Types as First-Class Values in Fuzion

Extended Abstract

Fridtjof Siebert
siebert@tokiwa.software
Tokiwa Software GmbH
Karlsruhe, Germany

Abstract

Using types as compile-time values provides possibilities for abstraction beyond what standard parametric types can offer. This extended abstract explains how the Fuzion language unifies the handling of type arguments and value arguments in calls and how types can be equipped with callable features. Furthermore, the use of types to distinguish multiple instances of algebraic effects is shown.

CCS Concepts: **• Software and its engineering → Data types and structures**; **Polymorphism**.

Keywords: Languages, Types, Algebraic Effects

1 Introduction

The goal of the Fuzion project [2, 3] is to provide an industrial strength general purpose high-level language with a particular focus on safety-critical applications and their validation and verification. For this, the language is designed to be simple and safe, but also to provide powerful high-level abstractions and a simple intermediate representation enabling tools for automatic code analysis and creation of certification artifacts such as tests or correctness proofs.

Fuzion combines functional and object-oriented aspects by unifying elements from both paradigms into the concept of a Fuzion *feature*. In particular, a feature in Fuzion can take the role of what is called a class, interface, package, trait, method, member, field, argument, generic, function, lambda, routine, tagged union, enum, type class, etc. in other languages.

Fuzion supports two forms of polymorphism: parametric polymorphism using monomorphization for efficient code generation and object-oriented style reference types with inheritance and dynamic dispatch at runtime.

Any *feature* in a Fuzion program is of one of five kinds: *routine*, *abstract*, *field*, *type parameter* or *choice*. A routine can be either a *constructor* that create a new instance when called or a *function* that returns a value of an arbitrary type. Features define algebraic types: constructors define product types while choices define tagged sum types. A feature declared as a constructor or as a choice type hence implicitly declares a type with the same name.

Features can be nested, i.e., routines and choice features may contain nested inner features. Abstracts provide no code, they cannot be called, but they provide a template for inheritance. Routines and abstracts may have formal arguments which are inner features of kinds field or type parameter. On a call, actual values and types are assigned to formal arguments.

Constructors may inherit from other constructor features, i.e., inherit the parent's inner features. Inherited functions can be redefined, inherited abstract features can be implemented by inner features of the heir feature. A constructor with abstract inner features can take the role of a Haskell type class: features inheriting from it that implement the abstract inner features can be used like instance of that type class.

Fuzion is statically typed, but it uses extensive type inference to avoid the need of explicit types. Unlike languages in the ML tradition, type inference does not start with functions to infer the types of actual arguments in a call. Instead, constructors define a type and call to a constructor creates a value of that type. As in most object-oriented languages, calls are for the form $t.f\ a_1\ a_2$, where the function f that is called must be declared as an inner feature of the constructor that defines t's type. Function result types can be omitted and inferred from the result expression, while feature argument types in many cases can be inferred from the actual arguments in a call[1].

Fuzion features are *pure*, non-functional aspects are encapsulated in algebraic effects using handlers [8, 9]. Effects are features that inherit from the base library feature `effect`. Effects provide operations as inner features that may encapsulate non-pure behavior. These operations, when called, may either resume or abort. An effect must be installed while code using its operations is executed. Installed effects form a stack for each running thread.

Fuzion features can be grouped into a Fuzion module that can be stored as a module file. The module forms the basis for tools processing library code.

Fuzion applications are created from a set of Fuzion modules. These modules are converted into the *Fuzion intermediate representation* that is used by the back end to create target code. The intermediate representation is also the basis for whole-program static analysis tools to work with.

In the following sections, it will be explained how Fuzion uses types to solve two important issues: First, parametric

[1]An example for where this is not possible is a library feature that is not called within the library itself needs to declare the argument types

types in object-oriented languages typically can only define functions that require an instance of that type to be called on, while it is often desirable to have functions that do not require such an instance. Fuzion provides a solution to this by associating features to types. Second, the use of several handler instances of an algebraic effect requires some naming mechanism to distinguish these instances. It will be explained how Fuzion uses types to this end.

