Algebraic Effects and Static Analysis for Safety-Critical Applications in Fuzion

Fridtjof Siebert, Michael Lill, Max Teufel
Tokiwa Software GmbH, Karlsruhe, Germany; email: {siebert, michael.lill, max.teufel}@tokiwa.software

Abstract

This work-in-progress paper presents the introduction of algebraic effects to the Fuzion language and how algebraic effects can be used in the context of safety-critical systems.

Fuzion is a modern, general purpose programming language that unifies functional and object-oriented paradigms into a pure functional language. Algebraic effects are used to represent and manage non-functional aspects like I/O operations or mutable state. Static analysis is used extensively at several stages in the Fuzion toolchain to verify different correctness aspects of the application.

We start with a condensed overview of the Fuzion language to then describe how algebraic effects are used to represent non-functional aspects. The Fuzion toolchain will be explained and how static analysis is used to build and validate applications. Finally, it will be shown how algebraic effects can be used to model aspects relevant to safety-critical systems.

Keywords: programming languages, algebraic effects, static analysis, safety, security

1 Introduction

Fuzion is a work-in-progress project to design a new high-level language targeting safety critical applications [1]. 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 it is explicitly declared to do so.

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 [2].

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:

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

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 *installed* 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 installed 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 frontend such that —even though Fuzion is statically typed— types can be omitted in most cases.

2.8 Information Hiding

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

[2]that will be optimized by the backends

3 Algebraic Effects

Purely functional code brings a number of advantages for the correctness of a complex software system: The result of every call depending only on the call's arguments simplifies (automatic) reasoning about the code, the absence of mutable state results in thread-safety and the absence of side-effects permits optimizations since there is no observable effect if code is not executed or executed repeatedly. Also, if purity of code from an untrusted source can be verified, it can safely be used without compromising a system's security.

However, for code to interact with the outside world or for mere performance reasons, actual systems must be able to perform non-functional operations. Examples include i/o operations, mutation of data, inter-thread communication, access to hw timers, sensors, actuators, aborting an operation, and many more.

Algebraic Effect handlers [3,4,5,6] are used in recent programming languages as a means to handle operations that would break the purity by having non-functional side-effects. An effect defines a set of such operations, while an effect handler provides a concrete implementation of these operations. Code that calls these operations is then said to require an instance of the given effect in its environment. The environment is essentially a stack of effects that is used to find the innermost handler for each operation to be used.

3.1 Effects in Fuzion

Fuzion uses algebraic effects to wrap non-functional operations. An effect in Fuzion is a *constructor feature* that inherits from a base library *feature* called *effect* and that defines a set of operations as inner features. An effect can be instantiated and installed to run code that uses the effect's operations. Effects are identified by their type [7].

Feature declarations in Fuzion include an optional section to list all the effect types that are required to call that feature. Features that are marked as *public* must include this information, which will be verified by static analysis.

3.2 Effects in dynamic code

A major difficulty in specifying the effects of a feature originates in the presence of dynamic code: functions that are passed as arguments to routines may require additional effects that are unknown to the called routine. E.g., one might want to log calls to a function passed to a library routine, where the effect that performs this logging is unknown to the library. The same problem may occur if a redefinition of an inherited feature uses effects that were unexpected by the original routine.

To solve this, languages like Koka introduce effect polymorphism to declare required effects explicitly [8] while the Effekt language [9] simplifies effect polymorphism by viewing effects as capabilities [10].

In Fuzion, static analysis is used both at module level and at whole application level. At module level, effect polymorphism is analyzed only to the extend that control flow reaches the use of a given effect, while additional effects introduced by users of that module through function arguments or redefinition are ignored. At the application level, whole program data-flow analysis verifies that all uses of effects occur in environments that provide corresponding effect instances.

3.3 Effect Example

We will present a small example using an effect *temp* to model a temperature sensor that can read a temperature in degrees Centigrade:

```
temp (hdlr ()->f64) : simple_effect is
  read => hdlr()
```

Here, *temp* is the effect constructor and also its type, and *read* is the only operation, which gives a temperature reading. *read* is implemented by calling a handler function *hdlr*, which permits different implementations for different instances of this effect.

