Active Typechecking and Translation

Cyrus Omar Nathan Fulton Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
{comar, jonathan.aldrich}@cs.cmu.edu

Abstract

Researchers and domain experts typically describe new language-based abstractions as extensions of an existing language, but most statically-typed languages are *monolithic*: they do not give users the ability to specify the static and dynamic semantics of new types and operators from within. This has lead to a proliferation of mutually incompatible standalone languages, each built around a small collection of privileged constructs. An alternative to this *language-external* approach is to work in an extensible programming language where the compile-time behaviors determining the functionality of new core constructs are provided directly within user libraries. Designing such a *language-internal* extensibility mechanism that is both safe and expressive is non-trivial. This paper introduces a mechanism called active type-checking and translation (AT&T) that aims to address these challenges. By building upon type-level computation in a novel way, AT&T admits user specification of a wide range of compile-time behaviors over a fixed grammar in a safe and modular manner. We discuss two points in the design space: (1) a simple calculus designed to distill the essential concepts and admit formal safety theorems, and (2) a fully-implemented language called Ace that we use to demonstrate the expressive power of AT&T across several domains, including scientific computing, security, functional programming and object-oriented programming.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Extensible Languages; D.3.4 [*Programming Languages*]: Processors—Compilers; F.3.1 [*Logics & Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification Techniques

1. Introduction

Programming languages have historically been specified and implemented monolithically. To introduce new primitive constructs, researchers or domain experts have developed a new language or a dialect of an existing language, often with the help of tools like domain-specific language frameworks and compiler generators [**?**]. Unfortunately, using different languages for different components of an application, called a *language-oriented approach* [**?**], has problems when crossing language boundaries: the external interface of a library must only use constructs that can be expressed in all possible calling languages. (talk about typical design using a common language like the JVM) Thus, any specialized invariants cannot be checked statically, decreasing reliability. It also often requires that developers generate unnatural "glue" code, impacting performance and defeating a primary purpose of specialized languages: abstracting these low-level details away from developers.

Extensible programming languages promise to decrease the need for new standalone languages by providing more granular, language-internal support for introducing new primitive constructs (i.e. constructs that cannot be correctly and concisely expressed in terms of existing constructs). By using an extensible language, developers would gain the freedom to choose those constructs most suitable for their problem domain and development discipline. Researchers and domain experts would gain the ability to develop new constructs modularly and distribute them for evaluation by a broader development community without requiring either wholesale adoption of a new language or approval from the maintainers of widely-used languages, who are naturally risk-averse and are unlikely to cater to niche domains. The key difference between the language-external approach, characteristic of language frameworks, and this language-internal approach, characteristic of extensible languages, is illustrated in Figure 1.

A number of criteria constrain potential language-internal extensibility mechanisms, however. One key issue is that a candidate mechanism must maintain the safety properties of the language and compilation process in the presence of arbitrary combinations of user extensions. The mechanism must ensure that basic metatheoretic and global safety guaran-

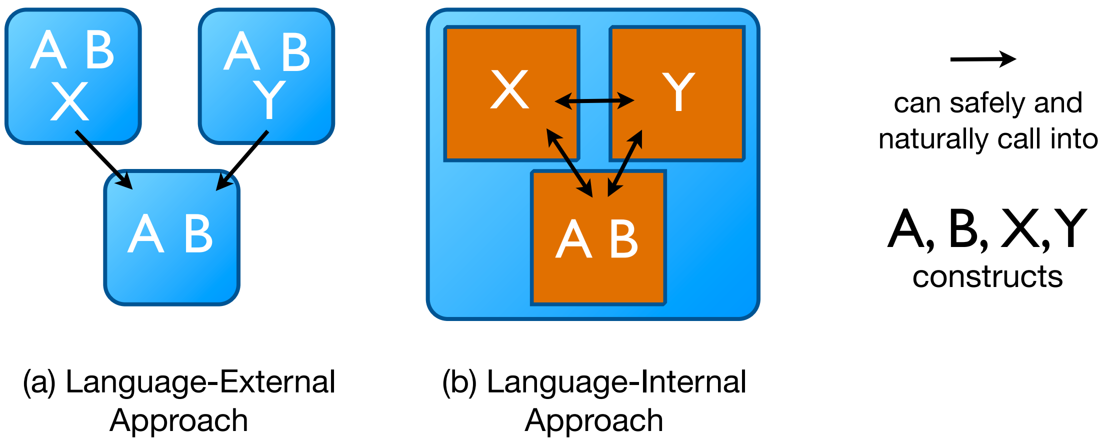
A B
X
A B
Y
A B
(a) Language-External Approach
X
Y
A B
(b) Language-Internal Approach
can safely and naturally call into
A, B, X, Y
constructs

