JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Toward a Lingua Franca for Memory Safety

Dimitri Racordon[*,†], **Aurélien Coet**[†], **and Didier Buchs**[†]
[*]Northeastern University, USA
[†]University of Geneva, Faculty of Science, Switzerland

ABSTRACT Memory safety checking seeks to protect programs from a wide spectrum of software problems related to memory access and management, such as using unallocated or uninitialized buffers. Despite decades of research, it remains an active and fruitful research topic, as issues of scalability and adoption continue to present open challenges. A popular approach to overcome these obstacles is to rely on type checking. Types are arguably one of the most scalable techniques to reason about a program's structural properties. They also offer a convenient tool to impose restrictions on source code, either to prohibit undesirable behaviors or to facilitate other analyses.

Within the plethora of type systems that have been proposed to combat memory bugs, one recurrent trend is to leverage uniqueness and/or immutability to limit the impact of mutation, to support local reasoning. Unfortunately, bringing these properties to existing languages is often met with a deterring engineering effort. Unlike many other features and mechanisms, these are difficult to encode within a host language via simple syntactic extensions or clever meta-programming. Instead, they require a deeper understanding of the program's semantics, which can only be obtained through preliminary analysis.

This paper presents results in our effort to address this difficulty. We introduce Fuel, a compiler framework designed to guarantee uniqueness and immutability properties in arbitrary programs with explicit memory management. Fuel is centered around a source and platform agnostic low-level intermediate representation, equipped with a capability-based type system, which can be targeted by compilers for higher level languages. It is able to guarantee freedom from invalid dereference and duplicate deallocations, and offers partial support to detect memory leaks. It also advocates for the use of dynamic checks to recover static type assumptions in places where static reasoning is either impractical or impossible. We present Fuel informally through a short series of examples and formalize a subset of its intermediate language.

KEYWORDS Type capabilities, memory safety, aliasing, uniqueness, immutability, borrowing, compiler toolchains, intermediate languages.

1. Introduction

Memory bugs (Szekeres et al. 2014) have plagued software development since Fortran proposed sharing data structures with the COMMON block. Not even high-level programming languages are immune to the curse, as memory initialization and cleanup remain pervasive issues (Kabir et al. 2020) in these as well. One successful approach to detect and/or prevent these bugs is type-checking. Types are one of the most scalable tools to formally reason about the structural properties of a program. Not only do they provide an elegant way to classify data, allowing developers to segregate values using names or structural properties, they can also uphold some aspects of a software specification. By attaching a type to each expression, one can discover nonsensical operations, such as subtracting a number from a function. This process allows compilers to reject ill-formed inputs, greatly improving the correctness and maintainability of software systems (Hanenberg et al. 2014).

Traditional type checking, however, struggles to handle situations in which the variables may represent mutable states. In response, recent years have witnessed a significant interest in flow-sensitive type systems as a countermeasure. Those combine type checking with data-flow analysis techniques to

JOT reference format:
Dimitri Racordon, Aurélien Coet, and Didier Buchs. *Toward a Lingua Franca for Memory Safety*. Journal of Object Technology. Vol. 21, No. 2, 2022.

http://dx.doi.org/10.5381/jot.2022.21.2.a3

describe properties that depend on a program's execution flow. They are particularly well-suited to express memory safety properties, therefore strengthening the static guarantees compilers can establish on a program's correctness.

Unfortunately, defining and implementing such type systems is often met with a surprisingly painful effort. Novel language features regularly emerge in "featherweight" formal calculi (e.g., (Racordon & Buchs 2020)), before they are reified under various forms in actual programming languages. Oftentimes, one can "encode" new features within an existing host language, using clever meta-programming (Ballantyne et al. 2020; Ichikawa & Chiba 2017), at a relatively low engineering cost. This technique is not limited to syntactic extensions. Static semantics too can be expanded into the host's type system (Imai et al. 2020). However, this approach fails when extensions are orthogonal to the existing concepts. In particular, flow-sensitive type systems aim to enforce additional program-wise restrictions to support general assumptions about safety guarantees. These restrictions use information that is often absent from a traditional type system. In other words, they require a more intimate understanding of the program's semantics.

One solution is to integrate these aspects with the host language at a deeper level, within its compiler. Compilers are usually implemented as so-called "toolchains" of specialized components. Each link of the chain consumes a specific input and generates a specific output, eventually leading to an executable program. This decomposition does not only contribute to more maintainable software architectures, it also provides a great opportunity to insert auxiliary analyzers. Conceptually, one simply has to "plug" their extension at the appropriate spot within the toolchain, and work on the intermediate representation (IR) that is exchanged between two components.

The advantages are manifold. IRs offer a refined description of the program's semantics and are often much smaller than surface languages by eliminating syntactic sugar and "lowering" some high-level abstractions. This property has been exploited by numerous tools, such as Java Pathfinder (Visser & Mehlitz 2005) and PhASAR (Schubert et al. 2019), which detect defects in Java and C++, respectively. Furthermore, IRs support some form of language-agnosticism. For instance, the LLVM compiler infrastructure (Lattner & Adve 2004) achieves relative language independence by the means of a universal RISC-like instruction set, called LLVM IR, specialized for program optimizations. This dramatically reduces the engineering effort, removing the need to re-implement common program optimizations and code generation for a specific CPU target when developing a new language front-end: a compiler can simply target LLVM IR to benefit from LLVM's implementation of optimizations and code generation.

IRs suitable to reason about flow-sensitive type systems are currently lacking. LLVM IR loses relevant information with respect to aliasing properties. For instance, inferring the lifetime of a pointer might be challenging, whereas the source language can provide valuable insights to determine its scope (with annotations, for example). Further, high-level constructs may translate to sophisticated patterns that become difficult to identify at a lower level, due for instance to type erasure, reliance on a runtime library, or intricate control flow. More specialized IRs, such as Java bytecode (Lindholm et al. 2014), may address some of these shortcomings, but at the cost of opinionated memory models which might fail to accurately capture certain semantics. For example, the Java bytecode does not support explicit memory management.

Finally, although more academic IRs come closer, such as Boogie (Barnett et al. 2005), Why (Filliâtre & Paskevich 2013) and Viper (Müller et al. 2017), they lean toward full functional correctness rather than more conservative safety properties. In other words, they offer stronger guarantees at the expense of complex specifications that are difficult to synthesize. These observations suggest the need for a new framework to soften the implementation burden of type-based memory safety checking. This endeavor prompts the following research questions:

RQ1 *What is the appropriate expressiveness to reason about mutable memory states?*

Over the years, numerous type disciplines and language mechanisms have been proposed to enforce memory safety properties, based on various principles (e.g. capabilities (Smith et al. 2000; Naden et al. 2012), ownership types (Clarke et al. 2013), effects (Nielson & Nielson 1999), etc.). Unsurprisingly, all come with their own set of strengths and weaknesses, turning the identification of a common denominator into a challenge.

RQ2 *What is the appropriate level of abstraction for a memory safety checking framework?*

This question relates to the above-mentioned issue with LLVM IR, and its inability to exploit insights from the surface language with respect to aliasing properties. A framework directed at a flow-sensitive analysis should sit at an abstraction level that is high enough to retain relevant structural information, but low enough to keep the analysis as simple and as generalizable as possible.

RQ3 *Where should a memory safety checking framework be plugged into a compiler toolchain?*

This third question follows naturally from the first one, as the framework's abstraction level correlates with its depth in the toolchain. While compilers are expected to preserve the semantics of the program they translate, its representation typically diverges from that of the original source code the further it moves along the chain. It is therefore tempting to stay as close as possible from the first stages. Nonetheless, jumping in at a later stage will contribute to language-agnosticism.

This paper reports results in our ongoing effort to answer these questions. Our objective is to lay down the foundation for a reusable, language-agnostic tool to express and check properties pertaining to memory safety. Drawing inspiration from LLVM, we present Fuel, a compiler framework centered around a low-level programming language that relies on type capabilities to keep track of mutable states and describe aliasing relationships. Fuel supports intra-procedural reasoning,

using annotations to thread typing assumptions across function boundaries, and leverages dynamic checks to recover static assumptions in places where static reasoning is impractical or impossible.

In section 2, we present some background on memory safety checking with a particular focus on type-based approaches. We then introduce our framework informally in section 4 by translating a handful of small C and Rust programs. In section 5, we formalize a subset of its IR, for which we describe the dynamic and static semantics, and we end in section 6 with a brief discussion on open challenges. This paper is accompanied by an artifact, a library and standalone compiler for Fuel programs, whose implementation is distributed as open-source software on GitHub: https://github.com/kyouko-taiga/fuel.

2. Background and related work

The term "memory safety" can bear different meanings, depending on the context in which it is being used. This paper is interested in properties aimed at ensuring that access to a machine's memory do not cause crashes or undefined behaviors. In the vast realm of errors that can violate such properties, we specifically focus on the following categories:

Invalid dereference errors These occur when a program attempts to read a value at a location that is no longer allocated (a.k.a. use-after-free) or that does not exist (e.g., `null` dereference).

Invalid deallocation errors These occur when a program attempts to deallocate memory that was already freed, or memory that does not reside in the heap and must not be freed manually (e.g., stack-allocated and static memory).

Memory leaks These occur when a program does not deallocate memory that is no longer needed or accessible.

While it is sometimes believed that only unsafe low-level languages (e.g., C/C++) suffer from these types of errors, we stress that they also impact safer systems. For instance, `null` dereferences and memory leaks are quite common in Java or Python.[1]

There exists a large body of work dedicated to analysis techniques and language mechanisms for memory safety. This section introduces some background on the topics most closely related to Fuel's principle and objectives, and briefly reviews related literature.

2.1. Static program analysis

Static analysis encompasses an array of techniques designed to discover and/or enforce specific properties without actually executing a program.

2.1.1. Pointer analysis

The above-mentioned memory errors relate to the use of pointers or references, made pervasive not only in the context of low-level system programming, but also with the success of Smalltalk-inspired object-oriented languages, where every object is represented by a reference to a container holding its internal representation. As a response, pointer analysis (Smaragdakis & Balatsouras 2015) (sometimes called "points-to analysis") has emerged as a common countermeasure. The technique consists of determining the set of values to which a pointer may refer (a.k.a. its points-to set), in order to check for some invariant before it is used.

After three decades of research, pointer analysis is now generally well understood and has been adopted by a wide range of industry-ready applications (e.g. Clang Static Analyzer (Clang Project 2021) or Cppcheck (Marjamäki 2021)). Nonetheless, open challenges remain with respect to scalability, dynamic features (as popularized by modern scripting languages), and the increasing popularity of modular compilation (Thakur 2020). Both aspects press on an important weakness of pointer analysis, namely that it typically expects visibility on the whole program statically.

2.1.2. Type systems

A difficult problem pointer analysis has to overcome stems from the number of different situations it has to consider. In particular, low-level languages impose little to no constraint on how a program may manipulate memory. This opens the gates to a realm of wild features, such as pointer arithmetics or value reinterpretation (e.g., C++'s `reinterpret_cast`) that are difficult to analyze. One way to tackle this problem is to restrict these features' applicability to cases that are simpler to examine, typically through the means of a type system.

Classic type systems understand types as invariant assumptions on the nature of an expression, including program variables. In other words, if a variable is found to have type τ at some location, then it is assumed that it should have type τ everywhere else. This convenient presupposition is inherited from functional programming languages, where program variables are mere placeholders for immutable values.

For example, consider a simple expression `let y = f(x) in g(y, y)`. A classic type system will deduce that `f` has type $\tau_1 \to \tau_2$, that `y` has type τ_2, and that `g` has type $\tau_2, \tau_2 \to \tau_3$. Hence, if that expression is well-typed, then so is the same expression where `f(x)` is substituted for `y` (i.e., `g(f(x), f(x))`), thanks to referential transparency. Unfortunately, this reasoning fails in the presence of references to mutable state, as `f` could be a function that mutates the state referred by `x`.

This issue is particularly stringent in imperative languages, where computation is described through sequences of operations that interact with a mutable state, generally represented by the machine's memory. While variables can still be understood as value placeholders, they are no longer merely substituted. Instead, they denote memory containers from which one may read or to which one may write a value. Thus, declaring that a variable has type τ usually specifies that the values that it *stores* have type τ—a distinction often dismissed, in spite of its significance.

In Java, for instance, a method f with a type $\tau_1 \to \tau_2$ can be called with either a literal value of type τ_1 or a variable whose values have type τ_1. This simplification has an obvious practical justification, as the alternative would compel developers to provide an additional method to accept *containers* (i.e.,

[1] Although both languages provide means to "catch" null dereference at runtime, doing so is usually discouraged (Bloch 2008).

variables) holding values of type τ_1, or to use an explicit operation to dereference a variable. But it also has far-reaching consequences. Since the container's type is confounded with that of the contained value, the type system becomes unable to distinguish between them. It follows that it can only ignore the fact that a variable may not hold any value before it is used in an operation. A pedantic type checker could preserve the distinction and require explicit checks every time one expects to read a value from a container (Dietl et al. 2011). A less cumbersome approach is to run some form of dataflow analysis to determine whether a container can be safely used whenever it appears in an expression (Liu et al. 2020; Nieto et al. 2020; Kabir et al. 2020).

More advanced type systems recognize that type assumptions are actually *not* necessarily invariant, but may depend on the program's "flow". In other words, they accept that variables may have different types throughout the execution of the program, according to the operations in which they are involved, effectively expressing the container's state. A comprehensive review of such systems is beyond the scope of this paper. Instead, we settle for a brief description of the approaches most closely related to our work.

Substructural type systems Type-based formalisms are usually formalized in the form of a deductive system based on inference rules. The deduction process exploits three structural properties (exchange, contraction and weakening (Reynolds 1998b)) to determine well-typedness. Substructural type systems are formalisms in which one or several of these properties do not hold. They provide an elegant way to encode the variance of typing assumptions within the meta-theory, altering the classical way to apprehend type checking and inference.

Substructural type systems are suitable to reason about elaborate properties of memory, such as aliasing relationships and memory safety. Linear (Wadler 1990) and affine (Tov & Pucella 2011) types, in particular, have received the most attention and even found their way into mainstream programming languages, such as Rust and C++. Linear types require that typing assumptions be used *exactly* once, by waving contraction and weakening. This effectively imposes an outright ban on aliasing and can be leveraged to guarantee freedom from memory leaks. In contrast, affine types preserve weakening and thus only require that typing assumptions be used *at most* once. Such a relaxation is justified by the ubiquity of garbage collection mechanisms in modern programming languages.

Capability-based systems Originally proposed as a generalization of reference annotations (J. Boyland et al. 2001), type capability-based approaches encompass techniques that extend types with context-sensitive properties. A capability is a token describing the set of operations that can be performed on a specific program variable, such as reading its contents, and that flows in and out of a function. In other words, capabilities enable additional information to be transferred between a caller and a callee.

