The Fuzion Intermediate Representation

An IR for static analysis of safety-critical systems

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

Michael Lill
michael.lill@tokiwa.software
Tokiwa Software GmbH
Karlsruhe, Germany

Abstract

Fuzion is a new language targeting safety-critical systems by building on few but powerful concepts and enabling static analysis of applications to verify their correctness. The Fuzion Intermediate Representation (FUIR) plays a key role in providing a basis for static analysis tools as well as interpreters and code generators that produce runnable programs.

FUIR has a number of aspects that are different from most other intermediate representations: Code is grouped into Fuzion features that are a common abstraction for functions, classes, types, etc. There are only 10 commands used in FUIR, calls to intrinsic features are used for operations not present as explicit commands. There is no support for loop or conditional jumps, recursion is used instead. The usual distinction between stack frames and heap instances does not exist in the FUIR.

This work-in-progress paper will give a quick overview of the Fuzion language and explain the aspects of the FUIR in more detail before presenting the impact it has on the tools processing the FUIR code.

CCS Concepts: • **Software and its engineering → Compilers**; **Software verification**; **Formal software verification**.

Keywords: Languages, Intermediate Language, Static Analysis, Safety-Critical Systems

ACM Reference Format:
Fridtjof Siebert and Michael Lill. . The Fuzion Intermediate Representation: An IR for static analysis of safety-critical systems. In *Proceedings of (VMIL'24).* ACM, New York, NY, USA, 6 pages.

1 Introduction

Fuzion is a work-in-progress project to design a new high-level language and toolchain targeting safety critical applications [14]. It provides a novel way that combines aspects of object-oriented and functional programming paradigms in a coherent and simple way. Fuzion is *pure* in the sense that no *public routine* will have any non-functional side-effects unless explicitly declared.

The Fuzion Intermediate Representation (FUIR) is being designed not only as a source language for execution and code generation, but also as the basis for static analysis tools to statically analyze the code to verify different correctness aspects like the absence of runtime faults, the absence of data flow that could result in a leak of sensitive data or resource constraints (worst-case execution time or memory usage) during execution.

1.1 Goals for the IR Design

The goals for FUIR are derived from the intended application for the development of safety-critical systems in very different domains such as industrial control, automotive, avionic, medical devices, critical infrastructure, etc.

The problems to be solved by the IR are:

1. provide a static, monomorphized representation of a whole Fuzion application.
2. serve as input for code generator back ends targeting different languages such as JVM bytecode, LLVM [6], C-code, etc.
3. serve as input for direct code execution by virtual machines using interpretation or JIT-compilation.
4. provide a basis for static analysis for application correctness to, e.g.,
   - proof the absence of runtime errors like failed *pre* or *post*-conditions [11] or assertion *checks*,
   - verify timing such as finding worst-case execution times,
   - verify resource limits such as heap or stack memory usage,
   - integrate with formal proof assistants like Isabelle [7].
5. permit static analysis to enable local and whole-program optimizations
6. enable simple, certified code generators to facilitate safety certification of generated binary code, e.g., for airborne systems [1].

These goals result in the general requirement of having a simple IR that permits generic abstract interpreters to be built that form the basis for different static analysis, verification and code generation tools.

1.2 Organization of this Paper

Section 2 will give a dense overview of the Fuzion language, which will help understand some of the design decisions for the intermediate representation before section 3 gives an overview of the Fuzion toolchain using the intermediate

VMIL'24, October 20, 2024, Pasadena, CA, USA
2024.

representation presented in section 4. The implementation of tools currently using the IR is explained in 5 before some related work in section 6 and our conclusions in section 7 outline future work.

2 Fuzion Language Introduction

This section gives a condensed and recursive overview of the Fuzion language and an introduction to its terminology, a tutorial is available online [17].

2.1 Building Block: Feature

The building blocks of Fuzion applications are *feature* declarations. There are different kinds of *features*, the most common kinds are *field* and *routine*. A *field* is an immutable variable of a statically fixed *type* that is initialized with a value calculated from an *expression*. A *routine* is a callable *feature* with formal *arguments* and code given as an *expression*. A *routine* may be one of two kinds: a *function* that results in the value produced by evaluation of its *expression* or a *constructor* that defines a product type consisting of its inner *fields*. The *arguments* of a *routine* themselves are *features*, they may be *type parameters* or *fields*.

2.2 Types

Fuzion's *types* fall into two categories, product types that are defined by a *constructor* and *sum types* that are defined by *choice* features. A *choice* feature defines a tagged union type (*choice* type) of its *type parameters*, complementary to a *constructor* feature that defines a product type of its inner *fields*. Unlike *constructor* features, a *choice* feature may not be called in an *expression*. An instance of a *choice* type is created by an assignment of a value whose type is one of the *choice's* type parameters to a field of the *choice* type.

