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

Ace is an extensible statically-typed programming language embedded within the ubiquitous scripting language Python. Python serves as a metalanguage, enabling programmatic control over several aspects of language specification and compilation. The central innovation is a novel reification of types as first-class objects in the metalanguage, a refinement to the concept of active libraries that we call *active types*. We demonstrate the power of this design for HPC by defining all of the primitives of the OpenCL kernel language as a library of user-defined types in Ace, then extending this core with several higher-level first-class abstractions. The target of translation, called the *backend*, is user-defined as well, allowing us to implement the semantics of these primitives by direct translation to OpenCL, CUDA or C99.

Although Ace is built around a static type discipline, it supports a novel form of generic programming based on type propagation that, together with a form of extensible local type inference, eliminates most of the syntactic overhead usually associated with statically-typed languages. A companion host API allows type information from `numpy` arrays to directly propagate into Ace kernels, supporting seamless adoption of Ace into current scientific workflows.

We validate our overall design by developing a simulation framework that uses the features of Ace to allow users to specify, compile and orchestrate the execution of ensembles of neural simulations on 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 like 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 properly because verifying code written with them remains a difficult (and often ill-specified) problem.

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, even when they have been actively developed. 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 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.

Ace is a programming language intended to be used by professional end-users as well as researchers in areas related to high-performance computing and computational science. The primary contribution of this paper consists of describing several novel language mechanisms and novel combinations of existing mechanisms that Ace uses to cohesively address the design criteria above. We anticipate that many of these mechanisms and choices will also influence other languages in the future. Following a principled design methodology, we justify each design decision by appeal to this full set of design criteria and citations to relevant literature. These

criteria and methodology constitute another contribution of this work that we hope will lead to fewer *ad hoc* designs and serve as a foundation for evaluating competing designs in HPC and related domains in the future.

We begin in Section 2 with a high-level overview of the Ace language design and the reasoning behind the choices that were made. In Section 3 we provide further details by way of concrete examples employing Ace for low-level GPU programming, based on an internalization of the OpenCL programming language as an Ace library. Ace can be used both as a standalone language via the `acec` compiler and as an interactive compilation environment. This latter mode of use enables just-in-time specialization and direct invocation of Ace functions from within Python scripts, as well as deep integration between the **numpy** library, the OpenCL host API and the Ace compiler.

In Section 4, we describe the extensible type system, which is based on the novel idea of reifying types as metalanguage objects. This mechanism, called *active typing*, was used to implement the OpenCL library. The operations associated with *active types* are given meaning by translation to a user-defined backend. We show in Section 4.1 how this allows us to give semantics to OpenCL constructs by translation to OpenCL itself, as well as to CUDA and C99, increasing the operational flexibility of the OpenCL standard. Then in Section 4.2, we continue with several examples of higher-level parallel abstractions that can be implemented in Ace. In other systems, these have required deriving a new variant of an existing language or a new language, both strategies that violate at least two of the three new criteria above. In Ace, these exist within libraries and can safely be used within a single program.

To demonstrate that Ace can be used for realistic scientific workloads, we briefly describe in Section 5 a case study that used Ace as the foundation for a neurobiological simulation framework capable of orchestrating the execution of thousands of realizations of a large stochastic neural circuit model on clusters of GPUs. We conclude in Sections 6 and 7 with related work and a discussion of some limitations of Ace at the time of writing, as well as a discussion of future work building on both Ace itself as well as the concept of active typing central to Ace.

2. THE DESIGN OF ACE

It is now widely-acknowledged that static type systems are fundamental tools for performance and verifiability, both critical criteria in HPC, so Ace is fundamentally a statically-typed language. To achieve continuity, Ace is implemented as a library in the ubiquitous scripting language Python. The Ace grammar is precisely that of Python, and it share's Python's packaging system, so any tools that work with Python files and packages can immediately be used with Ace.

Another important tool that has become increasingly prominent is the use of code generation techniques to optimize code and orchestrate scientific workflows, so Ace includes a powerful

- Types are fundamental tools for both performance and verifiability.

- Programmers in high-performance computing tend to be wary of abstraction mechanisms that necessitate run-time overhead.