Figure 1. (a) With the language-oriented approach, novel constructs are packaged into separate languages. Users can only safely and naturally call into languages consisting of common constructs. (b) With the library-oriented approach, there is one language and novel constructs are packaged as normal libraries. Thus, interoperability is not a problem. in monolithic systems, this approach is less expressive.

tees of the language cannot be weakened, that extensions are safely composable, and that typechecking and compilation remain decidable. The correctness of extensions themselves should be modularly verifiable, so that developers can rely on them for verifying and compiling their own code. At the same time, the extensibility mechanism should be capable of expressing a variety of language constructs naturally (from the perspective of the developers using the constructs). For extension developers, using the extension mechanism should be comparable in difficulty to using DSL frameworks and other language-external tools. These criteria guide the work that we describe here.

We will begin by considering the typical compiler front-end, consisting of a typechecker and an initial translator to a high-level intermediate representation (e.g. bytecode for a virtual machine). The extensibility problem considered in this context can be cast as an instance of the *expression problem* first examined by Reynolds [**?**] and so-named by Wadler [**?**]. Due to the particular structure of this domain, it is possible to sidestep the problem by delegating to functions associated with the internal representation of types, a form of *inversion of control*.

Building an extensible compiler front-end by exposing these representations via some API is a tempting approach, but it remains a *language-external* one. That is, a compiler that allows for new language constructs (by any mechanism) creates, *de facto*, a new language (an underappreciated fact in many language communities!) Thus, programs compiled against different collections of extensions may be mutually incompatible with one another. Moreover, such extensions cannot be used by different compilers for the same language.

All is not lost, however. It can be observed that types are represented both within the compiler front-end and within the language itself. In most languages, the mechanisms for defining and referring to types are purely declarative. In languages featuring support for *type-level computation*, however, types are a form of value that can be manipulated in various ways at compile-time via an internal *type-level language* [**?**]. This observation forms the foundation of the extensibility mechanism we describe here, called *active typechecking and translation* (AT&T), so named because it can be seen as a refinement of the general concept of *active libraries* proposed by veldHuizen [**?**] (see Section **??** for further discussion).

We will introduce AT&T by way of two artifacts: a fully-implemented language, Ace, and a minimal type-theoretic formulation, λ_A. Ace demonstrates the practicality of AT&T across a number of problem domains, beginning with low-level, performance-critical programming (targeting the OpenCL language and compilation infrastructure) and then demonstrate support for language-internal extensions that add a variety of higher-level abstractions in Section **??**. Ace, as an implemented prototype that makes some concessions for the sake of usability, does not readily admit formal safety theorems. For this reason, we introduce λ_A, a terse and somewhat more limited formalism based on a typed lambda calculus with type-level computation of higher kind. This calculus captures the core elements of the mechanism and admits several important safety theorems.

2. Inversion of Control in a Compiler

3. Structure and Usage

3.1 Example 1: Hello, World!

To demonstrate the somewhat unconventional structure of Ace programs and libraries, we begin in Listing 2 with a variant of the canonical "Hello, World!" example written using the `OpenCL` module. We note that this module, which

```
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!"
```

Figure 2. `[hello.py]` A basic Ace program demonstrating the two-phase structure of Ace programs and libraries.

```
$ acec hello.py
Hello, compile-time world!
Goodbye, compile-time world!
$ cat hello.cl
__kernel void main() {
    printf("Hello, run-time world.");
}
```

Figure 3. Compiling `hello.py` using the `acec` compiler.

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 **??**.

Perhaps the most immediately apparent departure from the standard "Hello, World!" example comes on lines 3 and 10 of Listing 2, 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 3. The `acec` compiler operates as follows:

```
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])
```

Figure 4. `[listing4.py]` A generic data-parallel higher-order map function written using the OpenCL user module.

```
from listing3 import map
from ace.OpenCL import global_ptr, double, int

@OpenCL.fn
def add5(x):
    return x + 5

A = global_ptr(double); B = global_ptr(int)
map_add5_dbl = map.compile(A, A, add5.ace_type)
map_add5_int = map.compile(B, B, add5.ace_type)
```

Figure 5. `[listing5.py]` The generic `map` function compiled to map the `add5` function over two types of input.

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 2) 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.

3.2 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 4 shows an imperative, data-parallel map primitive written using the OpenCL library introduced above. In OpenCL, users can define functions, called *kernels*, that execute across thousands of threads. Each kernel has access to a unique index, called its *global id*, which can be used by the programmer to ensure that each thread operates on different parts of the input data (Line 5). The `map` kernel defined in Listing 4 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

```
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 }
```

Figure 6. [`listing6.cl`] The OpenCL code generated by running `acec listing2.py`.