2.3 Expressions

An *expression* is code that can be evaluated to produce a result value of the expression's *type*. Any code that is executed must therefore produce a result value when it returns, but there are types like *unit* for values that do not contain any information or *void* to indicate that the expression does not return[1].

The most important expression is a *feature call* to a *routine*. On a *call*, actual values are assigned to the *routine's* formal *arguments*: For *arguments* that are *type parameters*, the actual values must be *types*, while for *argument fields* corresponding *expressions* must be given. The result of a call to a *function* is the value of the *function's expression*, while the result of a call to a *constructor* is the instance of the product type defined by that *constructor* with *arguments* set to the actual *types* and values and inner fields initialized to their *initial values*.

The only expression that permits conditional code is a *match* that takes an expression that evaluates to an instance of a *choice type*. Depending on the original type stored in the choice, evaluation proceeds with one of several expressions.

Finally, Fuzion expressions allow nested declarations described in the next sub-section:

2.4 Nested Features

A feature declaration itself is a Fuzion expression that results in a unit type result value. This permits fields to be nested within *routines*, but also permits the nesting of *routines*. An inner *routine* may access features declared in all of its outer *routines*. On a call to a *routine*, a reference to the outer instance is passed as an implicit argument.

2.5 Inheritance and Dynamic Binding

Constructors may inherit from other *constructor* features by adding *calls* to these *parents* in the declaration. As a result, the *child* inherits the *parents'* inner features, which the child may *redefine*.

Since Fuzion uses value semantics, using inheritance and redefinition does not require dynamic binding. However, a constructor may be defined as a *reference* type. If this is the case for the parent, the child becomes assignable to fields of the parent type and dynamic binding will be used.

2.6 Algebraic Effects

Fuzion *routines* are pure, i.e., their result depends only on the values of the actual arguments including the implicit outer instance. The only means to perform state changes or to interact with the outside world is via algebraic effects. These are features that inherit from a base feature *effect* and add effect *operations* as inner features. Effects can be *instated* to run code that can access the effect's operations in its environment. Static analysis is used to verify that all effects required for certain code are actually instated in the code's environment.

2.7 Syntactic Sugar

Fuzion uses extensive syntactic sugar to provide a more human-readable syntax for common code patterns.

Conditionals. of the form *if-then-else* are internally handled like *match* expressions. This is possible since type *bool* in Fuzion's base library is defined as a *choice type* of unit types *FALSE* and *TRUE*.

Loops. are supported via a powerful syntax that is, internally, mapped to tail-recursive calls[2] and *match* expressions.

Type inference. is used extensively in the front end such that —even though Fuzion is statically typed— types can be omitted in most cases.

[1] e.g., as the result type of a call to *panic* that aborts with an error

[2] that will be optimized by the backends

2.8 Information Hiding

Visibility of features and types can be restricted as *private* (same source file, default), *module* (same module) and *public*.

2.9 Code Example

Here is a small example that declares a feature *example* that requires the *io.out* effect. As inner features it declares a product type *point* that combines two values of type *f64* and has an inner *function d* that calculates the distance to the origin.

A polymorphic *function add* is declared that adds three numeric values whose type is given by a type parameter *T*.

```
example ! io.out is

  point(x, y f32) is
    d => f32.sqrt x*x+y*y

  p := point 3 4
  say p.d

  add(a, b, c T : numeric) => a + b + c
  say (add 3 4 5)
  say (add 3.141 2.718 1.141)
```

3 Fuzion Toolchain

The Fuzion toolchain (Fig 1) starts by compiling a set of Fuzion source files **.fz* into a Fuzion module *name.fum*. Modules may depend on other modules and compilation is done against pre-compiled modules. The front end phase checks that the source code respects the language validity rules that include type checks, visibility rules, etc.

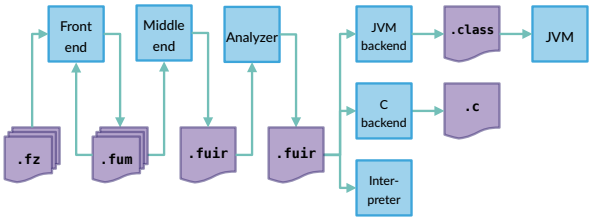

Figure 1. *Fuzion toolchain and intermediate data.*

Module files have unique version numbers, any change or recompilation of one module requires recompilation of all modules that depend on that module. There is hence no need for mechanisms to detect incompatible changes at link or load time as in other languages[3].

