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 [10, 26, 27]. As a result, many developers cannot use abstractions they might prefer because these abstractions are only available in languages they cannot adopt [25, 26]. This issue was perhaps most succinctly expressed by a participant in a recent study by Basili et al. [2] 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 [2].

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 5.

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. [1, 29]). 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* ([36], see Sec. 5). 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. 5).

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. 5, we compare Ace to related work on language extensibility and metaprogramming. We conclude in Sec. 6 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 [15]. Each thread executes the same kernel but has access to a unique index, called its *global ID*. Kernel code is written in a variant of C99 extended with some new primitive types and operators, which we will introduce as needed in our examples below.

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

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.

```
import ace, examples.clx as clx, ast, astx

_fn = ace.fn(clx.base, clx.opencl)

scale = _fn(ast.parse("""def scale(x, s):
    return x * s"""))

negate = _fn(astx.specialize(scale.ast, "negate",
    s=ast.parse("-1"))
```

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).

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.

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

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.
```

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 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. 3.2).

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 2.6 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, 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,

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(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 }
```

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 [15]). It is constructed by applying `clx.Ptr`, which is an Ace type constructor corresponding to pointer types, to a value representing the address space, `clx.global_`, and the type being pointed to. That type, `clx.float`, is in turn the Ace type corresponding to `float` in OpenCL (which, unlike C99, is always 32 bits). The `clx` library contains a full implementation of the OpenCL type system (including behaviors, like promotions, inherited from C99). Ace is *unopinionated* about issues like memory safety and the wisdom of such promotions. We will discuss how to implement, as libraries, abstractions that are higher-level than raw pointers in Sec. 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. 6). 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 dirertly 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 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). `clx.Cplx` truly extends the type system, it is not a type alias.

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 extracting a type from it. 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 1.6 operates by using the types of the provided arguments 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 5. In other words, types *propagate* into generic functions – we didn’t need to compile `negate` explicitly. This also explains the error printed on line 4.3-4.4: when this type was inadvertently assigned to the first argument `input`, the indexing operation on line 1.6 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.

2.6 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 [27, 30]. 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 [21]. The OpenCL language was designed for this workflow, exposing a retargetable compiler and data management routines as an API, called the *host API* [15]. The `pyopencl` library exposes this API and supports basic interoperability with `numpy`, a package for safely manipulating contiguously-allocated numeric arrays in Python [21].

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 6 shows an example of this workflow where the user chooses a compute device (line 6.3), constructs a `numpy` array (line 6.5), transfers it to the device (line 6.6), allocates an empty equal-sized buffer for the result of the computation (line 6.7), launches the generic kernel `map` from Listing 1 with these device arrays as well as the function `negate` from Listing 2 (line 6.9) choosing a number of threads equal to the number of elements in the input array (line 6.10), and transfers the result back into main memory to check that the device computed the expected result (line 6.12).

Listing 6 [`listing6.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()
```

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 possibly largely due to 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.double` by default, its shape is `(1024,1024)` and its order is row-major by default. The `pyopencl` library that our mechanism is built atop uses this information to automatically call the underlying OpenCL host API function for transferring byte arrays to the device without requiring the user to calculate the size. Our wrapper of `pyopencl` further retains this information in the Python wrappers around the device arrays, `d_input` and `d_output`. The Ace active type of such wrappers can be designated to be an instance of `clx.NPArray` parametermized by this metadata. This type knows how to typecheck and translate operations like indexing automatically (see Sec. 3.1).

This allows us to call the generic function `map` directly on these Python data structures (as well as the generic function `negate`) 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 also compiled for the device we selected using the OpenCL host API’s compiler infrastructure, and cached.

The same program written using the OpenCL C API directly is an order of magnitude longer and significantly more difficult to comprehend. OpenCL not support higher-order functions nor is there any way to write `map` in a type-generic manner. If we instead use the `pyopencl` library and apply the techniques described in [21], 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. 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 (e.g. the type for `numpy` arrays, which ensures that indexing respects ordering; see Sec. 4 for more interesting possibilities).

3. Extensibility

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 7 shows an example of such a class: the `clx.Cplx` class used in Listing 3, which implements the logic of complex numbers. The constructor takes as a parameter any numeric type in `clx` (line 7.2).

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, so all the semantics would be 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. `e1[e2]`), the compiler defers control over typechecking and translation to the active type of a designated subexpression (e.g. `e1`) determined by Ace's fixed *dispatch protocol*. Below are examples of these choices. Due to space constraints, we do not show the full protocol.

is it n-ary?

- 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. 5 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 [?] (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 7.7). 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

Listing 7 [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.translate(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.translate(node.left)
34    r_x = context.translate(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(right_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)
```

is raised otherwise (lines 7.8-7.10). We note that error messages are an important and sometimes overlooked facet of ease-of-use [23]. 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, rather than the semantics.