One advantage of this technique over substructural type-systems is that it dissociates pointer values, which can be copied freely, from the permission to use them (Smith et al. 2000). Hence, it supports the encoding of mutable self-referential data structures (e.g., graphs), albeit through relatively sophisticated constructs (Walker & Morrisett 2000; Maeda et al. 2011).

Capabilities generally aim at formalizing two properties: *uniqueness* and *immutability*. The former tightly couples capability-based approaches with substructural type systems, while the latter strives to tame unintended mutations. While these properties lead to a quite restrictive typing discipline, several mechanisms have been proposed to relax them, notably including borrowing (Naden et al. 2012), nesting (J. T. Boyland 2010), and adoption/focus (Fähndrich & DeLine 2002). Capabilities are of particular interest in the context of concurrent programming, often to guarantee data race freedom. Successful realizations include Pony (Clebsch et al. 2015) and Encore (Brandauer et al. 2015).

Rather than asking for capabilities to perform certain operations, a function may simply advertise the kind of side effects in a type-and-effect system (Rytz et al. 2013). Effects can express the same kind of safety guarantees as type capabilities, but in a more declarative fashion. In fact, both approaches are equivalent (Gordon 2020).

Ownership types Ownership types (Clarke et al. 2013) propose to uphold the encapsulation principle to limit the visibility of object mutations and prevent "spooky actions at a distance". The essence of this approach is to partition the memory into topologically nested regions whose ownership is attributed to a particular object. Intuitively, a region corresponds to the container holding the object's internal representation, while its owner is a reference to that container. From there, various restrictions can be applied to the access to a region's contents. Ownership types are deeply intertwined with object-oriented programming. Hence, most restriction policies are based on the idea that regions are associated with methods, attributing them privileged access to the object's internal representation.

Unlike the other approaches we reviewed so far, ownership types are traditionally flow-insensitive and rely on access restrictions to enable local reasoning. Therefore, although they can be leveraged to speed-up static analysis, they are not as well-suited to address the kind of memory errors that we have listed.

2.2. Dynamic program analysis

Another common technique to detect and/or fend memory bugs is to instrument the program with runtime guards to detect improper uses. There exist two main implementation strategies: dynamic binary instrumentation (DBI) (Buck & Hollingsworth 2000) and compile time instrumentation (CTI) (Stepanov & Serebryany 2015). The former consists of interpreting native code while inserting probes and other analysis routines to monitor the application. DBI does not even require access to the program's source, and is therefore completely agnostic of the input language. However, it comes at the price of a significant overhead in both time and space, restricting its use for debugging and profiling purposes. One successful application of this approach is Valgrind (Nethercote & Seward 2007) a powerful memory analyzer able to detect various errors including buffer overflows, leaks and uses of free or uninitialized memory.

In contrast, CTI sacrifices flexibility for better performance. Some language agnosticism can be achieved nonetheless, by relying on compiler IRs rather than actual language sources. For instance, AddressSanitizer (Serebryany et al. 2012) instruments LLVM IR (Lattner & Adve 2004) to support multiple languages, including C/C++, Swift and Rust, while running only two times slower than the uninstrumented program (in comparison, Valgrind is 20 times slower). Another approach consists of disassembling native code to feed it back to a CTI tool (Lyu et al. 2014). Other frameworks use code instrumentation to fix or remove memory errors automatically for the sake of software resilience (e.g., (Berger & Zorn 2006; Nguyen & Rinard 2007; Long et al. 2014)).

While Fuel is a static tool, it relates to dynamic approaches through its use of runtime checks, guarding portions of code for which static analysis is either impractical or impossible. In fact, static analysis can be understood as a way to optimize runtime checks away wherever conservative assumptions are affordable. This observation has already been noted in the past (Chalupa et al. 2020; Midi et al. 2017; Huang & Morrisett 2013).

2.3. Hardware based approaches

More recently, the CHERI project (Watson et al. 2019) has proposed to enforce memory safety guarantees directly in hardware, through the use of specialized instruction sets and CPUs. The approach relies on so-called *architectural capabilities*, which describe how pointers can be used and to what specific area of memory they provide access. These capabilities are encoded as metadata directly attached to memory addresses, stored in specialized hardware registers and manipulated by *capability-aware* CPU instructions. A set of hardware enforced rules for capability use guarantee that they cannot be tampered with and that all memory accesses that pass through them are safe. Any attempts to illegally access memory protected by capabilities are caught by the CPU instructions and raise hardware interrupts.

The CHERI model is formally verified (Armstrong et al. 2019) and has already been implemented in several CPU designs. The framework has also been tested with multiple case studies, notably to guarantee memory safety properties in C/C++ (Chisnall et al. 2015; Memarian et al. 2019).

Just like DBI, an important difference between CHERI and Fuel is that the former is that the former is entirely runtime-based: illegal accesses to memory are caught solely during programs' execution, raising hardware interrupts. In contrast, Fuel leverages runtime checks to feed additional properties to a purely static analysis. Hence, illegal accesses are caught at compile-time.

2.4. Programming languages

The principles underlying the type systems we have discussed above have made their way into multiple fully-fledged programming languages, providing precious insights for the design of a language-agnostic IR. We briefly review two of them.

Mezzo Our work owes a lot to the Mezzo programming language (Balabonski et al. 2016). Mezzo is a variant of OCaml that uses type capabilities to enforce restrictions on aliasing. Capabilities flow in and out of a function to describe what kind of operation the execution context (or thread) is allowed to perform. The language distinguishes between linear capabilities, representing uniquely owned memory objects, and non-linear ones, representing immutable values. It also advocates for a dynamic mechanism to deal with situations where static capability checking would impose too severe constraints.

In Mezzo, objects can *adopt* a cell, capturing a strong capability. A reverse operation *abandons* an adopted cell, producing a strong capability. This mechanism fits nicely into the traditional encapsulation principle. Objects can carry their own set of related capabilities, while hiding them from the environment.

Rust Rust (Klabnik & Nichols 2018) is aimed at verifying strong memory safety properties by the means of a typing discipline based on uniqueness and ownership. The language associates each pointer with a memory region (Tofte et al. 2004), delimiting its lifetime. The type system then ensures that a pointer can never be stored at a location whose lifetime exceeds that of its region, and uses uniqueness to guarantee data race freedom. Unlike Mezzo, Rust merges the possession of a pointer's value with the capability to dereference it. This results in an arguably more traditional, albeit quite restrictive type system, at the cost of an elaborate region model. Fortunately, the language goes to great lengths to infer region annotations and alleviate the syntactic overhead.

2.5. The three-phase compiler architecture

A well-established engineering technique for taming the complexity of large software systems is to split them into simpler, independent, and ideally interchangeable modules. Modern compilers follow this formula and are implemented as "toolchains" of small programs.[2] While interchangeability is a difficult property to achieve at every step, the whole process can be decomposed into three relatively independent phases, as depicted in Figure 1.

2.5.1. Front-end

The roles of the front-end are twofold: extract and verify the semantics of a program's source. The latter is often provided in the form of human-readable text which has to be tokenized and parsed to create a more abstract representation, called an abstract syntax tree (AST). We refer to this task as the syntactic analysis.

Once syntactic analysis has been completed, various passes of semantic analysis can be carried out on the AST, to detect ill-formed programs, which is notably where type-checking occurs. Compilation may not progress past this point when the program contains errors from which it cannot recover. Instead, the front-end issues an error report, prompting the user to fix the problems encountered.

2.5.2. Optimization and code generation

The optimizer consumes semantically sensible programs and rewrites them in such ways that they should run faster on a machine. This phase typically involves the use of specialized IRs, such as control flow graphs (CFGs) (Alpern et al. 1988) and value dependence

[2] By "program" we mean "loosely coupled components", not necessarily individual executable entities.

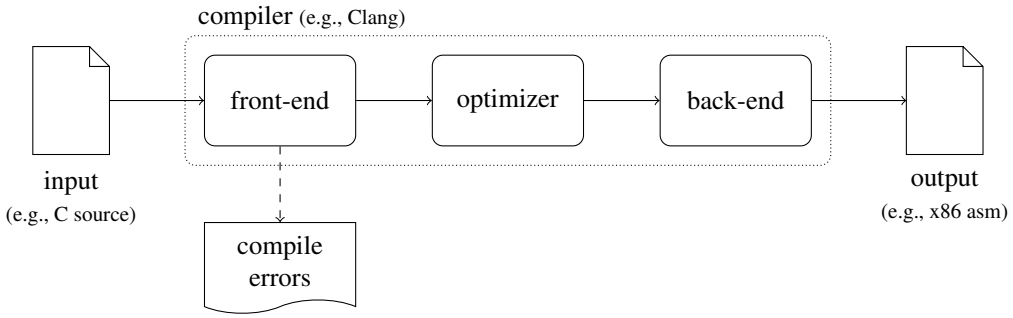

Figure 1 The "three-phase" compiler architecture.

graphs (VDGs) (Weise et al. 1994), which may be more suitable than trees to analyze the program's possible behaviors.

The back-end finally generates code for the target machine. Just as parsing is highly dependent on the source language, code generation is highly dependent on the target. Thus, the back-end may additionally perform some further target-specific optimizations that require knowledge about the target's language and/or hardware (e.g., support for specialized SIMD instructions). It follows that the optimizer is often more tightly coupled with the back-end than it is with the front-end.

Rather than rephrasing the input into some other language, one may directly "execute" the semantic representation produced by the back-end. A compiler operating this way is commonly referred to as an *interpreter*. The boundary between compilation and interpretation is somewhat blurry, as one may argue that a CPU is merely an interpreter for machine code. Nonetheless, the general understanding is that an interpreter is itself a program running another program on top of an existing system.

LLVM Loosening coupling between components not only improves maintainability, but also reusability. If a compiler can be split into three independent phases, then it is tempting to design these phases as reusable components in different toolchains. One influential product of this observation is LLVM (Lattner & Adve 2004). Originally designed as a common framework to perform program optimizations, LLVM has blossomed into a popular middleware in numerous compilers, including Clang (C/C++ and Objective-C), GHC (Haskell) and rustc (Rust). The framework can be seen as the end of a compiler toolchain. It comprises a highly customizable code optimizer and features code generators for various hardware architectures. Its biggest strength is that it is centered around a single IR, called LLVM, that provides a common, front-end agnostic language to conduct static program analysis, apply code optimizations, and generate machine code. This enables compiler developers to unlock a huge collection of tools at the comparatively low engineering cost of translating source code into LLVM IR.

LLVM IR does not feature any control flow structure beyond basic instruction blocks. Loops and conditional statements are expressed by the means of conditional jumps and phi instructions, at the risk of obfuscating some of the original structural properties of a program. Nonetheless, the language features a collection of attribute annotations to specify properties and declare restrictions on pointers. While those are typically not verified by the LLVM architecture itself, they can be leveraged by static analysis tools and code generators.

3. The Fuel framework

Fuel is a compiler framework designed to check memory safety properties on programs with explicit memory management. It is centered around a low-level programming language, Fuel IR. The latter is equipped with a capability-based type system that guarantees freedom from invalid dereference and invalid deallocation errors, and has limited support for leak detection. Hence, static analysis is carried out by simply type checking programs. The entire framework is provided as a library, intended to be integrated within an existing compiler toolchain. It also accepts Fuel IR programs in a human-readable textual form. The current implementation also features a code generator based on LLVM that produces machine code, allowing the entire framework to be used as a standalone compiler.

Figure 2 depicts the envisioned integration of Fuel into an existing compiler. The front-end phase of a compiler (see Figure 1) typically performs four main activities during semantic analysis: *name resolution*, which links identifiers to their declaration, *type inference*, which (semi-)automatically detects the type of the program's expressions, *type checking*, which verifies that the type of each expression—determined by inference or otherwise—is correct, and finally *dataflow analysis*, which relates to flow-sensitive checks. Remark that these activities cannot always be performed in a straight sequence. In particular, name resolution may require hints from type inference to distinguish between overloaded symbols. Nonetheless, the overall process converges as semantic ambiguities are eliminated from the AST, which generally occurs *before* dataflow analysis.

This observation is not surprising. Dataflow analysis requires definite knowledge about the program's control flow, which may depend on information that can only be derived once name resolution and static method dispatch have been settled. Concurrently, this means that a compiler front-end should hold an IR sufficiently rich to run some forms of memory safety analysis at this point, which is exactly where Fuel is intended to be plugged.

At the end of the (flow-insensitive) type checking analysis,

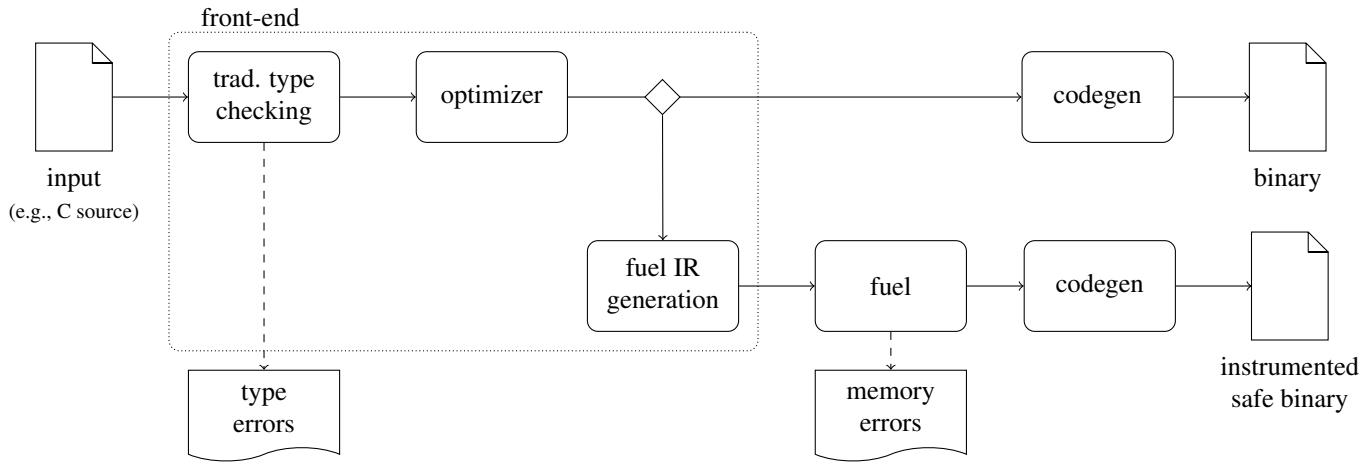

Figure 2 Overview of the envisioned integration of Fuel into a compiler toolchain.

the front-end has two choices. The first is to proceed forward with its current IR to the optimizer phase. The second is to branch off the regular compilation path and translate the program into Fuel IR to perform memory safety checking. In the absence of error, Fuel then forwards an instrumented version of the program to a code generator to terminate the compilation process. Thanks to the combination of static and dynamic checks performed by the framework, the binaries generated at the end of this pipeline are always guaranteed to be free from any memory related errors. Static type checking verifies that all accesses to memory are performed through the required capabilities, and dynamic checks are inserted in the binaries to ensure that all the assumptions made during type checking hold at runtime.