Our application main loop now requires this *temp* effect to repeatedly perform a temperature reading and printing the result unless it is larger than $41°C$, when it should call *panic*:

```
main ! temp =>
  do
    t := temp.env.read
    if t > 41
      panic "***␣get␣doctor␣***"
    say "ok:␣$t℃."
    time.nano.sleep (time.durations.ms 500)
```

In a deployed system, this would run with an instance of *temp* using a handler that reads the temperature from a thermometer. In a test setup, we can simulate the hardware using a handler *test_temp* that reads and modifies a mutable field *cur_temp*:

```
cur_temp := mut 37.0
test_temp =>
  t := cur_temp.get
  cur_temp <- t+0.3
  t
(temp test_temp).use main
```

This illustrates that, when using effects, we can separate the program logic, *main* in this example, from the implementation of non-functional aspects like reading a sensor.

4 Compilation Phases

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 frontend phase checks that the source code respects the language validity rules that include type checks, visibility rules, etc.

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].

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

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 backend that create C source code to be processed by clang/llvm or other C toolchains and an interpreter that directly executes the intermediate code.

5 Static Analysis in Fuzion

The use of static code analysis is essential in all compilation phases of Fuzion: During the frontend phase, static data-flow analysis is used to verify that dependencies on effects are declared for public features, while whole application analysis can be used for a variety of applications.

5.1 Intermediate Representation

The basis of the whole-program analysis is the Fuzion Intermediate Representation *FUIR*. This representation consists of a collection of features that were monomorphized, i.e., all type parameters are replaced by actual *runtime types* while features are duplicated for every combination of runtime types found by the middle end.

Expressions that provide the code of routines are encoded using a stack-based bytecode format, comparable to Java bytecode [11]. However, there are currently only ten different bytecode instructions:

- *AdrOf* — used for call-by-references for outer instances
- *Assign* — assign a value to a field
- *Box* — create a reference instance from a value instance
- *Call* — perform a call to a routine
- *Comment* — only used for debugging
- *Const* — create an instance from serialized byte data

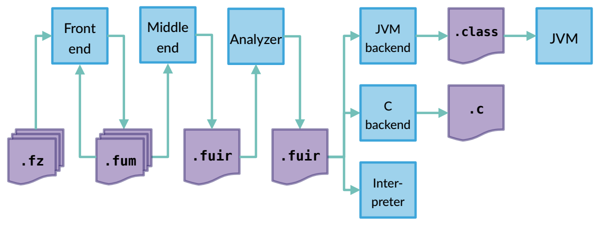

Figure 1: *Fuzion toolchain and intermediate data.*

- *Current* — return the instance of the current feature
- *Env* — obtain current instance of an algebraic effect type
- *Pop* — discard a value from the stack
- *Match* — extract the original value from a tagged union type value and branch to code that processes that value
- *Tag* — create tagged union type value from a value of one of the choice types

The low number of instructions helps to simplify the implementation of static analyzers and code generators.

There are no instructions for basic arithmetic operations. Instead, fundamental features like addition of values of type *i32* are calls to features of kind *intrinsic* that must be provided by the backends and handled by static analyzers correctly[4].

5.2 Whole Application Analysis

The whole application analysis performs a data-flow analysis over an application until a fix-point is reached. As a result, upper-bound sets of possible values for all fields and expressions in different call contexts are found.

This analysis can be used for a number of purposes, e.g.

- Elimination of deactivated code
- proof of absence of errors[5]
- verification of pre- and post-conditions
- specialization of code for actual values
- determination of life-spans of instances, e.g., for automatic stack or static allocation
- user feedback on heap allocation

Furthermore, we expect the application wide data-flow analysis to be useful during the verification and validation process by producing evidence for deactivated code, developing test cases for better code coverage, or even correctness proofs by verification of pre- and post-conditions.

6 Algebraic Effects for Safety and Security

Algebraic effects can be used to manage non-functional aspects related to the safety and security of the system:

6.1 Security along SW Supply Chain

Static, program-wide analysis will find all effects required by code, including all effects required by third-party libraries. This could be used to increase the security by providing harmless handlers to suppress undesired functionality. An example is the log4shell vulnerability [12] that enabled downloading and execution of arbitrary code in a widely used logging library for Java. Static analysis would first help to detect that such effects are used. Then, the library code could be sandboxed by applications using effect handlers that prohibit operations like network access or execution of arbitrary code.

