Fuzion – Safety through Simplicity

Dr. Fridtjof Siebert
Tokiwa Software GmbH, Karlsruhe, Germany; email: siebert@tokiwa.software

Abstract

Fuzion is a modern, general purpose programming language that unifies concepts found in structured, functional and object-oriented programming languages into the concept of a Fuzion feature. It combines a powerful syntax and safety features based on the design-by-contract principle with a simple intermediate representation that enables powerful optimizing compilers and static analysis tools to verify correctness aspects.

Fuzion maps different concepts into the concept of a Fuzion feature and uses a simple intermediate language that is friendly for static analysis tools as well as for optimizing compilers.

Fuzion was influenced by many other languages including Java, Python, Eiffel, Rust, Ada, Go, Lua, Kotlin, C#, F#, Nim, Julia, Clojure, C/C++, Scala, and many more. The goal of Fuzion is to define a language that has the expressive power present in these languages and allow high-performance implementation and powerful analysis tools. Furthermore, Fuzion addresses requirements for safety-critical applications by adding support for contracts that enable formal specification and detailed control over runtime checks.

Keywords: *safety, static analysis, programming language, certification*

1 Introduction

Many current programming language are getting more and more overloaded with new concepts and syntax to solve particular development or performance issues. Languages like Java/C# provide classes, interfaces, methods, packages, anonymous inner classes, local variables, fields, closures, etc. And these languages are currently extended further by the introductions of records/structs, value types, etc. The possibility for nesting of these different concepts results in complexity for the developer and the tools (compilers, VMs) that process and execute the code.

For example, the possibility to access a local variable as part of the closure of a lambda expression in Java may result in the compiler allocating heap space to hold the contents of that local variable. Hence, the developer has lost control over the allocation decisions made by the compiler.

In Fuzion, the concepts of classes, interfaces, methods, packages, fields and local variables are unified in the concept of a Fuzion feature. The decision where to allocate the memory associated with a feature (on the heap, the stack or in a register) is left to the compiler just as well as the decision if dynamic type information is added or not. The developer is left with the single concept of a feature, the language implementation takes care for all the rest.

2 Fuzion Features

A Fuzion applications consists of nested feature declarations. The main operation performed on a feature is a feature call.

2.1 Components of a Feature Declaration

Feature Name A Fuzion feature is identified by a name, similar to the name of a class or a function in other languages. The name may be a plain identifier, or an operator such as *infix* + or *postfix !*, the difference between calling a named feature or applying an operator is purely syntactical.

Formal Arguments Features may have formal arguments, which are themselves features implemented as fields. On a call to a feature with formal arguments, actual arguments have to be provided to the call.

Result Type The result of a feature call is an instance of the feature. Alternatively, a feature may declare a different result type, then it must return a value of that type on a call.

Parametric Types Features may have type parameters.

Closure Features are nested, i.e., every feature is declared within the context of an outer feature. The only exception is the universe, which is the outermost feature in any Fuzion system. A feature can access features declared in its outer features. This means, a feature declaration also declares a closure of the feature and its context.

Inheritance Clause Fuzion features can inherit from one or several other features. When inheriting from an existing feature, inner features of the parent become inner features of the heir. Inherited features can be redefined. In particular, when inheriting from a feature with abstract inner features, one can implement the inherited abstract features.

Inheritance and redefinition in Fuzion does not require dynamic binding. By default, types defined by features are value types and no runtime overhead for dynamic binding or heap allocation is imposed by inheritance.

Contract A feature may declare a contract that specifies what the features does and under which conditions the feature may be called [1]. This will be handled in more detail below in the section *Design by Contract*.

Implementation Features are implemented as one of

- a routine providing code that is executed on a call,
- a field providing a memory slot that stores a value and whose contents are returned on a call,
- an abstract feature with no implementation and that cannot be called directly, but that can be redefined,
- a choice defining a union type, or
- an intrinsic implemented by the compiler or interpreter.