The integration of Fuel in a compiler's pipeline requires language engineers to implement two additional components. One translates the front-end's IR into Fuel IR, whereas the other translates Fuel IR into a binary. Note that the latter requirement is not coupled with the front-end nor the surface language, as Fuel's output can be sent directly to an optimizer.[3]

Dynamic instrumentation

Just as the other memory safety checking tools we discussed in Section 2, Fuel is intended to statically determine whether a program satisfies the properties it is capable of enforcing. In other words, inputs for which Fuel does not produce any error report should be considered safe. Therefore, the reader may wonder about the purpose of code instrumentation.

Types generally offer a very coarse form of abstract interpretation which struggles to describe certain patterns, such as iterators and callbacks (Gordon 2020). Although different techniques have been proposed to address this shortcoming statically, we advocate for a simpler approach, at the cost of a runtime overhead. Rather than forcing pointer capabilities to be tracked across all possible statements, Fuel lets this information be lost during compilation. In exchange, it requires that any access to such pointer be guarded by a dynamic check. These are leveraged to recover assumptions on pointer capabilities.

During the program's execution, dynamic assumptions are checked using metadata managed by the language's runtime, which essentially describe runtime type information about allocated memory. It is maintained in an ad-hoc data structure, threaded through the program's evaluation, but separate from any object's runtime representation. This allows the system to easily erase runtime metadata from programs that can rely on static guarantees alone, without the need for dynamic checks.

This technique is reminiscent of occurrence typing (Tobin-Hochstadt & Felleisen 2008) (a.k.a. flow typing). Occurrence typing is a feature that lets a type checker "deduce" the contextual type of a variable based on the flow of the program. This approach lets type checking "recover" information that might have been lost statically, due to lacking or missing specification. Consider the following example, written in TypeScript:

```
1 function foo(x: any) {
2   if (typeof(x) === "number") {
3     console.log(x + 1)
4   } else {
5     console.log(x - 1) // <- type error
6   }
7 }
```

The function `foo` accepts a single parameter of any type. At line 2, it checks whether the type of that variable is `number`. Hence, at line 3, TypeScript's type checker is able to deduce that `x` *must* have type `number` at that specific program point. Consequently, it can guarantee statically that the addition `x + 1` is type-safe. The same cannot be said at line 5, where knowledge about the type of `x` is still too coarse to guarantee that subtracting is type-safe, causing the type checker to raise an error.

4. Fuel IR

Fuel IR is a hierarchical, typed intermediate language designed to express and check flow-sensitive type assumptions. On a practical side, Fuel IR is heavily inspired by LLVM IR (Lattner & Adve 2004). Although the language features a handful

[3] Our implementation currently supports code generation via LLVM.

of hierarchical control structures, conditional statements and anonymous scopes, it mostly resembles a RISC-like instruction set where most operations serve to describe memory traffic. From a more theoretical side, it borrows from the theory underlying alias types (Smith et al. 2000) and the Mezzo programming language (Balabonski et al. 2016). Fuel IR uses capabilities to keep track of ownership and permissions and offers runtime checks on type assumptions as an escape hatch.

This section introduces the core concepts underlying Fuel IR informally, through a series of examples. We use C and Rust as surface languages to illustrate how common memory operations translate into the IR.

4.1. Tracking initialization

Consider the program in Figure 3a. It declares two variables, one Boolean and one integer, at lines 1 and 2 respectively. The former is assigned to `1` (for *true*) and read to determine the second's value. The corresponding program in Fuel IR is shown in Figure 3b.

Instructions of the form `x = instruction` declare local, temporary registers. Like variables of static single assignment (SSA) form (Cytron et al. 1991), these registers cannot be reassigned. Instead, they designate immutable values, valid for the duration of their declaration scope. The two variables from the C program are translated as explicit stack allocations. Each results in the creation of a *memory cell*, whose address is assigned to a register. Remark that the cell itself is given a name, using the notation `at name` to capture and track memory assignment—we will come back to the purpose of this feature later.

Reading from and writing to the program's memory is carried out explicitly in Fuel IR. The initialization of the variable `b` in the C program is translated to an explicit store instruction, at line 3, which updates the contents of the memory cell located at the address assigned to `breg`. Perhaps more curiously, the memory cell is loaded immediately after, at line 4. This instruction corresponds to the explicit access to the variable's contents, which is performed in the C program at line 5. The loaded value is then used to evaluate the conditional statement's guard before a store instruction finally updates the contents of the cell located at the address assigned to `ireg`.

```
1 bool b;
2 int i;
3 b = 1;
4
5 if (b) { i = 2; }
6   else { i = 4; }
```

(a) C implementation

```
1 breg = salloc Bool at m0
2 ireg = salloc I32 at m1
3 store true, breg
4 bval = load breg
5 if bval { store 2, ireg }
6    else { store 4, ireg }
```

(b) Equivalent code in Fuel IR

Figure 3 Program declaring and initializing local variables.

To illustrate how Fuel uses capabilities to check memory safety, let us revisit the program from Figure 3 while considering its typing semantics. Fuel prescribes that all register declarations create a new type assumption, mapping the register to its type. We refer to these assumptions as *register capabilities*, as they describe the operations that are supported on the register's value. The stack allocation at line 1 produces such a capability. But unlike C, Fuel does not map `breg` to `Bool`. Instead, it creates a capability `[breg: !m0]`. The type `!m0` designates a singleton type (Aspinall 1994) whose unique inhabitant is the address of a memory cell named `m0`.

The stack allocation also creates a *cell capability* for the allocated cell. Here, the cell capability is denoted by `[m0: Junk<Bool>]`, capturing the intuition that, although the cell was declared so that it contains Boolean values, it currently contains uninitialized garbage. The parameterization serves to further specify the cell's layout. The cell contains nonsensical data, but has the size of a Boolean value nonetheless.

At line 3, the store instruction *consumes* `[m0: Junk<Bool>]` to *produce* a new capability `[m0: Bool]`, effectively resulting in a strong update (Smith et al. 2000) denoting the cell's state transition, going from uninitialized to initialized. The load instruction at line 4 yields the capability `[bval: Bool]`, which is used at line 5 to type check the conditional guard.

Consider now an altered version of the C program, where `b`'s assignment at line 3 would be delayed until after the conditional statement (e.g., at line 7). Such a program would expose undefined behavior by using the value of an uninitialized Boolean, but would be statically rejected in Fuel. Indeed, in the absence of the store instruction at line 3, the capability for the cell `m1` would remain unchanged. Hence, the load instruction at line 4 would be ill-typed.

4.2. Tracking aliases

Representing memory locations at the type system's level provides a straightforward mechanism to keep track of aliases. Consider the program in Figure 4a. Two variables are declared, the second of which is a pointer. The address of the former is assigned to the latter at line 3. Notice that `foo`'s value is not initialized until line 4, meaning that after the assignment at line 3, `bar` points to uninitialized memory. Line 8 finally dereferences `bar`, increments its value and assigns the result to `foo`.

```
1 float  foo;
2 float* bar;
3 bar = &foo;
4 foo = 13.37f;
5
6
7
8 foo = 1.0f + *bar;
```

(a) C implementation

```
foo = salloc F32 at m0
bar = salloc ∃a.!a at m1
store foo, bar
store 13.37f, foo
t0 = load bar
t1 = load t0
t2 = call add, 1.0f, t1
store t2, foo
```

(b) Equivalent code in Fuel IR

Figure 4 Program dereferencing a pointer.

The equivalent program in Fuel IR, depicted in Figure 4b, relies on slightly more involved concepts. In particular, the second stack allocation uses an existential type (Pierce 2002, Chapter 24) `∃a.!a` to represent the pointer type `float*` from the C program. Intuitively, an existential type captures the idea that some specific details about the actual type are hidden. In this case, the specific cell `a` whose address will be stored is unknown at the time of the register declaration, and must therefore be existentially quantified.

Remark that the address alone does not specify the kind of values to which it refers. In other words, ∃a.!a is not as precise as `float*`. That is because a stack allocation of the form `x = salloc τ at m` uses the type argument only to specify the layout of the values stored in `m`. Since the layout of an address does not depend on the values stored at this address, any address type will do. Besides, recall that type safety is flow-sensitive in Fuel. Hence, it makes sense that the declaration type of a cell is not sufficient to determine whether all subsequent uses of this cell's value are safe. A similar observation can be made between τ and `Junk<τ>`.

As in our first example, both stack allocations result in the creation of two capabilities. The second pair maps `m1` into `Junk<∃a.!a>`. This capability is consumed at line 3 to strongly update `m1`, resulting in `[m1: !m0]`. Two load instructions translate the expression `*bar` from the C program. The first, at line 5, loads the contents of the cell `m1` into the register `t0`, that is the address of `m0`. That results in a capability `[t0: !m0]`, revealing that `t0` has an address type. The second, at line 6, loads the contents of the cell `m0`, the actual floating point number, into the register `t1`. That results in a capability `[t1: F32]`. The remainder of the program is straightforward. The call instruction applies the function `add`, whose result is eventually stored into `m0`.

The C program from Figure 4a would exhibit undefined behavior if `foo`'s initial assignment was omitted. Remark, however, that the issue would not be caused by the assignment at line 3. Taking the address of an uninitialized variable is not inherently wrong. Instead, the problem would occur at line 8, when `bar` is dereferenced, which is translated by two load instructions, at lines 5 and 6 of the Fuel version. Similarly as in our previous example, the latter would not type check, as it would attempt to dereference a junk value.

4.3. Crossing function boundaries

Intra-procedural approaches have the advantage to support modular analyses, where portions of the codes can be ignored or simply abstracted away (e.g., when the actual source code is unavailable). In exchange, they require function signatures to serve as complete interfaces, describing how data flows in and out. Reference semantics further complicate this requirement because of possible side effects. If a function modifies the contents of a cell referred by one of its arguments, then one cannot rely on its codomain alone to determine the result of its application.

Consider, for instance, the following function header in C: `int libfoo_f(int*)`. This function may simply consult the value pointed by its argument, causing no side effect, but it may also modify this value or even free the pointer. The inability to reason locally about its behavior is the cause of most memory safety challenges. To address this issue, Fuel allows capabilities to occur in domains and/or codomains. In other words, a function's signature does not only describe the type of its parameters, but also captures pre- and post-conditions about its caller's typing environment.

An example is shown in Figure 5. The function has no implementation, which suggests that it is defined externally, perhaps in a different module compiled separately. As a result,

```
1  int libfoo_f(int*)
2
3
4  int main() {
5    int i;
6    i = 42;
7
8    i = libfoo_f(&i);
9
10   return i;
11 }
```

(a) C implementation

```
func libfoo_f(_0):
  ∀a.!a+[a: I32]
    -> I32+[a: I32]
func main(): () -> I32 {
  i = salloc I32 at m0
  store 42, i
  v = call libfoo_f , i
  store v, i
  u = load i
  return u
}
```

(b) Equivalent code in Fuel IR

Figure 5 Program using an external function.

we cannot analyze its behavior to deduce its possible side effects and can only rely on its signature. The Fuel representation expresses such a signature with a universal type (Pierce 2002, Chapter 23), quantifying `a` over all possible memory cells. This polymorphism is necessary to abstract over the actual name of the cell with which the function will be called (Tofte et al. 2004). Intuitively, it indicates that the function is defined for all possible memory locations. The domain is defined as a bundle, wrapping a type together with a capability. The notation `!a +[a: I32]` reads as "the singleton type `!a` and the assumption that the cell named `a` contains an integer value". The codomain is defined similarly, signaling that the function preserves the capability. We will elaborate on this mechanism later.

The function call at line 7 demands that the call site satisfies the assumptions the signature makes about its typing environment. Specifically, it must hold a capability `[a: I32]`, where `a` is the name of the cell passed as an argument. The requirement holds after `[m0: I32]` is produced by the store instruction at line 6, without which the program would be ill-typed.

4.4. Exploiting uniqueness and borrowing

We said earlier that the function `libfoo_f` from Figure 5b preserved the capability `[a: I32]`. At a finer level of detail, the capability is actually consumed from the function's domain, and a new one is produced with its codomain. In other words, it is treated as a linear resource (Wadler 1990), meaning that one's ability to interact with the contents of a cell is restricted to the scope of a single function at a time. When `libfoo_f` is applied, its caller loses the capability to dereference or use load/store instructions, until the function gives it back. One can exploit this mechanism to fight against dangling references. Deallocating a memory cell requires its associated type capability to be consumed. Therefore, the function `libfoo_f` is guaranteed not to deallocate its argument, as doing otherwise would prevent it from returning the associated capability. Similarly, it could not cause the capability to escape silently by packaging it into an existential bundle type. Another consequence is that the same address cannot be provided to two different arguments, unless they share the same singleton type. For instance, consider the following function signature:

```
1 func libfoo_g(x, y):
2     ∀a,b.(!a, !b)+[a: I32, b: I32] -> Void
```

One cannot instantiate the variables `a` and `b` with the same memory cell `c` to call `libfoo_g` if there is no way to duplicate `[c: I32]`. The reader may remark that this constraint only manifests because the domain features type capabilities. Indeed, if one is willing to drop the capability `[b: I32]` from the second parameter, then `b` could be instantiated with the same value as `a`. However, the function would lose the ability to infer statically whether dereferencing its second argument is safe. The intuition is that copying pointer values does not cause problems, while using them requires more attention.

The use of capabilities to enforce resources linearity can be used to express the mechanism of mutable borrowing in languages with affine or linear type systems, like Rust or Pony. Figure 6 shows an example. The Rust function `swap` borrows two mutable references, to mutate their contents. In Fuel, the full capabilities for the arguments are consumed and then returned by the function, effectively transferring user and mutation rights back to the caller.

```
fn swap<'a>(
  x: &'a mut i32,
  y: &'a mut i32
) -> () {
  let z = *x;
  *x = *y;
  *y = z;
}
```

(a) Rust implementation

```
func swap(x, y):
  ∀a,b.(!a, !b)
  +[a: I32, b: I32]
  -> [a: I32, b: I32] {
  z = load x
  store y, x
  store z, y
}
```

(b) Equivalent code in Fuel IR

Figure 6 Function borrowing its arguments to mutate them.

Of course, exchanging capabilities can be done at any depth in nested calls. Put differently, a caller is free to transfer a capability to a callee regardless of the way it came into its possession. Hence, the Rust-like borrowing showcased in Figure 6 is allowed in nested and recursive calls. Lifetime constraints are encoded by the return type annotation of a function.

