Ace: An Actively-Typed Compilation Environment for High-Performance Computing

Cyrus Omar, Nathan Fulton and Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
http://www.acelang.org/

***Abstract*—We introduce the Ace compilation environment and demonstrate its suitability as a foundational tool for both research and practice in high-performance computing. Ace consists of a statically-typed programming language with *user-extensible semantics* specified via a compile-time metalanguage, Python. Users specify new primitive types and their operations by equipping type definitions, which are first-class objects in the metalanguage, with methods that the compiler selectively invokes when type checking and translating expressions, a mechanism we call *active type-checking and translation (AT&T)*. We demonstrate the flexibility of this mechanism by implementing primitives from several widely-known languages, including the entirety of OpenCL, as libraries. Ace also supports more general forms of metaprogramming, and functions can be launched directly from Python with standard numeric data structures as arguments. Using these features, we designed a scientific simulation framework that allows users to modularly specify, compile and orchestrate the execution of parameterized families of scientific simulations on clusters of GPUs. This framework has been used to successfully conduct large-scale, high-performance neuroscience simulations, providing initial evidence that Ace is useful in practice today.**

***Keywords*-programming languages, metaprogramming, type systems, extensibility, heterogeneous computing**

I. Introduction

Computer-aided simulations and data analysis techniques have transformed science and engineering. Surveys show that scientists and engineers now spend up to 40% of their time writing software [1, 2]. Most of this software targets desktop hardware and about 20% of scientists also target either local clusters or supercomputers for more numerically-intensive computations [2]. To fully harness the power of these platforms, however, these *professional end-user developers* [3] must write parallel programs.

Professional end-users today generally use high-level scripting languages like MATLAB, Python, R or Perl for tasks that are not performance-sensitive, such as small-scale data analysis and plotting [4]. For portions of their analyses where these interpretted languages are too slow, they will typically call into code written in a statically-typed, low-level language like C or Fortran and use low-level parallel abstractions like pthreads or MPI [5, 6]. Unfortunately, these low-level languages and abstractions are notoriously error-prone and difficult to use, even for expert programmers.

The research community has proposed dozens of novel language features and parallel programming abstractions that aim to more equitably balance important concerns relating to performance, portability, verifiability and ease-of-use. Unfortunately, professional end-users are hesitant to adopt these, with many being generally skeptical regarding the practicality of new approaches that come from the research community. This viewpoint was perhaps most succinctly expressed by a participant in a recent study by Basili et al. [6], who stated "I hate MPI, I hate C++. [But] if I had to choose again, I would probably choose the same." Although this sentiment is easy to dismiss as paradoxical, we believe that it demands direct examination by researchers working to advance high-performance computing.

Many of the barriers to adoption of new abstractions stem from the fact that new abstractions are often developed and distributed together with a new programming language. Some prominent examples where this *language-oriented approach* has been followed include:

- UPC [**?**], CAF [**?**], X10 [**?**] and Chapel [**?**], built around first-class support for a partitioned global address space (PGAS) programming model;
- Erlang [**?**], Charm++ [**?**], Occam [**?**] and Axum [**?**], built around first-class support for message passing;
- Cilk [**?**] and NESL [**?**], built around first-class support for nested data parallelism; and
- Add something about automatic parallelization of functional programs.

There is evidence suggesting that developers would prefer first-class support for a parallel abstraction over a library-based solution, if all else is held constant [**?**]. Unfortunately, adopting a specialized research language comes with a host of ancillary challenges. Learning a new language can be time-consuming, new languages often lack well-developed libraries and tools, and it can be difficult to port large legacy codebases. Indeed, the most widely adopted parallel programming abstractions have been those that are distributed as libraries for an existing, widely-used language,

Listing 1 `hello.py`

```
1  from ace.OpenCL import OpenCL, printf
2
3  print "Hello, compile-time world!"
4
5  @OpenCL.fn
6  def main():
7      printf("Hello, run-time world!")
8  main = main.compile()
9
10 print "Goodbye, compile-time world!"
```

Listing 2 Compiling `hello.py` using `acecc`.

```
1  $ acecc hello.py
2  Hello, compile-time world.
3  Goodbye, compile-time world.
4  $ cat hello.cl
5  __kernel void main() {
6      printf("Hello, run-time world.");
7  }
```

such as MapReduce, Global Arrays and MPI. This *library-oriented approach* has proven more practical because it allows developers to progressively introduce parallelism into existing programs, leverage a familiar and well-developed general-purpose programming language when not directly invoking parallel primitives, and utilize different abstractions within a single program without encountering the friction inherent to language boundaries.

Ace aims to solve this problem by allowing developers to define and distribute new *first-class language constructs*, in a safely composable manner, directly within user libraries, via a mechanism we introduce called *active typechecking and translation (AT&T)*. In particular, AT&T is a mechanism that allows developers to extend the semantics of a language that has a fixed but flexible grammar. The Ace grammar is a large subset of Python's and we show that it is flexible enough that we are able to define all of the primitives of the C99 and OpenCL languages in a natural manner, aided by a form of type inference. These primitives can then be supplemented with higher-level user-defined primitives, examples of which we discuss in Section **??**. Notably, all of these primitives are imported via the normal library loading mechanism; they are not defined within domain-specific languages, nor as extensions for a particular compiler for a language.

