Reinvestigating Typed Assembly Language for the Garbage Collection Interface

Hayley Patton

A thesis submitted for the degree of
Bachelor of Computing (Honours)
The Australian National University

October 2024

Except where otherwise indicated, this thesis is my own original work.

Hayley Patton
25 October 2024

To my dearest friend Jaidyn.

Acknowledgments

I must thank immensely, and apologise to, Steve Blackburn for his supervision of this curve-ball of an honours project. Peter Höfner, Fabian Muehlboeck and Michael Norrish were equally helpful with the formal methods and type theory aspects of this project. Somehow the lab managed to make having meetings every Monday morning into the best part of the week. In these meetings Kunal Sareen guided me through the Android Runtime, and Zixian Cai, Tianle Qiu and Wenyu Zhao explained their experiences with OpenJDK (and its bug tracker).

The Simple project by Cliff Click and his merry-people of the Coffee Compiler Club was an instrumental guide for writing my sea-of-nodes-based compiler. Matt Dziubinski's reading lists and his recall thereof consistently beat the usual search engines when looking for references. The scope of this thesis was shaped early on by my conversations with Ben Titzer.

I also thank Gilbert Baumann, Gilad Bracha, James Harland, Robert Strandh and the many others who encouraged me to continue with my education by taking an honours year. Gnuxie thought this project sounded excellent and that everyone should already be using typed assembly languages; contrariwise, Eliza Scragg thought this would be ineffective and wouldn't scale with compiler optimisations. I finally thank them both for their encouragement.

Abstract

The garbage collectors in high-performance runtimes can be varied and flexible in order to achieve high performance for both the mutator and the collector. Different collectors and runtimes can have different methods for allocation, and requirements on initialisation and when the mutator must run write barriers. Such variation and flexibility however leads to complex interfaces and invariants which the mutator must uphold, and thus can misuse – indeed some bugs in commonly-used language runtimes can be attributed to the compiler generating code which misuses the interface with the garbage collector. Miscompilation can result in a vector for security exploits when compiling untrusted code, as is done in web browsers. A typed assembly language can protect against miscompilation from exposing such an exploit vector by validating that instructions are applied to the correct types, and thus ensuring memory and type safety. But previous work has paid little attention to the garbage collection interface.

My thesis statement is that *typed assembly language should be adapted to and used in modern runtimes, in order to prevent memory safety bugs caused by miscompilation from becoming security vulnerabilities.*

To this aim, this thesis introduces the Pulstar typed assembly language for AArch64, which is able to validate that the mutator upholds the invariants of the garbage collector, and thus ensuring that the mutator will correctly use the garbage collection interface.

Pulstar can validate the assembly emitted by an optimising compiler, when the compiler performs allocation folding, the compiler optimises the initialisation of objects and the compiler eliminates write barriers. The type checker introduces an geometric mean of 13% time overhead on compilation, and the time spent type checking is quadratic with the program size in practice.

Such a typed assembly language could be used in production, protecting users from compiler bugs which introduce memory unsafety, without drastically slowing down user applications. A typed assembly language could also be used by runtime implementors to provide feedback which is more precise than alternative approaches, when they are debugging how their compilers interact with the garbage collector. A typed assembly language furthermore could be useful for validating the safety of code sent over a network or cached on disk.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction **1**
1.1 Project Statement . . . 1
1.2 Contribution . . . 2
1.3 Thesis Outline . . . 3

2 Background and Related Work **5**
2.1 The Garbage Collector Interface . . . 5
2.1.1 Eliminating Zero-Initialisation . . . 5
2.1.2 Write Barrier Elimination . . . 6
2.1.3 Allocation Folding . . . 7
2.2 Typed Assembly Language . . . 7
2.3 Handling an Untrusted Compiler . . . 9
2.3.1 Sandboxing . . . 9
2.3.2 Translation Validation . . . 10
2.4 Verifying the Garbage Collection Interface . . . 11
2.4.1 IR Verification . . . 11
2.4.2 Heap Verification . . . 12
2.4.3 Heap and Pointer Poisoning . . . 12
2.5 Summary . . . 13

3 Design of a Typed Assembly Language **15**
3.1 AArch64 . . . 15
3.2 Abstract Interpretation . . . 15
3.3 Type System . . . 17
3.3.1 Definitions . . . 18
3.4 Values . . . 20
3.5 Fixnum Operations . . . 21
3.6 Magic Instructions . . . 24
3.7 Initialisation . . . 25
3.8 Records . . . 26
3.9 Stack Frames . . . 27
3.10 Calling Convention . . . 29
3.11 Summary . . . 32

4 Implementation **33**
4.1 Compiler Design 33
4.1.1 Initialisation 34
4.2 Type Inference Engine Implementation 36
4.3 Summary 38

5 Evaluation **41**
5.1 Test Cases 41
5.2 Type Checking Speed 42
5.3 Write Barrier Elimination Algorithms 43
5.4 Summary 46

6 Conclusion **49**
6.1 Future Work 50
6.1.1 Decoupling Representations from the Typed Assembly Language 50
6.1.2 Verifying Type System Soundness 50
6.1.3 Separate Inference and Checking 51
6.1.4 Removing Magic Instructions 51

Bibliography **53**

List of Figures

2.1 The write to `o.f3` is always preceded by some write to `o`, despite not being dominated by either of the prior writes. . . . 12

3.1 The subset of AArch64 instructions typed by Pulstar. . . . 16
3.2 The syntax of typing rules. . . . 17
3.3 The types, union ($\sqcup$), intersection ($\sqcap$), difference ($-$) and subtyping ($\leq$) rules in the type system. . . . 19
3.4 The algorithms for equality and unions of states s_1 and s_2. . . . 22
3.5 Magic instructions in Pulstar. . . . 24
3.6 The stack in AArch64. . . . 30

4.1 Type-based alias analysis in the sea of nodes. . . . 35
4.2 Using a construction fence to prevent the publication of uninitialised objects. . . . 37
4.3 Use of the `define-rule` macro to define RET, STACK-LOAD and BCC-LOCAL. . . . 39

5.1 The type checking time versus instructions and basic blocks per function. 44
5.2 The average visits of basic blocks in each function versus the size of each function. . . . 45
5.3 The store in `set-nesting-level!` (instruction #8) requires a write barrier (instruction #9), but the second store in `root!` (instruction #16) does not. . . . 47

List of Tables

5.1 The benchmarks and support code ported to Utena. 42
5.2 Compile times (in milliseconds) with and without the typed assembly phase. 43
5.3 How many write barriers are eliminated by each algorithm. 46

Chapter 1

Introduction

1.1 Project Statement

> With SPECjbb2015 the assertion fails after a few seconds, which gives a hint that this scary stuff is probably happening for real, and has been broken since forever.
>
> [Österlund, 2020]

Programming languages and their implementations are used to provide software-based isolation not (entirely) provided by the operating system. For example, a web browser might rely on the memory safety of JavaScript to safely run untrusted code downloaded from the Web. The memory safety combined with the capability model in the web browser, where the user must allow a web application to access files and devices, allows for guaranteeing isolation and access control even if the operating system cannot provide such fine-grained control. Unfortunately, bugs in an implementation may expose security vulnerabilities, and attackers can then exploit these vulnerabilities by having the implementation run their code.

A garbage collector is often used to ensure memory safety, as the application cannot use an object after freeing the object if the application cannot directly free an object. There are formally verified garbage collectors which are proven to be correct and will not prematurely free an object [McCreight et al., 2007; Sandberg Ericsson et al., 2017; Gammie et al., 2015], but the correct behaviour of many garbage collectors depends on the mutator cooperating with the collector. Concurrent and generational collectors often require the mutator to execute *write barriers* when updating references in the heap.[1] Many collectors also require the mutator to initialise objects to some extent before garbage collection can occur, if the allocator does not produce memory which is safe to trace. Precise collectors often require the compiler to identify which registers and stack slots contain references in a *stack map* which the compiler must emit as part of compilation.

[1] The CakeML generational collector [Sandberg Ericsson et al., 2017] however does not use a write barrier. All old-to-new references in ML are formed through mutable `ref` cells, and all old cells are scavenged at the start of a nursery collection. Michael Norrish told me that such cells are expected to be rare (though he doesn't know of any relevant measurements).

Some bugs in production runtimes can be attributed to the mutator not correctly following the aforementioned interface. The new Maglev compiler in V8 incorrectly implemented allocation folding, leaving the possibility of a new allocation being uninitialised during a garbage collection and then producing a dangling pointer afterwards [Jimenez and Rao, 2024]. The older Turbofan compiler in V8 incorrectly eliminated write barriers by falsely inferring that some values are small integers [Tiszka, 2021; Google, 2017]. Sometimes the fixes to such bugs are too cautious and can restrict optimisations: for example, the C2 compiler in HotSpot scheduled write barriers to be separated across safepoints from their respective writes [Lozano and Österlund, 2023], and the fix prevented some code motion, although the authors argue little optimisation was possible to begin with.

A *typed assembly language* may be used to validate some properties of the output of a compiler, by assigning types to all the values used in an assembly program, and rejecting programs which attempt to execute instructions with values of the wrong types. A typed assembly language provides type safety by design, and a typed assembly language can further provide memory safety by restricting how the application can manipulate references. For example, the program cannot possibly perform a *use-after-free* if programs can only allocate objects from a garbage-collected heap, and cannot explicitly deallocate objects somehow. Using a typed assembly language to ensure safety can substantially reduce the *trusted computing base* of a runtime which was relying on the compiler to produce safe code (under the very likely assumption that the typed assembly language is simpler than the compiler).

Existing typed assembly languages however do not consider the garbage collection interface, or make concessions which prevent the languages from being usable by high-performance runtimes. For example, they do not ensure that the garbage collector can only observe all fields being initialised, preventing the use of precise garbage collection without expensive bulk-zeroing of memory. They do not track the correct use of write barriers, preventing the use of inlined code for write barriers and preventing the compiler from eliminating redundant write barriers.

1.2 Contribution

In this thesis I will be considering two uses of a typed assembly language extended to validate the garbage collection interface. Typed assembly can ensure that compiled code is always memory- and type-safe, preventing compiler bugs which produce memory-unsafe code from becoming exploits in production. Typed assembly can be used to declare the invariants that a garbage collector needs in a precise and machine-checkable manner, which further can provide precise feedback to compiler authors on when compiled code might violate these invariants.

To these aims I have designed and implemented a typed assembly language *Pulstar* for the AArch64 instruction set, which can validate that the mutator correctly uses the garbage collection interface. In particular:

1. Pulstar tracks which fields of objects are initialised, to ensure that all reachable

objects are fully initialised when the garbage collector may run.

2. Pulstar tracks when objects are known to be adjacent to each other, to ensure that the mutator can only extract objects from allocation folding (under the assumption that they are adjacent) before the garbage collector may move and separate the objects.

3. Pulstar tracks which objects may have been updated to reference young objects, to ensure that the mutator always performs write barriers to inform the garbage collector of these updates.

Pulstar only requires type annotations for the arguments, return values and constant values of each function, and is able to infer the types of all intermediate values, so that a compiler may adopt a typed assembly language without needing to preserve its knowledge of types to the assembly level. Pulstar also supports down-casting, as needed by object-oriented and dynamically-typed programming languages. Pulstar furthermore tracks aliasing between registers and stack slots, so that the aforementioned information is not lost when register allocation causes values to move between registers and the stack.

I implemented an optimising compiler for a simple dynamic object-oriented language in order to test the performance of type checking in a realistic scenario. The compiler exercises the typed assembly language by performing allocation folding and write barrier elimination, and by explicitly initialising objects and eliminating redundant writes. The compiler also performs some of the optimisations present in production runtimes, such as inlining, specialisation, loop-invariant code motion and type-based alias analysis.

1.3 Thesis Outline

Chapter 2 introduces the details of the garbage collection interface, the concept of a typed assembly language, as well as other debugging tools and defenses against buggy compilers. Chapter 3 describes my typed assembly language, and how it validates correct use of the garbage collection interface. Chapter 4 describes the compiler used to generate assembly code to validate and how I have implemented the type checker. Chapter 5 presents the corpus used to test the typed assembly language and the performance of the type checker. Chapter 6 concludes the thesis and discusses future work which could improve the reusability of typed assembly languages.

Chapter 2

Background and Related Work

In this chapter I introduce the garbage collection interface, alongside existing typed assembly languages and how they interact with the garbage collection interface. Then I introduce other approaches to handling an untrustworthy compiler, and to validating correct usage of the garbage collection interface.

2.1 The Garbage Collector Interface

A tracing garbage collector functions by identifying all *reachable* objects and then reclaiming all unreachable objects. An object is reachable when a traversal (*trace*) from the *roots* reaches that object; the roots consist of all local and global variables used by the user program (which is called the *mutator*).

The mutator must thus supply the garbage collector with the roots; in particular, the collector must know which registers and stack slots (herein *locations*) hold references to objects. The *precise* approach is that the compiler generates *stack maps* which map values of the program counter to sets of which locations have references. The *conservative* approach entails that the collector assumes all locations might have references, and must identify whether each value in each location contains a valid reference. The collector identifies whether a reference is valid either by traversing the internal structure of the heap [Boehm and Weiser, 1988; Bartlett, 1988], or by reading from a bitmap of which addresses have the starts of objects which the mutator and collector must maintain [Shahriyar et al., 2014; Patton, 2023]. This thesis will focus on the invariants required by precise garbage collection; although conservative garbage collection has the invariant that the mutator cannot obfuscate references, such as by encoding two references in one field in a xor-linked list.