2 Types as Values

Fuzion features can have type parameters very similar to generics supported by many languages[4]. However, the syntax for the declaration of type parameters is essentially the same as that of value arguments. As an example, the following code shows the definition of a Fuzion feature `pair` that contains two values of the same type.

```
# a pair of two values a, b of the same type T
pair(T type,
     a, b T)
is
  redef as_string =>
    "pair of type $T: $a, $b"
```

Type parameters can be used as targets of a call, in this example, the feature `as_string` is called to create a text representation of type parameters `T` and value arguments `a` and `b`[2]. Type parameters can be assigned to fields and used to define new types that, e.g, may be passed as type parameters in calls.

`pair` is a constructor, i.e., it defines a type with the same name, `pair`. On a call, it creates one instance of that type. The following code creates three instances assigning different types and values to the formal arguments:

```
p1 := pair i32 47 11
p2 := pair String "Hello" "World!"
p3 := pair (option String) nil "xyz"
```

Actual values for type parameters can be inferred, so the code for `p1` and `p2` can be simplified as:

```
p1 := pair 47 11
p2 := pair "Hello" "World!"
```

In contrast, the arguments to the third call to `pair` are of types `nil` and `String` which are not compatible with one another and different to desired choice type `option String`. Consequently, that type cannot be inferred automatically, an attempt to use type inference would result in a compile-time error.

When a constructor or choice name is used as a type, actual type parameters are required and value arguments are omitted. An example is the following code that uses types `pair i32` and `pair (option String)` as types for the formal argument `p`.

```
# add the elements in a pair of i32 values
add(p pair i32) => p.a + p.b

# from pair of option String, get `a` as String
first_string(p pair (option String)) =>
  # `option String` is a choice of `nil` or
  #  `String`, pattern match it
  match p.a
    nil =>  "**error**"
    s String => s
```

3 Type Features

Every Fuzion constructor feature f defines a type t. Values created by the constructor f can be assigned to argument fields of that type t in a call. Similarly, the type t can be assigned to type parameters arguments in a call.

The type of this type value t is defined implicitly: Every constructor feature f implicitly declares a *type feature* f_t. Values of f_t correspond to types and are assigned to type parameters in calls.

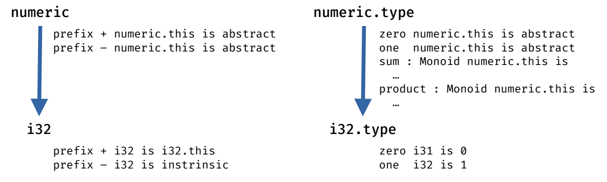
Type features can define inner features that can be called via a given type parameter.

An example from Fuzion's base library is `numeric` that defines abstract features like *infix +*. `numeric` is used as the parent of concrete numeric types `i32`, `f64` or `fraction` (Figure 1). `numeric` defines `zero` and `one` as inner functions of `numeric`'s type feature:

```
numeric is
  . . .
  type.zero numeric.this is abstract
  type.one  numeric.this is abstract
```

Just like many other features of `numeric`, these type features are abstract and have to be implemented for each concrete feature that inherits from `numeric`. For this to be possible, the type features form an inheritance graph parallel to the inheritance graph of the underlying features.

Note that the result type for `zero` and `one` is `numeric.this`, which is a placeholder for the actual heir type that implements these type features.



```
prefix + numeric.this is abstract
prefix - numeric.this is abstract

prefix + i32 is i32.this
prefix - i32 is instrinsic

zero numeric.this is abstract
one  numeric.this is abstract
sum : Monoid numeric.this is
  …
product : Monoid numeric.this is
  …

zero i31 is 0
one  i32 is 1
```

Figure 1. *Partial inheritance graph for numeric and the corresponding type feature.*

[2]Within a string literal, v is short-hand for inserting the result of the call *v.as_string* at the given position.

A feature such as i32 that inherits from numeric can provide implementations of these type features as follows.

```
i32 : numeric is
  . . .
  fixed type.zero i32 is 0
  fixed type.one  i32 is 1.
```

These features are marked as fixed to prevent passing this implementation down to heirs of i32. Those heirs would have to provide their own implementation with corresponding result type.

