

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

Cyrus Omar and Jonathan Aldrich  
Carnegie Mellon University, Pittsburgh, PA  
{comar,aldrich}@cs.cmu.edu

## ABSTRACT

Researchers developing languages and abstractions for high-performance computing must consider a number of design criteria, including performance, verifiability, portability and ease-of-use. Despite the deficiencies of legacy tools and the availability of seemingly superior options, end-users have been reluctant to adopt new language-based abstractions. We argue that this can be largely attributed to a failure to consider three additional criteria: continuity, extensibility and interoperability. This paper introduces Ace, a language that aims to satisfy this more comprehensive set of design criteria. To do so, Ace introduces several novel compile-time mechanisms, makes principled design choices, and builds upon existing standards in HPC, particularly Python and OpenCL. OpenCL support, rather than being built into the language, is implemented atop an extensibility mechanism that also admits abstractions drawn from other seemingly disparate paradigms. The core innovation underlying this and other features of Ace is a novel reification of types as first-class objects at compile-time, representing a refinement to the concept of active libraries that we call *active types*. We validate our overall design by considering a case study of a simulation framework enabling the modular specification and efficient execution of ensembles of neural simulations across clusters of GPUs.

## 1. INTRODUCTION

Computer-aided simulation 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, while 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 so-called *professional end-user developers* [3] must increasingly write parallel programs.

Professional end-users today generally use dynamically-typed high-level languages like MATLAB, Python, R or Perl

for tasks that are not performance-sensitive, such as small-scale data analysis and plotting [4]. For portions of their analyses where the performance overhead of dynamic type checking and automatic memory management is too high, they will typically call into code written in a statically-typed, low-level language, most commonly C or Fortran, that uses low-level parallel abstractions like pthreads and MPI [5, 6]. Unfortunately, these low-level languages and abstractions are notoriously difficult to use and automatic verification is intractable in general.

Researchers and domain experts often respond to these challenges by proposing novel language features that aim to strike an intermediate balance between **performance**, **verifiability**, **portability** and **ease-of-use**. Unfortunately, professional end-users rarely adopt new languages. Indeed, many end-users have become skeptical that novel approaches that originate in the research community can be practical. 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 it may seem paradoxical, the ubiquity of this sentiment demands direct examination by researchers proposing novel abstractions and languages for eventual use by professional end-users in HPC.

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.

In this paper, we introduce Ace, a programming language targeting professional end-users as well as researchers across high-performance computing and related domains. Ace has followed a principled design methodology guided by this more comprehensive set of design criteria in order to avoid many of the issues that have hindered previous language-based approaches in high-performance computing. These criteria, this design methodology, and the novel mechanisms and designs developed to satisfy these criteria constitute the generalizable contributions of this paper. We also hope that Ace itself will be useful to the HPC community<sup>1</sup>, and we describe several use cases and an initial case study in order to preliminarily validate its utility.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

<sup>1</sup>Ace is openly available at <http://acelang.org/>.

We begin in Section 2 with simple examples that show how Ace can be used for low-level GPU programming and introduce the fundamental decisions made in its design. We discuss the motivations, based on the criteria above, behind key design decisions, including the use of static typing, a fixed syntax based on Python, a novel type propagation and inference scheme, and an explicit phase separation between compile-time and run-time logic. Each of these can be seen in Listings 1 and 2, which we will discuss further below.

All examples in Section 2 are based on an internalization of the OpenCL kernel programming language as a library. This library is built atop a novel extensibility mechanism that we call *active typechecking and translation* (AT&T) and detail in Section 3. This mechanism relies on the core idea of representing types as first-class objects at compile-time. We refer to these objects as *active types* by analogy with *active libraries* [7] (see Section 6). Unlike prior approaches to extensibility where users globally modify the grammar or semantics of a language, and thus introduce conflicts between extensions, AT&T guarantees that extensions are composable by construction by pairing new rules with new types and limiting their scope to expressions of that type.

To demonstrate the flexibility of this mechanism beyond OpenCL, we continue in Section 3.4 by outlining examples of other, higher-level parallel abstractions that can be cleanly implemented using AT&T, including global address spaces, message passing and functional data parallelism. First-class support for each of these has required a new language in the past. AT&T allows compile-time logic to be safely included within libraries, and thus these first-class abstractions can be coexist naturally within a single program or workflow.