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 7.17-

7.29. Because arithmetic operations are meant to be symmetric, the dispatch protocol checks both subexpressions for support, favoring the left to ensure that the semantics remain deterministic. In either position, the implementation begins by recursively assigning a type to the other operand in the current context via the `context.type` method (line 7.24). If supported, it applies the C99 rules for arithmetic operations to determine the resulting type ([**?**], 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 7.50-7.57), raising a type error if the arguments cannot be promoted to a common type according to the rules of C99 or if two arguments have not been provided (an exercise for the reader: modify this method to also allow a single argument for when the imaginary part is 0).

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 targets*. 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 target.

Translation of terms of the form `c.ni` and `c.i` operates via `trans_Attribute`. Because we are implementing complex numbers as vector types in OpenCL, we generate an access to the left element, accessed via the attribute `s0`, to access the non-imaginary component and and `s1` to access the imaginary component. The target provides a programmatic API for generating code in the target language via its methods, as can be seen. Translation can assume typechecking succeeded, so the implementation assumes that if the attribute is not `ni`, it is `i`.

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).

3.2 Active Targets

3.2.1 Target Types

Every active type must translate to a unique *target type* – a type in the target language (dynamically typed target languages can be thought of as only having a single type, `dyn` [**?**]). Target types are constructed via methods of a provided active target, called `target` throughout this listing. Here, a complex number translates to a pair of values, each having the type that is the target type of `t`, the numeric type that was passed into the constructor (lines 7.7-7.8). The `tuple` method returns a representation of an actual type (not an active type) in the target language as the return value of the `tuple` method (e.g. 2-tuples of integers compile to the vector type `int2` in OpenCL). This design still allows the complex number type being defined here to supports any target for which tuples can be defined directly however (e.g. CUDA, Python, etc.).

3.3 Active Bases

nathan's work on representational consistency?

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 – the semantics of statements and expressions that do not have a clear "primary" subexpression.

For example, the type and translation of a numeric literal is determined by the active base. The active base `clx.base`, for example, assigns integer literals the type `clx.int32` while floating point literals have type `clx.double`, to be consistent with the semantics of OpenCL and C99. The `clx.base` object is an instance of `clx.Base`. This class is written so that this choice can be made differently if desired. For example, the base `clx.Base(flit_t=clx.float)` changes the base semantics so that floating point literals have type `clx.float`. This is useful because some OpenCL devices do not support double-precision floating point, or impose a significant performance penalty for its use. For applications where this higher precision is not necessary, this is convenient. Similarly, a base where integer literals are represented using true integers, rather than 32-bit integers, might be useful in applications where avoiding accidental overflows is more important than performance. True integers can be implemented as an active type (not shown).

Indeed, to even use `double` in OpenCL, an appropriate `#pragma` must be inserted. The base detects uses of `double` anywhere inside the function

3.3.1 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

4.1 Parallel Programming

4.1.1 OpenCL

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].

4.1.2 Partitioned Global Address Spaces

A number of recent languages designed for clusters use a partitioned global address space model, including UPC [5], Chapel [9] and others. These languages provide first-class support for accessing data transparently across a massively parallel cluster. Their type and emit code that hides the FFI from users, achieving **verifiability** and **ease-of-use**.

4.4 Domain-Specific Type Systems

4.4.1 Units of Measure

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 [19]. 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.

5. Related Work

5.1 Active Libraries

Libraries that contain compile-time logic have been called *active libraries* in prior proposals [36]. A number of projects, such as Blitz++, have taken advantage of the C++ preprocessor and template-based metaprogramming system to implement domain-specific optimizations [35]. 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*.

5.2 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 [22]. 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.

5.3 Run-Time Indirection

Operator overloading [34] and *metaobject dispatch* [20] 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.

5.4 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 [24], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [31]), 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.

5.5 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 [14]). 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.

5.6 Extensible Languages

Extensible languages like SugarJ [13] 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.

6. 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.

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**).

Approach	Examples	Library	Extensible Syntax	Extensible Type System	Extensions Compositional	Alternative Targets
Active Types	Ace	●	○	●	●	●
Desugaring	SugarJ [13], Sugar* [?]	○	●	○	○	○
Rule Injection	Qi [?], Typed Racket [33], A?	1+2	○	●	○	○
Static macros / Metaprogramming	Scala [4], MorphJ [16], OJ [32] Template Haskell [?]	○	○	○	●	○
Cross-Compilation	Delite [8]	●	○	○	○	●
EDSL Frameworks	?	●	●	●	○	○
Type-Specific Literals	Wyvern [?]	○	●	○	●	○

Figure 1. Comparison to related approaches

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 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 [12]. 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 [28]. 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.

fix python lang reference citation

References

[1] The python language reference, 2013.

[2] 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.

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

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

[5] 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.

[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.

[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] 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.

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

[12] 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.

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

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

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

[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 S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.