2.1.1 Eliminating Zero-Initialisation

The mutator must also ensure that, for each object which is reachable, each field of the object either has a non-reference type (either by a static type or by using tagged values), has a valid reference to an object, or has some other value which the collector knows not to trace (such as a null pointer). One simple approach to ensuring that this invariant is held is to arrange that the allocator always provides memory which

is filled with zeroes, when a null pointer or some immediate value is encoded as a zero. This approach is convenient for languages such as Java, which have null pointers and zero-initialise all fields of an object before the constructor for that object runs. This approach is not convenient for languages such as Common Lisp [X3J13, 1994, §7.13] and Smalltalk [Goldberg and Robson, 1983] which pre-initialise fields to a value other than zero, nor is it convenient for languages such as ML which never allow uninitialised fields to be observed by the programmer.

Zero initialisation however takes time and memory bandwidth [Yang et al., 2011], regardless of whether filling memory with zeroes or some other value corresponds to any part of initialisation in the programming language. Yang et al. describe zero initialisation as being either performed in *bulk* before the allocator gives memory to the mutator, or performed on the *hot path* when the mutator has just allocated an object. Hot-path zero initialisation is amenable to the *dead store elimination* optimisation, where all but the last of subsequent stores to a field can be removed, so long as the field can never be read in between stores. The compiler must however count running the garbage collector as reading all fields of objects when eliminating initialising stores in particular, as otherwise the optimisation would cause the collector to read uninitialised fields.

2.1.2 Write Barrier Elimination

The collector having to trace all live objects in every garbage collection yields poor performance, both in terms of latency and throughput. A *stop-the-world* collector must pause the mutator from running while the collector traces the heap. But a *concurrent* garbage collector [Dijkstra et al., 1978] allows the mutator to continue running while the collector traces the heap, with the requirement that the mutator notify the collector which references the mutator has updated in the heap.

The collector has to trace each object in each collection that the object survives (regardless of whether the collector is concurrent or not) which wastes time re-tracing objects which survive multiple collections. Fortunately, many programs exhibit a bimodal distribution of the lifetimes of objects (the *weak generational hypothesis*), so a *generational* garbage collector [Lieberman and Hewitt, 1983] can focus its efforts on collecting just the younger objects which are more likely to die than the older objects. A generational garbage collector partitions the heap into a young generation and an old generation, and allocates objects into the young generation. The collector then can trace the young generation without tracing the old generation. The collector infrequently promotes surviving objects into the old generation, and infrequently collects objects in both generations when the old generation grows too much in size. In order for the garbage collector to correctly trace the young generation without tracing the old generation, the collector must be able to enumerate all references from objects in the old generation to objects in the young generation, for which the mutator must again notify the collector of which references the mutator has updated in the old generation.

A compiler may be able to tell that a write barrier is not necessary for either a

concurrent or generational collector. In the case of a generational collector, there is no need to emit a write barrier for a write to an object which is always in the new generation. Most concurrent collectors do not need to be notified that an object was modified twice. For example, the Android runtime uses write barrier elimination for a card-marking write barrier. The compiler only emits one write barrier for multiple writes to the same referencee when the writes are not separated by control flow (i.e. they are in the same *basic block*).

Many runtimes using *tagged pointers*, and runtimes implementing languages implementing nullable references can eliminate the write barrier when storing an *immediate* value and null respectively. Such writes cannot produce old-to-new references as an immediate value is not in the heap and has no generation.

Two cases of incorrect write barrier elimination in Turbofan [Tiszka, 2021; Google, 2017] were caused by an optimisation accidentally widening the type of a value from being a small integer, which is not allocated in the heap, to "any number" which might be allocated in the heap. An optimisation should never cause types to widen, as doing so would invalidate the assumptions made by previous optimisations; in this case, a prior optimisation eliminated the write barrier under the assumption that a value was a small integer. Such interactions between optimisations can cause compiler bugs, which are not directly related to the garbage collection interface, to cause the mutator to violate its invariants nonetheless. A typed assembly language tracks types of values in a program, and thus would be suited for validating such a rule for write barrier elimination.

2.1.3 Allocation Folding

Allocation folding [Clifford et al., 2014] coalesces separate allocations for separate objects into one allocation, to speed up allocation. Allocation folding is used by V8, GraalVM[1] and Erlang [Ericsson AB, 2024]. V8 additionally uses allocation folding to help its compilers identify objects which are known to be young, in order to eliminate write barriers. Each allocation may trigger a garbage collection, and the garbage collector may choose to promote any young objects to the old generation, so a compiler cannot determine what the generation of an object will be after another object is allocated. Thus the compiler folding multiple allocations into fewer allocations allows the compiler to determine that more objects will be young, and thus the compiler can eliminate more write barriers.

2.2 Typed Assembly Language

Morrisett et al. [1999] introduced the concept of a *typed assembly language* and a compiler from polymorphically-typed lambda calculus (*System F*) to a typed assembly language. They propose using typed assembly language to guarantee safety without

[1]Allocation folding is demonstrated in GraalVM in https://ionutbalosin.com/2024/02/jvm-performance-comparison-for-jdk-21/#analysis-of-wrapper_obj_baseline; I haven't found it mentioned in the GraalVM documentation though.

relying on a compiler (or a programmer directly writing assembly) to preserve safety. Their type system tracks initialisation as they treat allocation and initialisation as separate instructions, and using an uninitialised object where an initialised object is expected would violate type safety. But they do not use type information to verify the garbage collector interface: they use the conservative Boehm-Demers-Weiser collector, which ignores uninitialised fields in objects out of necessity; the collector cannot identify which fields are not pointers anyway. The collector also uses hardware memory protection, as the collector cannot instruct the compiler to emit write barriers. However, they note that their typed assembly language prevents the mutator from obfuscating references in ways which would cause a conservative collector to not recognise a reference to an object as a reference, and thus the mutator could not produce a dangling pointer after deobfuscating the pointer.

Their type system also relies on the program not having any loops in the control flow in a function; they are able to compile interesting programs despite this requirement by compiling to *continuation-passing style* code wherein control flow is only performed by tail-calling between functions, and the call stack is replaced with a caller calling its callee with a *continuation* function for the callee to call with its "return" value [Steele, 1976]. Compiling to continuation-passing style side-steps the requirement, as the typed assembly language can verify arbitrary calls between between functions, but few compilers generate code in continuation-passing style. Later work by the authors allowed for arbitrary control flow in a function and the use of a traditional call stack in a *stack-based typed assembly language* [Morrisett et al., 2002], but they require basic blocks to be annotated with the types at the start of each basic block.[2] This requirement would require compilers to *preserve* type information up to and through emitting assembly, which may require substantial modifications to the compiler.

Chang et al. [2005] introduces the *Coolaid* verifier, which uses typed assembly language as a debugging aid for students in a compiler construction course. They introduce some concepts necessary for verifying the output of a compiler which compiles a Java-like language, such as nullable types and existential types for method dispatch; but they do not address the garbage collection interface. They use *abstract interpretation* to eliminate the annotations for each basic block, and track *values* in registers and stack slots to propagate types between locations which are known to refer to the same objects. Students who took the course when Coolaid was introduced wrote compilers which were more likely to pass the unit tests used to assess student work, and most students found the verifier useful. Chang et al. note that Coolaid "not only can expose compilation bugs that simple execution with [an emulator] might not cover, but can also pinpoint the offending instruction, as opposed to simply producing the wrong [...] output." This precision is unique to typed assembly language and translation validation (to be introduced in Subsection 2.3.2), and the typed assembly language can also statically detect type errors even when they are not encountered at

[2]This requirement is equivalent to the requirements of the original typed assembly language, that function types must be annotated and that functions are compiled in continuation-passing style, as jumping to a basic block is equivalent to calling a continuation function [Steele, 1976].

run time.

Tate et al. [2010] use a typed assembly language *iTalX* in the Bartok compiler for C# (as used in the Singularity operating system), which introduces additional constructs for type-checking the soundness of array bounds check elimination and the correct usage of stack allocation. They however outline allocation and write barrier code, as their type system cannot verify either, leading to 2.3% mutator overhead [Chen et al., 2008]; which is however less overhead than the 4.8% overhead found by Blackburn and McKinley [2002] for using outlined barriers instead of partially inlined barriers.

2.3 Handling an Untrusted Compiler

We may have to deal with a compiler which we do not trust to be correct, for which some production runtimes have implemented mitigations against miscompilation. Two approaches have been deployed for this problem: running untrusted code in a sandbox limits the damage that the code can inflict, and translation validation can establish trust that the compiler correctly compiled a specific program.

2.3.1 Sandboxing

V8 uses a *sandbox* to contain all JavaScript objects, to contain the effects of buggy compiler output and of the C++ runtime. A memory safe language with encapsulation theoretically should suffice to implement access control [Rees, 1996], so additional sandboxing should not be necessary. But experience with bugs in V8, which invalidate the memory safety guarantees of the language, have lead the developers to conclude that "memory safety cannot be guaranteed by the compiler if a compiler is directly part of the attack surface" [Groß, 2024] as is the case when miscompiling untrusted code to a memory-unsafe language (machine code).

Sandboxing is essentially the same isolation that a kernel provides between processes, but implemented in a user-space program; such sandboxing could be rather expensive if it was implemented like process isolation. Aiken et al. [2006] observe a 25–33% time overhead when using the conventional approach using the *memory management unit* to implement process isolation, but observe less than 5% overhead when using software-based isolation. The sandboxing technique in V8 however only requires a shift and an add instruction before loading a reference to an object internal to the heap, and an additional indirection for references external to the heap, producing "around 1% or less" time overhead.

Validating the output of the JIT compiler however could not help with the memory-unsafe runtime code written in C++, whereas the sandboxing contains the possible effects of buggy C++ and JavaScript code to the JavaScript heap. Validation also would not mitigate against side-channel attacks, which effectively bypass type and bounds checks; V8 instead implements some additional bespoke mitigations [McIlroy et al., 2019].

2.3.2 Translation Validation

Another approach to handling untrusted passes of a compiler which only introduces compile-time overhead is to employ *translation validation* [Pnueli et al., 1998] and prove that all possible semantics of the output of the pass are equivalent to possible semantics[3] of the input of the pass.

Rideau and Leroy [2010] present a translation validator for the register allocator of the formally verified CompCert compiler, for which most passes are formally verified using the Coq theorem prover. Instead Rideau and Leroy wrote the register allocator as unverified OCaml code, wrote a formally verified translation validator, and require that the output of the register allocator passes the validator in order for compilation to succeed. The validator and its correctness proof are an order of magnitude smaller than a correctness proof of the register allocator itself. They note that "the compile-time overhead of the validator is very reasonable: validation adds 20% to the time taken by register allocation and 6% to the whole compilation time." The validator is also decoupled from the register allocator, in that the register allocator can be modified freely without having to modify the correctness proof of the validator.

The validator performs a data-flow analysis to establish an equivalence between input code in an intermediate language close to assembly language, but with infinitely many *virtual* registers (*register transfer language*), and output code in another language which uses the registers of the target instruction set and stack slots (*location transfer language*). The analysis however requires the register allocator to leave in no-operation instructions in place of move instructions if the register allocator *coalesces* two virtual registers into one physical register. This requirement guarantees that the position of each instruction is the same between input and output, and so the analysis only has to compare instructions in lockstep to find the equivalence. The Cranelift compiler backend has a similar validator for register allocation; Fallin [2021] states that this validator is simpler than the validator of Rideau and Leroy, by having the register allocator explicitly mark which instructions move virtual registers between physical registers and the stack (*spills* and *reloads*).

A translation validator can be more thorough and validate more of the compiler. At the limit, validating the whole compiler from source code to assembly would provide a proof that the generated assembly is memory-safe. Alive2 [Lopes et al., 2021] and TurboTV [Kwon et al., 2024] validate transformations on the intermediate representations of LLVM and Turbofan (in V8) respectively. Both validators function by translating the intermediate representations into functions for a *satisfiability modulo theories* (SMT) solver, and have the solver validate that all behaviours of the output program correspond to behaviours of the input program.

TurboTV however cannot validate functions with loops. Alive2 can validate functions with loops, but is intentionally unsound by only considering the execution of a bounded number of loop iterations (by *unrolling* the body of the loop up to that

[3]This requirement is not the same as (and is instead more relaxed than) for the input and output of a pass to have equivalent behaviour: the output of a pass may have less undefined behaviour or less non-determinism than the input to that pass. For example, coalescing atomic operations in a multi-threaded program will eliminate some of the non-determinism [Bastien, 2015].

many times); an attacker could fool Alive2 with a program which only executes miscompiled code after more loop iterations than the bound. This unsoundness however limits the resource usage of Alive2, as the size of the SMT problem to solve scales with the limit on unrolling. TurboTV also required on average 0.71 seconds to validate a randomly-generated JavaScript function, which is too slow to deploy in production. The compilation speed is crucial for just-in-time compilation, although such overhead would also be excessive for an ahead-of-time compiler given the constraints on control-flow.