Not all capabilities need to be linear. As noted in previous work (Crary et al. 1999), linear capabilities are only necessary to perform operations that require uniqueness. Indeed, even in a concurrent setting, unrestricted read accesses to shared memory are safe, as long as all parties agree not to perform any mutating operation. Fuel leverages this observation to support safe, non-linear cell capabilities through borrowing (Naden et al. 2012). Capabilities qualified by the `@brw` modifier are non-linear. They can be used to read a cell's contents and can be duplicated at will. In exchange, a few restrictions apply. First, they do not grant permission to perform any mutation, nor to deallocate a cell. Second, they can only be obtained by weakening a linear capability at function boundaries. This mechanism naturally delineates the duration of the loan, which necessarily expires when the function returns. Finally, just as all non-linear capabilities, they cannot be packaged into an existential bundle type (e.g. $\exists$`a.!a+[a: `τ`]`). This third constraint guarantees that borrowed capabilities cannot escape the stack.

Figure 7 shows an example of borrowed capabilities. The two parameters are declared as bundled types, each wrapping a singleton type with an associated capability. Since the capabilities are qualified by `@brw`, the variables `a` and `b` can be instantiated with the same memory cell. Line 15 is therefore legal. The linear capability `[m0: I32]` is temporarily weakened as a non-linear capability `[@brw(m0: I32)]`, which can be duplicated for each argument. The function `equals` does not need to return any capability. The call site automatically reinstates the linear capability, based on the assumption that the loan expires when the function returns. Within the function, the typing environment holds two *different* borrowed capabilities, one for each argument, supporting both `load` instructions at lines 4 and 6. Furthermore, they effectively allow the creation of a "may-alias" relation: both arguments *may* refer to the same memory cell. Soundness is preserved nonetheless because mutation is forbidden.

```
1  func equals(x, y):
2    ∀a, b.(!a, !b)+[@brw(a: I32), @brw(b: I32)]
3    -> Bool {
4    xv = load x  // uses [m0: I32] borrowed as
5                 // [@brw(a: I32)]
6    yv = load y  // uses [m0: I32] borrowed as
7                 // [@brw(b: I32)]
8    rv = call eq, xv, yv
9    return rv
10 }
11
12 func main(): () -> I32 {
13   i = salloc I32 at m0    // => [m0: Junk<I32>]
14   store 42, i             // => [m0: I32]
15   b = call equals, i, i   // lends [m0: I32]
16                           // temporarily
17   return 0
18 }
```

Figure 7 Program using borrowed capabilities.

4.5. Exploiting runtime checks

One weakness of the type system we have presented so far is that it does not allow a function to conditionally consume or produce linear capabilities. Imagine a function `free_one(x, y)` that would accept two pointers and free one of them and leave the other intact, depending on some condition. Such a function would necessarily have to consume two linear capabilities, or it would not be able to free either of them, de facto precluding borrowing. A defensive strategy would be to assume both arguments `x` and `y` are freed. Unfortunately, that would introduce a memory leak, as the call site would definitely lose the ability to deallocate the memory pointed by the pointer left intact.

We address both limitations with dynamic checks to test for specific type capabilities at runtime. In addition to borrowed capabilities, capabilities qualified by the `@dyn` modifier (for *dynamic*) are also treated non-linearly. Unlike the former, these can be used to perform mutating operations, provided their use is guarded by a runtime check that verifies whether the associated cell can be safely dereferenced.

Figure 8 showcases the use of dynamic capabilities. The function `free_one(x, y)` is signed so that it accepts two dynamic capabilities. Hence, within the function, the environment holds two different dynamic capabilities for each argument, just as in

```
1  func free_one(x, y):
2    ∀a, b.(!a, !b)+[a: @dyn(I32), b: @dyn(I32)]
3    -> Void
4  func main(): () -> I32 {
5    i = halloc I32 at m0     // => [m0: Junk<I32>]
6    j = halloc I32 at m1     // => [m1: Junk<I32>]
7    store 42, i              // => [m0: I32]
8    store 24, j              // => [m1: I32]
9    _ = call free_one, i, j // trades
10                            // [m0: I32, m1: I32]
11                            // for [@dyn(m0:I32),
12                            //      @dyn(m1:I32)]
13   assuming i: i32 { free i } // recover [m0: I32]
14                              // to free m0
15   assuming j: i32 { free j } // recover [m1: I32]
16                              // to free m1
17   return 0
18 }
```

Figure 8 Program using dynamic capabilities.

the example shown in Figure 7. However, neither of them can be used directly, even for read accesses. Instead, they must be guarded by dynamic checks similar to that of line 13. The statement tests whether `[@dyn(a: I32)]` can be traded for a regular capability `[a: I32]`, in which case `free` is executed, consuming `[a: I32]` to delete the associated memory cell. Meanwhile, the call site permanently weakens the capability `[m0: I32]` produced by the `store` instruction at line 7 to create `[@dyn(m0: i32)]`. This capability is no longer linear. Hence, it can be copied once for each of `free_one`'s arguments.

One important restriction must be guaranteed to preserve the soundness of this mechanism. Specifically, a dynamic capability cannot be traded for a regular one if another capability already exists for the same memory cell, with a different symbol. In other words, dynamic capabilities do not allow mutable may-alias relations, as those may compromise the alias tracking mechanism.

4.6. Capability erasure

Capabilities only exist at compile time: they represent the resources that the type checker uses to reason about memory. It follows that a code generator can simply erase all information pertaining to static capabilities once type checking has been completed. The picture is slightly more complex for dynamic capabilities, because `assuming` statements require the runtime to record sufficient information to test their existence during the program's evaluation.

Once again, we can draw an analogy with occurrence typing. Types only exist at compile type: they represent information that the type checker knows about the nature of a specific expression, and they are typically erased during code generation. Nonetheless, an operator like TypeScript's `typeof` requires the runtime to record sufficient information to test the type of a particular value during the program's evaluation.

A solution to keep track of the necessary information to check dynamic capabilities is to store additional metadata in the representation of all pointers. For instance, one could allocate extra memory for every pointer in the program to store its type and keep track of its uniqueness. This technique is typically used by binary instrumentation tools such as Valgrind or CHERI.

Another solution is to store additional metadata in an ad-hoc data structure managed by the runtime and instrument the binary so that memory operations (e.g., `load`) appropriately update the data structure. This approach allows erasure during code generations for pointers that are known to be never associated with a dynamic capability. Hence, no runtime overhead will incur for programs that can be type-checked without any `assuming`, such as the one shown in Figure 6b.

5. Formal definition

This section describes Fuel IR formally. We first introduce some of the notations we use, before we delve into the language's syntax and semantics.

5.1. Notations

Let $f : A \to B$ be a function, $dom(f)$ and $codom(f)$ denote its domain and codomain, respectively. If f is a partial function, then $dom(f)$ is the subset $A' \subseteq A$ for which f is defined. We write $f = [\bot]_{A\to B}$ to represent a partial function $f : A \to B$ with $dom(f) = \varnothing$. We write $f = [a \mapsto b]_{A\to B}$ to represent a partial function f such that $f(a) = b$ with $dom(f) = \{a\}$. We write $f = [a \mapsto f'(a) \mid p(a)]_{A\to B}$ for a partial function from A to B such that $\forall a \in \{a \in A \mid p(a)\}, f(a) = f'(a)$. For example, $[i \mapsto -i \mid i < 0]_{\mathbb{Z}\to\mathbb{Z}}$ denotes a function that maps each negative integer number to its absolute value. We omit the subscript $A \to B$ when domains and codomains are obvious from the context.

We write $f \lceil a$ for a partial function that is not defined for a but is identical to f everywhere else. More formally, if $f' = f \lceil a$ then $dom(f') = dom(f) \setminus a$ and $\forall a' \in dom(f'), f'(a') = f(a')$. We write $f \ll f'$ for a function that returns $f'(a)$ if $a \in dom(f')$ or $f(a)$ otherwise, and whose domain is $dom(f) \cup dom(f')$. More formally, $\forall a \in dom(f) \cup dom(f')$,

$$(f \ll f')(a) = \begin{cases} f'(a) & \text{if } a \in dom(f') \\ f(a) & \text{otherwise} \end{cases}$$

Let A be an enumerable set. We write A^* for the set of words over A, that is the set of sequences of elements of A. Most often, we use an overbar to distinguish sequences from other objects. Let $\overline{w}, \overline{w}' \in A^*$ be two words, $\overline{w} \cdot \overline{w}'$ denotes their concatenation. For example, let $\overline{w} = xy$ and $\overline{w}' = yz$ be two words over $\{x, y, z\}$, then $\overline{w} \cdot \overline{w}' = xyyz$. We write ϵ for the empty sequence, that is $\overline{w} \cdot \epsilon = \overline{w}$. We write $\overline{w}_i$ for the letter at the i-th position in $\overline{w}$. For example, if $\overline{w} = xyz$, then $\overline{w}_1 = x$ and $\overline{w}_3 = z$. Finally, we write $\overline{w}_{i\ldots}$ for the sub-sequence in $\overline{w}$ that starts at its i-th position. For example, if $\overline{w} = xyz$, then $\overline{w}_{2\ldots} = yz$.

Let $\overline{f} = f_1 \ldots f_n$ be a sequence of (partial) functions $A \to B$. We write $\overline{f}(a)$ as a shorthand for $\overline{f}_i(a)$, where $\overline{f}_i$ is the first occurrence in $\overline{f}$ where $a \in dom(f_i)$. For example, given a sequence of functions $\overline{g} = [a_0 \mapsto b_0] \cdot [a_0 \mapsto b_1] \cdot [a_1 \mapsto b_1]$, then $\overline{g}(a_0) = b_0$ and $\overline{g}(a_1) = b_1$. We also write $\overline{f} \lceil a$ for the sequence of functions where a is removed from the domain

of the first function in which it was defined. For example, $\overline{g}\lceil a_0 = [\bot] \cdot [a_0 \mapsto b_1] \cdot [a_1 \mapsto b_1]$.

5.2. Syntax

The language's syntax appears in Figure 9. A program is a sequence of function declarations, one of which is designated as its entry point. We follow C's convention and hold this function to be named `main`. We assume that all registers are uniquely named in the entire lexical scope of a function. Put differently, we assume that an identifier x never appears twice on the left side of a static assignment in a function's body. We assume that return statements never appear in nested scopes (e.g., in the branch of a conditional statement). Although they are part of the abstract syntax, statements of the form $x = \{\ \overline{s}\ \}$ only represent the intermediate steps involved in the evaluation of a function's body in the operational semantics and do not appear in written programs. Finally, note that the `pack` and `unpack` instructions have no operational significance; they only serve to rearrange the static typing context.

Remark that functions cannot be created locally (e.g., within another function's scope) and do not have closures. However, their name is globally visible, providing support for reentrancy and mutual recursion. Besides, function names can be manipulated as first-class values (i.e., stored in registers, passed as arguments, etc.). Just like in C, this means that one can emulate higher-order functions by associating a defunctionalized (Reynolds 1998a) function symbol with a data structure storing the values in its closure. For the sake of syntactic regularity, we assume that all functions have a universal type. Functions that are not quantified by memory cells are simply written $\forall\epsilon.\langle\tau \to \tau', \epsilon\rangle$.

Note that the formal syntax introduced in figure 9 slightly differs from the concrete syntax of Fuel programs. Consider for example the signature of function `swap` in figure 6b, written `∀ a,b.(!a, !b)+[a: I32, b: I32] -> [a: I32, b: I32]` in the IR's concrete syntax. This signature translates to $\forall a, b.\langle !a, !b \to \mathit{Unit}, \texttt{@own}(a : I32), \texttt{@own}(b : I32) \to \texttt{@own}(a : I32), \texttt{@own}(b : I32)\rangle$ in our formal syntax.

5.3. Operational semantics

Despite sitting at a relatively low level, the language abstracts away most details about memory representation. Both registers and memory cells are arbitrary large blocks of memory, but fixed in the sense that their size is determined statically. This size is agnostic of the actual representation on a specific target machine. Simple scalar values, such as booleans or addresses are assumed to occupy a single cell (but not necessarily of the same size), and data aggregates are as large as their member count. We also assume that function objects occupy the same space if they share the same type, regardless of their implementation.

Definition 5.1 (Values)**.** A runtime value v is a constant, a cell name, an aggregate of values or a function. More formally,

$$v ::= c \mid a \mid \{\ v_1, \ldots, v_n\ \} \mid \texttt{func}\ x(y)\ \{\ \overline{s}\ \}$$

We write V to denote the set of all values.

A program is evaluated in a runtime environment that keeps track of a machine's state defined by three components. The first represents the machine's memory and maps cell names to runtime values. The second is a list of partial functions, each of which maps local register names to runtime values. The third relates to dynamic capability checks.

Definition 5.2 (Runtime Environment)**.** Let T be the set of types. A runtime environment is a triple $\mathcal{R} = \langle m, \overline{r}, \overline{\gamma}\rangle$ where:

- $m : A \to V$ represents the machine's memory,
- $\overline{r} \in (X \to V)^*$ associates local registers to their value,
- $\overline{\gamma} \in (X \cup A \to T)^*$ associates local registers and cell names to their type.

The most relevant pieces of the operational semantics appear in Figure 10, which defines a small-step evaluation operator of the form $\overline{s}, \mathcal{R} \rightsquigarrow \overline{s}', \mathcal{R}'$. All rules are relatively straightforward, but we draw the reader's attention to a few details. The notation $\mathcal{R} \vdash e \Downarrow v$ denotes the interpretation of an expression e in an environment $\mathcal{R}$. Such an interpretation does not involve any "computation" and, consequently, has no side effect on the machine's state. Similarly, the notation $\mathcal{R} \vdash e : \tau$ denotes the evaluation of an expression's type τ, given a runtime environment. Note that we cannot determine a value's type from its raw representation alone, as the environment only maps symbols (i.e., register and cell names) to their types. Nonetheless, since the judgment operates on syntactic representations, we can assume that the latter carry enough information.

$$\frac{}{\langle m, \overline{r}, \overline{\gamma}\rangle \vdash c \Downarrow c} \qquad \frac{\overline{r}(x) = v}{\langle m, \overline{r}, \overline{\gamma}\rangle \vdash x \Downarrow v}$$

$$\frac{c \in \mathit{dom}(\tau)}{\langle m, \overline{r}, \overline{\gamma}\rangle \vdash c : \tau} \qquad \frac{\overline{\gamma}(x) = \tau}{\langle m, \overline{r}, \overline{\gamma}\rangle \vdash x : \tau}$$

For simplicity, we only formalize stack memory management abstractly and do not model automatic deallocation in the operational semantics. All allocations result in the creation of a new, unique cell name which is not removed from the environment's domain when the corresponding scope is exited (e.g., when a function returns). This omission does not have any effect on well-formed programs. Dangling references and attempts to deallocate stack memory are ill-typed under our runtime semantics. A cell's initial value is obtained via a helper function *junk*, which accepts a type and produces uninitialized data with a compatible layout. For instance, $\mathit{junk}(\texttt{Bool}) = \texttt{junk}\langle\texttt{Bool}\rangle$. Finally, given a sequence of instructions $\overline{s}$, we write $\overline{s}[a/a']$ for the sequence in which occurrences of a are substituted for a'.