Ace can be used as a standalone language via the **acec** compiler and also as an interactive compilation environment. This mode of use, described in Section 4, uses just-in-time specialization to integrate the widely-adopted **numpy** library (including its internal type system) and OpenCL’s host API (by a mechanism also available to other host APIs, such as CUDA’s) with the Ace compiler itself to enable the direct invocation of Ace functions from within Python scripts with minimal overhead. Achieving this deep level of integration makes use of Ace’s novel type representation as well.

To demonstrate the utility of Ace for scientific workloads, we describe in Section 5 a case study where Ace was used in this mode to develop a scientific simulation framework. This framework has been used to specify and efficiently execute thousands of realizations of a large stochastic neural circuit model on clusters of GPUs, achieving the same performance as raw OpenCL code while being more modular and concise than was feasible before.

We conclude in Sections 6 and 7 with related work and a discussion of the limitations of Ace at the time of writing, as well as a discussion of planned future work further validating Ace and building upon the concept of active typing.

## 2. LANGUAGE DESIGN AND USAGE

A variant of the standard “Hello, World!” example written in Ace is shown in Listing 1 and its compilation to statically-typed OpenCL kernel code is demonstrated in Listing 2. The OpenCL kernel language is a variant of C99 with some additions and restrictions to facilitate execution on GPUs and other accelerators, in addition to conventional CPUs [?]. We will discuss it further in subsequent examples. We emphasize that this module, which we use throughout the

**Listing 1** [hello.py] A basic Ace program demonstrating the two-phase structure of Ace programs and libraries.

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

**Listing 2** Compiling hello.py using the acec compiler.

```
1  $ acec hello.py
2  Hello, compile-time world!
3  Goodbye, compile-time world!
4  $ cat hello.cl
5  __kernel void main() {
6      char* hello = "Hello, run-time world!";
7      printf(hello);
8  }
```

paper, is simply a library like any other. The core of Ace gives no special treatment to it; it is distributed together with Ace for convenience. Aspects of its implementation will be described in Section 3.

This example demonstrates several key design decisions that characterize Ace: static typing of run-time behavior, a Python-based syntax, a phase distinction between compile-time and run-time logic, and programmatic compilation. We discuss each of these in the next three sections.

### 2.1 Static Typing and Syntax

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 3.4 for examples). Because **verifiability** and **performance** are key criteria and static typing is a core technique, Ace is fundamentally statically-typed.

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 [8]. 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

---

**Listing 3** [listing3.py] A generic data-parallel higher-order map function written using the 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 gp_ptr, double, int
3
4  @OpenCL.fn
5  def add5(x):
6      return x + 5
7
8  D = gp_ptr(double); I = gp_ptr(int); A = add5.ace_type
9  map_add5_dbl = map.compile(D, D, A)
10 map_add5_int = map.compile(I, I, A)

```

---

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 [?]. 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.

## 2.2 Phase Separation

Ace’s **continuity** with Python does not stop at its syntax. Perhaps the most immediately apparent departure from the standard “Hello, World!” example comes on lines 3 and 10 of Listing 1, which contain Python **print** statements that are executed at *compile-time*. Ace programs and libraries 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.

A consequence of this choice is that Ace can leverage Python’s well-developed package system and associated distribution infrastructure directly (e.g. Line 1). This serves to address another key **ease-of-use** issue often associated with C-based languages: the fragility of the preprocessor-based packaging system they historically have relied upon.

Ace functions defining run-time logic are introduced by a decorator. The **@OpenCL.fn** decorator on Line 5 indicates that the **main** function is a statically-typed Ace function targeting the OpenCL backend for code generation (Section ??). 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

---

**Listing 5** [listing4.cl] The OpenCL code generated by running **acec** 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 }

```

---

Python function. Rather, it is a compile-time Python object representing an OpenCL primitive residing in the **OpenCL** module. It has an associated Ace type that controls what types of input it can receive and the type of its output given its input type (see Section 3).

