and user communities that it serves, and the number of seemingly promising abstractions that emerge continuously targeting various subsets of this problem space.

It is therefore a concern that 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 algorithms and abstractions of their design. Although highly-expressive general-purpose mechanisms have been developed (such as object systems or algebraic 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, or if defining an abstraction in terms of existing mechanisms is unnatural or verbose (in summary, to push the boundaries of **verifiability**, **performance** and **ease-of-use**). In these situations, it would be desirable to have the ability to modularly extend existing systems (**continuity**) with new compile-time logic (**extensibility**) and be assured that such extensions will never interfere with one another when used in the same program (**interoperability**).

3.5 Active Typechecking and Translation

To achieve these criteria, Ace has been designed around a minimal, extensible core. Users introduce the compile-time logic associated with primitive types and operations from *within libraries*, as opposed to by some language-external mechanism such as a domain-specific language framework or extensible compiler (see Section **??**).

The two phases of compilation that can be controlled by users are together referred to as *active typechecking and translation (AT&T)*. They are invoked the first time the compile method of a generic function is called, or when type propagation occurs, with a particular type assignment (a cached concrete function is returned subsequently).

3.5.1 Active Typechecking

When the compiler encounters an expression, it must first verify its validity by either assigning it a type or raising a meaningful type error. Rather than defining fixed logic for this, Ace defers control to the type of the *primary operand* according to a *dispatch protocol*. Because types are metalanguage instances of user-defined types derived from ace.Type (cf. Section 2.5), dispatch corresponds to calling a method of this base class.

Let us again consider the data-parallel map example of Section 2. When map is compiled on Line 9 of Listing 3, the first two argument types are gptr(double). This type is an instance of the user-defined OpenCL.PtrType which inherits from ace.Type. When the compiler encounters the expression input[gid] on Line 6, the dispatch protocol is to defer control to the type of input by invoking the method named resolve_Subscript.

The relevant portion of OpenCL.PtrType as well as gptr is shown in Listing 7. The resolve_Subscript method on Line 8 receives a context and the syntax tree of the node being considered. The context contains information about variables in scope and other contextual information, and also exposes a method, resolve_type, that this method uses to recursively resolve the types of other subexpressions, here the slice which will be gid, as

Listing 7 [ace.OpenCL] A portion of the implementation of OpenCL pointer types implementing subscripting logic using the Ace extension mechanism, AT&T.

```
1  import ace
2
3  class PtrType(ace.Type):
4    def __init__(self, target_type, addr_space):
5      self.target_type = target_type
6      self.addr_space = addr_space
7
8    def resolve_Subscript(self, context, node):
9      slice_type = context.resolve(node.slice)
10     if isinstance(slice_type, IntegerType):
11       return self.target_type
12     else:
13       raise TypeError('<error message>', node)
14
15   def translate_Subscript(self, context, node):
16     value = context.translate(node.value)
17     slice = context.translate(node.slice)
18     return ace.copy_node(node,
19       value=value, slice=slice,
20       code=value.code + '[' + slice.code + ']')
21
22   # ...
23
24 def gptr(target_type):
25     return PtrType(target_type, "__global")
```

needed. The context will have that `gid` is the machine-dependent integer type `size_t` as discussed in Section 2. On Line 10, it confirms that this type is an instance of an integer type and thus, it assigns the whole expression, `input[gid]`, the target type of the pointer, `double`. Had the function attempted to index `input` using a non-integer expression, the method would take the other branch of the conditional and raise a type error, with a custom error message, on Line 13. We note that error messages are an important component of **ease-of-use** [22]. Indeed, a widely-reported frustration with C++ is that it produces overly verbose and cryptic error messages, particularly when templates are used in clever ways.

3.5.2 Dispatch Protocol

Below are examples of the rules that comprise the Ace dispatch protocol. Due to space constraints, we do not list the entire dispatch protocol, which contains a rule for each possible syntactic form in the language.