[4]which, depending on the analysis, often permits ignoring them.

[5]which essentially means error handling. like calls to *panic.* is deactivated

6.2 Safety and Real-Time Aspects

With the presence of algebraic effects as a powerful means to describe behavior that is not purely functional, we expect to use these effects to address aspects that are of particular importance to many safety-critical systems, e.g., to describe and verify real-time behavior.

The following gives a list of possible effects that we want to implement and apply in Fuzion:

- *constant time* — code contains no conditionals
- *bounded time* — code contains no unbounded recursion[6]
- *non blocking* — code performs no blocking operations
- *no heap* — code performs no heap allocation[7]
- *interruptible* — code may be aborted asynchronously
- *worst-case execution time* — execution time limit, enforced by analysis or at runtime

The flexible way that algebraic effects permit to introduce scopes of arbitrary types into the program in conjunction with the ability of static analysis to attach semantics to these effects and verify these semantics appears to give a powerful tool to handle safety and real-time aspects.

7 Conclusions and Future Work

We have presented the Fuzion project that develops a pure functional language using algebraic effects and a corresponding toolchain and showed how we expect the use of Fuzion to help enhance security and help during safety verification and validation.

The project is still in an early prototype state, we have basic implementations of the frontend, middle-end, first versions of static analysis using data-flow analysis and three different backends.

The current focus of our work is on improving the base library, in particular to model a variety of non-functional aspects using algebraic effects. The Fuzion intermediate representation is kept simple to encourage the integration with different powerful tools, an integration with proof assistants like Rocq/Coq [13] or Isabelle [14] might increase the power of correctness proofs significantly.

Fuzion has a powerful interface to call Java code, but we will need other foreign function interfaces, in particular for C, to inter-operate with legacy code. The C backend currently uses the garbage collector by Hans Boehm [15], for use of Fuzion in real-time systems, we will need to ensure that all allocation can be performed statically or use a real-time GC [16].

[6]this implies bounded loops since loops use recursion

[7]and, with a suitable GC, is hence never preempted by GC work

References

[1] F. Siebert, “Fuzion - safety through simplicity,” *Ada Lett.*, vol. 41, p. 83–86, oct 2022.

[2] “Fuzion Portal Website.” https://fuzion-lang.dev, 2024.

[3] G. Plotkin and J. Power, “Algebraic operations and generic effects,” *Applied categorical structures*, vol. 11, pp. 69–94, 2003.

[4] G. Plotkin and M. Pretnar, “Handlers of algebraic effects,” in *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP ’09, (Berlin, Heidelberg), p. 80–94, Springer-Verlag, 2009.

[5] G. D. Plotkin and M. Pretnar, “Handling algebraic effects,” *Logical methods in computer science*, vol. 9, 2013.

[6] M. Pretnar, “An introduction to algebraic effects and handlers. invited tutorial paper,” *Electron. Notes Theor. Comput. Sci.*, vol. 319, p. 19–35, dec 2015.

[7] F. Siebert, “Types as first-class values in fuzion.” Talk at TyDe 2023: 8th ACM SIGPLAN International Workshop on Type-Driven Development, https://fuzion-lang.dev/talks/tyde23types, sep 2023.

[8] D. Leijen, “Koka: Programming with row polymorphic effect types,” *arXiv preprint arXiv:1406.2061*, 2014.

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

[10] J. I. Brachthäuser, P. Schuster, and K. Ostermann, “Effekt: Lightweight effect polymorphism for handlers (technical report),” tech. rep., University of Tübingen, Germany, 2020.

[11] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 21 Edition*. Oracle America, Inc., sep 2023.

[12] “Cve-2021-44228 apache log4j2 jndi features do not protect against attacker controlled ldap and other jndi related endpoints.” https://www.cve.org/CVERecord?id=CVE-2021-44228, dec 2021.

[13] Coq Development Team, “The coq proof assitant.” https://coq.inria.fr/.

[14] Isabelle Contributors, “Isabelle proof assitant.” https://isabelle.in.tum.de/.

[15] H.-J. Boehm, “Space efficient conservative garbage collection,” in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI ’93, (New York, NY, USA), p. 197–206, Association for Computing Machinery, 1993.

[16] 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.