The middle end then builds an application *name.fuir* from a main module that defines a main feature plus all the modules the main module depends on. The middle end performs monomorphization, i.e., all type parameters are replaced by actual types, features called with different type parameters are specialized for all combinations of type parameters that are used in the application.

Consequently, the intermediate code used for the application is fairly simple, all types except runtime types of reference values are known. Whole program static analyzers can now process the application that is represented using the Fuzion intermediate representation.

Finally, the intermediate representation is used by one of the Fuzion backends to produce executable code. Currently, three backends are implemented: a JVM bytecode generator, one back end that creates C source code to be processed by clang/llvm or other C toolchains and an interpreter that directly executes the intermediate code.

4 Intermediate Representation

The exact specification of the FUIR format is work-in-progress. In many parts, it will be similar to the Fuzion module file format that is specified online [18]. This section gives an overview of the general features of the format followed by a description of the contents.

4.1 Binary format using offsets

Both, the Fuzion Module File Format and the Fuzion Intermediate File Format are binary formats. For references between files and within one file, byte offsets are used. This permits efficient accesses between files, but it requires strict versioning of files. A modification in any module file requires recreation of all files that directly or indirectly depend upon that file.

This has two important consequences: Names of features or types are effectively irrelevant and present only for human interaction purposes. Furthermore, modifications cannot result in runtime problems like Java's *IncompatibleClassChangeError* [8] since such modifications are strictly forbidden without rebuilding of all dependent files. File hashes are used to enforce this.

Another aspect that is taken into account when designing the file format is that related data is kept close together. In particular, inner declarations like the argument fields of a routine are stored together while information that might not be needed at all in the common case like the source code context to be used in error messages is kept towards the end of the files. The intend is tp speed up processing by permitting fast memory-mapping and direct use of the important parts of a file while avoiding to even load parts that end up not being used at all.

4.2 IR contents

The Fuzion Intermediate Representation (FUIR) is a collection of monomorphic features created by a data-flow analysis over a Fuzion application that detects all runtime types that could be used by the application. Unlike Fuzion module files

[3] Java produces an *IncompatibleClassChangeError* in some cases, *C* could fail during linking or crash at runtime in this case.

that define features that are open to be used with different type parameters and that permit new children to inherit from their features, the FUIR is fairly static: All types and children are known and fixed.

4.2.1 Monomorphic Features. Features fall in one of the following categories

- *routine* — a feature containing executable code. A routine may define inner features, including its argument fields. Routines define a product type of the types of its direct inner fields, a call to a routine allocates an instance of this type. In case the routine is a function, the fields include an implicit result field. Otherwise, the routine is a constructor whose result is the instance of its type. A flag indicates if the type of a routine is a *value* type or a *ref* type.
- *abstract* — a feature similar to a routine but without executable code that cannot be called at runtime.
- *field* — inner features of routines or abstract features can be fields. Fields have a result type.
- *choice* — a feature containing no fields nor executable code that defines a tagged sum type of a fixed number of choice types. A choice may contain inner feature declarations except for direct inner fields.
- *intrinsic* — a function with arguments and a result type that does not contain explicit code, but implicit code created by the execution engine.
- *native* — a function that provides a way to call foreign language functions implemented in other languages.

4.2.2 Types. The Fuzion intermediate representation does not have an explicit notion of a type. Instead, types are defined by features of the routine and choice categories. When types are referenced in the FUIR, a reference to the defining feature is used, there is no explicit representation of a type.

4.2.3 Fields. Fields may store calculated values within the instance of a routine. Even though the Fuzion language does not permit mutable fields, the intermediate language does have commands to assign a value to a field. Static analysis is used to ensure that no call is made to read a field that has not been initialized and that fields that were initialized will not be overwritten.

Fuzion defines special fields for routine argument values and for the result value returned by a function.

Furthermore, whenever code to access instances of outer features might be required, an implicit *outer ref* field pointing to the instance of an outer routine or choice will be created automatically and initialized implicitly on a call with a reference to the target value of that call.

4.2.4 Code. Routines may contain code that is executed on a call to that routine. The code consists of a very small set of bytecode commands.

To represent executable code, the FUIR uses a stack machine with only ten different commands. The commands are

Current: Obtain a reference to the current instance of the routine feature we are executing. This instance is a value of the product type defined by the fields of this routine. It permits access to the inner fields of that routine. A reference to the current value will be stored on the top of the stack after this command.

Call: A call to a routine. Calls are performed on a target value given a fixed list of argument values.

