Key Research Contributions

Peter Vanderbilt
pvanderbilt@acm.org

Abstract

This white paper describes a handful of key research contributions. I (the author) describe these contributions in terms of an example programming language called P3, designed by me, which is similar to Java [5] and Scala [8]. P3 is not an end in itself, but rather a vehicle for explanation—I expect that the features described here could be integrated into a range of languages.

1. Key research contributions

This white paper describes a handful of key research contributions. I (the author) describe these contributions in terms of an example programming language called P3, designed by me, which is similar to Java [5] and Scala [8]. P3 is not an end in itself, but rather a vehicle for explanation—I expect that the features described here could be integrated into a range of languages.

1.1 Complex content and hybrids

One focus of my research is to allow containers, such as classes, to have complex content, which is content that includes types, classes, and other complex objects (as well as regular, simple data). The most basic complex container is what I call a hybrid, which is a record-like construct that is a cross between a record and a type group. A type group is as described by Bruce [1] and is also considered a form of operational semantics, in that the execution of a hybrid uncovers subtle details of the higher-level construct. Also, encodings could be integrated into a range of languages.

1.2 Encodings and compound constructs

My methodology is to define higher-level constructs in terms of encodings, which are mappings to lower-level terms, typically hybrids. These encodings define the semantics of the higher-level construct, and also provide insight into the typing of it, typically leading to higher-level typing rules.

The following is an encoding of simple classes:

\[ \text{closed class } C \{ \text{ const } \bar{x} : \bar{T} \} \rightarrow (1.1) \]

\[ \text{type } C@ = \{ \bar{x} : \bar{T} \}; \]

\[ \text{func } C\_\text{new} () : C@ = \mu(\text{self}) \{ \bar{x} = \bar{e} \}; \]

The encoding arrow \( \rightarrow \) indicates that the higher-level definition to its left encodes as the series of definitions to its right. The italic identifiers (like \( C \) and \( \bar{x} \)) are pattern variables that are instantiated when the left-hand side is matched against an actual class definition and expanded in the right-hand side to yield the actual encoding.

Without going into detail, encoding (1.1) indicates that the class definition to its left encodes as a type definition for \( C@ \) and a function definition for \( C\_\text{new}() \) (aka \texttt{new C}). Type \( C@ \) is the type of \( C \)’s instances (see 1.4) and \( C\_\text{new}() \) is the function that creates them. Class encodings become considerably more complex when they handle features such as inheritance, complex content, abstract fields and static fields. My complex class encoding [11] is one such accomplishment.

An encoding can be seen as defining the detailed semantics of a higher-level construct. Often working through the details of an encoding uncovers subtle details of the higher-level construct that would not be otherwise obvious. Also, encodings could be considered a form of operational semantics, in that the execution of a higher-level construct would be accomplished by the execution of its encoding.

Encodings can be used to derive or justify typing rules. In particular, the conditions required for the result of an encoding to be well-typed become premises of the higher-level rule. For example, when typing the right-hand side of encoding (1.1) at the hybrid level, the following conditions fall out:

\[ \Gamma \vdash \bar{T} :: * \quad (1.2) \]

\[ \Gamma; \text{ const self } : \{ \text{ const } \bar{x} : \bar{T} \} \vdash \bar{e} : \bar{T} \]

The first ensures that types appearing the class are well-formed and the second ensures that the implementations are well-typed. These conditions can be seen as necessary and sufficient for the type correctness of the class being encoded by (1.1).

Encodings also push on lower level technologies, uncovering requirements that might not be obvious from simpler examples. For instance, some encodings require the type of a function’s parameter to depend on the value of the function. Since encodings may be nested within encodings, these requirements can work their way down several levels.

Note that the right-hand side of (1.1) is a sequence of two definitions. We call constructs that encode to more than one thing compound constructs. Classes and modules are compound constructs. Compound constructs arise for two main reasons: (1) the construct
specifies several related aspects such as interface and implementation and (2) constructs have both a generic aspect needed for inheritance and a specialized aspect for the thing itself. Note that when embedded in some container construct, a compound construct can be viewed as complex content.

1.3 Extensible modules with mutually recursive content

There are cases, such as the Subject/Observer pattern [9], where mutually recursive classes cannot be extended in a statically type-safe fashion while maintaining their recursive relationship. I have developed a module construct that permits type-safe extension of collections of complex (see 1.1) elements. These modules can have fairly arbitrary, mutually recursive relationships between components and these relationships are maintained in extension. There is a mechanism for type-safe polymorphism over all extensions of a module, while permitting full use of the actual module. I have encodings (see 1.2) and typing rules for modules of varying complexity.

Within a module definition, the special variable “mself” denotes the future module and so, acts as the root of type paths; for example, mself.Module@ might appear in a module’s Subject class as a reference to the type of instances of the module’s future Observer class.

There are cases, such as the Subject/Observer pattern [9], where non-top-level modules (“submodules”) also have the special variable “msuper” as a way to reference implementations (including type values) of its “super-module”. This gets tricky because, for example, in a series of module extensions, m1 to mk, where each overrides inner class A with something that extends msuper.A, the effective value of m1.mA@ depends on the definitions of A in each module m1 to mk−1.