A feature implemented as a routine or as a choice can contain inner feature declarations.

2.2 Feature Example

Here is an example that declares a feature *point* that is similar to a struct or record in other languages:

```
point(x, y i32) is { }  # declare point as a feature with 2 args
p1 := point 3, 4        # create instance of point
```

2.3 Features as Types

A feature declaration implicitly declares a type of its instances. In the example above, the feature declaration for *point* declares the type *point* that can be used to declare a field, so we could, e.g., declare a new feature that takes an argument of type *point*:

```
draw(p point) is
  drawPixel p.x, p.y
```

Features implemented as a routine define a product type whose components are the types of the fields defined for that routine.

3 Specific Language Features

3.1 Loops

Fuzion has a powerful syntax for loops. Nevertheless, loops are syntactic sugar that is translated into feature declarations and tail recursive calls [2]. Loop index variables are automatically immutable and analysis of loop code is simplified.

3.2 Tuples

Tuples in Fuzion are provided by a generic standard library feature *Tuple*. The open list of generic parameters specifies the types of each element and their number. Here is an examples of a feature that splits a 16-bit unsigned integer *v* into two bytes returned as a tuple:

```
bytes(v u16) => ((v >> 8) & 255, v & 255)
```

The tuple can be unpacked on a call to *bytes*:

```
(hi, lo) := bytes(12345)
say "Bytes:␣$hi␣$lo"
```

Similar to the type defined for a routine, a tuple defines a product type, but without giving it an explicit name.

3.3 Choice Types

Fuzion provides choice types (also called tagged union or variant types). The simplest example of a choice type is the type *bool*, which is a choice between types *TRUE* and *FALSE*. *TRUE* and *FALSE* are themselves declared as features with no state, i.e., no fields containing any data.

Another example for a choice type from the standard library is *Option<T>*, which is a generic choice type that either holds a value of type *T* or *nil*, while *nil* is a feature with no state declared in the standard library.

A match statement can be used to distinguish the different options in a choice type, e.g.,

```
mayBeString Option<string> = someCall()
match mayBeString
  s String  => say s
  _ nil     => say "no␣string"
```

Together with the product types provided by tuples or routines, Fuzion provides algebraic data types.

3.4 First-class Functions

Fuzion offers first-class functions (lambdas) and maps these to Fuzion features that inherit from a standard library feature called *Function*.

4 Design by Contract

Fuzion features can be equipped with pre- and post-conditions to formally document the requirements that must be met when a feature is called and the guarantees given by a feature. An example is a feature that implements a square root function for 32-bit integers:

```
sqrt(a i32) i32
  pre
    a >= 0
  post
    # result^2 shall be less or equal toa
    result * result <= a,
    # (result+1)^2 shall be larger than a, may overflow i32
    (result + 1) > a / (result + 1),
    # result shall not positive
    result >= 0
is
  if a == 0
    0
  else
    # iteratively calculate root
    for
      # start iteration with 1
      r := 1, next
      # next in iteration is middle of r, a/r
      next := (r + a/r) / 2
    until r == next
      r
```

In this case, the function defines the pre-condition that its argument *a* is non-negative. A call of this function with a negative value will result in a runtime error. On the other hand, its post-conditions make a clear statement about the result: The result will be the largest value that, when squared, is <= a.

4.1 Checking Pre- and Post-conditions

Pre- and post-conditions can be classified for different purposes. Default qualifiers provided in the standard library are

safety protects pre-conditions that are required for the safety of an operation.

An example is the index check pre-condition of the intrinsic operation to access an element of an array: Not performing the index check would allow arbitrary memory accesses and typically would break the applications safety.