Although the AT&T mechanism, a key feature of Ace, is designed primarily for use by researchers and domain experts, Ace is intended to be highly practical and ready for adoption by professional end-user developers today. It is based on the Python programming language, which enjoys broad adoption in this community. Indeed, the language itself is implemented entirely as a Python library, and many existing tools, such as editor modes, documentation tools and style checkers, can be used without modification. Ace functions are generated programmatically and can be called directly from Python, with numeric arrays as arguments, so developers can integrate Ace immediately within their existing workflows to define portions of code that may benefit from static compilation and support for parallelism. We demonstrate the practicality of Ace in Section **??** by describing an Ace-based framework that supports specifying and orchestrating families of large-scale neurobiological circuit simulations on clusters of GPUs.

II. Ace for Professional End-Users

A. Example 1: Hello, World!

To demonstrate the structure of programs and libraries in Ace, let us begin in Listing 1 with a variant of the canonical "Hello, World!" example written using the OpenCL module distributed with Ace. We note that this module, which we use throughout the paper in examples, is simply a library like any other. The core of Ace gives no special treatment to it; it is distributed with Ace for convenience.

Perhaps the most immediately apparent departure from the standard "Hello, World!" example comes on lines 3 and 10 of Listing 1, which contain `print` statements that are executed at *compile-time*. Indeed, Ace programs are Python scripts at the top-level, rather than a list of declarations as in most conventional languages. In other words, Python is the *compile-time metalanguage* of Ace; Ace programs are embedded within Python. A consequence of this choice is that Ace can leverage Python's well-developed package system and distribution infrastructure directly (Line 1).

The `main` function defined on Lines 5-7 contains a single call to the `printf` primitive, imported from the `ace.OpenCL` module on Line 1. The function is introduced using the `@OpenCL.fn` decorator, which indicates that it is a statically-typed Ace function containing run-time logic targeting the `OpenCL` backend, also imported on Line 1. Without this decorator, the function would simply be a conventional Python function that could be called only at compile-time (doing so would fail here: `printf` is not a Python primitive.)

On Line 8, this *generic function* defined on Lines 5-7, is compiled to produce a *concrete function* that we also identify as `main` (overwriting the previous definition). The distinction between generic and concrete functions will be explained in our next example. For now, we note that only concrete functions are exported when running the Ace compiler, called `acecc`, shown operating at the shell in Listing 2. The `acecc` compiler operates as follows:

1) Executes the provided Python module (`hello.py`)
2) Produces source code associated with any concrete functions (functions produced using the `compile` method on Line 8 of Listing 1) that are in the top-level environment. Each backend may produce one or more files (here, `hello.cl` is produced).

Using the `acecc` executable is simply a convenience; the generated code is also available as an attribute called of

Listing 3 [listing3.py] A generic data-parallel higher-order map function written using OpenCL user module.

```
1  from ace.OpenCL import OpenCL, get_global_id
2
3  @OpenCL.fn
4  def map(input, output, f):
5      gid = get_global_id(0)
6      output[gid] = f(input[gid])
```

Listing 4 [listing4.py] The generic map function compiled to map the add5 function over two types of input.

```
1  from listing3 import map
2  from ace.OpenCL import global_ptr, double, int
3
4  @OpenCL.fn
5  def add5(x):
6      return x + 5
7
8  A = global_ptr(double); B = global_ptr(int)
9  map_add5_dbl = map.compile(A, A, add5.ace_type)
10 map_add5_int = map.compile(B, B, add5.ace_type)
```

the concrete function immediately after it has been defined, accessed as `main.code`. We will show in Section **??** that, for backend languages with Python bindings, such as OpenCL and C, generic functions can be executed directly, without the intermediate compilation step, if desired.

B. Example 2: Higher-Order Map for OpenCL

The "Hello, World!" example demonstrates the basic structure of Ace programs, but it does not require working with types. Listing 3 shows an imperative, data-parallel map primitive written using the OpenCL library introduced above. In OpenCL, users can define functions, called *kernels*, that execute across thousands of threads. Each kernel has access to a unique index, called its *global id*, which can be used by the programmer to ensure that each thread operates on different parts of the input data (Line 5). The `map` kernel defined in Listing 3 applies a transfer function, `f`, to the element of the input array, `input`, corresponding to its global id. It writes the result of this call into the corresponding location in the provided output array, `output`.

As above, `map` is a *generic function* (specifically, an instance of the class `ace.GenericFn`). This means that it's arguments have not been assigned types. Indeed, the functionality given by the `map` definition is applicable to many combinations of types for `input` and `output` and functions, `f`. In this sense, `map` is actually a family of functions defined for all types assignments for `input`, `output` and `f` for which the operations in the function's body are well-defined.

Running `acecc listing3.py` would produce no output, however. To create a *concrete function* (an instance of the class `ace.ConcreteFn`) that can be emitted by the compiler, types must be assigned to each of the arguments of any

Listing 5 [listing4.cl] The OpenCL code generated by running `acecc listing4.py`.

```
1  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2
3  double add5(double x) {
4      return x + 5;
5  }
6
7  __kernel void map_add5_dbl(
8    __global double* input,
9    __global double* output)
10 {
11     size_t gid;
12     gid = get_global_id(0);
13     output[gid] = add5(input[gid]);
14 }
15
16 int add5__1(int x) {
17     return x + 5;
18 }
19
20 __kernel void map_add5_int(
21   __global int* input,
22   __global int* output)
23 {
24     size_t = gid;
25     gid = get_global_id(0);
26     output[gid] = add5__1(input[gid]);
27 }
```

externally callable functions. Listing 4 shows how to use the `compile` method to specialize `map` in two different ways to apply the `add5` function, defined on Lines 4-6, to arrays that reside in global memory (OpenCL has a notion of four different memory spaces). Line 9 produces a version specialized for arrays of `double`s and Line 10 produces a version for arrays of `int`s. The output of compilation is shown in Figure 5.