5.4. Typing semantics

Flow-sensitive typing is carried out by running an abstract interpretation of the program, which checks whether type capabilities are used appropriately. This process is described with a judgment of the form $\mathcal{C} \vdash \overline{s} \dashv \mathcal{C}'$, partially formalized in Figure 11. The judgment holds if the sequence of statements $\overline{s}$ is well-formed in a typing context $\mathcal{C}$, and its interpretation results in the updated context $\mathcal{C}'$. It follows that a typing context can be seen

$$x, y \in \mathbb{X} \quad \textit{(identifier)}; \quad a \in \mathbb{A} \quad \textit{(cell name)}; \quad i \in \mathbb{N} \quad \textit{(member offset)}$$

$$
\begin{array}{llll}
\textit{func. decl.} & d & ::= & \texttt{func}\ x(y) : \forall \overline{a}.\langle \tau \to \tau', \overline{\kappa} \to \overline{\kappa}' \rangle\ \{\ \overline{s}\ \} \\
\textit{type} & \tau & ::= & \texttt{Unit} \mid \texttt{Junk}\langle \tau \rangle \mid \texttt{Bool} \mid !a \mid \{\ \overline{\tau_1}\ \} \\
 & & \mid & \exists \overline{a}.\langle \tau, \overline{\kappa} \rangle \mid \forall \overline{a}.\langle \tau \to \tau', \overline{\kappa} \to \overline{\kappa}' \rangle \\
\textit{capability} & \kappa & ::= & \texttt{@own}(\sigma : \tau) \mid \texttt{@brw}(a : \tau) \mid \texttt{@dyn}(a : \tau) \\
\textit{symbol} & \sigma & ::= & x \mid a \\
\textit{expression} & e & ::= & x \mid c \\
\textit{constant} & c & ::= & \texttt{unit} \mid \texttt{junk}\langle \tau \rangle \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil}\langle \tau \rangle \\
\textit{statement} & s & ::= & x = \texttt{salloc}\ \tau\ \texttt{at}\ a \mid x = \texttt{halloc}\ \tau\ \texttt{at}\ a \mid \texttt{free}\ x \\
 & & \mid & x = \texttt{load}\ y \mid \texttt{store}\ e, x \mid x = \texttt{extract}\ e, i \mid x = \texttt{insert}\ e, e', i \\
 & & \mid & x = \texttt{call}\ y, e \mid \texttt{return}\ e \mid x = \{\ \overline{s}\ \} \mid \{\ \overline{s}\ \} \\
 & & \mid & \texttt{if}\ e\ \{\ \overline{s}\ \}\ \texttt{else}\ \{\ \overline{s}'\ \} \mid \texttt{unwrapping}\ x\ \{\ \overline{s}\ \}\ \texttt{else}\ \{\ \overline{s}'\ \} \\
 & & \mid & \texttt{assuming}\ x : \tau\ \{\ \overline{s}\ \}\ \texttt{else}\ \{\ \overline{s}'\ \} \\
 & & \mid & \texttt{pack}\ x\ \texttt{with}\ \overline{\kappa}\ \texttt{as}\ \exists \overline{a}.\langle \tau, \overline{\kappa}' \rangle \mid \texttt{unpack}\ x\ \texttt{with}\ \overline{a}
\end{array}
$$

Figure 9 Formal syntax

as the static counterpart of a runtime environment, representing the abstract machine's state.

Definition 5.3 (Typing Context)**.** Let $Q = \{\texttt{@own}, \texttt{@brw}, \texttt{@dyn}\}$ denote the set of capability qualifiers and T be the set of types. A typing context is a triple $\mathcal{C} = \langle \mu, \overline{\rho}, \overline{\alpha} \rangle$ where:

- $\mu : A \to Q \times T$ keeps track of cell capabilities,
- $\overline{\rho} \in (X \to Q \times T)^*$ keeps track of register capabilities,
- $\overline{\alpha} \in \mathcal{P}(A)^*$ keeps track of managed cells.

Cells allocations and deallocations Memory cells are allocated explicitly, either on the stack (Rule T-STACK-ALLOC) or on the heap (Rule T-HEAP-ALLOC). In both cases, a new entry is inserted in the cell capability table μ, resulting in a strong capability (marked by `@own`) mapping the new cell name to a junk type. The register capability table is also updated to associate the new register with the new cell. Finally, stack-allocations update the current set of managed cells $\overline{\alpha}_1$, indicating that the freshly created capability must be reclaimed automatically at the end of the current scope.

Rule T-FREE formalizes explicit deallocation of unmanaged memory. It requires that the given target register be known to contain a cell name, for which the typing environment holds a strong capability. If both conditions are met, then the cell is discarded from the domain of the capability table μ. Note that the register's value does not need to be modified. The forfeiture of the capability is enough to guarantee that the cell cannot be dereferenced.

Rule T-BLOCK describes the automatic deallocation of managed memory. It removes all the cells bound to the current scope from the capability table. A key detail is the check ensuring that the current set of managed cells $\overline{\alpha}_1$ is contained within the table's domain, prescribing that all stack allocated cells must not be permanently consumed by any instruction (e.g. a `free` statement) before the end of the scope.

The reader may wonder why we do not identify invalid deallocations with the Rule T-FREE using the set of managed cells $\overline{\alpha}$ to determine which cells cannot be deallocated. While that approach would work for cells allocated in the same function as the `free` instruction and likely result in more precise error reporting, it would not be applicable beyond function boundaries. The problem is that we cannot determine where a particular cell has been allocated when it has been passed as an argument, as $\overline{\alpha}$ only contains the name of the cells allocated *within* the function being type-checked. Unfortunately, extending the framework to support this distinction would complicate the type system substantially.

Example 5.1 (Premature deallocation)**.** Consider this snippet:

```
1 {
2   // μ = [⊥]
3   foo = salloc Bool at a
4   // μ = [a ↦ @own(Junk⟨Bool⟩)]
5   free foo
6   // μ = [⊥]
7 }
8 // a ∉ dom(μ)
```

The system fails to type-check the last line. The rule T-BLOCK does not apply because the cell `a` is missing from the domain of the capability table μ. Remark that the same code would be well-typed if the stack allocation at line 3 was replaced by heap allocation, as `a` would not have been added to the set of managed cells.

STACK-ALLOC

$$\frac{\begin{array}{c}a' \text{ fresh in } m \text{ and } \overline{s} \\ m' = m \ll [a' \mapsto \mathit{junk}(\tau)] \qquad \overline{r}' = \overline{r}_1 \ll [x \mapsto a'] \cdot \overline{r}_{2\ldots} \\ \overline{\gamma}' = \overline{\gamma}_1 \ll [x \mapsto\, !a, a \mapsto \texttt{Junk}\langle\tau\rangle] \cdot \overline{\gamma}_{2\ldots}\end{array}}{x = \texttt{salloc}\ \tau\ \texttt{at}\ a \cdot \overline{s}, \langle m, \overline{r}, \overline{\gamma}\rangle \rightsquigarrow \overline{s}[a/a'], \langle m', \overline{r}', \overline{\gamma}'\rangle}$$

FREE

$$\frac{m' = m\lceil\overline{r}(x) \qquad \overline{\gamma}' = \overline{\gamma}\lceil\overline{r}(x)}{\texttt{free}\ x \cdot \overline{s}, \langle m, \overline{r}, \overline{\gamma}\rangle \rightsquigarrow \overline{s}, \langle m', \overline{r}, \overline{\gamma}'\rangle}$$

LOAD

$$\frac{\begin{array}{c}\overline{r}' = \overline{r}_1 \ll [x \mapsto m(\overline{r}(y))] \cdot \overline{r}_{2\ldots} \\ \overline{\gamma}' = \overline{\gamma}_1 \ll [x \mapsto \overline{\gamma}(\overline{r}(y))] \cdot \overline{\gamma}_{2\ldots}\end{array}}{x = \texttt{load}\ y \cdot \overline{s}, \langle m, \overline{r}, \overline{\gamma}\rangle \rightsquigarrow \overline{s}, \langle m, \overline{r}', \overline{\gamma}'\rangle}$$

STORE

$$\frac{\begin{array}{cc}\langle m, \overline{r}, \overline{\gamma}\rangle \vdash e \Downarrow v & m' = m \ll [\overline{r}(x) \mapsto v] \\ \langle m, \overline{r}, \overline{\gamma}\rangle \vdash e : \tau & \overline{\gamma}' = \overline{\gamma} \ll [\overline{r}(x) \mapsto \tau]\end{array}}{\texttt{store}\ e, x \cdot \overline{s}, \langle m, \overline{r}, \overline{\gamma}\rangle \rightsquigarrow \overline{s}, \langle m', \overline{r}, \overline{\gamma}'\rangle}$$

RETURN

$$\frac{\begin{array}{cc}\langle m, \overline{r}, \overline{\gamma}\rangle \vdash e \Downarrow v & \overline{r}' = \overline{r}_2 \ll [x \mapsto v] \cdot \overline{r}_{3\ldots} \\ \langle m, \overline{r}, \overline{\gamma}\rangle \vdash e : \tau & \overline{\gamma}' = \overline{\gamma}_2 \ll [x \mapsto \tau] \cdot \overline{\gamma}_{3\ldots}\end{array}}{x = \{\ \texttt{return}\ e\ \} \cdot \overline{s}, \langle m, \overline{r}, \overline{\gamma}\rangle \rightsquigarrow \overline{s}, \langle m, \overline{r}', \overline{\gamma}'\rangle}$$

CALL

$$\frac{\begin{array}{c}\overline{r}(y) = \texttt{func}\ x'(y')\ \{\ \overline{s}'\ \} \\ \langle m, \overline{r}, \overline{\gamma}\rangle \vdash e \Downarrow v \qquad \overline{r}' = [y' \mapsto v] \cdot \overline{r} \\ \langle m, \overline{r}, \overline{\gamma}\rangle \vdash e : \tau \qquad \overline{\gamma}' = [y' \mapsto \tau] \cdot \overline{\gamma}\end{array}}{x = \texttt{call}\ y, e \cdot \overline{s}, \langle m, \overline{r}, \overline{\gamma}\rangle \rightsquigarrow x = \{\ \overline{s}'\ \}, \langle m, \overline{r}', \overline{\gamma}'\rangle}$$

ASSUMING-TRUE

$$\frac{\begin{array}{c}\overline{\gamma}(x) = !a \\ \forall \sigma \in \mathit{dom}(\overline{\gamma}_1), \sigma \neq x \implies \overline{\gamma}_1(\sigma) \neq !a \qquad \overline{\gamma}(a) = \tau\end{array}}{\begin{array}{c}\texttt{assuming}\ x : \tau\ \{\ \overline{s}\ \}\ \texttt{else}\ \{\ \overline{s}'\ \} \cdot \overline{s}'', \langle m, \overline{r}, \overline{\gamma}\rangle \\ \rightsquigarrow \{\ \overline{s}\ \} \cdot \overline{s}'', \langle m, \overline{r}, \overline{\gamma}\rangle\end{array}}$$

Figure 10 Operational semantics (selected rules)

Loading and storing values As mentioned earlier, regular values in Fuel can be copied at will, even memory cell names. On the other hand, capabilities that grant the ability to perform mutating operations must be treated linearly. Hence, to preserve soundness, packaging a linear capability together with a value has to result in a linear object. Linear objects cannot be copied. Instead, they should be *moved*, invalidating the register or cell previously holding them. We define a predicate *lin* to identify types of linear objects (i.e., linear types), formally defined as the minimal predicate satisfying the following rules:

$$\frac{}{\mathit{lin}(\exists \overline{a}.\langle \tau, \overline{\kappa}\rangle)} \qquad \frac{\exists i, \mathit{lin}(\tau_i)}{\mathit{lin}(\{\ \tau_1, \ldots, \tau_n\ \})}$$

Rule T-LOAD-COPY formalizes the static semantics of `load` instructions on non-linear types. It requires a strong or borrowed capability to access the cell referenced by the source register, whose value gets copied into the target register. In contrast, Rule T-LOAD-MOVE applies when the source value is linear. In this case, a strong cell capability has to be consumed to invalidate the cell's value, by changing its type. Both rules additionally check that the loaded cell does not have a junk type.

Example 5.2 (Loading a linear object)**.** Consider a `load` instruction of the form $x = \texttt{load}\ y$, type-checked in a context such that $\mu = [a \mapsto \texttt{@own}(\exists b.\langle !b, \texttt{@own}(b : \texttt{Bool})\rangle)]$ and $\overline{\rho}_1 = [y \mapsto !a]$. Rule T-LOAD-MOVE applies, because a's type is linear. It updates the capability table such that $\mu = [a \mapsto \texttt{@own}(\texttt{Junk}\langle\ldots\rangle)]$, effectively invalidating a's value. In other words, one can no longer perform a `load` instruction on y.

The two rules for `store` instructions are defined similarly: T-STORE-COPY applies for non-linear values whereas T-STORE-MOVE applies on linear values, held in a temporary register. In both instances, a strong cell capability is required to update its type. The notation $\tau \sim \tau'$ additionally ensures that τ and τ' have a compatible memory layout.

Function calls As Fuel relies purely on an intraprocedural analysis, function calls play a central role in Fuel's typing semantics, formalized by Rules T-CALL-COPY. Let $x = \texttt{call}\ y, e$ be a call statement, the rule first ensures that y holds a function object, by verifying that it has a function type $\forall \overline{a}.\langle \tau \to \tau', \overline{\kappa} \to \overline{\kappa}'\rangle$. Next, it checks that the type τ_e of the expression e matches that of the function's domain τ, prompting the instantiation of the universal type. We define a partial function *match* for this purpose. Given two types τ_e and τ, as well as a sequence of capabilities $\overline{\kappa}$, the function determines whether there exists a substitution s mapping the universally quantified names $\overline{a}$ to concrete names from the typing context so that τ_e matches τ (i.e., τ_e describes the same layout as τ) and that all capabilities $\overline{\kappa}$ can be derived from the context; for example:

$$[a_0 \mapsto \texttt{@own}(\texttt{Unit})], \epsilon \vdash \mathit{match}(!a_0, !a, \texttt{@brw}(a : \texttt{Unit})) = [a \mapsto a_0]$$

Provided a substitution exists, the next step consists of consuming the capabilities required by the function from the typing context. We define another partial function *take* for this purpose. The function accepts three arguments, namely a sequence of capabilities to consume $\overline{\kappa}$, a capability table μ and a borrowed capability table β. The latter serves to handle the creation of borrowed capabilities. We formalize its semantics in Figure 12. Consuming a strong capability (Rule T-TAKE-STRONG) removes it from the context, while consuming a borrowed or dynamic capability (Rule T-TAKE-WEAK) leaves it unchanged. Two rules further describe how strong capabilities are weakened. T-TAKE-MAKE-DYN permanently trades a strong capability for a dynamic one. Finally, T-TAKE-MAKE-BRW creates a borrowed capability by inserting a new entry in β. Although the capability is not removed from the context, notice that this prevents further applications of T-TAKE-STRONG and T-TAKE-MAKE-DYN, hence avoiding unsound borrows. The rule finally merges the cell capabilities produced or returned by the function, and creates a register capability for the return value.