On a call, a reference to the target value together with a fixed number of argument values will be taken from the stack and assigned to the outer ref field and the argument fields of the called feature.

In case the target value is a ref type, the call may use dynamic binding, the called feature will be determined dynamically depending on the target value's type.

Assign: Will assign the value on the top of the stack to a given field in the instance referred to by the second topmost value on the stack.

Tag: Will convert a value on the stack to a tagged union type (choice) value that will replace the original value on the stack. Examples are converting number *42* of type *i32* to a value of type *option i32*.

Match: This is the only instruction that branches execution. For this, a tagged union type value is taken from the stack and, depending on the actual type, one of several code blocks will be executed after the original value was recovered and assigned to a field.

Box: Will convert a value that is on top of the stack to a corresponding ref type. This permits value types like *f64* to be passed as a reference with type information to a routine that expects a reference like *Any*.

Const: Creates a constant instance of given type from an array of bytes that contain the value in a serialized form. This works for simple types like *u64* well as for product or sum types.

Pop: Drop the top value from the stack.

Env: Obtain the current value of an effect of a given types.

Comment: A No-op, used to transfer debug comments from earlier code transformation phases.

4.2.5 No Loops. FUIR has no commands for loops or, at a lower level, conditional or unconditional jumps that would allow building a loop. Instead, tail-recursive calls are used to implement loops. This simplifies static analysis that will have to be able to handle recursion anyway, so there is no need for loop analysis. However, back end code generators

or VMs must be able to optimize tail calls to avoid excessive stack usage.

4.2.6 Intrinsics. The commands listed above lack many operations that are typically present in intermediate formats. In the FUIR, these operations are provided via calls to intrinsic features that are part of the Fuzion base library. These intrinsics cover the following operations

- compile-time reflection such as *Type.name*
- atomic operations like *compare_and_set*
- debug levels
- access to command line arguments and env vars
- exit
- integer and float arithmetic
- low-level arrays
- effect handling: instantiation and abortion
- handling threads
- foreign function interface related operations

Currently, there are intrinsics for basic I/O operations and access to file and networking as well. However, these should become native features once we have a foreign function interface for C library functions.

4.2.7 No Heap vs. Stack distinction. In the FUIR, a call requires the allocation of memory for the current instance that consist of all direct inner fields of a called feature. However, there is no distinction between heap and stack allocation like, e.g, in Java bytecode [10] where the *new* instruction would allocate an instance of a class on the heap while the *invoke** instructions implicitly allocate a stack frame. Also, FUIR does not make a distinction between an access to a local variables or to a field fields in a heap allocated instance.

Instead, a call to a routine requires the allocation of memory for the called routine's current instance such that the life span of that instance covers all accesses to the inner fields.

Often, this life span ends when a function returns while it lives on after returning form a ref type constructor. However, the presence of inner features that have access to the fields of their outer features via outer ref fields in their instances may extend the outer instance's life span in case the inner instances remain accessible, e.g., by being returned as part of a function result.

A data-flow analysis performed on the FUIR is used to determine the life span of all instances that backends then use to decide which instances require heap allocation and which instances may be stack allocated, or use a completely differen allocation context like a region [19].

5 Implementation

Currently, the Fuzion compiler is implemented in Java and available freely on github [16]. Access to the Fuzion Intermediate Representation is performed via a single Java class *dev.flang.fuir.FUIR*, which provides an abstraction for the actual file format.

An *AbstractInterpreter* class has been implemented using FUIR to facilitate the processing of the intermediate code. Any code processing the intermediate code can use this class by implementing a few abstract methods that correspond to the commands in the FUIR code and by providing implementations of those intrinsics that are relevant for the specific processing. All subsequent processors in the Fuzion implementation, i.e., data flow analyzer and backends, are built on top of this *AbstractInterpreter*.

5.1 Data-Flow Analysis

A data-flow analysis (DFA) using a variant of object-sensitivity [12] has been implemented using the *AbstractInterpreter* class. This permits accurate tracking of data flow.

Currently, this essentially serves as a powerful smart-linker for the backends that get fed with an optimized FUIR created by the DFA.

5.2 Interpreter

The simplest execution environment for FUIR code is an interpreter implemented in Java that directly executes Java code for each of the ten FUIR commands and that provides implementations of the intrinsics. The interpreter currently does rely on the underlying Java VM's garbage collector, all instances are allocated on the heap.

5.3 JVM back end

A second back end uses the *AbstractInterpreter* class to create Java bytecode that can then either be loaded and executed directly by the current JVM or saved as jar-file for later execution. A small runtime system is implemented in Java and contains code for Thread handling, an effect stack implementation, etc.