- Responsibility over **unary operations** like `-x` is given to the type assigned to the operand, `x`.
- Responsibility over **binary operations** is first handed to the type assigned to the left operand. If it indicates that it does not understand the operation, the type assigned to the right operand is handed responsibility, via a different method call. Note that this operates similarly to the Python run-time operator overloading protocol; see Section **??**.
- Responsibility over **attribute access** (`e.attr`), **subscript access** (`e[idx]`) and **calls** (`e(e1, ..., en)`) is handed to the type assigned to `e`.
- In support of the type inference mechanism described in Section **??**, responsibility to resolve a type given multiple assignments can be taken by any of the types of the assignments, with priority given to later types.

3.5.3 Active Translation

Once typechecking a method is complete, the compiler must subsequently translate each Ace source expression into an expression in the target language. This has been OpenCL in the examples thus far, but we describe a generalization of this in the next section.

It does so by again applying the dispatch protocol to call a method of the form `translate_`X, where X is the syntactic form of the expression. This method is responsible for returning a copy of the expression's ast node with an additional attribute, `code`, containing the source code of the translation, represented here as a string though it may also be represented in a structured manner. In our example, it is simply a direct translation to the corresponding OpenCL attribute access (Line 20), using the recursively-determined translations of the operands (Lines 16-17). More sophisticated abstractions may insert arbitrarily complex statements and expressions during this phase. The context also provides some support for non-local insertions, such as new top-level type declarations, imports and helper code (not shown).

3.5.4 User-Defined Backends

Thus far, we have discussed using OpenCL itself as a backend for our implementation of the OpenCL primitives. This backend is called `OpenCL` in the `ace.OpenCL` module. Ace supports the definition of new backends in a manner similar to the introduction of new types, by extending the `ace.Backend` base class. Backends are specified for a function by using the decorator form `@backend.fn`, as can be seen in the previous examples. Backends are responsible for some aspects of the grammar that do not admit simple dispatch to the type of a subterm, such as number and string literals or statement forms like `while` (though `for` is handled by an iterator-like protocol, not shown).

In addition to the OpenCL backend, preliminary C99 and CUDA backends are available (with the caveat that they have not been as fully developed or tested as of this writing.) This allows us to use the OpenCL kernel language, which offers a simplified variant of C99, without relying on the full OpenCL stack, which may not be well-supported by vendors with proprietary solutions such as CUDA. Backends not based on the C family are also possible, but we leave such developments for future work. During the translation phase, types can access the backend via the context and emit different code for different supported backends (not shown above due to space considerations).

3.5.5 Composability and Interoperability

Because a type can only exert control over typechecking and translation of operations where an expression of that type is the primary operand, extensions defined using AT&T can not interfere with one another by construction. This does not imply that there are no **interoperability** issues to consider, however. A type may need to know about other types (e.g. pointer types need to know about integer types) and if this is done without consideration of future extensions, it may be difficult to integrate data produced by one type system with another. These issues cannot easily be addressed by a language design, however.

4. Examples

The development of the full OpenCL language using only the extension mechanisms described above provides evidence of the power of this approach. However, to be truly useful, the mechanism must be able to express a wide array of higher-level primitive abstractions. We briefly describe a number of other abstractions that are possible using this mechanism. Many of these are currently available in existing languages either via libraries or as primitives of some specialized language. A study comparing a language-based concurrency solution for Java with an equivalent, though less clean, library-based solution found that language support is preferable but leads to many of the issues we have described [7].

4.0.6 Parallel Programming

OpenCL

perative syntax does not preclude writing programs in a reasonable functional style (due largely to its support for tuples).

4.0.7 Other Use Cases in HPC

Interoperability Layers The design criteria of **continuity** and **interoperability** require consideration of the large body of codes written in a variety of existing languages, across a number of paradigms. Although it is often possible to call from one language to another using a foreign function interface (FFI), this is almost never natural or statically safe. In Ace, however, extensions could be written that internalize the foreign language's type system (note that dynamically-typed languages can be considered to have a single type, often called `dyn`) and emit code that hides the FFI from users, achieving **verifiability** and **ease-of-use**.

Specialized Optimizations In many cases, specialized code optimizations requires statically tracking invariants throughout a program. Often these optimizations can be encoded as a type system for this reason. For instance, a course project using Ace implemented substantial portions of the GPU-specific optimizations described in [33] as a library, using types to track affine transformations of the global ID in order to construct a summary of the memory access patterns of the kernel. This information can be used both for single-kernel optimization (as in [33]) and for future research on cross-kernel fusion and other optimizations.