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 5 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 6.

3.3 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 6, 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.Globa` instantiated with the target type, `double`, as a constructor argument. The types `double` and `int` (imported from `ace.OpenCL` on Line 4.2) are instances of `ace.OpenCL.FloatType` and `ace.OpenCL.IntegerType`, respectively. As we will describe further in Section **??**, this notion of types as metalanguage objects is a key to Ace's flexibility.

3.4 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 5.

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.

3.5 Type Inference

On Line 5 in the generic `map` function in Listing 4, 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

```
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
```

Figure 7. [listing6.py] A function demonstrating whole-function type inference when multiple values with differing types are assigned to a single variable.

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 6.

Inference is not restricted within single assignments, as in the map example. Multiple assignments to the same variable with values 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 7, 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 **??**). 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).

3.6 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 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.

```
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)
```

Figure 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.

3.7 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 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.

3.8 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

```
 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()
```

Figure 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.

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 runtime 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 4 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 `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.

4. 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.

4.1 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`,

```
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')
```

Figure 10. `[listing9.py]` A portion of the implementation of OpenCL pointer types implementing subscripting logic using the Ace extension mechanism.

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

4.1.1 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 4 once again. When it is compiled on Line 9 of Listing 5, 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 `ace.OpenCL.GlobalPtrType` is shown in Listing 10. 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.

4.1.2 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.)

4.2 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 4. Backends are responsible for some aspects of the grammar that do not admit simple dis-

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

patch 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.

4.3 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 [**?**]. 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. [**?**]).

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 [**?**] 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 [**?**]) 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.

4.4 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 [**?**]).

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.

4.5 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.

4.6 Specification Languages

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

5. 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 [**?**] 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 4. 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.

6. 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)

7. Acknowledgments

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

8. Background

The approach we describe, *active type-checking and translation* (AT&T), makes use of type-level computation in a novel way. To review, in languages supporting type-level computation, the syntactic class of types is not simply declarative. Instead, it forms a programming language itself (the *type-level language*). Types themselves are one kind of value in this language, but there can be many others. To ensure the safety of type-level computations, *kinds* classify type-level terms, just as types classify expression-level terms. The simplest example of a language featuring type-level computation is Girard's System F$_\omega$ [**?**]. In F$_\omega$, types have kind $\star$ and type-level functions have kinds like $\star \to \star$. A growing number of implemented languages now feature more sophisticated type-level languages (see Section 5). We emphasize that type-level computation occurs during compilation, rather than at run-time, because type-level terms that are used where types would normally be expected must be reduced to normal form before type-checking can proceed.

In this work, we wish to allow extensions to strengthen the static semantics of our language. Naturally, extension specifications will also need to be evaluated during compilation and manipulate representations of types. This observation suggests that the type-level language may be able to serve directly as a specification language. In this paper, we show that this is indeed the case. By introducing some new constructs at the type-level, developers can specify the semantics of operators associated with newly-introduced families of primitive types with type-level functions. The compiler front-end invokes these functions to synthesize types for and assign meanings to expressions, by translation into a *typed internal language*. Unlike conventional metaprogramming systems, these *type-level specifications* do not directly manipulate or rewrite expressions. Instead, they examine and manipulate the types of these expressions. By using a sufficiently constraining kind system and incorporating techniques from typed compilation into the type-level language directly, the global safety properties of the language and compilation process can be guaranteed. In other words, users can only *increase* the safety of the language.

We focus on extending the static semantics of a language with a fixed, though flexible, grammar. Techniques for extensible parsing have been proposed in the past (e.g. [**?**]), and we conjecture that these can be made compatible with the mechanism described in this paper with some simple modifications, but we do not discuss this further here. We also focus on *functional*, rather than declarative, specifications of language constructs. Extracting a compiler from a declarative language specification (e.g. in Twelf [**?**]) has not yet been shown practical, but we note that a future mechanism of this sort could safely target a language implementing the mechanism we discuss here.

9. Type-Level Specifications in $\lambda_{\mathbf{A}}$

9.1 Example: Natural Numbers in $\lambda_{\mathbf{A}}$

We begin with a simple calculus with no primitive notion of natural numbers, nor any more general notion of an inductive data type. We can, however, concretely specify both the static and dynamic semantics of natural numbers, including the natural recursor of Gödel's T [**?**], using type-level specifications. Let us begin in Figure 1 with the type **nat** and its constructors, **z** and **s**.