T-STACK-ALLOC

$$\frac{\begin{array}{c}\overline{\alpha}' = \overline{\alpha}_1 \cup \{a\} \cdot \overline{\alpha}_{2...} \\ \mu' = \mu \ll [a \mapsto \texttt{@own}(\texttt{Junk}\langle\tau\rangle)] \\ \overline{\rho}' = \overline{\rho}_1 \ll [x \mapsto \texttt{@own}(!a)] \cdot \overline{\rho}_{2...}\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash x = \texttt{salloc}\ \tau\ \texttt{at}\ a \dashv \mu', \overline{\rho}', \overline{\alpha}'}$$

T-HEAP-ALLOC

$$\frac{\begin{array}{c}\mu' = \mu \ll [a \mapsto \texttt{@own}(\texttt{Junk}\langle\tau\rangle)] \\ \overline{\rho}' = \overline{\rho}_1 \ll [x \mapsto \texttt{@own}(!a)] \cdot \overline{\rho}_{2...}\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash x = \texttt{halloc}\ \tau\ \texttt{at}\ a \dashv \mu', \overline{\rho}', \overline{\alpha}}$$

T-BLOCK

$$\frac{\begin{array}{c}\mu, \overline{\rho}, \overline{\alpha} \vdash \overline{s} \dashv \mu', \overline{\rho}', \overline{\alpha}' \qquad \overline{\alpha}'_1 \subseteq dom(\mu') \\ \mu'' = \mu' \lceil \overline{\alpha}'_1 \qquad \overline{\rho}'' = \overline{\rho}'_{2...} \qquad \overline{\alpha}'' = \overline{\alpha}'_{2...}\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash \{\ \overline{s}\ \} \dashv \mu'', \overline{\rho}'', \overline{\alpha}''}$$

T-FREE

$$\frac{\overline{\rho}(x) = \texttt{@own}(!a) \qquad \mu(a) = \texttt{@own}(\tau) \qquad \mu' = \mu \lceil a}{\mu, \overline{\rho}, \overline{\alpha} \vdash \texttt{free}\ x \dashv \mu', \overline{\rho}, \overline{\alpha}}$$

T-LOAD-COPY

$$\frac{\begin{array}{c}\overline{\rho}(y) = \texttt{@own}(!a) \qquad \mu(a) = q(\tau) \\ q \neq \texttt{@dyn} \qquad \neg lin(\tau) \qquad \tau \neq \texttt{Junk}\langle\tau'\rangle \\ \overline{\rho}' = \overline{\rho}_1 \ll [x \mapsto \texttt{@own}(\tau)] \cdot \overline{\rho}_{2...}\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash x = \texttt{load}\ y \dashv \mu, \overline{\rho}', \overline{\alpha}}$$

T-LOAD-MOVE

$$\frac{\begin{array}{c}\overline{\rho}(y) = \texttt{@own}(!a) \qquad \mu(a) = \texttt{@own}(\tau) \\ lin(\tau) \qquad \mu' = \mu \ll [a \mapsto \texttt{@own}(\texttt{Junk}\langle\tau\rangle)] \\ \overline{\rho}' = \overline{\rho}_1 \ll [x \mapsto \texttt{@own}(\tau)] \cdot \overline{\rho}_{2...}\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash x = \texttt{load}\ y \dashv \mu', \overline{\rho}', \overline{\alpha}}$$

T-STORE-COPY

$$\frac{\begin{array}{c}\overline{\rho}(x) = \texttt{@own}(!a) \qquad \mu(a) = \texttt{@own}(\tau) \\ \mu, \overline{\rho} \vdash e : \tau' \qquad \neg lin(\tau') \qquad \tau \sim \tau' \\ \mu' = \mu \ll [a \mapsto \texttt{@own}(\tau')]\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash \texttt{store}\ e, x \dashv \mu', \overline{\rho}, \overline{\alpha}}$$

T-STORE-MOVE

$$\frac{\begin{array}{c}\overline{\rho}(x) = \texttt{@own}(!a) \qquad \mu(a) = \texttt{@own}(\tau) \qquad \mu, \overline{\rho} \vdash y : \tau' \\ lin(\tau') \qquad \tau \sim \tau' \qquad \mu' = \mu \ll [a \mapsto \texttt{@own}(\tau')] \\ \overline{\rho}' = \overline{\rho}_1 \ll [x \mapsto \texttt{@own}(\texttt{Junk}\langle\tau\rangle)] \cdot \overline{\rho}_{2...}\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash \texttt{store}\ y, x \dashv \mu', \overline{\rho}', \overline{\alpha}}$$

T-CALL-COPY

$$\frac{\begin{array}{c}\overline{\rho}(y) = \texttt{@own}(\forall \overline{a}.\langle \tau \to \tau', \overline{\kappa} \to \overline{\kappa}' \rangle) \qquad \mu, \overline{\rho} \vdash e :\ \tau_e \qquad \neg lin(\tau_e) \\ \mu, \overline{\rho} \vdash match(\tau_e, \tau, \overline{\kappa}) = s \qquad \mu' = take(\overline{\kappa}[/s], \mu, [\bot]) \qquad \mu'' = \mu' \ll \overline{\kappa}'[/s] \qquad \overline{\rho}' = \overline{\rho}_1 \ll [x \mapsto \texttt{@own}(\tau'[/s])] \cdot \overline{\rho}_{2...}\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash x = \texttt{call}\ y, e \dashv \mu'', \overline{\rho}', \overline{\alpha}}$$

T-ASSUMING

$$\frac{\begin{array}{c}\overline{\rho} = \texttt{@own}(!a) \qquad \mu(a) = q(\tau) \qquad q \neq \texttt{@brw} \qquad \mu' = \mu_a \sqcup \mu_b \qquad \overline{\rho}' = \overline{\rho}_a \sqcup \overline{\rho}_b \\ \mu \ll [a \mapsto \texttt{@own}(\tau)], \overline{\rho}, \overline{\alpha} \vdash \{\ \overline{s}\ \} \dashv \mu_a, \overline{\rho}_a, \overline{\alpha} \qquad \mu \lceil a, \overline{\rho}, \overline{\alpha} \vdash \{\ \overline{s}'\ \} \dashv \mu_b, \overline{\rho}_b, \overline{\alpha}\end{array}}{\mu, \overline{\rho}, \overline{\alpha} \vdash \texttt{assuming}\ x : \tau\ \{\ \overline{s}\ \}\ \texttt{else}\ \{\ \overline{s}'\ \} \dashv \mu', \overline{\rho}', \overline{\alpha}}$$

Figure 11 Statements' typing semantics (selected rules): $\mathcal{C} \vdash \overline{s} \dashv \mathcal{C}'$

Example 5.3 (Consuming capabilities for a call)**.** Let $take(\mu, \overline{\kappa}, [\bot]) = \mu'$, where $\mu = [a \mapsto \texttt{@own}(\tau), b \mapsto \texttt{@own}(\tau), c \mapsto \texttt{@dyn}(\tau)]$ be a capability table and $\overline{\kappa} = \texttt{@own}(a : \tau) \cdot \texttt{@brw}(b : \tau) \cdot \texttt{@dyn}(c : \tau)$ a sequence of capabilities to consume. Rule T-TAKE-STRONG applies first, removing a from μ's domain. Rule T-TAKE-MAKE-BRW applies next, weakening b's capability by creating an entry in β. Rule T-TAKE-WEAK applies, ascertaining that the context contains a dynamic capability for c without modifying it. Finally, Rule T-TAKE-EMPTY applies on an empty list of capabilities.

Notice that Rule T-CALL-COPY prescribes that the argument's type be non-linear. Just like in the case of a `store` instruction, a register capability may be taken when τ is a linear type, packaging a capability within an existential type. The sister rule handles this situation similarly to Rule S-STORE-MOVE.

Capability assumptions tests Rule T-ASSUMING formalizes type-checking for capability assumption tests. It first requires that the tested capability be at least *mentioned* in the typing context, with any qualifier but `@brw`. Recall that assumption tests are meant to recover strong capabilities. Obviously, this can never succeed if the capability is already statically known to be weak. The whole point of the construction is to leverage dynamic checks to recover static knowledge about the typing context. Hence, it only makes sense that the static semantics should explore both branches of the conditional. Nonetheless, notice that each of them is interpreted in a different context. The first branch is type-checked after updating a's capability to a strong one, while the second branch is type-checked after removing a's capability altogether. This effectively simulates two possible situations: either a is indeed available in the environment, or it is not.

The contexts resulting from both branches are then *joined*. This operation, denoted by the operator $\sqcup$[4], merges two capability tables, resolving conflicts by introducing dynamic capabilities. More formally, $t \sqcup t' = t''$ if and only if:

[4] We extend the operator to sequences of capability tables of equal lengths by joining them pairwise, that is $\overline{t} \sqcup \overline{t}' = \overline{t}_1 \sqcup \overline{t}'_1, \ldots, \overline{t}_n \sqcup \overline{t}'_n$.

$$\text{T-TAKE-EMPTY}\quad \frac{}{take(\epsilon, \mu, \beta) = \mu} \qquad \text{T-TAKE-STRONG}\quad \frac{\mu(a) = \texttt{@own}(\tau) \quad a \notin dom(\beta) \quad \mu' = take(\overline{\kappa}, \mu[a, \beta)}{take(\texttt{@own}(a : \tau) \cdot \overline{\kappa}, \mu, \beta) = \mu'}$$

$$\text{T-TAKE-WEAK}\quad \frac{(\mu \ll \beta)(a) = q(\tau) \quad q \neq \texttt{@own} \quad \mu' = take(\overline{\kappa}, \mu, \beta)}{take(q(a : \tau) \cdot \overline{\kappa}, \mu, \beta) = \mu'}$$

$$\text{T-TAKE-MAKE-BRW}\quad \frac{\mu(a) = \texttt{@own}(\tau) \quad a \notin dom(\beta) \quad \mu' = take(\overline{\kappa}, \beta \ll [a \mapsto \texttt{@brw}(\tau)])}{take(\texttt{@brw}(a : \tau) \cdot \overline{\kappa}, \mu, \beta) = \mu'}$$

$$\text{T-TAKE-MAKE-DYN}\quad \frac{\mu(a) = \texttt{@own}(\tau) \quad a \notin dom(\beta) \quad \mu' = take(\overline{\kappa}, \mu \ll [a \mapsto \texttt{@dyn}(\tau)], \beta)}{take(\texttt{@dyn}(a : \tau) \cdot \overline{\kappa}, \mu, \beta) = \mu'}$$

Figure 12 Capability consumption in function calls

- $\forall \sigma \in dom(t) \cap dom(t'), t(\sigma) = q(\tau) \wedge t'(\sigma) = q'(\tau') \implies \tau = \tau'$
- $dom(t'') = dom(t) \cup dom(t')$
- $\forall \sigma \in dom(t''), t''(\sigma) =$
$$\begin{cases} t(\sigma) & \text{if } t(\sigma) = t'(\sigma) \\ \texttt{@dyn}(\tau) & \text{if } \sigma \notin dom(t') \vee \sigma \notin dom(t) \\ \texttt{@dyn}(\tau) & \text{if } \{t(\sigma), t'(\sigma)\} = \{\texttt{@own}(\tau), \texttt{@dyn}(\tau)\} \end{cases}$$

Remark that joining is not defined for tables that map the same symbol to different types (e.g., $\texttt{@own}(\texttt{Bool})$ and $\texttt{@own}(\texttt{Junk}\langle\texttt{Bool}\rangle)$). This prevents the two branches of a conditional statement from leaving the same register or memory cell in different memory configurations. One way to lift this limitation would be to equip the type system with a subtyping relation to substitute mismatching types with their least supertype. However, such an extension would also call for the addition of a runtime subcasting mechanism.

Function declarations A program is well-formed if all its function declarations are well-typed. Each declaration is typed-checked in an isolated typing context, built solely from the signatures of all globally visible symbols, thus including that of the declaration itself. Therefore, just as call statements do not assume anything about the function being called beyond their signature, functions do not make any more assumptions about their caller. Rule T-FUNC (see Figure 13) describes well-typed function declarations formally. In a nutshell, the rule creates a new typing context, type-checks the function's statements and confirms that its post-conditions are satisfied.

The cell capability table μ is created solely from the capabilities taken by the function. For example, given a function signature $\forall a.\langle !a \to \texttt{Unit}, \texttt{@own}(a : \tau) \to \texttt{@own}(a : \tau)\rangle$ (i.e., a function accepting a pointer to a cell containing a value of type τ), the table μ would be initialized as $[a \mapsto \texttt{@own}(\texttt{Bool})]$. The symbol F denotes a mapping from global identifiers to their declarations, created at the start of the type-checking process by gathering all global declarations. The register capability $\overline{\rho}$ is created by prefixing F with a mapping associating the argument y to the type of the function's parameter. These two tables serve to type-check the function's statements, before the rule finally looks at the function's post-conditions. It checks that all stack-allocated memory can be reclaimed (similar to how it is done by Rule T-BLOCK), that all returned capabilities are available in the context, and that the returned value has the expected type. Note that the special identifier $\$r$ designates the return register.

$$\text{T-FUNC}\quad \frac{\mu = \{\overline{\kappa}_i \mid 1 \leq i \leq |\overline{\kappa}|\} \quad \overline{\rho} = [y \mapsto \tau] \cdot F \quad \mu, \overline{\rho}, \varnothing \vdash \overline{s} \dashv \mu', \overline{\rho}', \alpha' \quad \alpha' \subseteq dom(\mu') \quad \{\overline{\kappa}'_i \mid 1 \leq i \leq |\overline{\kappa}'|\} \subseteq \mu' \quad \overline{\rho}'(\$r) = \texttt{@own}(\tau')}{F \vdash \texttt{func}\ x(y) : \forall \overline{a}.\langle \overline{\tau} \to \overline{\tau}', \overline{\kappa} \to \overline{\kappa}'\rangle\ \{\ \overline{s}\ \}}$$

Figure 13 Typing semantics of function declarations

A noteworthy property of this model is its ability to predict heap memory leaks. Since one cannot access the contents of a memory cell without a capability, any remaining strong capability at the end of a function scope that is not mentioned in its signature denotes a leak. Indeed, since the call site does not have access a callee's typing context, it has no way to recover the capabilities that are not returned, nor automatically reclaimed. Similarly, linear objects that remain in the register capability table also constitute leaks. While the register's value can be assumed to be deallocated, the memory cells referred to by the packaged capabilities cannot. Furthermore, those are known to be heap-allocated, otherwise the function would be ill-typed. Unfortunately, dynamic capabilities may still introduce untraceable leaks, precisely because they represent uncertainty about the actual state of the runtime environment. Hence, their presence may denote a false positive.