This qualifier should therefore never be disabled unless you are running code in an environment where performance is essential and safety is irrelevant.

debug is generally for debugging, it is set if debugging is enabled.

debug(n) is specific for enabling checks at a given debug level, where higher levels include more and more expensive checks.

pedantic is for conditions that a pedantic purist would require, that more relaxed hacker would prefer to do without.

analysis is used for conditions that are generally not reasonable as runtime checks, either because they are prohibitively expensive or even not at all computable in this finite universe. These conditions may, however, be very useful for formal analysis tools that do not execute the code but perform techniques such as abstract interpretation or formal deduction to reason about the code.

Runtime checks for pre- and post-conditions can be enabled or disabled for each of these qualifiers. This gives a fine-grain control over the kind of checks that are desired at runtime. Usually, one would always want to keep safety checks enabled in a system that processed data provided from the outside to avoid vulnerabilities such as buffer overflows. However, in a closed system like a controller of a launcher, it might make sense to disable checks if a runtime error would mean definite loss of the mission, while an unexpected intermediate value may still result in a useful final result of a calculation.

5 Immutability in Fuzion

Fuzion encourages the use of immutable data by simple syntax for the declaration of immutable fields. Also, the use of tail calls for loops automatically converts index variables used in that loop into immutable variables with a life span of a single loop iteration.

Since immutability is essential to ensure correctness of parallel execution within threads that do not rely on locks or similar synchronization mechanisms, Fuzion's analyzer will verify that data shared between threads is immutable.

6 Memory Management in Fuzion

Fuzion to a large extend relies on static analysis to reduce memory management overhead. Instances are by default value instances that do not require heap allocation. Furthermore, immutability in many cases avoids the need to keep a shared copy on the heap. For dynamic calls, heap allocation and dynamic binding overhead is avoided by specialization of calls.

Only for those instances for which all of these optimizations would fail, in particular instances shared between threads or long-lived instances with mutable fields, heap allocation will be required. Memory allocated on the heap will be reclaimed by a real-time garbage collector [3].

7 Fuzion Implementation

The Fuzion implementation consists of several, independent parts from a front-end performing parsing and syntax-related tasks that creates the intermediate representation (IR), via a middle-end that collects the modules needed by a Fuzion application, to the analyzers, optimizers and several backends.

7.1 Fuzion IR

Fuzion has a very simple intermediate representation. The dominant instruction is a call. The only control structure is a match operation. Loops are replaced by tail recursive calls, so there is no need in the compiler or analysis tools to handle loops as long as (tail) recursion is supported and optimized.

Clazzes in the Fuzion IR When creating the intermediate representation, Fuzion features from the source code are instantiated with actual type parameters. This are referred to as Fuzion *clazzes* (sic). Any analysis tool or compiler working on the IR hence does not need to handle parametric types.

Clazzes come in one of five flavors depending on their underlying feature:

- *Routine* – 'normal' Fuzion feature with code to execute.
- *Field* – a feature that can store data.
- *Abstract* – an abstract Fuzion feature
- *Choice* – a Fuzion feature that defines a union type.
- *Intrinsic* – a feature implemented by the compiler.

All clazzes except *choices* may be equipped with contracts.

Instructions in the Fuzion IR A clazz of type routine contains code that consists of very simple (bytecode-) instructions using a simple stack machine. There are only eight intermediate instructions:

- *Assign* – an assignment to a field
- *Box* – boxing a value to a (heap-allocated) reference type
- *Call* – a call to a clazz.
- *Current* – the current instance
- *Const* – a constant value of primitive of compound type.
- *Match* – a match instruction to determine the type of a value of a choice type
- *Tag* – create an instance of a choice type from a value whose type is one of the type parameters of the choice.

- *Pop* – remove value from the execution stack, used when call result is ignored.

The Fuzion IR can be stored in a file called a Fuzion Application (*fapp*) such that the IR can be applied to a variety of different following steps for static analysis, code generation, dynamic analysis (code coverage), optimizers, etc.