C. Types as Metalanguage Objects

The types of the arguments of the functions being compiled on Lines 4.9 and 4.10, if written directly in OpenCL as in Listing 5, are `__global double*` and `__global int*`, respectively. The corresponding types in Ace are `global_ptr(double)` and `global_ptr(int)`, abbreviated for convenience as `A` and `B` respectively on Line 8 (note that here, standard variable assignment in the metalanguage is serving the role that a `typedef` declaration would fill in a C-like language.)

Types in Ace are values in the metalanguage, Python. More specifically, types are *object instances* of user-defined classes that inherit from the Ace-provided `ace.Type` class. For example, `global_ptr(double)` is an instance of `ace.OpenCL.GlobalPtrType` instantiated with the target type, `double`, as a constructor argument. The types `double` and `int` (imported from `ace.OpenCL` on Line 4.2) are instances of `ace.OpenCL.FloatType` and `ace.OpenCL.IntegerType`, respectively. As we will

describe further in Section **??**, this notion of types as metalanguage objects is a key to Ace's flexibility.

D. Type Propagation and Higher-Order Functions

The type assigned to the third argument, `f`, on Lines 9 and 10, is given as `add5.ace_type`. The `ace_type` attribute of a generic function is an instance of `ace.GenericFnType`, the type of Ace generic functions, that corresponds to that particular function, `add5` in this case. Generic functions are compiled automatically when they are called. 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, and similarly with the `int` type when compiling `map_add5_int`. This mechanism is called *type propagation*. In other words, we did not need to run `add5.compile(double)` before compiling `map_add5_dbl` (although doing so would also be valid). Only functions that are never called in the process of compiling other functions in a concrete module need type information explicitly provided, whereas `add5` is a function that is only called within `map`.

We note that 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 the same as the `ace.GenericFnType` for a superficially similar function, such as `add6` (defined as you would expect.) To put it in type theoretic terms, `ace.GenericFnType`s are *uniquely inhabited* by a single generic function (and similarly for `ace.ConcreteFnType`s), and thus it is impractical to use them as full first-class values (i.e. they cannot put into a run-time array.) In practice, this limitation is rarely a problem, particularly in parallel programming where such forms of specialization are already common in order to avoid the potential performance penalty associated with a function pointer dereference per call.

We note that fully-featured first-class functions can be implemented when targeting a backend that supports them, such as C99, or by using a jump table implementation, but we do not discuss this further due to lack of space.

E. Type Inference

Within the `map` function in Listing 3, on Line 5, the variable `gid` is initialized with the result of calling the `get_global_id` primitive (OpenCL supports multidimensional global ids, so the argument specifies which dimension to use.) Note that a type for the `gid` variable is not listed explicitly. This is because Ace supports a form of within-function *type inference*. In this case, `gid` will 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.) This can be seen on Lines 11 and 24 in Listing 5.

Listing 6 A function demonstrating whole-function type inference when multiple values with differing types are assigned to a single variable.

```
1  from ace.OpenCL import OpenCL, int, double, long
2
3  @OpenCL.fn
4  def threshold_scale(x, scale):
5      if x <= 0:
6          y = 0
7      else:
8          y = scale * x
9      return y
10 f = f.compile(int, double)
11 g = g.compile(int, long)
12 assert f.return_type == double
13 assert g.return_type == long
```

Inference is not restricted to single assignments, as in the `map` example. Multiple assignments to the same variable with values of differing types and multiple return statements can be unified, such that the variable or return type is given a common supertype. For example, in Listing 6 the variable `y` in the first branch of the conditional is assigned the `int` literal `0`. However, in the second branch of the loop, its type depends on the types of `x` and `scale`. We show two examples where this is the case on Lines 10 and 11. However, type inference correctly unifies these two types according to the C99 rules governing numeric types (which can be defined by the user, as we show in Section **??**). This example would also work correctly if the assignments to `y` were replaced with `return` statements were placed directly in the branches of the conditional.

F. Attribute and Extension Inference

In addition to type annotations, OpenCL introduces additional syntactic burdens. Developers must annotate functions that meet the requirements to be called from a host with the `__kernel` attribute. The `Ace.OpenCL.OpenCL` backend is able to check these requirements and add this annotation automatically. Additionally, several types (notably, `double`) and specialized functions require that an OpenCL extension be enabled with a `#pragma`. The OpenCL backend can also automatically infer many of these cases as well and add the appropriate `#pragma` declaration, consistent with the principle of conciseness followed for the Ace design as a whole. An example of this can be seen in Listing 5.