## 2.3 Programmatic Compilation

As defined on Lines 5-7, **main** is a *generic function*. This terminology in Ace is used to indicate a function for which types have not yet been assigned to arguments. Invoking the **compile** method on a generic function with a sequence of argument types invokes the active typechecking and translation mechanism we will describe in Section 3 to produce a *concrete function* – one with a single type for each argument, internal variable (such as **hello**) and the return value. In Listing 1, we name this concrete function **main**, overwriting the generic function of the same name that it is derived from. The compiler does nothing apparently interesting: there are no arguments, the single internal variable takes the OpenCL string type **char\*** from its value and the return type is **void**.

Before moving on to more interesting examples, let us discuss the **acec** compiler shown operating at the shell in Listing 2. The **acec** compiler operates in two steps:

1. Executes the provided Python file (**hello.py**) which contains Ace functions and (in future examples) types and compile-time code generation logic.
2. Produces source code for concrete functions (produced using the **compile** method) in the top-level Python environment and any other concrete functions, type declarations and other program items required by or generated by these functions. This may produce one or more files (here, just **hello.cl**).

We will show in Section ?? that for backends with Python bindings, such as OpenCL, CUDA and C, generic functions can be executed directly, without this explicit compilation step to concrete functions, if desired.

## 2.4 Example 2: Higher-Order Map

The “Hello, World!” example demonstrates the 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. To review, 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 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, it is an instance of the class `ace.GenericFn`). This means that its arguments have not been assigned types. The functionality given by the `map` definition is in fact 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` such that the operations in the function’s body are well-defined.

Running `accc listing3.py` would produce no output. To create a *concrete function* (that is, an instance of the class `ace.ConcreteFn`) that can be emitted by the compiler, types must be assigned to each of the arguments. 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 associates a memory space with pointer types). Line 9 results in a version specialized for arrays of `doubles` and Line 10 results in a version for arrays of `ints`. The output of compilation is shown in Listing 5.

## 2.5 Types as Metalinguage Objects

The `compile` method assigns types to the arguments of a generic function. In Listing 4, the types we are using are given shorter names, for convenience, on Line 8 (that is, variables in the metalinguage can be used like `typedef` is used in a C-like language). The types `int` and `double` imported from the `OpenCL` module correspond to the OpenCL types of the same name. The types `gptr(int)` and `gptr(double)` correspond to `__global int*` and `__global double*`.

These types are *objects in the metalinguage*, Python. More specifically, types are instances of user-defined classes that inherit from the Ace-provided `ace.Type` class. For example, `gptr(double)` is an instance of `OpenCL.PtrType` instantiated with the target type, `double`, and memory space, `__global`, as constructor arguments. The types `double` and `int` are instances of `OpenCL.FloatType` and `OpenCL.IntegerType`, respectively. This notion of types as metalinguage objects is key to the Ace compilation model and also enables other mechanisms that we will discuss in subsequent sections.

## 2.6 Type Propagation

The type assigned to the third argument, `f`, on both Lines 4.9 and 4.10, is `add5.ace_type`. The `ace_type` attribute of a generic function is an instance of `ace.GenericFnType`, the type of Ace generic functions. Ace generic functions are compiled to concrete functions automatically at all internal call sites. That is, when the compiler encounters the call to `f` inside `map` when compiling `map_add5_double`, it compiles a version of `add5` specialized to the `double` type (seen

---

**Listing 6** [listing6.py] A function demonstrating whole-function type inference when multiple values with differing types are assigned to a single identifier, `y`.

---

```
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
11 f = threshold_scale.compile(int, double)
12 g = threshold_scale.compile(int, long)
13 assert f.return_type == double
14 assert g.return_type == long
```

---

on Line 5.3), and similarly when compiling `map_add5_int` (on Line 5.16, automatically given a unique name to avoid conflicts). This mechanism is called *type propagation*. We did not need to use `add5.compile(double)` before compiling `map_add5_double` because only functions that are never called in the process of compiling other functions in a module need type information explicitly provided, supporting *ease-of-use* by increasing conciseness.

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

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

Type propagation via generic functions can be compared to template specialization in C++, where both the template headers (containing nested template parameters for function arguments) and specialization parameters at any call sites are inferred automatically from usage. This significantly simplifies a sophisticated feature of C++ and introduces it to OpenCL and C, which do not support templates. Other uses for C++ templates are subsumed by the metaprogramming features discussed in Section 8.