7.2 Fuzion Analyzer

Static analysis on the Fuzion IR will be used to ensure different aspects of correctness of the application. Tasks for the analyzer include

- field initialization: ensure that fields are initialized before accessed. Usually, this is ensured by the Fuzion syntax, but calls in a routine's body could result in accesses to uninitialized fields (as final fields in Java [4]).
- thread safety: ensure that fields accessed by different threads are immutable when these accesses occur to achieve thread-safety similar to Rust [5].
- module safety: for a library module to be safely usable in different context, static analysis should ensure that no external call could access undefined or intermediate state.
- ensure that contracts are satisfied. In particular, preconditions of intrinsics that are qualified with *safety* such as indexed accesses to arrays could be analyzed statically [6].
- ensure higher level of correctness defined in contracts qualified with *analysis* using quantors (using tools like KeY [7]).

Currently, the analyzers are still work on in progress, only the basic analysis required by the back ends is done.

7.3 Fuzion Optimizer

The Fuzion Optimizer modifies the intermediate representation of a Fuzion application. In particular, it determines the life spans of values to decide if they can be stack allocated or need to be heap allocated and it specializes feature implementations for the actual argument types and the actual generic arguments provided at each call.

This means that runtime overhead for heap allocation and garbage collection will be avoided as much as possible, most values can be allocated on the runtime stack. Additionally, runtime type information such as vtables will be required only in very few cases where dynamic binding cannot be avoided. An example is a data structure like a list of some reference type with elements of different actual types that are stored in this list.

7.4 Fuzion Back-Ends

Fuzion currently has two back-ends: An interpreter written in Java running on OpenJDK and a back-end creating C source code processed by gcc or clang. It is planned to add further back-ends, in particular for LLVM and Java bytecode.

Thanks to the very simple intermediate code, the back ends are relatively simple. The core of the C backend, e.g., consists of less than 1000 lines of well documented Java code.

7.5 Certification Artifacts

Due to the simple structure of the backend it is expected that certification of generated code will be simplified. The IR removes syntactic sugar, but should remain simple enough to allow for manual analysis and creation of certification artifacts. Furthermore, it can be expected that qualification of a backend for use in safety-critical systems will be simpler compared to other languages.

8 Conclusion / Next Steps

The Fuzion language definition and implementation is still in an early stage. A first version of the language and its implementation was presented at FOSDEM 2021 [8]. The main areas of ongoing work are the definition of a standard library, interfaces to other languages, IDE integration and documentation. I expect the presented approach to have a huge potential, in particular in the safety-critical domain. The combination of a powerful language with a simple core that is open for static analysis tools can improve the software development productivity, the safety of the resulting software as well as the performance.

9 Availability

Fuzion is available as source code on github [9]. The Fuzion portal website *flang.dev* [10] provides an interactive tutorial with many examples, a tutorial and many background documents on the design of Fuzion.

References

[1] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, p. 40–51, Oct. 1992.

[2] W. D. Clinger, "Proper tail recursion and space efficiency," *SIGPLAN Not.*, vol. 33, p. 174–185, May 1998.

[3] F. Siebert, "Concurrent, parallel, real-time garbage-collection," in *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, (New York, NY, USA), p. 11–20, Association for Computing Machinery, 2010.

[4] "Use of uninitialized final field - with/without 'this.' qualifier." `https://stackoverflow.com/questions/13864464/`, 2012.

[5] S. Klabnik and C. Nichols, *The Rust Programming Language: Ch. 16 Fearless Concurrenty*. USA: No Starch Press, 2018.

[6] R. Rugina and M. C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," *ACM Trans. Program. Lang. Syst.*, vol. 27, p. 185–235, Mar. 2005.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of Object-Oriented Software: The KeY Approach*. Berlin, Heidelberg: Springer-Verlag, 2007.

[8] "FOSDEM." `https://fosdem.org`, 2021.

[9] "Fuzion Souces." `https://github.com/fridis/fuzion`.

[10] "Fuzion Portal." `https://flang.dev`, 2021.