5.5. Soundness

Our static semantics guarantees that well-typed programs can always progress to a *valid configuration*. A configuration is a sequence of statements $\overline{s}$ paired with a runtime environment $\mathcal{R}$. It is *valid* if either of these two conditions holds:

- there is no more statement to evaluate (i.e., $\overline{s} = \epsilon$); or
- the program is not stuck (i.e., there exists a reduction rule such that $\overline{s}, \mathcal{R} \rightsquigarrow \overline{s}', \mathcal{R}'$).

We first describe a relation between runtime environments and typing contexts, which holds if the latter is a faithful description of the former. Intuitively, it prescribes that a capability that appears in a typing context must be matched by a corresponding entry in the environment. Notice the asymmetry: a symbol σ being mapped onto a value v in $\mathcal{R}$ does not need to be matched by a corresponding capability in $\mathcal{C}$. However, all capabilities featured in $\mathcal{C}$ must be verified in $\mathcal{R}$. This allows typing contexts to underspecify environments.

Definition 5.4 (Environment Typing)**.** Let $\mathcal{R} = \langle m, \overline{r}, \overline{\gamma} \rangle$ be a runtime environment and $\mathcal{C} = \langle \mu, \overline{\rho}, \overline{\alpha} \rangle$ be a typing context. We write $\mathcal{R} : \mathcal{C}$ if and only if:

- $\forall a \in dom(\mu), \mu(a) = q(\tau) \implies m(a) : \tau$
- $\forall x \in dom(\overline{\rho}), \overline{\rho}(x) = q(\tau) \implies \overline{r}(x) : \tau$

Our typing rules focus on maintaining an environment-typing invariant. We express this idea formally with the following soundness property; its proof is left for future work.

Theorem 5.1 (Soundness)**.** *Let $\mathcal{C}$ be a typing context and $\mathcal{R}$ be a runtime environment such that $\mathcal{R} : \mathcal{C}$. If $\mathcal{C} \vdash \overline{s} \dashv \mathcal{C}'$ and $\overline{s}, \mathcal{R} \rightsquigarrow^* \overline{s}', \mathcal{R}'$, then $\overline{s}', \mathcal{R}'$ is a valid configuration.*

6. Conclusion

We have presented Fuel, a compiler framework designed to verify memory safety properties on programs using explicit memory management. Drawing inspiration from LLVM, Fuel features a low-level, typed intermediate language, called Fuel IR, intended as a front-end agnostic target for any source language. The language uses type capabilities to eliminate initialization issues, null dereference, use-after-free, doubly freed memory and inaccessible memory. All properties are verified statically on programs transformed in Fuel IR, using a type-checking analysis that leverages dynamic checks to recover static assumptions about the typing environment. We have introduced our framework informally, through a short tutorial showcasing how to translate common low-level idioms from C and Rust to Fuel IR, and formalized a subset for which we have described the operational and static semantics.

6.1. Current limitations

A handful of outstanding limitations prevent Fuel from being used in a concrete setting in its current state. In particular, Fuel IR's lacks the following critical features:

Self-recursive data structures Our language is currently unable to represent self-recursive data structures, such as linked-lists or trees. Two features are missing: a support for recursive definitions and optional types (a.k.a., maybe monads). The former is a simple extension of Fuel IR's type system, while the latter calls for minor amendments to its operational semantics. Specifically, it requires the addition of a conditional statement able to branch behavior depending on the presence or absence of a value.

Arrays Arrays are another fundamental data structure that Fuel IR is currently unable to represent. In particular, the language lacks the ability to compute indices dynamically. Unfortunately, such a feature would introduce uncertainty as to determine which member of the tuple is involved in a particular operation. One possible countermeasure would be to leverage dynamic checks, updating the type of each member after any mutating operation.

Another issue is to determine whether a computed index actually falls within the array's bounds. Dynamic solutions typically instrument code with runtime checks (Duck & Yap 2016), while static alternatives use symbolic execution (Rugina & Rinard 2005) or dependent types (Xi & Pfenning 1998) to remove them.

Concurrency Fuel does not yet provide any support for concurrency. While extending the language with threads should be possible, this feature suggests further questions about the additional mechanisms that should come with it. Low-level languages typically offer locks and synchronization primitives, while their higher-level counterpart may rely on more elaborate patterns, such as actors and promises.

6.2. Discussion and future work

Our long-term objective is to develop a scalable, off-the-shelf framework to express and verify memory safety properties in existing compiler toolchains. We identified three main research questions related to this goal, which this work partially answers:

RQ1 *What is the appropriate expressiveness to reason about mutable memory states?*

By dissociating the possession of a pointer value from the capability to read or modify memory, Fuel's type system is more amenable than traditional substructural approaches. We have presented a formal semantics for Fuel IR. The proof is expected to follow from the traditional style of Wright and Felleisen (Wright & Felleisen 1994).

The use of runtime assumptions in concert with static type checking is our most significant contribution, which can be understood as occurrence typing (Tobin-Hochstadt & Felleisen 2008) for flow-sensitive memory properties. It uncovers promising leads to support dynamic features that are typically too impractical to check statically.

RQ2 *What is the appropriate level of abstraction for a memory safety checking framework?*

We believe that starting from LLVM IR, adding explicit scoping constructs (e.g., `assuming`), and supporting richer functions APIs yields a suitable model to express universal memory operations while remaining non-opinionated about high-level features, such as class-based polymorphism or closure implementations.

Scopes in particular provide an elegant way to delimit the lifetime of an assumption and reason about the state of the memory at a particular point in the program. In other words, Fuel is more *structured* than LLVM IR.

RQ3 *Where should a memory safety checking framework be plugged into a compiler toolchain?*

This question requires more future research. As we observed in this paper, all information required to perform memory safety checking is typically available within the front-end phase, after semantic analysis has fixed the type of each expression. Besides, structural information should still be available at this stage, allowing scope information to be translated into Fuel IR.

Nonetheless, it remains to more formally identify what properties of a source language are strictly necessary to

generate well-formed Fuel IR code and discover how common code patterns can be translated optimally. Another issue is to determine whether Fuel IR code generation is reasonably automatable. Only a thorough practical evaluation can provide definite answers.

Other avenues for future work include the exploration of Fuel's applicability to other domains than memory safety checking. One particularly exciting prospect is to leverage its type system in the context of JIT compilation. By providing richer foreign function interfaces, advertising their possible side effects, one can hope to leverage local reasoning to exploit more optimization opportunities.

References

Alpern, B., Wegman, M. N., & Zadeck, F. K. (1988). Detecting equality of variables in programs. In J. Ferrante & P. Mager (Eds.), *Conference record of the fifteenth annual ACM symposium on principles of programming languages, san diego, california, usa, january 10-13, 1988* (pp. 1–11). ACM Press. Retrieved from https://doi.org/10.1145/73560.73561 doi: 10.1145/73560.73561

Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K. E., Norton-Wright, R., … others (2019). Isa semantics for armv8-a, risc-v, and cheri-mips.

Aspinall, D. (1994). Subtyping with singleton types. In L. Pacholski & J. Tiuryn (Eds.), *Computer science logic, 8th international workshop, CSL '94, kazimierz, poland, september 25-30, 1994, selected papers* (Vol. 933, pp. 1–15). Berlin: Springer. Retrieved from https://doi.org/10.1007/BFb0022243 doi: 10.1007/BFb0022243

Balabonski, T., Pottier, F., & Protzenko, J. (2016). The design and formalization of mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems*, *38*(4), 14:1–14:94. Retrieved from http://dl.acm.org/citation.cfm?id=2837022

Ballantyne, M., King, A., & Felleisen, M. (2020). Macros for domain-specific languages. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 229:1–229:29. Retrieved from https://doi.org/10.1145/3428297 doi: 10.1145/3428297

Barnett, M., Chang, B. E., DeLine, R., Jacobs, B., & Leino, K. R. M. (2005). Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, & W. P. de Roever (Eds.), *Formal methods for components and objects, 4th international symposium, FMCO 2005, amsterdam, the netherlands, november 1-4, 2005, revised lectures* (Vol. 4111, pp. 364–387). Springer. Retrieved from https://doi.org/10.1007/11804192_17 doi: 10.1007/11804192_17

Berger, E. D., & Zorn, B. G. (2006). Diehard: probabilistic memory safety for unsafe languages. In M. I. Schwartzbach & T. Ball (Eds.), *Proceedings of the ACM SIGPLAN 2006 conference on programming language design and implementation, ottawa, ontario, canada, june 11-14, 2006* (pp. 158–168). ACM. Retrieved from https://doi.org/10.1145/1133981.1134000 doi: 10.1145/1133981.1134000

Bloch, J. J. (2008). *Effective java, 2nd edition*. Addison-Wesley. Retrieved from https://www.worldcat.org/oclc/255160742

Boyland, J., Noble, J., & Retert, W. (2001). Capabilities for sharing: A generalisation of uniqueness and read-only. In *European conference on object-oriented programming, ecoop 2001* (pp. 2–27). Retrieved from https://doi.org/10.1007/3-540-45337-7_2 doi: 10.1007/3-540-45337-7_2

Boyland, J. T. (2010). Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, *32*(6), 22:1–22:33. Retrieved from https://doi.org/10.1145/1749608.1749611 doi: 10.1145/1749608.1749611

Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E. B., Pun, K. I., … Yang, A. M. (2015). Parallel objects for multicores: A glimpse at the parallel language encore. In M. Bernardo & E. B. Johnsen (Eds.), *Formal methods for multicore programming - 15th international school on formal methods for the design of computer, communication, and software systems, SFM 2015, bertinoro, italy, june 15-19, 2015, advanced lectures* (Vol. 9104, pp. 1–56). Springer. Retrieved from https://doi.org/10.1007/978-3-319-18941-3_1 doi: 10.1007/978-3-319-18941-3_1

Buck, B. R., & Hollingsworth, J. K. (2000). An API for runtime code patching. *International Journal of High Performance Computing Applications*, *14*(4), 317–329. Retrieved from https://doi.org/10.1177/109434200001400404 doi: 10.1177/109434200001400404

Chalupa, M., Strejcek, J., & Vitovská, M. (2020). Joint forces for memory safety checking revisited. *International Journal on Software Tools for Technology Transfer*, *22*(2), 115–133. Retrieved from https://doi.org/10.1007/s10009-019-00526-2 doi: 10.1007/s10009-019-00526-2

Chisnall, D., Rothwell, C., Watson, R. N., Woodruff, J., Vadera, M., Moore, S. W., … Neumann, P. G. (2015). Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. *ACM SIGARCH Computer Architecture News*, *43*(1), 117–130.

Clang Project. (2021). *Clang static analyzer.* Retrieved from https://clang-analyzer.llvm.org ((accessed January 9, 2021))

Clarke, D., Östlund, J., Sergey, I., & Wrigstad, T. (2013). Ownership types: A survey. In D. Clarke, J. Noble, & T. Wrigstad (Eds.), *Aliasing in object-oriented programming. types, analysis and verification* (Vol. 7850, pp. 15–58). Springer. Retrieved from https://doi.org/10.1007/978-3-642-36946-9_3 doi: 10.1007/978-3-642-36946-9_3

Clebsch, S., Drossopoulou, S., Blessing, S., & McNeil, A. (2015). Deny capabilities for safe, fast actors. In E. G. Boix, P. Haller, A. Ricci, & C. Varela (Eds.), *International workshop on programming based on actors, agents, and decentralized control, agere! 2015* (pp. 1–12). ACM. Retrieved from https://doi.org/10.1145/2824815.2824816 doi: 10.1145/2824815.2824816

Crary, K., Walker, D., & Morrisett, J. G. (1999). Typed memory management in a calculus of capabilities. In A. W. Appel & A. Aiken (Eds.), *POPL '99, proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages, san antonio, tx, usa, january 20-22, 1999* (pp. 262–275). New York: ACM. Retrieved from https://doi.org/

10.1145/292540.292564 doi: 10.1145/292540.292564

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, *13*(4), 451–490. Retrieved from https://doi.org/10.1145/115372.115320 doi: 10.1145/115372.115320

Dietl, W., Dietzel, S., Ernst, M. D., Muslu, K., & Schiller, T. W. (2011). Building and using pluggable type-checkers. In R. N. Taylor, H. C. Gall, & N. Medvidovic (Eds.), *Proceedings of the 33rd international conference on software engineering, ICSE 2011, waikiki, honolulu , hi, usa, may 21-28, 2011* (pp. 681–690). ACM. Retrieved from https://doi.org/10.1145/1985793.1985889 doi: 10.1145/1985793.1985889

Duck, G. J., & Yap, R. H. C. (2016). Heap bounds protection with low fat pointers. In A. Zaks & M. V. Hermenegildo (Eds.), *Proceedings of the 25th international conference on compiler construction, CC 2016, barcelona, spain, march 12-18, 2016* (pp. 132–142). ACM. Retrieved from https://doi.org/10.1145/2892208.2892212 doi: 10.1145/2892208.2892212

Fähndrich, M., & DeLine, R. (2002). Adoption and focus: Practical linear types for imperative programming. In J. Knoop & L. J. Hendren (Eds.), *Proceedings of the 2002 ACM SIGPLAN conference on programming language design and implementation (pldi), berlin, germany, june 17-19, 2002* (pp. 13–24). ACM. Retrieved from https://doi.org/10.1145/512529.512532 doi: 10.1145/512529.512532

Filliâtre, J., & Paskevich, A. (2013). Why3 - where programs meet provers. In M. Felleisen & P. Gardner (Eds.), *Programming languages and systems - 22nd european symposium on programming, ESOP 2013, held as part of the european joint conferences on theory and practice of software, ETAPS 2013, rome, italy, march 16-24, 2013. proceedings* (Vol. 7792, pp. 125–128). Springer. Retrieved from https://doi.org/10.1007/978-3-642-37036-6_8 doi: 10.1007/978-3-642-37036-6_8

Gordon, C. S. (2020). Designing with static capabilities and effects: Use, mention, and invariants (pearl). In R. Hirschfeld & T. Pape (Eds.), *34th european conference on object-oriented programming, ECOOP 2020, november 15-17, 2020, berlin, germany (virtual conference)* (Vol. 166, pp. 10:1–10:25). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Retrieved from https://doi.org/10.4230/LIPIcs.ECOOP.2020.10 doi: 10.4230/LIPIcs.ECOOP.2020.10

Hanenberg, S., Kleinschmager, S., Robbes, R., Tanter, É., & Stefik, A. (2014). An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, *19*(5), 1335–1382. Retrieved from https://doi.org/10.1007/s10664-013-9289-1 doi: 10.1007/s10664-013-9289-1

Huang, D., & Morrisett, G. (2013). Formalizing the safecode type system. In G. Gonthier & M. Norrish (Eds.), *Certified programs and proofs - third international conference, CPP 2013, melbourne, vic, australia, december 11-13, 2013, proceedings* (Vol. 8307, pp. 211–226). Springer. Retrieved from https://doi.org/10.1007/978-3-319-03545-1_14 doi: 10.1007/978-3-319-03545-1_14

Ichikawa, K., & Chiba, S. (2017). User-defined operators including name binding for new language constructs. *The Art, Science, and Engineering of Programming*, *1*(2), 15. Retrieved from https://doi.org/10.22152/programming-journal.org/2017/1/15 doi: 10.22152/programming-journal.org/2017/1/15

Imai, K., Neykova, R., Yoshida, N., & Yuen, S. (2020). Multiparty session programming with global protocol combinators. In R. Hirschfeld & T. Pape (Eds.), *34th european conference on object-oriented programming, ECOOP 2020, november 15-17, 2020, berlin, germany (virtual conference)* (Vol. 166, pp. 9:1–9:30). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Retrieved from https://doi.org/10.4230/LIPIcs.ECOOP.2020.9 doi: 10.4230/LIPIcs.ECOOP.2020.9

Kabir, I., Li, Y., & Lhoták, O. (2020). ιdot: a DOT calculus with object initialization. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 208:1–208:28. Retrieved from https://doi.org/10.1145/3428276 doi: 10.1145/3428276

Klabnik, S., & Nichols, C. (2018). *The rust programming language*. USA: No Starch Press.