Translation validation is nonetheless very successful as a compiler development tool. Contributors to the InstCombine pass of LLVM must provide proofs using Alive2 that their transforms are correct[4] and 1,275 pull requests on GitHub[5] mention Alive2 at the time of writing. TurboTV allows a *fuzzer* to find more bugs in a fuzzing setup despite the time overhead of TurboTV, as TurboTV considers all possible arguments of the function rather than selecting one set of arguments.

2.4 Verifying the Garbage Collection Interface

2.4.1 IR Verification

The compiler of the Android runtime has an IR checker which, among other properties, asserts that all write barriers which were eliminated are preceded in the same basic block by some write barrier which was not eliminated. (The compiler does not represent write barriers as separate instructions in its intermediate representation. Instead each write instruction is annotated to indicate whether a barrier should be emitted.)

Erik Österlund implemented a similar check while debugging the implementation of weak references in OpenJDK [Österlund, 2020]. He associates loads of the values of weak references and their write barriers, and then checks that no safepoints appear in the dominator basic blocks between the write barrier and load.

Österlund's dominator check would allow for some degree of code motion, and dominators may be used to verify that write barriers were correctly eliminated too. But we may face code where a write barrier is not dominated by another barrier, but still always is preceded by some barrier, as in Figure 2.1. A compiler perhaps should hoist and merge the write barriers out of the conditional blocks to reduce code size, which would make dominance sufficient to check correctness, but this optimisation should not be necessary to check correctness.

[4] https://llvm.org/docs/InstCombineContributorGuide.html#proofs

[5] This count includes pull requests which don't mention InstCombine; Alive2 is also used e.g. in pull requests involving the SLP vectoriser and ValueTracking: https://github.com/llvm/llvm-project/issues?q=alive2

```
1 if (p) {
2     o.f1 = /* barrier */ v;
3 } else {
4     o.f2 = /* barrier */ v;
5 }
6 o.f3 = /* no barrier */ v;
```

Figure 2.1: The write to `o.f3` is always preceded by some write to `o`, despite not being dominated by either of the prior writes.

2.4.2 Heap Verification

Steel Bank Common Lisp has a *heap verifier*[6] which checks that invariants hold on the heap, such as that old-to-new references always reside on dirty cards. The verifier can be configured to run before garbage collections to validate the mutator behaviour, or after garbage collections to validate the collector behaviour. But the verifier introduces a huge slowdown on garbage collections, slowing programs by up to an order of magnitude, making it unsuitable for use in production. The verifier is inherently imprecise as a debugging tool, as it only runs just before and after a garbage collection, and the mutator may break an invariant long before collection. The verifier can also only react after the mutator breaks an invariant, which may seldom occur for code which is infrequently executed, or when a bug involves some very specific interleaving of events in concurrent code.

2.4.3 Heap and Pointer Poisoning

One issue not addressed by IR verification or heap verification is whether the checks are insufficient and still allow for memory safety bugs to occur. *Heap poisoning* and *pointer poisoning* both attempt to force the mutator to crash when the mutator attempts to use an invalid reference. The crash would likely occur sooner than heap verification would detect an inconsistency in the heap, but poisoning also can still only react some time after the mutator breaks an invariant.

Heap poisoning fills unused memory with a value which corresponds to an invalid reference, so that loading and using a reference loaded from unused memory will cause a segmentation fault. The OpenJDK debug setting ZapUnusedHeapArea configures the garbage collector to fill unused memory with the word `0xBAADBABE`[7]. Filling unused memory is sufficiently slow however that it was disabled by default in debug builds of OpenJDK [Saha, 2009]. Pointer poisoning requires the mutator to deobfuscate a pointer before use. The Android runtime negates the values of compressed references[8] so that a compressed reference cannot be used without decompressing the reference beforehand and so that the mutator must perform the use barrier before using a reference.

[6] https://github.com/sbcl/sbcl/blob/sbcl-2.4.3/src/runtime/gencgc.c#L4824

[7] https://github.com/openjdk/jdk/blob/jdk-23%2B17/src/hotspot/share/gc/shared/spaceDecorator.cpp#L85

[8] ART uses the name heap poisoning for their pointer poisoning. https://android.googlesource.com/platform/art/+/fb90365d945e73ac5bed6249c2dbcf6e98031efd/runtime/mirror/object_reference.h#101

2.5 Summary

An implementation of a memory-safe language using a garbage collector requires compiled code to correctly follow the interface to the garbage collector. This interface broadly consists of how to allocate and initialise memory, and what the mutator needs to communicate to the collector; a mutator which does not uphold the invariants of this interface can introduce memory unsafety.

Typed assembly languages and sandboxing remove a compiler from the trusted computing base, and make a runtime resistant to memory safety bugs in its compiler. Typed assembly languages can also be used as a debugging tool, to detect precisely where the compiler generates ill-typed code. But existing typed assembly languages currently do not validate much of the garbage collection interface; those that do are limited to using conservative garbage collection, or pessimise code by require the mutator to call trusted functions in order to allocate objects and to update fields with references.

Heap verification and heap and pointer poisoning can detect misuse of the garbage collector interface at run-time, but these introduce significant time overheads, precluding their use in production and slowing down the process of debugging. Heap verification is also inherently imprecise, making it difficult to debug precisely how and when the mutator broke its invariants. In contrast, the typed assembly language I will present runs at compile-time and can identify exactly which instructions will cause the mutator to break its invariants.

Chapter 3

Design of a Typed Assembly Language

This chapter presents a typed assembly language for AArch64, with extensions to validate use of the garbage collection interface. Section 3.1 introduces the subset of AArch64 used, Sections 3.2 through 3.4 introduce the structure of the type system and type checker, and Section 3.5 onward contain the individual type rules.

3.1 AArch64

The typed assembly language covers the subset of the AArch64 instruction set presented in Figure 3.1. The state of an AArch64 machine is contained in thirty-two 64-bit general-purpose registers named `x0` through `x31`, a program counter *PC*, a flags register which stores the results of a comparison, and random access memory. The highest four general-purpose registers have special purposes and names: `x28` references the thread-local state (`tls`), `x30` is the link register (`lr`) which provides a callee with the address to return after a function call, `x29` stores the frame pointer (`fp`) and `x31` stores the stack pointer (`sp`) which together bound the current stack frame. The only requirement on the contents of registers is that the stack pointer must always be aligned to 16 bytes.

3.2 Abstract Interpretation

A type system for assembly language differs from type systems for high-level languages in ways which require an unconventional approach to type-checking. A register or stack slot may be allocated to different variables of different types, so the type system must be able to assign different types to the same location at different times (known as *flow-sensitivity* [Hunt and Sands, 2006]). The assembly is structured as an unstructured control-flow graph rather than a syntax tree, so the usual approach of recursively traversing a syntax tree is at least inconvenient, and at most will not

Instructions	Arguments	Behaviour
(register) mov	d, s	$\text{Reg}_d \leftarrow \text{Reg}_s$
(immediate) mov	d, #constant	$\text{Reg}_d \leftarrow \mathit{constant}$
add, sub, mul, div, and, orr, eor, asr, lsl	d, s, t	$\text{Reg}_d \leftarrow \text{op}(\text{Reg}_s, \text{Reg}_t)$
b	label	$PC \leftarrow \mathit{label}$
bl	label	$\text{Reg}_{30} \leftarrow PC + 4;\ PC \leftarrow \mathit{label}$
ret		$PC \leftarrow \text{Reg}_{30}$
cmp	s, t	set flags by comparing Reg_s and Reg_t
blt, ble, beq, bne, bge, bgt	label	if flags match condition, $PC \leftarrow \mathit{label}$
tbz	s, #i, label	if ith bit of Reg_s is zero, $PC \leftarrow \mathit{label}$
tbnz	s, #i, label	if ith bit of Reg_s is one, $PC \leftarrow \mathit{label}$
		With $a = \text{Reg}_s + o$:
ldr	d, [s, #o]	$\text{Reg}_d \leftarrow \text{Memory}_a$
str	v, [s, #o]	$\text{Memory}_a \leftarrow \text{Reg}_v$
ldp	d, e, [s, #o]	$\text{Reg}_d \leftarrow \text{Memory}_a$;
		$\text{Reg}_e \leftarrow \text{Memory}_{a+8}$
		(but undefined when $d = e$)
stp	v, u, [s, #o]	$\text{Memory}_a \leftarrow \text{Reg}_v$;
		$\text{Memory}_{a+8} \leftarrow \text{Reg}_u$

Instruction	Operation	Instruction	Operation
add	addition	sub	subtraction
mul	multiplication	div	division
and	bitwise and	orr	bitwise or
eor	bitwise exclusive or		
asr	bitwise shift right	lsl	bitwise shift left
blt	branch if lesser than	bgt	branch if greater than
ble	branch if lesser or equal	bge	branch if greater or equal
beq	branch if equal	bne	branch if not equal

Figure 3.1: The subset of AArch64 instructions typed by Pulstar.

$$
\begin{array}{rcl}
\mathit{rule} & ::= & \mathit{state}\ \mathit{instruction}\ \langle \mathit{state}^{\mathit{successor}} \rangle^{*} \\
& | & \dfrac{\mathit{condition}; \ldots ; \mathit{condition}}{\mathit{state}\ \mathit{instruction}\ \langle \mathit{state}^{\mathit{successor}} \rangle^{*}} \\
\mathit{state} & ::= & \{\mathit{condition}; \ldots ; \mathit{condition}\} \\
\mathit{successor} & ::= & +1 \mid \mathit{label}
\end{array}
$$

Figure 3.2: The syntax of typing rules.

work.[1] Instead I have adapted the use of Hoare logic for ARM assembly by Myreen et al. [2007] in order to specify what each rule requires of the state before an instruction, and what updates to that state are made in order to produce the (possibly zero or multiple) states for the successor instructions of the current instruction. Each state used by the type checker contains an abstraction of the ARM machine state, assigning known types to locations instead of concrete values.

A typing rule is expressed as a relation between an abstract state preceding the execution of the instruction, and the updates to that state required to produce any number of abstract states succeeding the execution of the instruction. Each successor state pertains to a location in control-flow that execution can proceed to, which is either a label or the next instruction (denoted by $+1$).

I have also adopted the use of *abstract interpretation* [Cousot and Cousot, 1977], following its use in Coolaid, to handle arbitrary control flow. Abstract interpretation infers sound types by iterating through the basic blocks, with the predecessor state of a basic block being set to the union of successor states of each of the predecessors of the basic block. Abstract interpretation starts propagating types from the first basic block, with an empty stack frame and the registers assigned the types of the parameters of the function. The iteration continues until no states change. Iterating in this manner is crucial to correctly inferring types for loops, where there is no (topological) order in which all predecessor states can be computed before computing the state for a basic block inside a loop, as some predecessors will depend on the successor state of that basic block.

The states used in this type system consist of mappings of registers and stack slots to the values they contain, mappings of values to their types, and a log. The log consists of a set of the values which need to be logged by the write barrier, and a set of the values which already have been logged by the write barrier.

3.3 Type System

The types used in the type system, presented in Figure 3.3 is similar to that of Morrisett et al. [1999] but with some modifications. There are top and bottom types

[1] Morrisett et al. [1999] however do use a recursive approach on a sequence of instructions, by specifying rules for each instruction which apply when the rest of the sequence is well-typed with a modified state. I personally find this notation hard to read, and this also approach would have to interact with abstract interpretation, which was not used in any of the type systems by Morrisett et al.

used by abstract interpretation: the top type denotes that the type of a location is unknown, and the bottom type denotes that a location cannot possibly have a value.

There are no function types or polymorphic types, as my implementation of method calls turns polymorphic method calls into calls to constant functions which dispatch on the types given to them (for which a similar process for higher-order functions is *defunctionalisation* [Reynolds, 1972]). The compiler may downcast only between *object types* which have their types tagged at run time; only Fixnum and fully initialised record types are object types. The least significant bit (*tag bit*) of a reference distinguishes between small integers (*fixnums*, tagged with a zero bit) and heap-allocated objects (*records*, tagged with a one bit). A record has type information in a *header word* to allow for testing whether a value is of a particular record type at run time. A record type tracks whether each field is initialised, but this information is not made available dynamically; partly initialised records are not first-class objects and not subtypes of Object, and downcasts cannot be performed to partly initialised record types.

The type system also tracks union types as the compiler may rely on its union types being exhaustive: for example, it may dispatch on the type $\mathsf{True} \cup \mathsf{False}$ by testing for the type True and assuming the type must be False if that test fails (by taking the difference $(\mathsf{True} \cup \mathsf{False}) - \mathsf{True}$. The type system tracks initialisation of each field.

3.3.1 Definitions

Almost all rules require knowing that some location or value l is of a type T, which is written as $l : T$. l may have a subtype of T when $l : T$, and need not have the exact type T; the exact type of l in a state is instead $\textsc{Type}(l)$. Updating a state to assign a type to a location entails assigning a new value of that type to that location. Some rules require knowing that some location l will refer to the values v, which is written as $l \rightsquigarrow v$.