G. Metaprogramming in Ace

Metaprogramming refers to the practice of writing programs that manipulate other programs. There are a number of use cases for this technique, including domain-specific optimizations and code generation for programs with a repetitive structure that cannot easily be captured using

Listing 7 Metaprogramming with Ace, showing how to construct generic functions from strings and abstract syntax trees, and how to manipulate syntax trees at compile time.

```
1  from ace.OpenCL import OpenCL
2  import ast, ace.astx as astx
3
4  plus = OpenCL.fn.from_string("""
5  def plus(a, b):
6      return a + b
7  """)
8
9  add5_ast = astx.specialize(plus.ast, b=5)
10 add5 = OpenCL.fn.from_ast(add5_ast)
```

Listing 8 A full OpenCL program using the `Ace.OpenCL` Python bindings, including data transfer and invocation of a generic function, `map`, as a kernel without compiling it first.

```
1  import numpy as np
2  import ace.OpenCL.bindings as cl
3  from listing3 import map
4  from listing4 import add5 # or Listing 7
5
6  ctx = cl.Context.for_device(0, 0)
7  d_in = ctx.to_device(np.ones((1024,)))
8  d_out = ctx.alloc(like=d_in)
9  map(d_in, d_out, add5,
10     global_size=d_in.shape, local_size=(128,))
11 out = ctx.from_device(d_out)
```

available abstractions. OpenCL itself in particular relies on code generation as a fundamental mechanism, partially justifying its lack of support for higher-order programming [**?**]. Ace supports programmatic compilation and higher-order constructs, as described above, and language extension, which we describe below, so several use cases for this feature have been eliminated. However, we demonstrate cases where this form of metaprogramming is useful in the case study described in Section **??**.

An Ace function can be constructed from a string containing its source using the `from_source` variant of `fn`, as demonstrated in Listing 7 on Lines 4-7. It can also be constructed directly from a Python abstract syntax tree (AST), available via the Python standard `ast` package, using the `from_ast` variant of `fn`, demonstrated on Line 10. The AST here is generated programmatically by transforming the syntax tree of the `plus` function such that the argument `b` becomes a constant, `5`. This transformation, called `specialize`, as well as several others are distributed with Ace in the `ace.astx` module for convenience.

H. Direct Invocation from Python

As discussed in the Introduction, a common workflow for professional end-users involves the use of a high-level scripting language for overall workflow orchestration and small-scale data analysis and visualization, paired with a low-level language for performance-critical sections. Python is already widely used by professional end-users as the high-level scripting language. It has support for executing code written in low-level languages in several ways. Developers can call into native libraries using its foreign function interface (FFI) or a library like `pycuda` for code compiled with CUDA, a proprietary language similar to OpenCL targeting nVidia GPUs [**?**].

The OpenCL language is designed specifically for this workflow, in that it exposes the compiler directly an API, rather than as a separate executable. Developers generate OpenCL source code as strings, compile it programmatically, then execute it using the run-time APIs that OpenCL also provides. Python bindings to the OpenCL APIs have been developed in a module called `pyopencl` [**?**].

Ace supports a similar workflow as an alternative to using `acecc` as described above. Both generic functions and concrete functions written for a backend that supports the direct execution interface (thus far, `OpenCL` and `C99`) can be called like regular Python functions. An example of this for the generic `map` function defined in Listing 3 is shown in Listing 8, with the call itself on Lines 9-10. The first two arguments to `map` are OpenCL buffers, generated using a simplified wrapper to the `pyopencl` APIs on Lines 7-8. This wrapper associated type information with each buffer, and this information is used to implicitly compile `map` as appropriate. That is, explicit calls to the `compile` method, as we have been showing, are unnecessary when using this method of invocation. The final two keyword arguments on Line 10 are parameters that OpenCL requires for execution that determine the number of threads (called the *global size*) and the grouping of these threads (the *local size*).

By way of comparison, the same program written using the OpenCL C API directly is an orders of magnitude larger and correspondingly more complex. A full implementation of the logic of `map` written using the `pyopencl` bindings and metaprogramming techniques as described in [**?**] is twice as large and significantly 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 – the reader is referred to the language documentation for additional details.

III. Ace for Researchers

Thus far, we have been largely discussing the OpenCL module in our examples. However, Ace gives no preferential treatment to this module; it is implemented entirely using the user-facing mechanisms described in this section.

Most programming languages are *monolithic* – the set of available primitives is determined by the language designers, and users must combine these primitives to produce desired behaviors. Although a number of very flexible primitives have been developed (e.g. object systems), these have proven

insufficient in specialized domains where developers and researchers need fine control over how certain operations are type checked and translated. High-performance computing is an example of such a field, since both correctness and performance have been difficult to achieve in general, and are topics of active research. As discussed in the Introduction, the proliferation of parallel programming languages, rather than library-based abstractions, indicates that researchers often find the abstractions in existing languages insufficient for their needs.

In view of these issues, Ace is fundamentally *extensible*, rather than monolithic. Users can introduce new primitive types and operations, fully controlling how they are type checked and translated. The target of translation can also be specified modularly by the user. Because Ace libraries can contain compile-time logic, written in the metalanguage as described in the previous section, these primitive definitions can be distributed within normal modules.

A. Active Typechecking and Translation (AT&T)

When the compiler encounters an expression, it must first verify that it typechecks, then translate it to produce code in a target language. Rather than containing fixed logic for this, however, the Ace compiler defers this responsibility to the *type* of a subexpression whenever possible, according to a fixed *dispatch protocol* for each syntactic form. 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.