Another innovation is the special variable “mdef” to denote (mself-parameterized) implementations within the current module. This supports sibling extension, where for example in the “expression problem” [1, 12], component class Neg extends its sibling class, mdef.Exp, while its field is of type mself.Exp. Variable mdef is also important in expressing relationships such as “the future Neg class must be an extension of the current class, mdef.Neg, and the future Exp class, mself.Exp”. Finally, mdef is crucial to nested virtual classes when using an encoding that results in multiple definitions. For example, in (1.1), the C@ as the return type of C.new is the one defined at the line before, not some future (overridden) version of C@.

1.4 Two instance types per complex class

Generally when classes have complex content, subclassing does not imply subtyping – given that the types of fields can reference virtual types that can vary arbitrarily, it is unlikely that such types subtype in the subclass. This lack of subclass-implied subtyping means that types that can vary arbitrarily, it is unlikely that such types subtype.

My approach is to define two types per class. For class C, C@ is its exact type and denotes instances of just class C, while C# is C’s general type and denotes instances of class C and its subclasses. My class encodings (see 1.2) yield specific, hybrid-level definitions of C@ and C#.

Type C@ has the actual, exact values for all type fields and, so, permits type-safe use of any field. Type C# has bounds on field types, but not actual values, so accesses to fields whose types reference virtual types are effectively disabled, thus avoiding the runtime errors mentioned above.

The expression “new C#” has C@ as the return type, which is a subtype of C#. For subclass B of C, “new B#” yields a B@ which is a subtype of B@ which is a subtype of C@, but B@ and C@ are generally incompatible types.

Thus variables of type C@ can be used in ways that those of type C# cannot. On the other hand, type C# permits subclass polymorphism, which C@ does not. If class C has only data (non-complex) fields, C@ and C# are equal and support both safe access to all fields and subclass polymorphism, as expected.

1.5 First-class classes

One branch of my research treats high-level class definitions as resulting in class objects, which are first-class in the sense that they can be passed as parameters and stored in data structures. A class object is a hybrid (see 1.1) with at least a type field, TE, and a function, new(). Type C@ (see 1.4) is redefined to be syntactic sugar for C._TE and expression “new C” is sugar for C.new().

In particular, for concrete class C, its high-level definition is encoded as several lower-level definitions, including one for name C. The following is a simple encoding of class definitions to class objects (replacing and extending (1.1)):}

```
@closed class C {  
  const x_m : T_m = e_m ;  
  static const x_s : T_s = e_s ;  
  }  
  C = m(C) {  
    type _TE = { x_m : T_m } ;  
    func new () : C@ = m(self) { x_m = e_m ; } : C@ ;  
    x_s : T_s = e_s ;  
  }
```

The first line of the object literal in the right-hand side defines the exact type, C@, the second line defines the implementation of “new C,” and the third line denotes a series of definitions for static fields (those labelled static in the left-hand side).

With class objects comes the notion of class types whose instances are class objects. I currently define one class type related to extension whose syntactic sugar is “<<<C)” and such that “X<<<C)” for variable X means that X ranges over all classes extending open class C (including C if it is concrete). For example, consider the following method in partial class Iterable[T]:

```
@func mapTo [R] (f:T->R) : C@ = m(C) {  
  const builder = C.newBuilder();  
  foreach (e in self.elements()) { builder.add(f(e)) ; }  
  return builder.finalize() ;  
}
```

The idea is that a call of the form C.mapTo(f/List[Int]) will collect the result in an instance of List[Int] while C.mapTo(f/IntBag) will collect the result in an IntBag instance. In more detail, c.mapTo[R][f,C] first creates a “builder” object by calling C’s static method newBuilder, then applies function f to every element of c, collecting the results in the builder object, and finally calls the builder’s finalize method to yield an instance of the class specified by parameter C. Note that the class argument C does double duty: (1) it defines the return type, C@, of the method and (2) it provides a way, C.newBuilder(), to create the builder object. Without going into details, parameter declaration C<<<Buildable[R]] ensures that C has a static method newBuilder() that yields an object with methods add:(X)->Void and finalize()()->C@.

I am also investigating the use of “Self” as a special variable (within class definitions) that refers to the future class object. In this case, Self<<<C and Self@ (aka Self._TE) is the type of self (so Self@ is the “self type” for “binary methods” [2, 3, 6, 7]). In this approach, I define C# as the union of the instances of all extensions:

```
C# = Self<<<C | Self@  
```

Self also supplies access to static members (as Self.x_). Consider the following additions to the Iterable[T] class of (1.4):
abstract static const DefC[X] << Buildable[X];  
(1.5)

func map [R] (f:(T)→R) : Self.DefC[R]@
= mapTo[R](f,Self.DefC[R])

The first line declares an abstract static generic class object, DefC; any concrete class extending Iterable[T] must supply a value for DefC. For example, one could define class List[T] with DefC=List, while Set[T] is defined with DefC=Bag (not Set because the default shouldn’t lose information).

The last two lines of (1.5) define a method, map, which is implemented by calling mapTo with Self.DefC[R] as the class parameter. Notice that the return type refers to the special variable Self, so typing calls of map can be non-trivial.

Further discussion

This white paper described a few of my cooler research contributions. Please contact me for further details and check out my repository of technical notes [10].

References