We can call these type features in generic code. Here is an example of a function that calculates the sum of the elements of a list of any numeric type:

```
sum(T type : numeric, l list T) =>
  match l
    nil    => T.zero
    c Cons => c.head + sum c.tail
```

In case the list is empty, the value returned by the actual numeric type's zero feature will be returned as the result of sum. This mechanism is used in Fuzion's base library to provide monoids sum and product for all numeric types using one and zero defined for numeric.type to implement the identity element e:

```
numeric is
  . . .
  # monoid of numeric with infix + operation.
  type.sum : Monoid numeric.this is

    # associative operation
    infix • (a, b numeric.this) => a + b

    # identity element
    e => zero

  # monoid of numeric with infix * operation.
  type.product : Monoid numeric.this is
    infix • (a, b numeric.this) => a * b
    e => one
```

4 Types to Identify Effects

Fuzion uses algebraic effects to support non-functional aspects such as I/O, mutable data, aborting operations by throwing exceptions, etc. Effects can be seen as capabilities that are required to execute code that requires the presence of a given effect in the current execution environment. This is similar to the Effekt language that regards effects as requirements[10]. For convenience, common effects in Fuzion such as io.out have default handlers that are installed automatically for programs that do not install a required handler explicitly.

An effect is essentially a set of operations encapsulated in a feature. The type of that feature is used to identify the effect. E.g., a simple example is the following *Hello World* code using the io.out effect:

```
hello ! io.out is
  io.out.env.println "Hello World"
```

The feature hello requires the io.out effect to be executed. This is documented using ! io.out following the feature name, where the effect is identified by its type io.out.

Within the feature implementation, the current effect instance can be accessed using the effect type followed by .env as in io.out.env.println in this example.

Fuzion permits installation of different handlers for an effect of a given type. E.g., we could install a handler for io.out that, instead of using stdout, would print to a log file. If hello was called with that handle installed, the access io.out.env would divert the output to the log file. The installed handlers form a stack, such that there is at most one current handler for each effect type.

Often, we might not only want to choose a particular handler for an effect, but we would like to work with several instances derived from the same effect simultaneously. So we need a mechanism to distinguish these. Fuzion uses types to do this as follows:

4.1 Using types to distinguish effect instances

Assume we want to implement a function that traverses a list and, as a side-effect, counts the number of even values found in the list. We could do this using a mutable variable as follows.

```
count_even(l Sequence i32) i32 ! mutate is
  res := mutate.env.new 0
  l.for_each (
    x -> if x %% 2 then res <- res.get + 1)
  res.get
```

Here, we use the mutate effect in the current environment to create a new mutable field res. This works, but unfortunately, this feature requires the mutate effect even though it does not perform any state changes to the system, all it does is using an auxiliary mutable variable res that is discarded when this function returns, the mutation is fully encapsulated in an otherwise pure function.

It would help if we were able to define a local instance of mutate that would permit us to create a mutable auxiliary variable but that would also document that this feature is effectively pure.

We can do this by defining a new type local_mutate that inherits from mutate, execute our code using this variant of the mutate effect and create auxiliary mutable variables locally using this effect.

The following code illustrates how this can be done:

```
count_even(l Sequence i32) i32 is
  local_mutate : mutate.
  local_mutate.go ()->
    res := local_mutate.env.new 0
    l.for_each (
      x -> if x %% 2 then res <- res.get + 1)
    res.get
```

The line `local_mutate : mutate.` defines a constructor feature that inherits from `mutate`. Since constructor features implicitly define a type, this defines the new type `local_mutate` that inherits all the operations from the `mutate` effect. One instance of this new effect is then installed. This instance is accessed via its type `local_mutate.env`, it is used to create and update the local mutable variable `res`. The surrounding feature `count_even` remains pure.

Since operations on effects are accessed via their type in an expression `effect_type.env.operation`, and types can be type parameters, it is now possible to pass effect types as type parameters in calls. As an example, the Fuzion base library provides a feature `Mutable_Linked_List` that expects a type parameter `LM type : mutate` that may be any type derived from `mutate`. This permits simultaneously using several instances of mutable linked lists that use different versions of mutate enabling fine-grain control of local mutability.