- Responsibility over a **unary operation** like `-x` is handed 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, with a different method call[1].
- Responsibility over **attribute access** (`obj.attr`) and **subscript access**, (`obj[idx]`) is handed to the type assigned to `obj`.

1) Active Typechecking: During the typechecking phase, the type of the primary operand, as determined by the dispatch protocol, is responsible for assigning a type to the expression as a whole. Let us consider the `map` function from Listing 3 once again. When it is compiled on Line 9 of Listing 4, its first two argument types are given as `global_ptr(double)`. As described in Section **??**, this type, abbreviated `A`, is an instance of `ace.OpenCL.GlobalPtrType` which inherits from `ace.OpenCL.PtrType` and ultimately from `ace.Type`. So when the compiler encounters the expression `input[gid]` on Line 6, it follows the dispatch protocol just described and assigns responsibility over typechecking it to `A`. This is done by calling the `resolve_`X method of the responsible type, where X is the syntactic form of the expression. In this case, the expression is of the `Subscript` form, so the compiler calls `A.resolve_Subscript`.

[1]Note that this operates similarly to the Python run-time operator overloading protocol; see Related Work.

Listing 9 A portion of the implementation of OpenCL pointer types implementing subscripting logic using the Ace extension mechanism.

```
1  import ace, ace.astx as astx
2
3  class PtrType(ace.Type):
4    def __init__(self, T, addr_space):
5      self.target_type = T
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 astx.copy_node(node,
19       value=value, slice=slice,
20       code=value.code + '[' + slice.code + ']')
21
22   # ...
23
24 class GlobalPtrType(PtrType):
25   def __init__(self, T):
26     PtrType.__init__(self, T, '__global')
```

The relevant portion of `ace.OpenCL.GlobalPtrType` is shown in Listing 9. The `verify_Subscript` method on Line 8 receives a context and the syntax tree of the node itself as input. The context contains information about other variables in scope, as well as other potentially relevant information, and also contains a method, `resolve_type`, that can be used recursively resolve the types of subexpressions. On Line 9, this method is used to resolve the type of the slice subexpression, `gid`, which is the machine-dependent integer type `size_t` as discussed in Section II.E. On Line 10, it confirms that this type is an instance of an integer type. Thus, it assigns the whole expression, `input[gid]`, the target type of the pointer, `double`. Had a user attempted to index `input` using a non-integer value, the method would take the other branch of the conditional and raise a type error with a relevant user-defined error message on Line 13.

2) Active Translation: Once typechecking a method is complete, the compiler must subsequently translate each Ace source expression into an expression in the target language, OpenCL in the examples thus far. It does so by again applying the dispatch protocol and calling 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. In this case, 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 effects, such as new top-level declarations (not shown.)

B. Active Backends

Thus far, we have discussed using OpenCL as a backend with Ace. The OpenCL extension is the most mature as of this writing. However, Ace supports the introduction of new backends in a manner similar to the introduction of new types, by extending the `clq.Backend` base class. Backends are provided as the first argument to the `@clq.fn` decorator, as can be seen in Figure 3. 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 basic statements like `while`.

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.) Backends not based on the C family are also possible, but we leave such developments for future work.

C. Use Cases

The development of the full OpenCL language using only the extension mechanisms described above provides evidence of the power of this approach. Nothing about the core language was designed specifically for OpenCL. However, to be truly useful, as described in Sections **??** and **??**, the language must be able to support a wide array of primitive abstractions. We briefly describe a number of other abstractions that may be possible using this mechanism. Many of these are currently available either via inconvenient libraries or in standalone languages. With the Ace extension mechanism, we hope to achieve robust, natural implementations of many of these mechanisms within the same language.

Partitioned Global Address Spaces: A number of recent languages in high-performance computing have been centered around a partitioned global address space model, including UPC, Chapel, X10 and others. These languages provide first-class support for accessing data transparently across a massively parallel cluster, which is verbose and poorly supported by standard C. The extension mechanism of Ace allows inelegant library-based approaches such as the Global Arrays library to be hidden behind natural wrappers that can use compile-time information to optimize performance and verify correctness. We have developed a prototype of this approach using the C backend and hope to expand upon it in future work.

Other Parallel Abstractions: A number of other parallel abstractions, some of which are listed in **??**, also suffer from inelegant C-based implementations that spurred the creation of standalone languages. 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]. The extension mechanism is designed to enable library-based solutions that operate as first-class language-based solutions, barring the need for particularly exotic syntactic extensions.

Domain-Specific Type Systems: Ace is a statically-typed language, so a number of domain-specific abstractions that promise to improve verifiability using types, as discussed in Section **??**, can be implemented using the extension mechanism. We hope that this will allow advances from the functional programming community to make their way into the professional end-user community more quickly, particularly those focused on scientific domains (e.g. [8]).

Specialized Optimizations: In many cases, code optimization requires domain-specific knowledge or sophisticated, parametrizable heuristics. Existing compilers make implementing and distribution such optimizations difficult. With active libraries in Ace, optimizations can be distributed directly with the libraries that they work with. For instance, we have implemented substantial portions of the NVidia GPU-specific optimizations described in [9] as a library that uses the extension mechanism to track affine transformations of the thread index used to access arrays, in order to construct a summary of the memory access patterns of the kernel, which can be used both for single-kernel optimization (as in [9]) and for future research on cross-kernel fusion and other optimizations.