Some rules require knowing that some address will be aligned, either to a single word if accessing a record, or to two words if adjusting the stack pointer. For clarity we define the predicates:

$$\textsc{Aligned}(x) = x \bmod 8 = 0$$
$$\textsc{Double-Aligned}(x) = x \bmod 16 = 0$$

Rules which produce values require that the destination register is "ordinary" and can be written by the normal means. The predicate Writable is true of all registers, except for the stack pointer (SP), thread-local state register (TLS) and two registers reserved by the AArch64 calling convention (x16 and x17). Rules which check the size of the current stack frame use the array Stack-Frame of a state.

Rules which involve instructions which might lead to running the garbage collector require that all reachable objects are initialised. The predicate All-Values-Initialised

types	τ	::=	$\top$ *(top)* $\mid \bot$ *(bottom)* $\mid$ Untagged $\mid o$
			$\mid$ LR $\mid$ SP $\mid$ Caller-FP $\mid$ Callee-FP
object types	o	::=	Object $\mid$ Fixnum $\mid$ Heap $\mid (o_1 \cup \ldots \cup o_n) \mid \text{Name}\langle o_1^1, \ldots, o_n^1 \rangle$
records	ρ	::=	$\text{Name}\left\langle o_1^\phi, \ldots, o_n^\phi \right\rangle$
initialisation flags	ϕ	::=	1 *(initialised)* $\mid 0$ *(uninitialised)*

No unions have only one member:	$t \sqcup t = t$
No Bottom in unions:	$t \sqcup \bot = t$
Unions are deduplicated:	$t \sqcup (o_1 \cup \ldots \cup o_n) = (o_1 \cup \ldots \cup o_n)$ when $t \in o$
Unions contain object types only:	$t_1 \sqcup t_2 = (t_1 \cup t_2)$ when $t_1 \leq \text{Object} \wedge t_2 \leq \text{Object}$
Everything else becomes Top:	$t_1 \sqcup t_2 = \top$ when $t_1 \not\leq \text{Object} \vee t_2 \not\leq \text{Object}$
Unions are symmetric:	$t_1 \sqcup t_2 = t_2 \sqcup t_1$

The intersection of the same type is that type:	$t \sqcap t = t$
The intersection of a union is a union of intersections:	$(t_1 \cup t_2) \sqcap s = (t_1 \sqcap s) \sqcup (t_2 \sqcap s)$
The intersection of two types is the subtype:	$t_1 \sqcap t_2 = t$ when $t_1 \leq t_2$
Everything else becomes Bottom:	$t_1 \sqcap t_2 = \bot$ otherwise
Intersections are symmetric:	$t_1 \sqcap t_2 = t_2 \sqcap t_1$

The difference of subtype and supertype is Bottom:	$t_1 - t_2 = \bot$ when $t_1 \leq t_2$
The difference of a union is a union of differences:	$(t_1 \cup t_2) - s = (t_1 - s) \sqcup (t_2 - s)$
...and when the other argument is a union too:	$t - (s_1 \cup s_2) = (t - s_1) \sqcup (t - s_2)$
The difference of a type and everything else is that type:	$t - s = t$ otherwise
(The difference is not symmetric.)	

Every type is a subtype of Top:	$t \leq \top$
Every type is a subtype of itself:	$t \leq t$
Bottom is a subtype of every type:	$\bot \leq t$
Fixnum is a subtype of Object:	$\text{Fixnum} \leq \text{Object}$
Heap is a subtype of Object:	$\text{Heap} \leq \text{Object}$
All fully-initialised record types are subtypes of Heap:	$r\langle f_1^1, \ldots, f_n^1 \rangle \leq \text{Heap}$
Unions are subtypes when all members are subtypes:	$(o_1 \cup \ldots \cup o_n) \leq t$ when $\forall o.\, o \leq t$
Unions are supertypes when some members are supertypes:	$t \leq (o_1 \cup \ldots \cup o_2)$ when $\exists o.\, t \leq o$

Figure 3.3: The types, union ($\sqcup$), intersection ($\sqcap$), difference ($-$) and subtyping ($\leq$) rules in the type system.

is true of a state when all values referenced by the registers and stack slots do not have any record types with uninitialised fields. Such rules also require that all written records have been logged by the write barrier. The predicate All-Logged is true of a log which contains no values which need to be logged. The function Needs-Barrier($log, value$) produces a log with an additional value which needs to be logged, and Log-Object($log, value$) produces a log with an additional value which was logged.

3.4 Values

If we would solely assign types to locations, spilling and unspilling registers could prevent the type system from correctly updating the state with updated types. Consider some object construction code (based on some code generated by my compiler) which spills all its values to the stack, which we attempt to type whilst only considering the types of each location:

```
1   alloc x0, T
2   ; {x0 : T⟨U⁰⟩} by Alloc
3   str x0, [sp, 0]
4   ...
5   ldr x1, [sp, 0]
6   ; {x2 : U; x1 : T⟨U⁰⟩; [sp, 0] : T⟨U⁰⟩} by Stack-Load
7   str x2, [x1, 7]
8   ; {x1 : T⟨U¹⟩; [sp, 0] : T⟨U⁰⟩} by Record-Store
9   ...
10  ldr x3, [sp, 0]
11  ; {x3 : T⟨U⁰⟩; [sp, 0] : T⟨U⁰⟩} by Stack-Load
12  ldr x4, [x3, 7] ; Error! The field of x3 is uninitialised.
```

`x25` is reserved as a temporary register used solely for loading values which have been allocated to a stack slot; it is reloaded before every use of a value in a stack slot (as on line 5), and it is written to the stack after every instruction which updates the value in a stack slot (as on line 3). The store on line 7 initialises the single field of the object, whilst the object is referenced by both `x25` and `[sp, 0]`, but only the type of `x25` is updated when the field is initialised. Since the reference itself is not updated by the store instruction, the reference is not written back to the stack. The subsequent code in lines 10 and 12 use the value in `[sp, 0]` and the type system reports that the field is uninitialised, despite that we did initialise the field.

The solution I use to associate the register and stack slot in this example together, is to indirect the types of locations through *values*, as is done in Coolaid. Thus locations have values and values have types. The separation of types and values also allows for introducing allocation folding without cluttering the definitions of operations on types: allocation folding is represented by the fact that a location may have multiple values of record types, when the records are known to be adjacent in the heap, due to allocation folding having allocated them so. Values are not propagated through the heap; doing so is complex and does not appear to allow for type-checking any more optimisations.

Abstract interpretation requires an equality function on states. If the state only contained types, we could define equality of two states to be that each location has the same type in either state. We must further require that the values are equivalent somehow when we extend states to track values. But requiring that the values of locations are identical between states is too strict, and may cause abstract interpretation to never terminate with loops: if abstract interpretation of a loop body produces states with new values, abstract interpretation will produce new values on every iteration. Instead I define equality of states to be that there is a one-to-one mapping (a *bijection*) of values between the states, and that the values related by the bijection have equal types. The algorithm for finding a bijection is presented in Figure 3.4. The exact same values must be logged between environments however, as otherwise a loop body could cause an unbounded number of new values to not be logged, and abstract interpretation would never terminate.

Abstract interpretation also requires a function to merge (union) all of the states preceding an instruction, in order to produce an approximation which correctly reflects the machine state regardless of whichever predecessor instruction was last executed by the machine. For simplicity I will describe the operation in terms of computing the union of just two input states, for which union of more than two input states is computed by reducing the union function over all input states. I use the algorithm for merging values from Coolaid, where a union of a value and itself is the same value, and otherwise each pair of values having their union taken is assigned a new value. The algorithm assigns types to values being the union of types from the input states.

3.5 Fixnum Operations

The type system allows for operations on fixnums, without having to remove the tags. Recall that fixnums are represented as integers with their tag bit set to zero, so that a fixnum is encoded as its value shifted left one bit, which is equivalent to multiplying the value by two. Addition of fixnums produces a fixnum, because the multiplication factor is preserved in $2x + 2y = 2(x + y)$; subtraction, bitwise-and, bitwise-or and bitwise-xor have the same property.

$\textsc{Writable}(\mathit{result})$

$\{\mathit{arg1} : \mathsf{Fixnum};\ \mathit{arg2} : \mathsf{Fixnum}\}$
$\texttt{add} \mid \texttt{sub} \mid \texttt{and} \mid \texttt{orr} \mid \texttt{eor}\ \mathit{result}, \mathit{arg1}, \mathit{arg2}$ (Fixnum-Tag-Preserving-Op)
$\{\mathit{result} : \mathsf{Fixnum}\}^{+1}$

However, bitwise-not inverts the tag bit from 0 to 1, which would appear to produce a reference to a record. Instead the result is typed as an *untagged* integer, which can then be turned into a fixnum by setting the tag bit to zero, which in turn is achieved by performing bitwise-and with an integer with all but the least significant

```
1  EQUAL(s1, s2) =
2    seen = new set
3    UNIFY(v1, v2) =
4      if (v1, v2) is in seen return true
5      if v1 or v2 appear in different pairs in seen
6        return false from EQUAL
7      if s1.type(v1) ≠ s2.type(v2)
8        return false from EQUAL
9      else add (v1, v2) to seen
10
11   UNIFYALL(l1, l2) =
12     if LENGTH(l1) ≠ LENGTH(l2) return false
13     for v1, v2 being pairwise elements of l1 and l2
14       UNIFY(v1, v2)
15
16   if the log of e1 ≠ the log of e2 error
17   for r1, r2 being pairwise values in registers of s1 and s2
18     UNIFYALL(r1, r2)
19
20   if LENGTH(the stack frame of e1) ≠ LENGTH(the stack frame of e2)
21     return false
22   for t1, t2 being pairwise values in stack slots of s1 and s2
23     UNIFYALL(t1, t2)
24
25   return true
```

```
1  UNION(s1, s2) =
2    values = new map
3    result = new state
4    UNIONVALUE(v1, v2) =
5      type = s1.type(v1) ⊔ s2.type(v2)
6      if v1 = v2 then value = v1
7      else if (v1, v2) is a key in values then value = values[(v1, v2)]
8      else
9        values[(v1, v2)] ← new value
10       value = values[(v1, v2)]
11     result.type(value) ← type
12     return value
13   UNIONVALUES(l1, l2) =
14     return [UNIONVALUE(v1, v2) : v1, v2 being pairwise elements of l1 and l2,
15             ignoring remaining elements when one is shorter]
16
17   for r being each register
18     result.values(r) ← UNIONVALUES(s1.values(r), s2.values(r))
19   for s being each stack slot in both s1 and s2
20     result.values(s) ← UNIONVALUES(s1.values(s), s2.values(s))
21
22   if s1.log.unlogged ≠ s2.log.unlogged then error
23   result.log ← log(s1.log.unlogged, s1.log.logged ∩ s2.log.logged)
24   return result
```

Figure 3.4: The algorithms for equality and unions of states s_1 and s_2.

bit set i.e. $\ldots 1110 = -2$.

$$\frac{\textsc{Writable}(\mathit{result})}{\begin{array}{l}\{\mathit{value} : \mathsf{Fixnum}\} \\ \texttt{not}\ \mathit{result}, \mathit{value} \\ \{\mathit{result} : \mathsf{Untagged}\}^{+1}\end{array}}\ (\textsc{Fixnum-Not})$$

$$\frac{\textsc{Writable}(\mathit{target})}{\begin{array}{l}\{\mathit{source} : \mathsf{Untagged}\} \\ \texttt{and}\ \mathit{target}, \mathit{source}, -2 \\ \{\mathit{target} : \mathsf{Fixnum}\}^{+1}\end{array}}\ (\textsc{Fixnum-Tag-Losing-Lsb})$$

Division eliminates the tag bit: common factors in the dividend and divisor are eliminated in a division such as $\frac{2x}{2y} = \frac{x}{y}$, producing an untagged result. The quotient thus needs to be shifted left by one bit to produce a fixnum; setting the tag bit to zero would also produce a fixnum, but would lose the least significant bit of the quotient.

$$\frac{\textsc{Writable}(\mathit{quotient})}{\begin{array}{l}\{\mathit{dividend} : \mathsf{Fixnum};\ \mathit{divisor} : \mathsf{Fixnum}\} \\ \texttt{div}\ \mathit{quotient}, \mathit{dividend}, \mathit{divisor} \\ \{\mathit{quotient} : \mathsf{Untagged}\}^{+1}\end{array}}\ (\textsc{Fixnum-Div})$$

$$\frac{\textsc{Writable}(\mathit{target})}{\begin{array}{l}\{\mathit{source} : \mathsf{Untagged}\} \\ \texttt{lsl}\ \mathit{target}, \mathit{source}, 1 \\ \{\mathit{target} : \mathsf{Fixnum}\}^{+1}\end{array}}\ (\textsc{Fixnum-Tag-Shift-Left})$$

Multiplication has the opposite problem of producing one too many zero bits as $2x \times 2y = 2 \times 2 \times xy$. We could shift the product right by one bit to get the right number of zero bits, but the most significant bit of the product would overflow in the multiplication. Instead we compensate for this by having one argument be untagged beforehand, in order to produce a product which is only shifted by one bit. Untagging

Instructions	Arguments	Behaviour
alloc	d, types	$\mathsf{Reg}_d \leftarrow$ allocate uninitialised records
write-barrier	s	log the record Reg_d
branch-instance-of	s, type, label	if Reg_s is an instance of *type*, $PC \leftarrow$ *label*
branch-not-instance-of	s, type, label	if Reg_s is not an instance of *type*, $PC \leftarrow$ *label*

Figure 3.5: Magic instructions in Pulstar.

is achieved by shifting a fixnum right by one bit.

$$\frac{\textsc{Writable}(\mathit{target})}{\begin{array}{l}\{\mathit{source} : \mathsf{Fixnum}\} \\ \texttt{asr}\ \mathit{target}, \mathit{source}, 1 \\ \{\mathit{target} : \mathsf{Untagged}\}^{+1}\end{array}}\ (\textsc{Fixnum-Untag-Shift-Right})$$

$$\frac{\textsc{Writable}(\mathit{product})}{\begin{array}{l}\{\mathit{arg1} : \mathsf{Fixnum};\ \mathit{arg2} : \mathsf{Untagged}\} \\ \texttt{mul}\ \mathit{product}, \mathit{arg1}, \mathit{arg2} \\ \{\mathit{product} : \mathsf{Fixnum}\}^{+1}\end{array}}\ (\textsc{Fixnum-Mul})$$

3.6 Magic Instructions

I introduce several *magic* instructions as in Figure 3.5, which stand in for instruction sequences, as verifying the instruction sequences would be difficult. The **`alloc`** instruction allocates (possibly multiple) uninitialised objects similar to the `malloc` instruction in Morrisett et al. [1999]; these objects are always allocated adjacent to each other. The `write-barrier` instruction performs a write barrier on its argument.