Domain-Specific Type Systems Although not strictly related to HPC, a number of domain-specific type systems related to computational science can be implemented within Ace. 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 existing abstraction mechanisms because this information should only be maintained statically to avoid excessive run-time overhead associated with tagging. The Ace extension mechanism allows this information to be tracked in the type system, but not included during translation.

Instrumentation Several sophisticated feedback-directed optimizations and adaptive run-time protocols require instrumenting code in various ways. The extension mechanism combined with support for patching classes dynamically using Python enables granular instrumentation that can consider both the syntactic form of an operation as well as its constituent types, easing the implementation of such tools.

This ability could also be used to collect data useful for more rigorous usability and usage studies of languages and abstractions, and we plan on following up on this line of research going forward.

4.1 Active Libraries

Libraries that contain compile-time logic have been called *active libraries* in prior proposals [32]. A number of projects, such as Blitz++, have taken advantage of the C++ preprocessor and template-based metaprogramming system to implement domain-specific optimizations [31]. In Ace, we replace these brittle mini-languages with a general-purpose language and significantly expand the notion of active libraries by consideration of types as objects in the metalanguage. We thus call these types *active types*.

4.2 Structural Polymorphism

Generic functions represent a novel strategy for achieving *function polymorphism* – the ability to define 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 approach is related to structural polymorphism, however [21]. Structural types make explicit the requirements on a function, unlike generic functions.

Structural typing can be compared to the more *ad hoc* approach taken by dynamically-typed languages, sometimes called "duck typing". It is more flexible than the parametric polymorphism found in many functional languages and in languages like Java (which only allow polymorphic functions that are valid for *all* possible types), but is comparable to the C++ template system, as discussed previously.

4.3 Run-Time Indirection

Operator overloading [30] 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 type-level specification. However, type-level specification is a *compile-time* protocol focused on enabling specialized verification and implementation strategies, rather than simply enabling run-time indirection.

4.4 Term Rewriting Systems

Many languages and tools allow developers to rewrite expressions according to custom rules. These can broadly be classified as *term rewriting systems*. Macro systems, such as those characteristic of the LISP family of languages [23], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [27]), and external rewrite systems also exist for many languages. These differ in their direct exposure to syntax trees and their difficulties with propagating type information, since it is not directly encoded in the syntax. The AT&T mechanism is a type-based mechanism that avoids these issues.

4.5 Language Frameworks and Extensible Compilers

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 portability issues that using compiler extensions can introduce. It is difficult or impossible for these language-external approaches to achieve interoperability, as discussed above.

4.6 Extensible Languages

Extensible languages like SugarJ [12] afford some of the extensibility benefits of Ace, but are not targeted toward HPC. They have largely focused on syntactic extensibility, while Ace relies on a fixed syntax and emphasizes semantic extensibility. They also generally allow users to extend languages globally, which leads to conflicts when multiple extensions are used. AT&T does not admit such conflicts.

5. 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. Many parallel programming abstractions are defined in terms of, or benefit from, a type system that enforces a communication protocol, ensures the consistency of data and simplifies the dynamics of the run-time system (see Section **??** for examples). Because **verifiability** and **performance** are key criteria and static typing is a core technique, Ace is fundamentally statically-typed.

Approach	Examples	Library	Extensible Syntax	Extensible Type System	Composable	Alternative
Active Types	Ace	●	○	●	●	●
Desugaring	SugarJ [12], Sugar*	○	●	○	○	○
Rule Injection	Qi, Typed Racket [29]/Clojure, A?	1+2	○	●	○	○
Static Metaprogramming	Scala macros [3], OJ [28], OC++ [10], MorphJ [15]	○	○	○	●	○
Cross-Compilation	Delite [8]	●	○	○	○	●
EDSL Frameworks	?	●	●	●	○	○
Type-Specific Literals	Wyvern	○	●	○	●	○