Instrumentation: Several sophisticated feedback-directed optimizations and adaptive run-time protocols require instrumenting code in other ways. The extension mechanism enables granular instrumentation based on the 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.

IV. Case Study: Neurobiological Circuit Simulation

An important criteria that practitioners use to evaluate a language or abstraction, as discussed in Section **??**, is whether significant case studies have been conducted with it. In this section, we briefly (due to space limitations) discuss an application of the Ace OpenCL library, Python host bindings and code generation features for developing a modular, high-performance scientific simulation library used

to simulate thousands of parallel realizations of a spiking neurobiological circuit on a GPU.

A. Background

A neural circuit can be modeled as a network of coupled differential equations, where each node corresponds to a single neuron. Each neuron is modeled using one or more ordinary differential equations. These equations capture the dynamics of physically important quantities like the cell's membrane potential or the conductance across various kinds of ion channels and can take many forms [**?**]. Single simulations can contain from hundreds to tens of millions of neurons each, depending on the specific problem being studied. In some cases, such as when studying the effects of noise on network dynamics or to sweep a parameter space, hundreds or thousands of realizations must be generated. In these cases, care must be taken to only probe the simulation for relevant data and process portions of it as the simulation progresses, because the amount of data generated is often too large to store in its entirety for later analysis.

The research group we discuss here (of which the first author was a member) was studying a problem that required running up to 1,000 realizations of a network of between 4,000 and 10,000 neurons each. An initial solution to this problem used the Brian framework, written in Python, to conduct these simulations on a CPU. Brian was selected because it allowed the structure of the simulation to be specified in modular and straightforward manner. This solution required between 60 and 70 minutes to conduct the simulations and up to 8 hours to analyze the data each time a parameter of the simulation was modified.

Unsatisfied with the performance of this approach, the group developed an accelerated variant of the simulation using C++ and CUDA. Although this produced significant speedups, reducing the time for a simulation by a factor of 40 and the runtime of the slowest analyses by a factor of 200, the overall workflow was also significantly disrupted. In order to support the many variants of models, parameter sets, and probing protocols, C preprocessor flags were necessary to selectively include or exclude code snippets. This quickly led to an incomprehensible and difficult to maintain file structure. Moreover, much of the simpler data analysis and visualization was conducted using Python, so marshalling the relevant data between processes also became an issue.

B. The cl.egans Simulation Library

In order to eliminate these issues while retaining the performance profile of the GPU-accelerated code, the project was ported to Ace. Rather than using preprocessor directives to control the code contained in the final GPU kernels used to execute the simulation and data analyses, the group was able to develop a more modular library called cl.egans[2] based on the language's compile-time code generation mechanism and Python and OpenCL bindings.

cl.egans leverages Python's object-oriented features to enable modular, hierarchical simulation specifications. For example, Figure 10 shows an example where a neuron model (`ReducedLIF`) is added to the root of the simulation, a synapse model (`ExponentialSynapse`) is then added to it, and its conductance is probed in the same way, by adding a probe model as a child of the synapse model. If interleaved analysis needed to be conducted as well, it would be specified in the same way.

Implementations of these classes do not evaluate the simulation logic directly, but rather contain methods that generate Ace source code for insertion at various points, called *hooks*, in the final simulation kernel. The hook that code is inserted into is determined by the method name, and code can be inserted into any hook defined anywhere upstream in the simulation tree. New hooks can also be defined in these methods and these become available for use by child nodes. Figure 11 shows an example of a class that inserts code in the `model_code` hook and defines several new hooks. This protocol is closely related to the notion of *frame-oriented programming*. Although highly modular, this strategy avoids the performance penalties associated with standard object-oriented methodologies via code generation.

Compared to a similar protocol targeting OpenCL directly, the required code generation logic is significantly simpler because it enables classes like `StateVariable` to be written generically for all types of state variables, without carrying extra parameters and *ad hoc* logic to extract and compute the result types of generated expressions. Moreover, because types are first-class objects in the metalanguage, they can be examined during the memory allocation step to enable features like fully-automatic parallelization of multiple realizations across one or more devices, a major feature of cl.egans that competing frameworks cannot easily offer.

Once the kernel has been generated and memory has been allocated, the simulation can be executed directly from Python using the bindings described in Section II-H. The results of this simulation are immediately available to the Python code following the simulation and can be visualized and further analyzed using standard tools. Once the computations are complete, the Python garbage collector is able to handle deallocation of GPU memory automatically (a feature of the underlying `pyopencl` library [10].)

Using this Ace-based framework, the benefits of the Brian-based workflow were recovered without the corresponding decrease in performance relative to the previous CUDA-based solution, leading ultimately to a satisfying solution for the group conducting this research.