The `branch-instance-of` and `branch-not-instance-of` instructions attempt to downcast their argument and branch if the downcast succeeds or fails, respectively.

$$\begin{array}{l}\{\mathit{register} \rightsquigarrow [\mathit{value}, \_...];\ \mathit{value} : \mathsf{Heap};\ \mathit{before\text{-}type} = \textsc{Type}(\mathit{value})\} \\ \texttt{branch-instance-of}\ \mathit{register}, \mathit{type}, \mathit{target} \qquad (\textsc{Branch-Instance-Of}) \\ \{\mathit{value} : \mathit{before\text{-}type} - \mathit{type}\}^{+1} \\ \{\mathit{value} : \mathit{before\text{-}type} \sqcap \mathit{type}\}^{\mathit{target}}\end{array}$$

$\{\mathit{register} \rightsquigarrow [\mathit{value}, \_...];\ \mathit{value} : \mathsf{Heap};\ \mathit{before\text{-}type} = \textsc{Type}(\mathit{value})\}$
`branch-not-instance-of` $\mathit{register}, \mathit{type}, \mathit{target}$ (Branch-Not-Instance-Of)
$\{\mathit{value} : \mathit{before\text{-}type} \sqcap \mathit{type}\}^{+1}$
$\{\mathit{value} : \mathit{before\text{-}type} - \mathit{type}\}^{\mathit{target}}$

3.7 Initialisation

The typed assembly language tracks whether each field of a record is initialised. A runtime could also pre-zero all memory to ensure that the garbage collector may safely traverse uninitialised objects, if zeroes always denote valid values (such as a null pointer for a field containing a reference). Pre-zeroing is convenient when the language requires that fields are zero-initialised before constructor code from the application runs. But pre-zeroing has considerable time and bandwidth overheads, and thus compilers attempt to optimise initialisation by attempting to replace zero stores with subsequent initialising stores.

We require that all objects are initialised at potential safepoints (function calls and **`alloc`** magic instructions), so that the garbage collector will only encounter initialised fields. Another option would be to allow fields to be uninitialised at safepoints, but have the stack map (which was generated from the inferred types) indicate to the collector which locations refer to objects with uninitialised fields; such objects are only accessible directly from the registers and stack slots, as references to such objects cannot be written to the heap.

This model of initialisation might be inadequate for concurrent programming languages with shared memory. An instruction set with a *weak memory model* such as AArch64 may allow other threads to observe an initialising write to an object and a write of a reference to that object out of order. Then other threads may observe the object being uninitialised, despite the thread which initialised the object observing that it performed the initialisation before writing any references to the object. A common solution is to issue a *fence* instruction to prevent writes from being reordered: a fence is performed after `final` fields in Java have been initialised, and runtimes which pre-zero allocated memory often fence after zeroing the memory so that other threads will at worst read zero values in fields. A compiler may also re-order writes when the compiler determines the order of the writes cannot affect behaviour; this has similar ramifications for concurrency, and I will explain one solution in Section 4.1.1.

However, some concurrent languages ensure other threads only read initialised values without requiring a fence after object construction, for which the current model of initialisation suffices: Multicore OCaml only fences when an object escapes a thread-local heap [Dolan et al., 2018], and the BEAM[2] for Erlang only ever copies messages between process-local heaps.

[2]Bogdan or Björn's Erlang Abstract Machine

3.8 Records

The mutator may only read fields of records which the mutator had initialised beforehand, and the mutator can initialise a field by writing values to a field. The displacement must be one byte less than word-aligned, so that adding the tag bit and the displacement will produce an aligned pointer. The first field is always one word after the start of the object, as the first word is the header word (which Pulstar does not provide a way for the mutator to access).

$\textsc{Writable}(\mathit{value});\ \textsc{Aligned}(\mathit{displacement}+1);\ \mathit{index} = \textsc{Floor}(\mathit{displacement}+1, 8) - 1;\ \mathit{index} \geq 0$

$\{\mathit{type} = \textsc{Type}(\mathit{record});\ \mathit{type} \text{ is a record type};\ \textsc{Field-Initialised}(\mathit{type}, \mathit{index})\}$
`ldr` $\mathit{value}, [\mathit{record}, \mathit{displacement}]$ (Record-Load)
$\{\mathit{value} : \textsc{Field-Type}(\mathit{type}, \mathit{index})\}^{+1}$

Recall that the log contains a set of values which the mutator has run a write barrier for. This set alone would suffice to check whether a write may be performed if all write barriers are run before their respective writes. But some write barriers are run after their respective writes; in order to support running write barriers after writes, the type system also tracks values which the mutator must log before the mutator can reach a safepoint.

The behaviour of Alloc and Record-Store encode which writes to records the garbage collector must be notified of. Allocation is involved in write barrier elimination because a newly-allocated object is known to initially be in the young generation, and the garbage collector may not need to be notified of writes to young objects. The information about which objects are new is encoded in the abstract state by following an approach to using a generational collector with a logging barrier: the allocator allocates objects such that the write barrier will treat the objects as if they were already logged (that they are *dirty*), so that the slow path will not be taken on newly allocated objects [Blackburn and McKinley, 2003]. The Alloc rule applies this approach to abstract states by treating the values for newly-allocated objects as having been logged in successor states.

The mutator also may not need to perform a write barrier when writing a value which is not a reference to the heap (an *immediate* value), depending on the garbage collector. All immediates in Pulstar are fixnums; runtimes without type-tagging (such as those for Java) may still have a *null* value which is immediate by this definition, and thus the same reasoning applies for null. If the collector uses an *insertion* barrier where the collector is informed of when the mutator creates new references to objects on the heap, a write barrier is not necessary when writing an immediate value. If the collector however uses a *deletion* barrier where the collector is informed of when the mutator removes references to objects on the heap, we could only eliminate a barrier if the previous value of the field was known to be immediate. It is unlikely we know the type of the previous value: in the cases that we can determine the type,

such as when the program performs multiple writes to the same field in one function, the compiler should have eliminated all but the last write, leaving no barriers to eliminate.

The following rules assume a generational collector, where objects are allocated dirty, and that the collector uses an insertion barrier which only needs to be run when the mutator creates references from old objects to new objects:

$$\frac{\textsc{Writable}(\mathit{register});\ \mathit{values} = [\textsc{Value}(\textsc{Uninitialised}(r)) : r \leftarrow \mathit{records}]}{\begin{array}{l}\{\textsc{All-Values-Initialised};\ \textsc{All-Logged}(\textsc{Log})\} \\ \texttt{alloc}\ \mathit{register}, \mathit{records} \\ \{\textsc{Erase-Multiple};\ \mathit{register} \rightsquigarrow \mathit{values};\ \textsc{Log} = \textsc{Log-Objects}(\mathit{values})\}^{+1}\end{array}}\ (\textsc{Alloc})$$

$$\frac{\textsc{Aligned}(\mathit{displacement}+1);\ \mathit{index} = \textsc{Floor}(\mathit{displacement}+1, 8) - 1;\ \mathit{index} \geq 0}{\begin{array}{l}\left\{\begin{array}{cc} \mathit{type} = \textsc{Type}(\mathit{record}); & \mathit{type}\ \text{is a record type;} \\ \mathit{value} : \textsc{Field-Type}(\mathit{type}, \mathit{index}); & \mathit{log} = \textsc{Log}; \\ \mathit{is\text{-}fixnum} = \mathit{value} : \mathsf{Fixnum}; & \mathit{record} \rightsquigarrow [\mathit{before\text{-}value}, \_...] \end{array}\right\} \\ \texttt{str}\ \mathit{value}, [\mathit{record}, \mathit{displacement}] \\ \left\{\begin{array}{c} \mathit{before\text{-}value} : \textsc{Initialise-Field}(\mathit{type}, \mathit{index}); \\ \textsc{Log} = \text{if}\ \mathit{is\text{-}fixnum}\ \text{then}\ \mathit{log}\ \text{else}\ \textsc{Needs-Barrier}(\mathit{log}, \mathit{before\text{-}value}) \end{array}\right\}^{+1}\end{array}}\ (\textsc{Record-Store})$$

$$\frac{}{\begin{array}{l}\{\mathit{target} \rightsquigarrow [v, \_...];\ \mathit{log} = \textsc{Log}\} \\ \texttt{write-barrier}\ \mathit{target} \\ \{\textsc{Log} = \textsc{Log-Object}(\mathit{log}, v)\}^{+1}\end{array}}\ (\textsc{Write-Barrier})$$

3.9 Stack Frames

A function may only modify the stack pointer by incrementing and decrementing it in multiples of two words, so that the stack pointer is always aligned to two words. The new stack slots produced by Grow-Stack are unitialised and have no values.

$$\frac{\mathit{diff} \geq 0;\ \textsc{Double-Aligned}(\mathit{diff})}{\begin{array}{l}\{\} \\ \texttt{sub sp, sp,}\ \mathit{diff} \\ \textsc{Grow-Stack}(\textsc{Floor}(\mathit{diff}, 8))^{+1}\end{array}}\ (\textsc{Grow-Stack})$$

$$\frac{\mathit{diff} \geq 0;\ \textsc{Double-Aligned}(\mathit{diff})}{\begin{array}{l}\{\textsc{Length}(\textsc{Stack-Frame}) - \textsc{Floor}(\mathit{diff}, 8) \geq 0\} \\ \texttt{add sp, sp,}\ \mathit{diff} \\ \textsc{Shrink-Stack}(\textsc{Floor}(\mathit{diff}, 8))^{+1}\end{array}}\ (\textsc{Shrink-Stack})$$

Values may be loaded from and stored to the stack frame. Recall that the state associates stack slots with values and not types, so the update rules are written in terms of values. AArch64 also allows for loading and storing two registers at once, which compilers often generate when spilling and reloading values around function calls. Loading two values to the one register is undefined in AArch64, so we reject `ldp` instructions which load to the same register twice. Write barriers are also not used for writes to the stack.[3]

$$\frac{\textsc{Aligned}(\mathit{offset});\ \textsc{Writable}(\mathit{target})}{\begin{array}{l}\{[\texttt{sp}, 8(\textsc{Floor}(\mathit{offset}, 8))] \rightsquigarrow \mathit{values}\} \\ \texttt{ldr}\ \mathit{target}, [\texttt{sp}, \mathit{offset}] \\ \{\mathit{target} \rightsquigarrow \mathit{values}\}^{+1}\end{array}}\ (\textsc{Stack-Load})$$

$$\frac{\mathit{target1} \neq \mathit{target2};\ \textsc{Double-Aligned}(\mathit{offset});\ \textsc{Writable}(\mathit{target1});\ \textsc{Writable}(\mathit{target2})}{\begin{array}{l}\{[\texttt{sp}, 8(\textsc{Floor}(\mathit{offset}, 8))] \rightsquigarrow \mathit{values1};\ [\texttt{sp}, 8(\textsc{Floor}(\mathit{offset}, 8) + 1)] \rightsquigarrow \mathit{values2}\} \\ \texttt{ldp}\ \mathit{target1}, \mathit{target2}, [\texttt{sp}, \mathit{offset}] \\ \{\mathit{target1} \rightsquigarrow \mathit{values1};\ \mathit{target2} \rightsquigarrow \mathit{values2}\}^{+1}\end{array}}\ (\textsc{Stack-Load-Pair})$$

[3]The `stp` and `ldp` instructions are not limited to the stack, and can be used for records; I simply did not write the analogous rules for records, nor does my compiler emit such instructions.

$$\frac{\textsc{Aligned}(\mathit{offset})}{\begin{array}{l}\{\mathit{source} \rightsquigarrow \mathit{values}\} \\ \texttt{str}\ \mathit{source}, [\texttt{sp}, \mathit{offset}] \\ \{[\texttt{sp}, 8(\textsc{Floor}(\mathit{offset}, 8))] \rightsquigarrow \mathit{values}\}^{+1}\end{array}}\ (\textsc{Stack-Store})$$

$$\frac{\textsc{Double-Aligned}(\mathit{offset})}{\begin{array}{l}\{\mathit{source1} \rightsquigarrow \mathit{values1};\ \mathit{source2} \rightsquigarrow \mathit{values2}\} \\ \texttt{stp}\ \mathit{source1}, \mathit{source2}, [\texttt{sp}, \mathit{offset}] \\ \{[\texttt{sp}, 8(\textsc{Floor}(\mathit{offset}, 8))] \rightsquigarrow \mathit{values1};\ [\texttt{sp}, 8(\textsc{Floor}(\mathit{offset}, 8) + 1)] \rightsquigarrow \mathit{values2}\}^{+1}\end{array}}\ (\textsc{Stack-Store-Pair})$$

3.10 Calling Convention

Pulstar can type check a subset of the AArch64 calling convention [Arm Limited, 2024]. The types `LR`, `SP`, `Caller-FP` and `Callee-FP` are used for the calling convention, being used to type the link register, stack pointer, frame pointer of the caller and frame pointer of the callee respectively. Performing a function call with the ARM **`bl`** instruction stores the address to return to in the *link register*, as opposed to the x86 `call` instruction pushing the address onto the stack. The stack pointer grows "downwards" to lower addresses, so the stack pointer must be decremented to allocate space for stack slots, and incremented to free all space before returning from a function (or before performing a tail-call).