Simple intrinsics are translated into inline bytecode instructions, while more complex ones are implemented as Java methods in the runtime system. Instances are currently mostly allocated on the Java heap, but infrastructure for stack allocation of short-lived instances is being implemented, even though the JVM implementations appear to perform surprisingly efficient heap allocation.

5.4 C back end

A third back end generates C source code, also using the *AbstractInterpreter* class and inline C code for all intrinsics. Boehm's garbage collector [3] is used as a temporary solution until a native GC is implemented.

6 Related Work

The first Java Specification Request JSR1 [13] addressed the desire to make Java a language suitable for real-time systems, which was followed by the recent release of an update of the Real-Time Specification for Java 2.0 in JSR282 [9]. There also exists a proposal for safety-critical Java JSR302[5]. These

proposals address ways how a high-level language like Java could be applied for safety-critical and real-time applications.

For Ada, the Ravenscar profile [4] and later SPARK [15] defined a safety-critical profile for the Ada programming language together with commercial tools. For Rust, the Ferrocene tool-chain targets critical systems [2].

In contrast to these attempts to bring high-level languages to safety-critical applications, Fuzion attempts to design a new high-level language for this application domain by providing a simplified language and intermediate code with supporting tools.

7 Conclusion

The Fuzion language definition, its implementation and the Fuzion Intermediate Representations presented here are work in progress. The Fuzion team has a number of ambitious development goals, the most important ones being stabilizing the Fuzion language definition itself, adding standard libraries, improving the tools for static analysis and providing better and more efficient back ends.

Nevertheless, we see that having a language and an intermediate representations that focus on simplicity and accessibility by analysis tools can bring significant advantages to a large number of software applications. In particular, we expect that the high engineering effort that is required for the validation and verification process during the certification of safety-critical systems can be reduced significantly, while the production of software in general could profit from a focus an simplicity for safety.

References

[1] 1992. RTCA/DO-178B Software Considerations in Airborne Systems and Equipment Certification.

[2] 2024. This is Rust for critical systems. https://ferrocene.dev/en/

[3] Hans-Juergen Boehm. 1993. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/155090.155109

[4] A. Burns, B. Dobbing, and G. Romanski. [n. d.]. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. *Proceedings of Ada-Europe 98* 1411 ([n. d.]), 263–275.

[5] C. Douglass Locke. 2006. JSR 302: Safety Critical Java Technology. http://jcp.org/en/jsr/detail?id=302

[6] The LLVM Foundation. [n. d.]. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

[7] Isabelle Contributors. [n. d.]. Isabelle proof assitant. https://isabelle.in.tum.de/.

[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman. 2023. *The Java® Language Specification.* Oracle America, Inc. https://docs.oracle.com/javase/specs/jls/se21/html/index.html

[9] James Hunt, Andy Wellings, David Bacon. 2024. JSR 282: RTSJ version 2.0. http://jcp.org/en/jsr/detail?id=282

[10] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2023. *The Java Virtual Machine Specification, Java SE 21 Edition.* Oracle America, Inc. https://docs.oracle.com/javase/specs/jvms/se21/jvms21.pdf

[11] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (Oct. 1992), 40–51. https://doi.org/10.1109/2.161279

[12] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. *SIGSOFT Softw. Eng. Notes* 27, 4 (jul 2002), 1–11. https://doi.org/10.1145/566171.566174

[13] Peter Dibble. 2006. JSR 1, Final Release 3: RTSJ version 1.02. http://jcp.org/en/jsr/detail?id=1

[14] Fridtjof Siebert. 2022. Fuzion - Safety through Simplicity. *Ada Lett.* 41, 1 (oct 2022), 83–86. https://doi.org/10.1145/3570315.3570323

[15] SPARK Team. [n. d.]. SPARK - The SPADE Ada Kernel (including RavenSPARK), Edition 7.2. https://docs.adacore.com/sparkdocs-docs/SPARK_LRM.htm. https://docs.adacore.com/sparkdocs-docs/SPARK_LRM.htm

[16] The Fuzion Team. 2023. Fuzion GitHub Repository. https://github.com/tokiwa-software/fuzion

[17] The Fuzion Team. 2024. Fuzion Portal Website. https://fuzion-lang.dev. https://fuzion-lang.dev

[18] The Fuzion Team. 2024. Fuzion • Fuzion Design • File Formats • Module File. https://fuzion-lang.dev/design/fum_file

[19] Mads Tofte and Lars Birkedal. 1998. A region inference algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (jul 1998), 724–767. https://doi.org/10.1145/291891.291894