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       printf("Hello, run-time world!")
8   main = main.compile()
9
10  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       printf("Hello, run-time world.");
7   }
```

- Compile-time code generation is often used to optimize code and orchestrate scientific workflows.

- Scripting languages, such as Python, are particularly appreciated for their ease-of-use.

- Library-based abstractions are more highly-adopted, but language-integrated abstractions are easier to use [**?**].

3. USAGE OF ACE

4. ACTIVE TYPECHECKING AND TRANSLATION

4.1 Active Translation

4.2 Use Cases

5. CASE STUDY

6. RELATED WORK

7. DISCUSSION

8. STRUCTURE AND USAGE

8.1 Example 1: Hello, World!

To demonstrate the somewhat unconventional structure of Ace programs and libraries, we begin in Listing 1 with a variant of the canonical "Hello, World!" example written using the `OpenCL` module. 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. Aspects of its implementation will be described in Section 4.

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])
```

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 `acec`, shown operating at the shell in Listing 2. The `acec` 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 `acec` executable is simply a convenience; the generated code is also available as an attribute of 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.

8.2 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

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)
```

Listing 5 [`listing5.cl`] The OpenCL code generated by running `acec listing2.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  }
```

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 `acec 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 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.

8.3   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 4, this notion of types as metalanguage objects is a key to Ace's flexibility.

8.4   Type Propagation and Higher-Order Functions

The type assigned to the third argument, `f`, on Lines 4.9 and 4.10, 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 from another function. That is, when the compiler encounters the call to `f` inside `map` when compiling `map_add5_double`, it compiles a version of `add5` specialized to the `double` type (seen on Line 5.3), and similarly with the `int` type when compiling `map_add5_int` (on Line 5.16, automatically given a unique name to avoid conflicts). This mechanism is called *type propagation*. In other words, we did not need to use `add5.compile(double)` before compiling `map_add5_dbl` (although we could if we would like). Only functions that are never called in the process of compiling other functions in a module need type information explicitly provided, whereas

`add5` is a function that is only called within `map` in Listing 4.

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 even 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 one would expect). To put it in type theoretic terms, `ace.GenericFnType`s are singleton types, *uniquely inhabited* by a single generic function, 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 is rarely a problem, particularly in parallel programming where explicit specialization is 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 simulated by using a jump table implementation in OpenCL, but we do not discuss this further due to lack of space.

8.5   Type Inference

On Line 5 in the generic `map` function in Listing 3, the variable `gid` is initialized with the result of calling the `get_global_id` primitive (the argument, `0`, is not important for our purposes.) Note that the type for the `gid` variable is never listed explicitly. This is because Ace supports a form of *whole-function type inference*. In this case, `gid` will be assigned 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. Multiple assignments to the same variable of differing types, or multiple return statements, can be unified such that the variable or return type is given a common supertype. For example, in the `threshold_scale` function defined 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 choices for these types on Lines 10 and 11. However, type inference correctly unifies these two types according to OpenCL's C99-derived rules governing numeric types (which are defined by the user in the `OpenCL` module, as we will describe in Section 4). We test 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).

8.6   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 `Ace.OpenCL.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` when used. The OpenCL backend automatically detects many of

Listing 6 [`listing6.py`] 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
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
```

Listing 7 [`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)
```

Listing 8 [`listing8.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()
```

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. Ace is designed to allow backends to observe the results of the type checking process to support this form of inference.

8.7 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 7) and modular simulation orchestration, described in Section ??.

On Lines 4-7 of Listing 7, 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.

8.8 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 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 `pyopencl` module exposes this API as well as the OpenCL memory management API to Python. With both `pyopencl` 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 `pyopencl` 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 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 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 `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; due to a lack of space, the reader is referred to the language documentation for additional details on these features.

9. 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. 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 abstractions, 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.

9.1 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

---

Listing 9 [`listing9.py`] 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')
```

---

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[1].

- 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

9.1.1 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

---

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

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 `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.

9.1.2   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.)

9.2   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 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.

9.3   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 [8]. 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. [9]).

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 [10] 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 [10]) 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[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 8.8. 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 [11].)

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 [`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      # ...
```

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 [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 **??**.

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 [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].)

11.3 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.

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 [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. Open-Java[22], Template Haskell[23], MetaML [21] 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 [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.

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 [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.

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 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.

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

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. 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] 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] 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.

[9] 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

[10] 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.

[11] 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.

[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_2000lnc

[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.