A function must first set up a frame pointer (a value of the type `Callee-FP`) by producing a pointer near the end of the stack frame, before it can perform another function call. The frame pointer is exactly two words lower than the upper end of the stack frame, leaving room for saving the stack pointer and link register (a *frame record*) across a function call, as depicted in 3.6. This information may also be required by a precise garbage collector to scan the roots: the runtime would use each saved value of the link register to find the stack map relevant for each stack, with the information in the stack map being relative to the stack pointer. The collector can traverse all frames by following the saved frame pointers, which form a linked list of frame records. Note that the contents of the frame record are checked by the rules involving function calls and not by the rule to establish a frame pointer, as a function could compute its frame pointer and then overwrite the record with other values before performing a

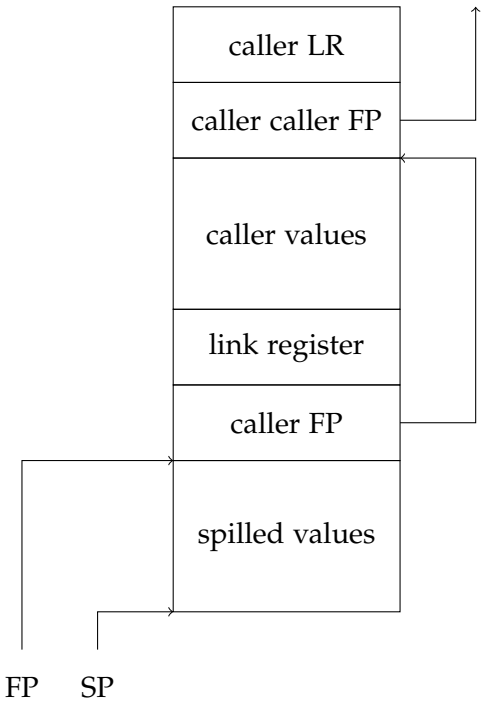


Figure 3.6: The stack in AArch64.

function call with an invalid frame record.

$$\frac{\mathit{diff} = 8 \times \textsc{Length}(\textsc{Stack-Frame}) - 2}{\begin{array}{l}\{\textsc{Length}(\textsc{Stack-Frame}) \geq 2\} \\ \texttt{add fp, sp,} \mathit{diff} \\ \{\texttt{fp : Callee-FP}\}^{+1}\end{array}} \quad (\textsc{Initialise-FP})$$

Values of each of `LR`, `Caller-FP` and `Callee-FP` may be spilled onto the stack and moved between registers, so long as they are reinstated to the appropriate registers when needed when returning and performing function calls[4]. These types cannot be stored to the heap as fields of records can only have object types; consequentially a function can only use values of the types `LR` and `Caller-FP` which were provided to it. The arguments are passed in lexicographic order in the registers not used for other purposes by the calling convention, excluding `x25` through `x27` for loading spilled values from the stack, leaving `x0` to `x15` and `x18` to `x25`. Thus we have the rules for

[4]The typed assembly language also permits functions which do not call other functions (*leaf functions*) to not establish a stack frame, though the compiler does not perform this optimisation.

calls, tail calls and returns:

$$\frac{\textit{slots} = \textsc{Length}(\textsc{Stack-Frame});\ \textit{return-type} = \textsc{Return-Type}(\textit{target})}{\left\{\begin{array}{cc} \texttt{fp} : \textsc{Callee-FP}; & [\texttt{sp}, 8(\textit{slots}-2)] : \textsc{Caller-FP}; \\ [\texttt{sp}, 8(\textit{slots}-1)] : \texttt{LR}; & \textsc{All-Values-Initialised}; \\ \textsc{Arguments-Match}(\textit{target}); & \textsc{All-Logged}(\textsc{Log}) \end{array}\right\}}$$

$$\texttt{bl}\ \textit{target} \qquad \text{(BL-Call)}$$

$$\left\{\begin{array}{cc} \textsc{Overwrite-Caller-Saves}; & \textsc{Erase-Multiple}; \\ \texttt{x0} : \textit{return-type}; & \textsc{Log} = \textsc{Empty-Log} \end{array}\right\}^{+1}$$

$$\frac{\textit{target}\text{ is a function}}{\left\{\begin{array}{cc} \textsc{All-Values-Initialised}; & \textsc{Arguments-Match}(\textit{target}); \\ 0 = \textsc{Length}(\textsc{Stack-Frame}); & \textsc{All-Logged}(\textsc{Log}) \end{array}\right\}}$$

$$\texttt{b}\ \textit{target} \qquad \text{(B-Tail-Call)}$$

$$\frac{\textit{return-type} = \textsc{Return-Type}(\textit{current-function})}{\left\{\begin{array}{cc} \texttt{lr} : \textsc{LR}; & \texttt{fp} : \textsc{Caller-FP}; \\ \texttt{x0} : \textit{return-type}; & 0 = \textsc{Length}(\textsc{Stack-Frame}); \\ \textsc{All-Logged}(\textsc{Log}) & \end{array}\right\}}$$

$$\texttt{ret} \qquad \text{(Ret)}$$

All arguments are passed in registers and all registers are caller-save for simplicity. Tail calls with arguments on the stack would be tricky otherwise, and the register allocator would have to distinguish between callee-save and caller-save registers. I did not implement a runtime accompanying the compiler and typed assembly language, but callee-save registers would also complicate stack scanning [Click, 2016]. Optimising the calling convention is generally less important with inlining, making callee-saves registers less appealing for compilers which inline heavily.[5] We could still introduce more opaque types to enforce that the callee preserves the callee-saves registers, similar to the generic types used by Morrisett et al.

[5]This theory came up in separate private discussions with Cliff Click and Robert Strandh.

3.11 Summary

I designed a typed assembly language for AArch64 which can handle arbitrary control flow through abstract interpretation and can precisely narrow the types of values aliased between registers and stack slots. The typed assembly language additionally tracks allocation folding, enforces that all objects are initialised at safepoints, and enforces that write barriers are performed correctly.

Chapter 4

Implementation

To practically evaluate the type system I need a compiler to produce code to type-check. One could (and probably would, in practice) retrofit the type checker onto a production compiler, but this appears complicated due to production compilers being difficult to extend; Steve Blackburn reported that retrofitting the MMTk garbage collector onto V8 was very complex [Blackburn, 2020] due to tight coupling in V8, which suggests that retrofitting a typed assembly language-based backend to V8 or another production compiler would take too much time for this thesis. Instead I wrote a simpler compiler for a simple source language, which still has the essential issues relevant to this thesis. I also need to implement the type inference engine in order to check that it can validate a corpus of code, and in order to measure its performance; a careful choice of representation and data structures allows for faster type inference whilst keeping the code close to the declarative rules which define the type system.

4.1 Compiler Design

I implemented a compiler which compiles code for the Utena abstract machine [Lulamoon, 2023; Applied Language, 2023] to AArch64 assembly.[1] Utena most closely resembles the Newspeak programming language; the abstract machine is very simple but retains some core concepts common with more mainstream languages like Java and JavaScript, which more strongly influence the compiler and runtime design than the rest of the concepts in the programming languages (such as the syntax of the programming languages). In particular, Utena is dynamically typed, object-oriented and uses automatic memory management. The Utena abstract machine further exacerbates the issues in implementing a high-performance runtime in several ways. The Utena abstract machine uses heap-allocated objects for storing arguments and local variables, uses method calls for control-flow and local variables like Self [Ungar and Smith, 1987], and uses objects as modules like Newspeak [Bracha et al., 2010]. Objects are also initialised to have *unbound* slots[2] alike CLOS before the slots are

[1] This compiler is available at https://gitlab.com/cal-coop/utena/movement-three.

[2] The word "slot" is used in Utena where "field" is used for Java, as CLOS and Newspeak use the word "slot". The language-level concept of a "slot" only appears in this section, and elsewhere in this

properly initialised with user values; an unbound slot is represented by storing a non-zero value in the slot, so pre-zeroing does not correspond to any language-level initialisation. A fast implementation therefore requires the optimising compiler to eliminate allocations, polymorphism and initialising stores when possible, and for the runtime to have fast allocation and method dispatch. Therefore this system especially exercises the allocation folding and initialisation tracking of Pulstar.

The compiler compiles the source language to class and method definitions for the Utena abstract machine, then to a sea-of-nodes-based intermediate representation [Click and Paleczny, 1995], then schedules into a control flow graph, then allocates registers. The compiler performs inlining, type-based alias analysis [Diwan et al., 1998], some constant folding, global value numbering, loop-invariant code motion, and linear-scan register allocation [Poletto and Sarkar, 1999]. It compiles a whole program ahead-of-time for simplicity; it specialises methods by argument types [Dean et al., 1995] in order to gain some of the type information that just-in-time compilation would find by profiling, and to better exercise the type inference engine with functions with different argument types.

There are two main limitations which affect what code can be used to evaluate Pulstar. The compiler notably does not support floating-point numbers, which are just another kind of unboxed data and do not introduce any new memory safety issues. The compiler would need to be able to allocate floating-point registers (which are separate from the general-purpose registers), and the compiler would need to be able to box and unbox floats around float operations[3]; so supporting floating-point numbers would require substantial effort which does not further exercise the garbage collection interface. Utena also currently does not have any inheritance mechanism to implement, so the compiler does not implement any inheritance mechanisms.

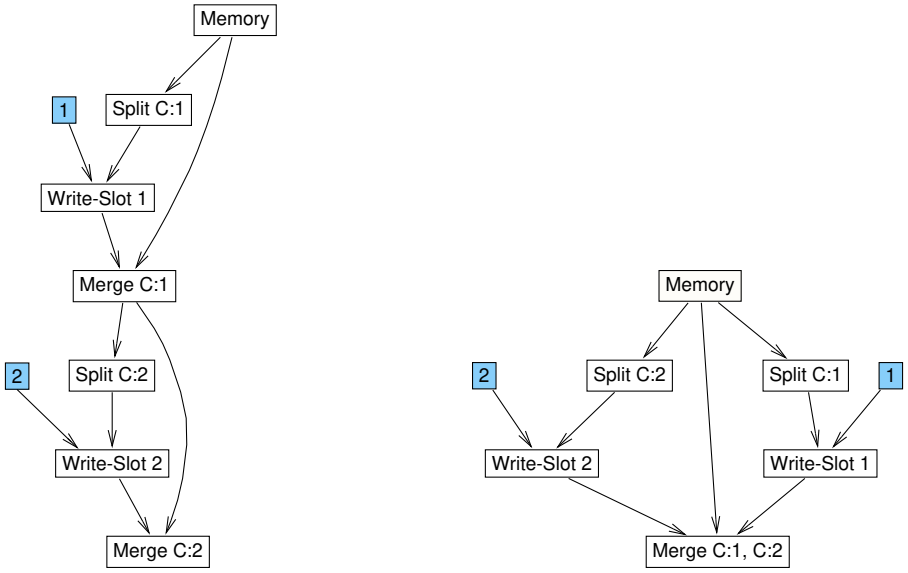
4.1.1 Initialisation

The sea-of-nodes is characterised by explicitly representing all dependencies as inputs and outputs to instructions (*nodes*), including the control flow and memory effects, and by having as few dependencies as possible between nodes. This representation has two main advantages: it first allows for more optimisations to be represented as simple graph rewriting rules (*peepholes*), including some restructuring of control flow and optimising redundant reads and writes to memory. Optimisations on control flow tend to be done on a separate *control-flow graph* which must be manipulated in a different way, when not using a sea-of-nodes compiler. The memory flow is also usually only implicit in other intermediate representations, so a different algorithm must be used to analyse and eliminate memory operations.