Lattner, C., & Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM international symposium on code generation and optimization (CGO 2004), 20-24 march 2004, san jose, ca, USA* (pp. 75–88). IEEE Computer Society. Retrieved from https://doi.org/10.1109/CGO.2004.1281665 doi: 10.1109/CGO.2004.1281665

Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2014). *The java virtual machine specification, java se 8 edition* (1st ed.). Boston: Addison-Wesley Professional.

Liu, F., Lhoták, O., Biboudis, A., Giarrusso, P. G., & Odersky, M. (2020). A type-and-effect system for object initialization. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 175:1–175:28. Retrieved from https://doi.org/10.1145/3428243 doi: 10.1145/3428243

Long, F., Sidiroglou-Douskos, S., & Rinard, M. C. (2014). Automatic runtime error repair and containment via recovery shepherding. In M. F. P. O'Boyle & K. Pingali (Eds.), *ACM SIGPLAN conference on programming language design and implementation, PLDI '14, edinburgh, united kingdom - june 09 - 11, 2014* (pp. 227–238). ACM. Retrieved from https://doi.org/10.1145/2594291.2594337 doi: 10.1145/2594291.2594337

Lyu, Y., Hong, D., Wu, T., Wu, J., Hsu, W., Liu, P., & Yew, P. (2014). DBILL: an efficient and retargetable dynamic binary instrumentation framework using llvm backend. In M. Hirzel, E. Petrank, & D. Tsafrir (Eds.), *10th ACM SIGPLAN/SIGOPS international conference on virtual execution environments, VEE '14, salt lake city, ut, usa, march 01 - 02, 2014* (pp. 141–152). ACM. Retrieved from https://doi.org/10.1145/2576195.2576213 doi: 10.1145/2576195.2576213

Maeda, T., Sato, H., & Yonezawa, A. (2011). Extended alias type system using separating implication. In S. Weirich & D. Dreyer (Eds.), *Proceedings of TLDI 2011: 2011 ACM SIGPLAN international workshop on types in languages design and implementation, austin, tx, usa, january 25, 2011*

(pp. 29–42). ACM. Retrieved from https://doi.org/10.1145/1929553.1929559 doi: 10.1145/1929553.1929559

Marjamäki, D. (2021). *Cppcheck*. Retrieved from http://cppcheck.sourceforge.net ((accessed January 9, 2021))

Memarian, K., Gomes, V. B., Davis, B., Kell, S., Richardson, A., Watson, R. N., & Sewell, P. (2019). Exploring c semantics and pointer provenance. *Proceedings of the ACM on Programming Languages*, *3*(POPL), 1–32.

Midi, D., Payer, M., & Bertino, E. (2017). Memory safety for embedded devices with nescheck. In R. Karri, O. Sinanoglu, A. Sadeghi, & X. Yi (Eds.), *Proceedings of the 2017 ACM on asia conference on computer and communications security, asiaccs 2017, abu dhabi, united arab emirates, april 2-6, 2017* (pp. 127–139). ACM. Retrieved from https://doi.org/10.1145/3052973.3053014 doi: 10.1145/3052973.3053014

Müller, P., Schwerhoff, M., & Summers, A. J. (2017). Viper: A verification infrastructure for permission-based reasoning. In A. Pretschner, D. Peled, & T. Hutzelmann (Eds.), *Dependable software systems engineering* (Vol. 50, pp. 104–125). IOS Press. Retrieved from https://doi.org/10.3233/978-1-61499-810-5-104 doi: 10.3233/978-1-61499-810-5-104

Naden, K., Bocchino, R., Aldrich, J., & Bierhoff, K. (2012). A type system for borrowing permissions. In J. Field & M. Hicks (Eds.), *Proceedings of the 39th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2012, philadelphia, pennsylvania, usa, january 22-28, 2012* (pp. 557–570). New York: ACM. Retrieved from https://doi.org/10.1145/2103656.2103722 doi: 10.1145/2103656.2103722

Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In J. Ferrante & K. S. McKinley (Eds.), *Proceedings of the ACM SIGPLAN 2007 conference on programming language design and implementation, san diego, california, usa, june 10-13, 2007* (pp. 89–100). New York: ACM. Retrieved from https://doi.org/10.1145/1250734.1250746 doi: 10.1145/1250734.1250746

Nguyen, H. H., & Rinard, M. C. (2007). Detecting and eliminating memory leaks using cyclic memory allocation. In G. Morrisett & M. Sagiv (Eds.), *Proceedings of the 6th international symposium on memory management, ISMM 2007, montreal, quebec, canada, october 21-22, 2007* (pp. 15–30). ACM. Retrieved from https://doi.org/10.1145/1296907.1296912 doi: 10.1145/1296907.1296912

Nielson, F., & Nielson, H. R. (1999). Type and effect systems. In E. Olderog & B. Steffen (Eds.), *Correct system design, recent insight and advances, (to hans langmaack on the occasion of his retirement from his professorship at the university of kiel)* (Vol. 1710, pp. 114–136). Springer. Retrieved from https://doi.org/10.1007/3-540-48092-7_6 doi: 10.1007/3-540-48092-7_6

Nieto, A., Zhao, Y., Lhoták, O., Chang, A., & Pu, J. (2020). Scala with explicit nulls. In R. Hirschfeld & T. Pape (Eds.), *34th european conference on object-oriented programming, ECOOP 2020, november 15-17, 2020, berlin, germany (virtual conference)* (Vol. 166, pp. 25:1–25:26). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Retrieved from https://doi.org/10.4230/LIPIcs.ECOOP.2020.25 doi: 10.4230/LIPIcs.ECOOP.2020.25

Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

Racordon, D., & Buchs, D. (2020). Featherweight swift: a core calculus for swift's type system. In R. Lämmel, L. Tratt, & J. de Lara (Eds.), *Proceedings of the 13th ACM SIGPLAN international conference on software language engineering, SLE 2020, virtual event, usa, november 16-17, 2020* (pp. 140–154). ACM. Retrieved from https://doi.org/10.1145/3426425.3426939 doi: 10.1145/3426425.3426939

Reynolds, J. C. (1998a). Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, *11*(4), 363–397. Retrieved from https://doi.org/10.1023/A:1010027404223 doi: 10.1023/A:1010027404223

Reynolds, J. C. (1998b). *Theories of programming languages*. Cambridge University Press.

Rugina, R., & Rinard, M. C. (2005). Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems*, *27*(2), 185–235. Retrieved from https://doi.org/10.1145/1057387.1057388 doi: 10.1145/1057387.1057388

Rytz, L., Amin, N., & Odersky, M. (2013). A flow-insensitive, modular effect system for purity. In W. Dietl (Ed.), *Formal techniques for java-like programs* (pp. 4:1–4:7). New York, NY: ACM. Retrieved from https://doi.org/10.1145/2489804.2489808 doi: 10.1145/2489804.2489808

Schubert, P. D., Hermann, B., & Bodden, E. (2019). Phasar: An inter-procedural static analysis framework for C/C++. In T. Vojnar & L. Zhang (Eds.), *Tools and algorithms for the construction and analysis of systems - 25th international conference, TACAS 2019, held as part of the european joint conferences on theory and practice of software, ETAPS 2019, prague, czech republic, april 6-11, 2019, proceedings, part II* (Vol. 11428, pp. 393–410). Springer. Retrieved from https://doi.org/10.1007/978-3-030-17465-1_22 doi: 10.1007/978-3-030-17465-1_22

Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. (2012). Addresssanitizer: A fast address sanity checker. In G. Heiser & W. C. Hsieh (Eds.), *2012 USENIX annual technical conference, boston, ma, usa, june 13-15, 2012* (pp. 309–318). USENIX Association. Retrieved from https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

Smaragdakis, Y., & Balatsouras, G. (2015). Pointer analysis. *Foundations and Trends in Programming Languages*, *2*(1), 1–69. Retrieved from https://doi.org/10.1561/2500000014 doi: 10.1561/2500000014

Smith, F., Walker, D., & Morrisett, J. G. (2000). Alias types. In G. Smolka (Ed.), *Programming languages and systems, 9th european symposium on programming, ESOP 2000, held as part of the european joint conferences on the theory and practice of software, ETAPS 2000, berlin, germany, march 25 - april 2, 2000, proceedings* (Vol. 1782, pp. 366–381). Berlin: Springer. Retrieved from https://doi.org/10.1007/3-540-46425-5_24 doi: 10.1007/3-540-46425-5_24

Stepanov, E., & Serebryany, K. (2015). Memorysanitizer: fast detector of uninitialized memory use in C++. In K. Olukotun,

A. Smith, R. Hundt, & J. Mars (Eds.), *Proceedings of the 13th annual IEEE/ACM international symposium on code generation and optimization, CGO 2015, san francisco, ca, usa, february 07 - 11, 2015* (pp. 46–55). IEEE Computer Society. Retrieved from https://doi.org/10.1109/CGO.2015.7054186 doi: 10.1109/CGO.2015.7054186

Szekeres, L., Payer, M., Wei, T., & Sekar, R. (2014). Eternal war in memory. *IEEE Security and Privacy*, *12*(3), 45–53. Retrieved from https://doi.org/10.1109/MSP.2014.44 doi: 10.1109/MSP.2014.44

Thakur, M. (2020). How (not) to write java pointer analyses after 2020. In *Proceedings of the 2020 acm sigplan international symposium on new ideas, new paradigms, and reflections on programming and software* (pp. 134–145). New York, NY, USA: ACM. Retrieved from https://dl.acm.org/doi/10.1145/3426428.3426923 doi: 10.1145/3426428.3426923

Tobin-Hochstadt, S., & Felleisen, M. (2008). The design and implementation of typed scheme. In G. C. Necula & P. Wadler (Eds.), *Proceedings of the 35th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2008, san francisco, california, usa, january 7-12, 2008* (pp. 395–406). ACM. Retrieved from https://doi.org/10.1145/1328438.1328486 doi: 10.1145/1328438.1328486

Tofte, M., Birkedal, L., Elsman, M., & Hallenberg, N. (2004). A retrospective on region-based memory management. *High. Order Symb. Comput.*, *17*(3), 245–265. Retrieved from https://doi.org/10.1023/B:LISP.0000029446.78563.a4 doi: 10.1023/B:LISP.0000029446.78563.a4

Tov, J. A., & Pucella, R. (2011). Practical affine types. In T. Ball & M. Sagiv (Eds.), *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2011, austin, tx, usa, january 26-28, 2011* (pp. 447–458). ACM. Retrieved from https://doi.org/10.1145/1926385.1926436 doi: 10.1145/1926385.1926436

Visser, W., & Mehlitz, P. C. (2005). Model checking programs with java pathfinder. In P. Godefroid (Ed.), *Model checking software, 12th international SPIN workshop, san francisco, ca, usa, august 22-24, 2005, proceedings* (Vol. 3639, p. 27). Berlin: Springer. Retrieved from https://doi.org/10.1007/11537328_5 doi: 10.1007/11537328_5

Wadler, P. (1990). Linear types can change the world! In M. Broy (Ed.), *Programming concepts and methods: Proceedings of the IFIP working group 2.2, 2.3 working conference on programming concepts and methods, sea of galilee, israel, 2-5 april, 1990* (p. 561). North-Holland.

Walker, D., & Morrisett, J. G. (2000). Alias types for recursive data structures. In R. Harper (Ed.), *Types in compilation, third international workshop, TIC 2000, montreal, canada, september 21, 2000, revised selected papers* (Vol. 2071, pp. 177–206). Springer. Retrieved from https://doi.org/10.1007/3-540-45332-6_7 doi: 10.1007/3-540-45332-6_7

Watson, R. N., Moore, S. W., Sewell, P., & Neumann, P. G. (2019). *An introduction to cheri* (Tech. Rep.). University of Cambridge, Computer Laboratory.

Weise, D., Crew, R. F., Ernst, M. D., & Steensgaard, B. (1994). Value dependence graphs: Representation without taxation. In H. Boehm, B. Lang, & D. M. Yellin (Eds.), *Conference record of popl'94: 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages, portland, oregon, usa, january 17-21, 1994* (pp. 297–310). ACM Press. Retrieved from https://doi.org/10.1145/174675.177907 doi: 10.1145/174675.177907

Wright, A. K., & Felleisen, M. (1994). A syntactic approach to type soundness. *Information and Computation*, *115*(1), 38–94. Retrieved from https://doi.org/10.1006/inco.1994.1093 doi: 10.1006/inco.1994.1093

Xi, H., & Pfenning, F. (1998). Eliminating array bound checking through dependent types. In J. W. Davidson, K. D. Cooper, & A. M. Berman (Eds.), *Proceedings of the ACM SIGPLAN '98 conference on programming language design and implementation (pldi), montreal, canada, june 17-19, 1998* (pp. 249–257). ACM. Retrieved from https://doi.org/10.1145/277650.277732 doi: 10.1145/277650.277732

About the authors

Dimitri Racordon is a post-doctoral researcher at the University of Geneva in Switzerland, and Northeastern University in the United States. His current research focuses on type-based approaches to memory safety, as well as language designs for safe and efficient concurrency. You can contact him at dimitri.racordon@unige.ch or visit https://kyouko-taiga.github.io.

Aurélien Coet is a PhD student at the University of Geneva, Switerland, in the *Semantics, Modelling and Verification* group, under the supervision of Professor Didier Buchs. His research focuses on the combination of static and dynamic approaches to enforce memory safety in programming languages. You can contact him at aurelien.coet@unige.ch.

Didier Buchs is a professor at University of Geneva, Switzerland, and the head of the *Semantics, Modelling and Verification* group. His research focuses primarily on formal specifications and validation techniques for complex, distributed systems. You can contact him at didier.buchs@unige.ch.