[2]...after *c. elegans*, a model organism in neuroscience

Listing 10 An example of a nested simulation tree, showing that specifying a simulation is both simple and modular.

```
# Create the root node of the simulation
sim = Simulation(ctx, n_timesteps=10000)
neurons = ReducedLIF(sim, count=N, tau=20.0)
e_synapse = ExponentialSynapse(neurons, 'ge',
    tau=5.0, reversal=60.0)
probe = StateVariableProbeCopyback(e_synapse)
```

Listing 11 An example of a hook that inserts code and also inserts new, nested hooks for downstream simulation nodes below that.

```
class SpikingModel(Model):
  """Base class for spiking neuron models."""
  def in_model_code(self, g):
    "idx_state = idx_model + count*" << g
    "(realization_n - realization_start)" << g
    self.insert_hook("read_incoming", g)
    self.insert_hook("read_state", g)
    self.insert_hook("calculate_inputs", g)
    self.insert_hook("state_calculations", g)
    self.insert_hook("spike_processing", g)
  # ...
```

V. Related Work

A. Active Libraries in Ace

Libraries that have such capabilities has been called *active libraries* in prior proposals [11]. A number of projects, such as Blitz++, have taken advantage of the C++ preprocessor and template-based metaprogramming system to implement domain-specific optimizations. In Ace, we replace these brittle mini-languages with a general-purpose language. This allows for several interesting uses that we discuss in the following sections.

1) Structural Polymorphism: In Section **??**, we discussed several strategies for achieving *polymorphism* – the ability to create functions and data structures that operate over more than a single type. In Ace, all functions are implicitly polymorphic and can be called with arguments of *any type that supports the operations used by the function*. For example, in Figure 3, `in` can be any type that supports indexing by a variable of type `size_t` to produce a value of a type that can be passed into `fn`, which must then produce a value consistent with indexing into `out`. OpenCL pointer types are consistent with these constraints, for example. Although powerful, this also demonstrates a caveat of this approach – that it is more difficult to give a function a concise signature, because arguments are constrained by capability, rather than to a single type [12].

Structural typing can be compared to the approach taken by dynamically-typed languages that rely on "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 of comparable strength to the template system found in C++. It can be helpful to think of each function as being preceded by an implicit template header that assigns each argument its own unique type parameter. At function call sites, these parameters are implicitly specialized with the types of the provided arguments. This choice is again motivated by the criteria of conciseness given in Section **??**.

B. Type-Level Computation

System XX with simple case analysis provides the basis of type-level computation in Haskell (where type-level functions are called type families [13]). Ur uses type-level records and names to support typesafe metaprogramming, with applications to web programming [14]. Ωmega adds algebraic data types at the type-level, using these to increase the expressive power of algebraic data types at the expression level [15]. Dependently-typed languages blur the traditional phase separation between types and expressions, so type-level computation is often implicitly used (though not always in its most general form, e.g. Deputy [16], ATS [17].)

C. Run-Time Indirection

Operator overloading [18] 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.

D. 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 [20], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [21]), and external rewrite systems also exist for many languages. These facilities differ from type-level specification in one or more of the following ways:

1) In type-level specification, the type of a value is determined separately from its representation; in fact, the same representation may be generated by multiple types.
2) We draw a distinction between the metalanguage, used to specify types and compile-time logic, the source grammar, used to describe run-time behavior, and the internal language, used to implement this behavior. Term rewriting systems generally do not draw this distinction. By doing so, each component language

can be structured and constrained as appropriate for its distinct role, as we show.

3) Many common macro systems and metaprogramming facilities operate at run-time. Compilers for some forms of LISP employ aggressive compile-time specialization techniques to attempt to minimize this overhead. Static and staged term-rewriting systems also exist (e.g. OpenJava [22], Template Haskell [23], MetaML [21] and others).

E. Language Frameworks

When the mechanisms available in an existing language prove insufficient, researchers and domain experts must 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 [24]).

A major barrier to adoption is the fact that interoperability is intrinsically problematic. Even languages which target a common platform, such as the Java Virtual Machine, can only interact using its limited set of primitives. Specialized typing rules are not checked at language boundaries, performance often suffers, and the syntax can be unnatural, particularly for languages which differ significantly from the platform's native language (e.g. Java).

Instead of focusing on defining standalone languages, type-level specification gives greater responsibility in a granular manner to libraries. In this way, a range of constructs can coexist within the same program and, assuming that it can be shown by some method that various constructs are safely composable, be mixed and matched. The main limitation is that the protocol requires defining a fixed source grammar, whereas a specialized language has considerable flexibility in that regard. Nevertheless, as Ace shows, a simple grammar can be used quite flexibly.

F. Extensible Compilers

An alternative methodology is to implement language features granularly as compiler extensions. As discussed in Section 1, existing designs suffer from the same problems related to composability, modularity, safety and security as extensible languages, while also adding the issue of language fragmentation.

Type-level specification can in fact be implemented within a compiler, rather than provided as a core language feature. This would resolve some of the issues, as described in this paper. However, by leveraging type-level computation to integrate the protocol directly into the language, we benefit from common module systems and other shared infrastructure. We also avoid the fragmentation issue.

G. Specification Languages

Several *specification languages* (or *logical frameworks*) based on these theoretical formulations exist, including the OBJ family of languages (e.g. CafeOBJ [25]). They provide support for verifying a program against a language specification, and can automatically execute these programs as well in some cases. The language itself specifies which verification and execution strategies are used.

Type-determined compilation takes a more concrete approach to the problem, focusing on combining *implementations* of different logics, rather than simply their specifications. In other words, it focuses on combining *type checkers* and *implementation strategies* rather than more abstract representations of a language's type system and dynamic semantics. In Section 4, we outlined a preliminary approach based on proof assistant available for the type-level language to unify these approaches, and we hope to continue this line of research in future work.