The compiler must then *schedule* its sea of nodes into a sequential representation for the compiler to emit assembly. The second advantage is that the scheduler can

thesis I use the word "field". Slots in Utena do not directly map to fields in Pulstar, as the compiler introduces another field to objects in order to store the *parent* object for lexical scoping.

[3]Strictly speaking, we could emit boxing and unboxing code around every floating point operation; then we can always reuse the same floating-point registers for every operations and not have to do any register allocation. However, this arrangement would produce rather poor code.



(a) Before the memory effect peepholes.

(b) After the memory effect peepholes.

Figure 4.1: Type-based alias analysis in the sea of nodes.

move nodes around arbitrarily, including by moving nodes out of loops and into rarely-executed branches, so long as the scheduler preserves the order of dependencies and places all dependent nodes after their dependencies. The scheduler can interact with type-based alias analysis, which assumes that accesses to different fields of different types can never affect each other. For example, in a program:

```
1  class C { a; b; }
2  C c = new C();
3  c.a = 1;
4  c.b = 2;
5  print(c.a);
```

the write to `c.b` cannot possibly affect `c.a`, so we can determine that the value of `c.a` must be 1. This information is represented in my compiler by introducing nodes which *split* off a particular field from the memory effects, and nodes which then *merge* a split-off effect with the rest of the effects (the *bulk*). Then a peephole rule rewrites a memory-split node which follows a memory-merge node: if the split is for a field which was involved in the merge, the rule replaces the split node with the effect prior to the merge. Otherwise the split is rewritten to split out from the bulk input to the merge, bypassing the merge. Another rule coalesces merge nodes when they immediately precede other merge nodes. An example of these peepholes is depicted in Figure 4.1. The results are that the scheduler may schedule the writes in any order, and that further peepholes on a subsequent read can quickly find the last write to its field.

However, the scheduler could cause a reference to an object to be written to a field before the object is initialised if it schedules other writes with an object before it schedules initialising writes to the object. Consider another program:

```
1 class O { x; }
2 class Box { y; }
3 Box b = new Box();
4 b.y = 0;
5 O o = new C();
6 o.x = 1;
7 b.y = o;
```

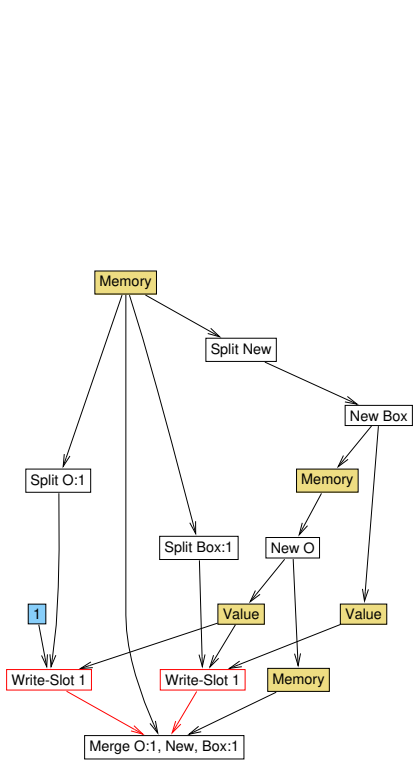
Using type-based alias analysis yields the graph in Figure 4.2(a). There are no dependencies between the two writes, so the scheduler may choose to schedule the writes on lines 6 and 7 in any order. If the scheduler chooses to schedule `b.y = o` first (and **`new`** does not initialise either object), it will have written a reference to an uninitialised object, which is ill-typed, and could expose other threads to uninitialised fields in a concurrent program. A solution, used by the C2 compiler in HotSpot, is to insert a *fence* node between the initialisation and uses of an object; in Figure 4.2(b) the compiler inserted Construct-Fence nodes which force a use of a new object to depend on the memory effects which initialised the object with unbound slots (denoted with the *unbound marker* value `+UNBOUND+`). The compiler however cannot optimise around the fence node, and now has written all of the unbound marker, a constant 0, and the object `o` to `b.y` in quick and needless succession. The write of the unbound marker can be replaced with the write of the constant 0 as in Figure 4.2(c), as the constant is always initialised, and so we cannot end up with another ill-typed schedule in doing so. We cannot, however, push the write of `o` before the fence, as that would produce a cycle in the memory effects.[4]

4.2 Type Inference Engine Implementation

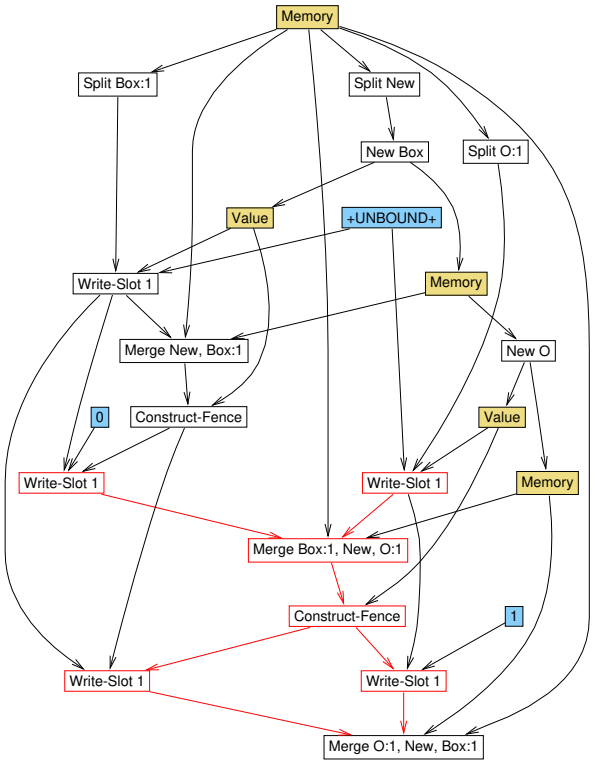
The type checker revolves around manipulating abstract states, of which the more complex part is to represent the relations between locations, values and types. I used arrays to map from locations to lists of values, as the number of registers is constant, and the stack slots have consecutive offsets from zero to the size of the stack frame (as established by Grow-Stack and Shrink-Stack). A simple approach to representing the relation between values and types in a state is to use a hash table, with values compared by *object identity*; but I found that lookups in hash tables and copying hash tables were too slow.[5] Instead, each state uses an array to represent the relation of values to types in a state, and each state also maps each location to a pair of a *local* index and a *global* ID. The local index indexes into an array of types of values: we only ever add values to states, and so we can assign the next index in the array to a value when we extend the array with the type of that value. This index however may not be the same for the same value in different states, when combined with the

[4] My compiler only allows writes of arguments and constants to be pushed before fences, but C2 uses a more elaborate test which permits more writes to be pushed: https://github.com/openjdk/jdk/blob/jdk23/src/hotspot/share/opto/memnode.cpp#L4581-L4693

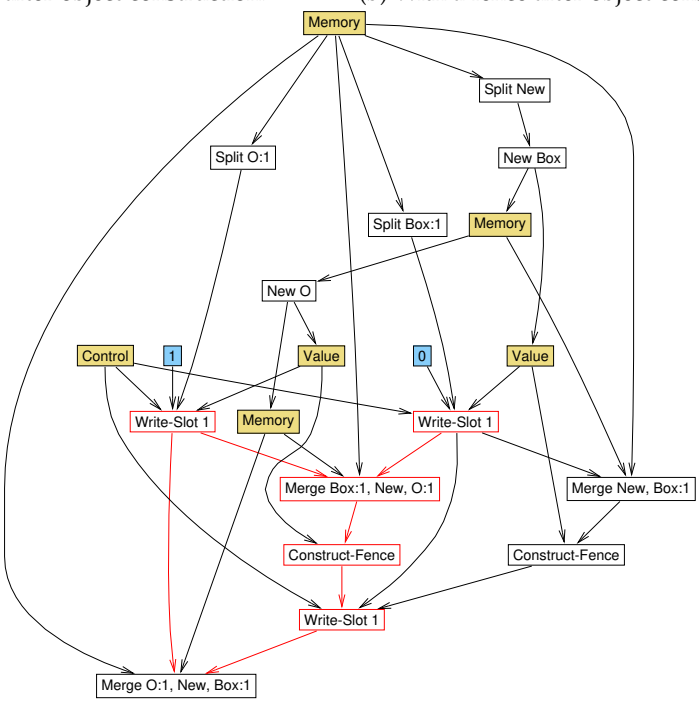
[5] Gábor Melis developed an adaptive hashing scheme [Melis, 2024] concurrently with the writing of this thesis, which was recently upstreamed in Steel Bank Common Lisp. His changes are likely to speed up the type checker, but I think my current approach would still be faster by avoiding hashing entirely.



(a) Without a fence after object construction.



(b) With a fence after object construction.



(c) With a fence after object construction, and optimising around the fence.

Figure 4.2: Using a construction fence to prevent the publication of uninitialised objects.

garbage collection scheme to be described. Instead the global ID is used to compare if two values are the same. The global ID does not index into any data structure, and so it is held stable between states.

The abstract state can accumulate garbage values which are not reachable from any locations. Two back-to-back instructions with the same destination register will create garbage for example, as the value produced by the first instruction will be unreachable after the second instruction. I did not implement any garbage collection technique, although reference counting would suffice to reclaim garbage, as states are acyclic. The amount of garbage is however bounded by the number of instructions in a basic block, as the union operation on states only inserts live values into the output state, similar to a copying garbage collector which copies the live values from one semispace to the other semispace [Fenichel and Yochelson, 1969].

I wrote the type checker and compiler in Common Lisp, which provides a macro system to extend the syntax of the programming language. Macros have allowed me to write the type inferencer in a manner similar to the mathematical notation used for describing the typing rules; I wrote a bespoke macro `define-rule` which allows for writing rules in a similar manner to Hoare logic, by specifying requirements of the predecessor state to a rule and how to construct successor states. The implementation of three rules is presented in Figure 4.3: a rule contains a name and a pattern[6] for an instruction to match (on lines 1, 13 and 20-21), values to bind from the predecessor state (on lines 2–4 and 16), conditions which must hold (lines 5–9, 14–15 and 21) and the successor states (lines 11, 18, and 25–26).

Pulstar runs the rules of the type system by calling a compiled function which performs the pattern-matching dispatch. Pulstar pre-computes this compiled function by collecting all the rules, combining the code of each into one `match` form, and compiling a function with that code by using the Common Lisp function `compile`. Pulstar also includes a *pretty printer* which generates LaTeX source code from the rules, which I used to present the rules in Chapter 3; some rules contain forms to guide the pretty printer, or replace the output outright (such as `emitting`, as on line 3).

4.3 Summary

This chapter presented a compiler which we will use to test the type system in Chapter 5, and the data structures used to efficiently implement the type system while keeping the code at a high level and without visible mutation.

[6] Common Lisp does not have pattern matching; I used the pattern matching library Trivia https://github.com/guicho271828/trivia/.

```
(define-rule ret ('arm:ret '())
  :given return-type :=
    (emitting "\\mathsc{Return-Type}(\\var{current-function})"
              (convert-mt-type (mt:code-return-type *code*)))
  :when (has-type-p before arm:*lr* +lr+) :else (error "LR not in LR")
  :when (has-type-p before arm:*fp* +fp+) :else (error "FP not in FP")
  :when (has-type-p before (mt:register 0) return-type)
  0 := (length (type-env-stack before))
  :when (logging-state-all-logged-p (type-env-log before))
  ---
  :stop)

(define-rule stack-load ('arm:ldr (list target (eq arm:*sp*) offset))
  :given (aligned-p offset)
  :given (writable-register-p target)
  values := (get-values before (stack-location (floor offset 8)))
  ---
  (update-type-env before (:values target) values))

(define-rule bcc-local ((or 'arm:blt 'arm:ble 'arm:beq
                            'arm:bne 'arm:bge 'arm:bgt)
                        (list target))
  :given (movement-three::basic-block) := target
  ---
  before
  target before)
```

Figure 4.3: Use of the `define-rule` macro to define RET, STACK-LOAD and BCC-LOCAL.

Chapter 5

Evaluation

The previous two chapters detailed the type system of Pulstar and how I have efficiently implemented Pulstar respectively. Chapter 4 also introduced my compiler for the Utena abstract machine which targets Pulstar. Now we may evaluate the efficacy of Pulstar. The main result is if we are able to type-check all of the output of the compiler. Then we also prefer to have a small implementation to minimise the trusted computing base, and we prefer fast type-checking in order to reduce the overhead of using a typed assembly language as a defense in production.

Pulstar is implemented in 829 lines of Common Lisp code, approximately a quarter of the size of my 3,454-line compiler. Either line count excludes the tests and visualisation tools I wrote for either codebase, which are not involved at all when using either codebase normally. Only the type inference engine itself is the trusted computing base when using a typed assembly language, and only the compiler is the trusted computing base when not using a typed assembly language (or some other mitigation against an untrusted compiler). This ratio is less extreme than, for example, the 14 thousand-line iTalX checking the output of the 200 thousand-line Bartok compiler. But an important effect of using a typed assembly language (or translation validation) is that the typed assembly language should seldom need to change when the compiler changes. My compiler would undoubtedly grow in size and complexity from adding some of the many missing optimisations. For example, the compiler should split control-flow to eliminate redundant tests on union types [Chambers and Ungar, 1990] and should use a better register allocation algorithm (such as *second-chance bin packing* [Traub et al., 1998]), and such optimisations could be implemented without modifying Pulstar.