5 Implementation

The Fuzion toolchain uses two intermediate steps: First, Fuzion source code is translated into Fuzion modules that essentially contain a set of feature declarations.

Second, an application is created from a set of these modules that are compiled into the Fuzion intermediate representation. In this representation, Fuzion features with type parameters are specialized for the actual types that are assigned to their type parameters. Finally, the back end creates code from this intermediate representations. Currently, there are two back ends, one creating C source code and an interpreter implemented in Java.

Type values in the intermediate representation are unit type values, i.e., they do not contain any data and no code is generated for an assignment of a type value.

Only when assigned to a reference type like `Any`, which is Fuzion's generic object reference type, these values get boxed into a heap allocated reference that can be used, e.g, to convert the type to a string using `as_string`.

The specialization of code for actual type parameters and the fact that type values are unit type values imply that this is a zero-cost abstraction, there is no code needed for assigning type values nor for any dynamic lookup when calling inner features defined in type features.

6 Related Work

Scala has a similar goal to Fuzion of unifying functional and object-oriented approaches[7]. Läufer and Odersky suggested explicit type variables by incorporating first-class abstract types as an extension of algebraic data types[5]. Moors et al present a solution to remove limitation of type parameters by allowing type constructors as type parameters and avoiding the additional boilerplate using Scala's implicits[6].

The challenge of escaping of named effects has been addressed by previous work[1, 12]. Xie et all recently proposed to treat effect handler names as first-class values[11].

The syntax of `comptime` types in Zig[13] is similar to that of Fuzion type parameters and their treatment as purely compile time values results in specialized code similar to the monomorphization perfomed for Fuzions intermediate code. Zig's `comptime` mechanisms, however, has a different goal of providing a a high-level macro mechanism.

7 Conclusion

Fuzion unifies different concepts in programming languages. Types in Fuzion are unit type values defined by implicitly generated type features. These type features are counterparts of constructor features. They allow the definition of inner features relative to a type. Type values can be passed as type parameters in calls where they can not only be used to define new types, but also as the target of calls that are specific to that type. This increases the power of generic functions without imposing a runtime cost.

Furthermore, types in Fuzion are used to name effects and can be used to create effects locally that permit safely limiting the scope of these effects.

Acknowledgments

Thanks to the anonymous reviewers for helpful comments and related work that I was unaware of. Also thanks to the Fuzion team at Tokiwa Software GmbH for providing important ideas and feedback.

References

[1] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (dec 2019), 29 pages. https://doi.org/10.1145/3371116

[2] Fridtjof Siebert et al. 2023. Fuzion GitHub Repository. https://github.com/tokiwa-software/fuzion

[3] Fridtjof Siebert et al. 2023. Fuzion Portal Website. https://flang.dev

[4] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. https://doi.org/10.1145/503502.503505

[5] Konstantin Läufer and Martin Odersky. 1994. Polymorphic Type Inference and Abstract Data Types. *ACM Trans. Program. Lang. Syst.* 16, 5 (sep 1994), 1411–1430. https://doi.org/10.1145/186025.186031

[6] Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a Higher Kind. *SIGPLAN Not.* 43, 10 (oct 2008), 423–438. https://doi.org/10.1145/1449955.1449798

[7] Martin Odersky and Tiark Rompf. 2014. Unifying Functional and Object-Oriented Programming with Scala. *Commun. ACM* 57, 4 (apr 2014), 76–86. https://doi.org/10.1145/2591013

[8] Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied categorical structures* 11 (2003), 69–94.

[9] Gordon D Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Logical methods in computer science* 9 (2013).

[10] The Effekt research team. 2023. Effekt Language — Effect Safety. https://effekt-lang.org/docs/concepts/effect-safety

[11] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-Class Names for Effect Handlers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 126 (oct 2022), 30 pages. https://doi.org/10.1145/3563289

[12] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-Safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (jan 2019), 29 pages. https://doi.org/10.1145/3290318

[13] Zig Software Foundation. 2023. Zig Compile-Time Concept. https://ziglang.org/documentation/master/#Introducing-the-Compile-Time-Concept