## 2.7 Whole-Function Type Inference

On Line 5 in the generic `map` function in Listing 3, the variable `gid` is initialized with the result of calling the OpenCL primitive `get_global_id`. The type for `gid` is never given explicitly. This is a simple case of Ace’s more general *whole-function type inference*. In this case, `gid` will be inferred

to have type `size_t` because that is the return type of `get_global_id` (as defined in the OpenCL specification [?], which the `ace.OpenCL` module follows). The result can be observed on Lines 11 and 24 in Listing 5.

Inference is not restricted within single assignments, as in the `map` example, however. Multiple assignments to the same identifier with values of differing types, or multiple return statements, can be combined if the types in each case are compatible with one another (e.g. by a subtyping relation or an implicit coercion). In Listing 6, the `threshold_scale` function assigns different values to `y` in each branch of the conditional. In the first branch, the value `0` is an `int` literal. However, in the second branch of the loop, the type depends on the types of both arguments, `x` and `scale`. We show two choices for these types on Lines 11 and 12. Type inference correctly combines these two types according to OpenCL’s C99-derived rules governing numeric types (defined by the user in the `OpenCL` module, as we will describe in Section 3). We can verify this programmatically on Lines 12 and 13. Note that this example would also work correctly if the assignments to `y` were replaced with `return` statements (in other words, the return value of a function is treated as an assignable for the purpose of type inference).

## 2.8 Annotation and Extension Inference

In addition to type annotations, OpenCL normally asks for additional annotations in a number of other situations. Users can annotate functions that meet certain requirements to be callable from the host with the `__kernel` attribute. The `OpenCL` backend is able to check these requirements and add this annotation automatically. Several types (notably, `double`) and specialized functions require that an OpenCL extension be enabled with a `#pragma` if used. The OpenCL backend automatically detects many of these cases as well and adds the appropriate `#pragma` declaration. An example of this feature can be seen on Line 5.1, where the use of the `double` type triggers the insertion of an appropriate `#pragma` automatically.

## 3. ACTIVE TYPECHECKING AND TRANSLATION (AT&T)

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. A C99 module has also been developed (which, due to its similarity to OpenCL, shares many of its implementation details), but we do not discuss it further in this paper. Extensions complementing these are described in Section C below.

Most programming languages are *monolithic* – the set of available primitives is determined by the language designers, and users must combine these primitives to produce any desired run-time behavior. Although many very general primitives have been developed (e.g. object systems and algebraic datatypes), these can be 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 abstrac-

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

---

```

1  import ace, 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')
```

---

tions, indicates that researchers often find the abstractions available in existing languages insufficient for their needs.

To address these use cases, Ace has been designed to be fundamentally *extensible*, rather than monolithic. Users can introduce new primitive types and operations and fully control how they are typechecked and translated. The backend 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 as modules, rather than as extensions to particular compilers or domain-specific languages.

### 3.1 Active Typechecking and Translation (AT&T)

We now explain how new primitive types and operators can be defined in Ace. When the compiler encounters an expression, such as `input[gid]`, it must first verify its validity by assigning it a type, then translate the expression to produce an expression in a target language. Rather than containing fixed logic for this, however, the Ace compiler defers this responsibility to the *type* of a subexpression, such as `input`, 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 in the language.

- 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<sup>2</sup>.

<sup>2</sup>Note that this operates similarly to the Python run-time operator overloading protocol; see Related Work.



- Responsibility over **attribute access** (`obj.attr`) and **subscript access**, (`obj[idx]`) is handed to the type assigned to `obj`.
- **Talk about multiple assignment here I think**

### 3.1.1 Active Typechecking

During the typechecking phase, the type of the primary operand, as determined by this 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`.

The relevant portion of `OpenCL.GlobalPtrType` is shown in Listing 7. 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.

### 3.1.2 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.)

## 3.2 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

**Listing 8** [listing7.py] Metaprogramming with Ace, showing how to construct generic functions from both strings and abstract syntax trees, and how to manipulate syntax trees at compile-time.

---

```

1  from ace.OpenCL import OpenCL
2  import ace.astx as astx
3
4  plus = OpenCL.fn.from_str("""
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)

```

---

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.