VI. Conclusion

In addition to the novel architecture of Ace as a whole, we note several individually novel features introduced here:

- The AT&T mechanism, which is a generalization of the concept of active libraries [11] where types are metalanguage objects.
- The method Ace uses to eliminate the need for type annotations in most cases, which combines a form of type inference with type propagation.
- The method Ace uses check correctness of generated code by checking representational consistency constraints associated with types, detailed in Section **??**.
- The type-aware simulation orchestration techniques used in the `cl_egans` library, described in Section **??**.

Readers familiar with the Python programming language will recognize the style of syntax used in Figure 3. In fact, Ace uses the Python grammar and parsing facilities directly. Several factors motivated this design decision. First, Python's syntax is widely credited as being particularly simple and readable, due to its use of significant whitespace and conventional mathematical notation. Python is one of the most widely-used languages in scientific computing, so its syntax is already familiar to much of the field. And significantly, a large ecosystem of tools already exist that work with Python files, such as code editors, syntax highlighters, style checkers and documentation generators. These can be used without modification to work with Ace files. Therefore, by re-using an existing, widely-used grammar, we are able to satisfy many of the design criteria described in Section **??** and the adoption criteria described in Section **??** without significant development effort.

Professional end-users demand much from new languages and abstractions. In this paper, we began by generating a concrete, detailed set of design and adoption criteria that we hope will be of broad interest and utility to the research community. Based on these constraints, we designed a new language, Ace, making several pragmatic design decisions and utilizing advanced techniques, including type inference,

structural typing, compile-time metaprogramming and active libraries, to uniquely satisfy many of the criteria we discuss, particularly those related to extensibility. We validated the extension mechanism with a mature implementation of the entirety of the OpenCL type system, as well as preliminary implementations of some other features. 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. Going forward, we hope that Ace (or simply the key techniques it proposes, by some other vehicle) will be developed further by the community to strengthen the foundations upon which new abstractions are implemented and deployed into professional end-user development communities.

VII. Availability

Ace is available under the LGPL license and is developed openly and collaboratively using the popular Github platform at https://github.com/cyrus-/ace. Documentation, examples and other learning materials will be available at http://acelang.org/. (by the time of the conference)

VIII. Acknowledgments

CO was funded by the DOE Computational Science Graduate Fellowship under grant number DE-FG02-97ER25308. NF was funded by ...

References

[1] J. Howison and J. Herbsleb, "Scientific software production: incentives and collaboration," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 2011, pp. 513–522.

[2] J. Hannay, C. MacLeod, J. Singer, H. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. IEEE Computer Society, 2009, pp. 1–8.

[3] J. Segal, "Some problems of professional end user developers," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, 2007, pp. 111–118.

[4] 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*. ACM, 2010, p. 12.

[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*, may 2007, pp. 550 –559.

[6] 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*, vol. 25, no. 4, pp. 29–36, 2008.

[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*. ACM, 2010, p. 9.

[8] A. Kennedy, "Types for units-of-measure: Theory and practice," in *CEFP*, ser. Lecture Notes in Computer Science, Z. Horváth, R. Plasmeijer, and V. Zsók, Eds., vol. 6299. Springer, 2009, pp. 268–305. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17685-2

[9] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *ACM SIGPLAN Notices*, vol. 45, no. 6. ACM, 2010, pp. 86–97.

[10] 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.

[11] 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. [Online]. Available: http://arxiv.org/abs/math/9810022

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

[13] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow, "Associated types with class," *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 1–13, Jan. 2005.

[14] A. Chlipala, "Ur: statically-typed metaprogramming with type-level record computation," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, B. G. Zorn and A. Aiken, Eds. ACM, 2010, pp. 122–133. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806612

[15] T. Sheard and N. Linger, "Programming in omega," in *CEFP*, ser. Lecture Notes in Computer Science, Z. Horváth, R. Plasmeijer, A. Soós, and V. Zsók, Eds., vol. 5161. Springer, 2007, pp. 158–227.

[16] C. Chen and H. Xi, "Combining programming with theorem proving," in *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, O. Danvy and B. C. Pierce, Eds. ACM, 2005, pp. 66–77. [Online]. Available: http://doi.acm.org/10.1145/1086365.1086375

[17] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *Proceedings of the 16th European Symposium on Programming, ESOP 2007*, ser. Lecture Notes in Computer Science, R. D. Nicola, Ed., vol. 4421. Springer, 2007, pp. 520–535.

[18] 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*, vol. 5, pp. 1–236, 1975.

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

[20] J. McCarthy, "History of lisp," in *History of programming languages I*. ACM, 1978, pp. 173–185.

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

[22] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano, "OpenJava: A class-based macro system for java," in *Reflection and Software Engineering, Papers from OORaSE 1999, 1st OOPSLA Workshop on Reflection and Software Engineering*, ser. Lecture Notes in Computer Science. Denver, Colorado, USA: Springer Verlag, 2000, vol. 1826, pp. 117–133, http://www.csg.is.titech.ac.jp/∼mich/openjava/papers/mich_2000lncs1826.pdf.

[23] T. Sheard and S. Peyton Jones, "Template metaprogramming for Haskell," in *ACM SIGPLAN Haskell Workshop 02*, M. M. T. Chakravarty, Ed. ACM Press, Oct. 2002, pp. 1–16.

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

[25] R. Diaconescu and K. Futatsugi, "Logical foundations of CafeOBJ," *Theoretical Computer Science*, 2001, this volume.