Figure 1. Comparison to related approaches

It is legitimate to ask, however, why dynamically-typed languages are so widely-used in HPC. 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 [11]. Statically-typed languages, particularly those that HPC programmers are exposed to, are 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. The dynamically-typed languages used in HPC, 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 low-level memory and execution model of C but syntactic overhead comparable to a high-level scripting language?*

Rather than designing a new syntax, or modifying the syntax of C, 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 achieves both **ease-of-use** and **continuity**. Python's whitespace-delimited syntax is widely regarded as both concise and readable, and Python is amongst the most widely-adopted languages in computational science [25]. By directly adopting Python's syntax, Ace's syntax is immediately *familiar* and *acceptable* to a significant segment of the intended audience. Moreover, a key benefit of adopting it without modifications is that any tools that handle Python source code, including parsers, editors, style checkers and documentation generators, can be used on Ace code without modification. Researchers often dismiss the importance of syntax. By using a well-developed syntax, they no longer need to worry about the equally "trivial" task of implementing tools for it.

Professional end-users demand much from new languages and abstractions. In this paper, we began by generating a concrete set of design and adoption criteria that we hope will be of interest and utility to the research community. Based on these constraints, we designed a new language, Ace, making several pragmatic design decisions and introducing several novel techniques, including type propagation via generic functions, extensible type inference, active typechecking and translation and type-aware Python-Ace-OpenCL bindings to uniquely satisfy many of the criteria we discussed, particularly the three criteria that are typically overlooked in other languages. We validated the extension mechanism with a mature implementation of the entirety of the OpenCL type system, as well as outlined a number of other use cases. Finally, we demonstrated that this language was useful in practice, drastically improving performance without negatively impacting the high-level scientific workflow of a large-scale neurobiological circuit simulation project.

Ace has some limitations at the moment. Debugging is only supported on the generated code, so if code generation introduces significant complexity, this can be an issue. The OpenCL library we have implemented is a reasonably straightforward internalization of OpenCL itself, however, so debugging has not been a problem thusfar. We believe that active types can be useful to control debugging, and plan to explore this in the future. We will also further explore the use cases and case studies that we have described to validate the design we propose here. We hope that Ace will be developed further by the community to strengthen the foundations upon which new abstractions are implemented and deployed into the HPC professional end-user community.

We suggest three mutually-related design criteria that, unlike those in bold above, many languages and language-integrated abstractions have failed to adequately consider: **continuity**, **extensibility** and **interoperability**. These criteria encompass the intuitions that new abstractions will not be adopted in a vacuum, that programming systems must support change, and that interacting components of an application or workflow should be able to make use of different abstractions naturally and without the possibility of conflict arising at their interface boundaries.

We anticipate that coding guidelines mandating the use of abstractions that can be shown to have certain desirable properties will replace language-mandated enforcement opinions

Future work: integrate with something like scalad http://lampwww.epfl.ch/ hmiller/scala2013/resources/pdfs/paper8.pdf.

References

[1] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *Software, IEEE*, 25(4):29–36, 2008.

[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] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[5] J. Carver, R. Kendall, S. Squires, and D. Post. Software development environments for scientific and engineering software: A series of case studies. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 550 –559, may 2007. doi: 10.1109/ICSE.2007.77.

[6] 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.

[7] 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.

[8] 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. ISBN 978-1-4503-0119-0.

[9] 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.

[10] S. Chiba. A metaobject protocol for c++. *SIGPLAN Not.*, 30(10): 285–299, Oct. 1995. ISSN 0362-1340.

[11] 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.

[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] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

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

[15] 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.

[16] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[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. ISBN 978-3-642-17684-5. URL `http://dx.doi.org/10.1007/978-3-642-17685-2`.

[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] J. McCarthy. History of lisp. In *History of programming languages I*, pages 173–185. ACM, 1978.

[24] 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.

[25] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

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

[27] T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608:207–??, 1999. ISSN 0302-9743.

[28] 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.

[29] 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.

[30] 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.

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

[32] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proc. 1998 SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. URL `http://arxiv.org/abs/math/9810022`.

[33] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *ACM SIGPLAN Notices*, volume 45, pages 86–97. ACM, 2010.