## 3.3 Active Translation

## 3.4 Use Cases

# 4. ACE: A COMPILATION ENVIRONMENT

## 4.1 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 available abstractions [?]. OpenCL in particular relies on code generation as a fundamental mechanism, which is cited as justification for its lack of support for higher-order programming. Ace supports programmatic compilation and higher-order constructs, as described above, and a flexible language extension mechanism, which we describe below, so several use cases for metaprogramming have been eliminated. However, cases where this form of metaprogramming include programmatic function specialization (Listing 8) and modular simulation orchestration, described in Section ??.

On Lines 4-7 of Listing 8, an Ace function is constructed from a string containing its source using the `from_source` variant of the `OpenCL.fn` method. It can also be constructed directly from an abstract syntax tree (AST), as implemented by the Python standard `ast` package, using the `from_ast` variant of `fn`, demonstrated on Line 10. The AST here is generated programmatically by calling the `specialize` function, which produces a copy of the syntax tree of the `plus` function with the argument, `b`, eliminated and its uses replaced with a constant, `5`. This transformation as well as some others are distributed in `ace.astx` for convenience.

## 4.2 Direct Invocation from Python

As discussed in the Introduction, a common workflow for professional end-users involves the use of a high-level script-

---

**Listing 9** [listing9.py] A full OpenCL program using the `Ace.OpenCL` Python bindings, including data transfer to and from a device and direct invocation of a generic function, `map`, as a kernel without explicit compilation.

---

```
1 import numpy as np
2 import ace.OpenCL.bindings as cl
3 from listing1 import map
4 from listing2 import add5
5
6 cl.ctx = cl.Context.for_device(0, 0)
7
8 input = np.ones((1024,))
9 d_in = cl.to_device(input)
10 d_out = cl.alloc(like=d_in)
11
12 map(d_in, d_out, add5,
13     global_size=d_in.shape, local_size=(128,))
14
15 out = cl.from_device(d_out)
16 assert (out == a + 5).all()
```

---

ing 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 a high-level scripting language and also features mature support for calling into code written in low-level languages. Developers can call into native libraries using its foreign function interface (FFI), or by using a wrapper library like `pycuda` for code compiled with CUDA, a proprietary language similar to OpenCL specifically targeting nVidia GPU hardware. Although CUDA’s compilers are separate executables on the system, the OpenCL language was designed for this workflow, in that it exposes the compiler directly as an API. The `pyopenc1` module exposes this API as well as the OpenCL memory management API to Python. With both `pyopenc1` and `pycuda`, developers generate source code as strings, compile it programmatically, then execute it using the run-time APIs that each library provides [?].

Ace supports this workflow as an alternative to using the `acec` compiler as described above to generate source code directly from the shell. For the OpenCL backend, these bindings are exposed as a wrapper on top of `pyopenc1` called `Ace.OpenCL.bindings`. 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 9, with the call itself on Lines 9-10. The first two arguments to `map` are OpenCL buffers, generated using a simplified wrapper to the `pyopenc1` APIs on Lines 7-8. This wrapper associates type information with each buffer, similarly to `numpy`, and this is used to implicitly compile `map` as appropriate the first time it is called for any given combination of input types. Explicit calls to the `compile` method, as we have been showing thusfar, are unnecessary if 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 thread grouping (the *local size*).

By way of comparison, the same program written using the OpenCL C API directly is an order of magnitude larger and correspondingly more complex. A full implementation of the logic of `map` written using the `pyopenc1` 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; due to a lack of space, the reader is referred to the language documentation for additional details on these features.

## 5. CASE STUDY

## 6. RELATED WORK

## 7. DISCUSSION

## 8. STRUCTURE AND USAGE

## 9. ACE FOR RESEARCHERS

### 9.1 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 [9]. 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. [10]).

### Specialized Optimizations.

In many cases, code optimization requires domain-specific knowledge or sophisticated, parametrizable heuristics. Existing compilers make implementing and distributing 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 [11] 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 [11]) 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.

## 10. 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.

### 10.1 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.

### 10.2 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`<sup>3</sup> 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.

<sup>3</sup>...after *c. elegans*, a model organism in neuroscience



---

**Listing 10** [listing10.py] An example of a nested simulation tree, showing that specifying a simulation is both simple and modular. The first argument to the constructor specifies each node’s parent.

---

```

1  sim = Simulation(None, n_timesteps=10000)
2  neurons = ReducedLIF(sim, count=N, tau=20.0)
3  e_synapse = ExponentialSynapse(neurons, 'ge',
4      tau=5.0, reversal=60.0)
5  probe = StateVariableProbeCopyback(e_synapse)

```

---

**Listing 11** [listing11.py] An example of a hook that inserts code and also inserts new, nested hooks for downstream simulation nodes below that.

---

```

1  class SpikingModel(Model):
2      """Base class for spiking neuron models."""
3      def in_model_code(self, g):
4          "idx_state = idx_model + count*" << g
5          "(realization_n - realization_start)" << g
6          self.insert_hook("read_incoming", g)
7          self.insert_hook("read_state", g)
8          self.insert_hook("calculate_inputs", g)
9          self.insert_hook("state_calculations", g)
10         self.insert_hook("spike_processing", g)
11         # ...

```

---

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.elems` 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 4.2. 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 `pyopenc1` library [12].)

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.

## 11. RELATED WORK

### 11.1 Active Libraries in Ace

Libraries that have such capabilities has been called *active libraries* in prior proposals [7]. 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.

#### 11.1.1 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 [13].

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 ??.

### 11.2 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 [14]). Ur uses type-level records and names to support typesafe metaprogramming, with applications to web programming [15]. Omega adds algebraic data types at the type-level, using these to increase the expressive power of algebraic data types at the expression level [16]. 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 [17], ATS [18].)

### 11.3 Run-Time Indirection

*Operator overloading* [19] 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.

### 11.4 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 [21], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [22]), 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[23], Template Haskell[24], MetaML [22] and others).

## 11.5 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 [25]).

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.

## 11.6 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.

## 11.7 Specification Languages

Several *specification languages* (or *logical frameworks*) based on these theoretical formulations exist, including the OBJ family of languages (e.g. CafeOBJ [26]). 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.

## 12. 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 [7] where types are metalinguage 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.

### 13. 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)

### 14. ACKNOWLEDGMENTS

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

### 15. 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. Sankaranarayanan, "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] 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>
- [8] J. Cordy, "Hints on the design of user interface language features: lessons from the design of turing," in *Languages for developing user interfaces*. AK Peters, Ltd., 1992, pp. 329–340.
- [9] 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.
- [10] A. Kennedy, "Types for units-of-measure: Theory and practice," in *CEFP*, ser. Lecture Notes in Computer Science, Z. Horváth, R. Plasmeijer, and V. Zsóok, Eds., vol. 6299. Springer, 2009, pp. 268–305. [Online]. Available: <http://dx.doi.org/10.1007/978-3-642-17685-2>
- [11] 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.
- [12] 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.
- [13] D. Malayeri and J. Aldrich, "Is structural subtyping useful? an empirical study," *Programming Languages and Systems*, pp. 95–111, 2009.
- [14] 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.
- [15] 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>
- [16] T. Sheard and N. Linger, "Programming in omega," in *CEFP*, ser. Lecture Notes in Computer Science, Z. Horváth, R. Plasmeijer, A. Soós, and V. Zsóok, Eds., vol. 5161. Springer, 2007, pp. 158–227.
- [17] 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>
- [18] 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.
- [19] 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.
- [20] G. Kiczales, J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. Cambridge, MA: MIT Press, 1991.
- [21] J. McCarthy, "History of lisp," in *History of*

- programming languages I*. ACM, 1978, pp. 173–185.
- [22] T. Sheard, “Using MetaML: A staged programming language,” *Lecture Notes in Computer Science*, vol. 1608, pp. 207–??, 1999.
  - [23] 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](http://www.csg.is.titech.ac.jp/~mich/openjava/papers/mich_2000lncs1826.pdf).
  - [24] 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.
  - [25] M. Fowler and R. Parsons, *Domain-Specific Languages*. Addison-Wesley Professional, 2010.
  - [26] R. Diaconescu and K. Futatsugi, “Logical foundations of CafeOBJ,” *Theoretical Computer Science*, 2001, this volume.