5.1 Test Cases

I ported many of the benchmarks in the are-we-fast-yet benchmark suite [Marr et al., 2016] to Utena, excluding the benchmarks which use floating-point numbers or inheritance, leaving the benchmarks in Table 5.1.[1] I also ported some support code, consisting of the custom pseudo-random number generator in the benchmark harness (Harness), some of the data structures in the Newspeak platform (Platform), and stan-

[1] These benchmarks are available at https://gitlab.com/cal-coop/utena/are-we-fast-yet/.

Table 5.1: The benchmarks and support code ported to Utena.

Name	Lines of code
Benchmarks	
Bounce	37
Havlak	334
List	31
Permute	28
Queens	46
Sieve	18
Storage	17
Towers	48
Support code	
Harness	7
Platform	53
SOM	177

dard versions of data structures which have been ported to every language used in benchmarks (taken from the Simple Object Model, SOM [Haupt et al., 2010]). Pulstar is able to type check all of the compiler output when compiling these benchmarks.

5.2 Type Checking Speed

Each benchmark was compiled 100 times with and without the type checker, resulting in the timings in Table 5.2. The time taken type checking is quadratic with regards to the number of instructions and to the number of basic blocks as in Figure 5.1; with the number of basic blocks being more correlated with time than the number of instructions. The compile time overheads vary between 12% and 17% with a geometric mean of 13%. A comparison with prior work is difficult, as the prior work on typed assembly languages seldom discusses type checking time, and if performance is mentioned, any comparisons would be between almost entirely dissimilar systems. Pulstar nonetheless introduces more overhead on average but less variable overhead than iTalX, which has overhead varying between "about 1%–35% of compilation time, with a geometric mean of 8%" [Tate et al., 2010]; again noting that these results compare different code being compiled, different compilers and different type systems on different hardware.

The quadratic behaviour is due to that the number of times a basic block is revisited grows linearly with the number of basic blocks, as in Figure 5.2. This is consistent with the results of Tate et al. who found that their "type inference is more sensitive to the control flow structures of methods" and complex control flow can cause "type inference [to take] much longer to reach a fixed point for preconditions

Table 5.2: Compile times (in milliseconds) with and without the typed assembly phase.

Program	Without TAL	With TAL	Ratio
Bounce	30.3	35.5	1.17
Havlak	252	286	1.13
List	44.0	50.1	1.14
Permute	26.4	29.5	1.14
Queens	20.1	22.6	1.12
Sieve	21.1	23.8	1.12
Storage	11.1	12.5	1.13
Towers	60.8	69.0	1.14
		Geometric mean	1.13

of basic blocks". But the number of times that basic block are re-visited is heavily dependent on the order in which basic blocks are traversed: basic blocks are on average traversed 1.41 times when using a queue of basic blocks to visit, which produces a breadth-first traversal. Basic blocks are on average traversed 2.94 times when using a stack of basic blocks to visit, which produces a depth-first traversal. If the inference algorithm produces a different successor state from visiting a basic block, the algorithm must then enqueue all successors of that basic block to be revisited later. Type inference using a stack performs more visits as the inference algorithm pops the successors to a merge in the control flow before it pops the predecessors leading to the merge, and then the algorithm must revisit the successors after visiting *any* predecessor to the merge. In contrast, breadth-first search visits all blocks immediately after a branch in the control flow before it visits the successors of any of those blocks, accumulating more predecessor states for those successors.

5.3 Write Barrier Elimination Algorithms

I implemented an analysis which tracks which objects are known to be logged, optionally including the allocate-logged optimisation. This analysis is performed both per basic block (as is done in the Android runtime) and as a data-flow analysis; the data-flow analysis eliminates exactly one more write barrier than the per-basic-block analysis.

The most complex analysis tracks the possible generations of objects involved in writes. This analysis is the most flexible, in that the garbage collection implementor can elect to eliminate or preserve write barriers based on any combinations of known generations. I chose to keep the barrier for only old-to-new references, which are those relevant to a generational stop-the-world tracing collector.

The static count of write barriers eliminated per analysis is presented in Table 5.3; most writes are initialising writes and write to objects allocated in the same basic block. Using a data-flow analysis rather than a per-basic-block analysis eliminates

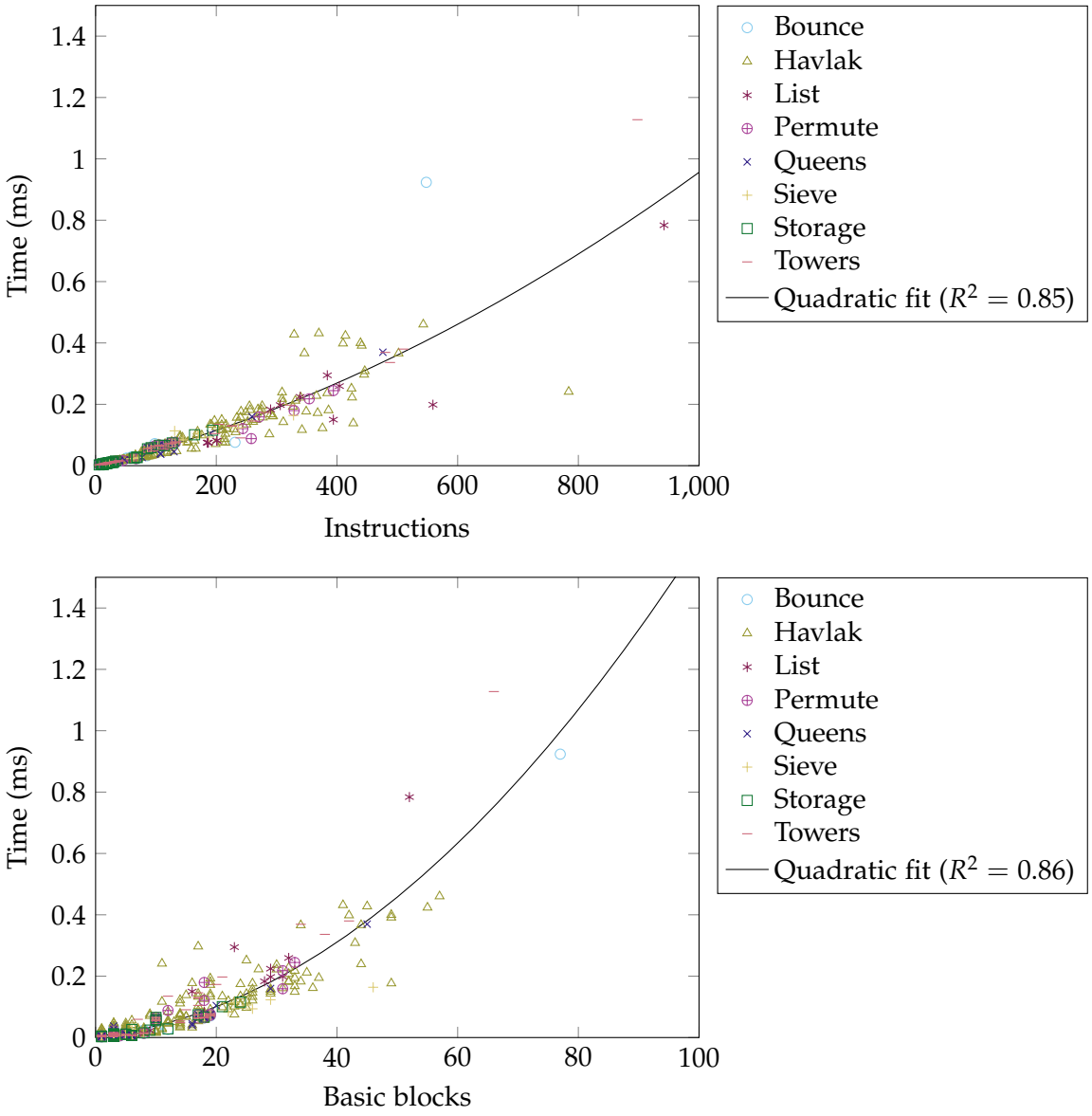
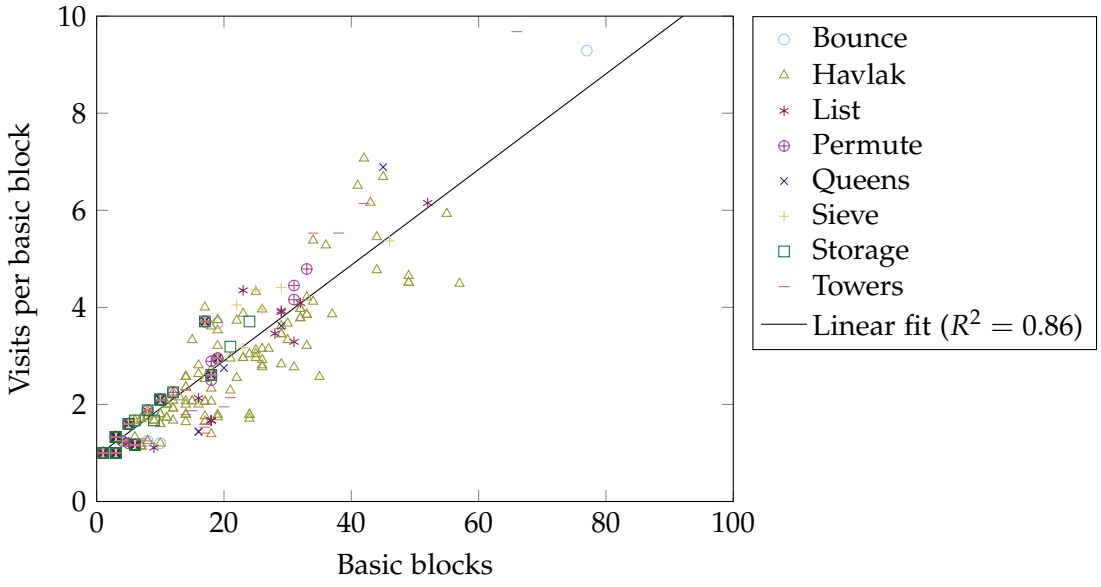
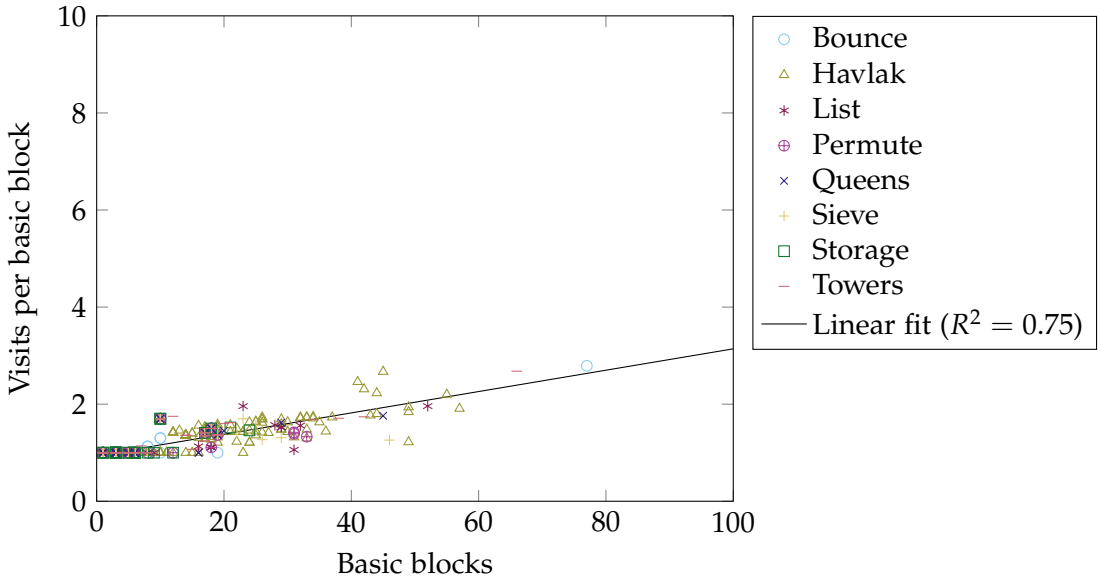


Figure 5.1: The type checking time versus instructions and basic blocks per function.



(a) Visits using depth-first traversal.



(b) Visits using breadth-first traversal.

Figure 5.2: The average visits of basic blocks in each function versus the size of each function.

Table 5.3: How many write barriers are eliminated by each algorithm.

Algorithm	Without folding	With folding
Logged before in same basic block	1209	1229
Logged before analysis	1210	1230
Same control input as allocation site	–	3534
Logged before in same basic block + pre-logging	3666	3569
Logged before analysis + pre-logging	3667	3570
Generation analysis	3793	3630
Total writes	3940	3804

exactly one more write barrier in my analysis: a method `set-nesting-level!` in Havlak takes an argument *level*, sets a field of the receiver to *level*, and then calls `root!` if *level* is zero. `root!` then sets another field of the receiver. My compiler inlines `root!` into `set-nesting-level!`, producing the assembly in Figure 5.3, which allows the write in `root!` to not need a barrier.

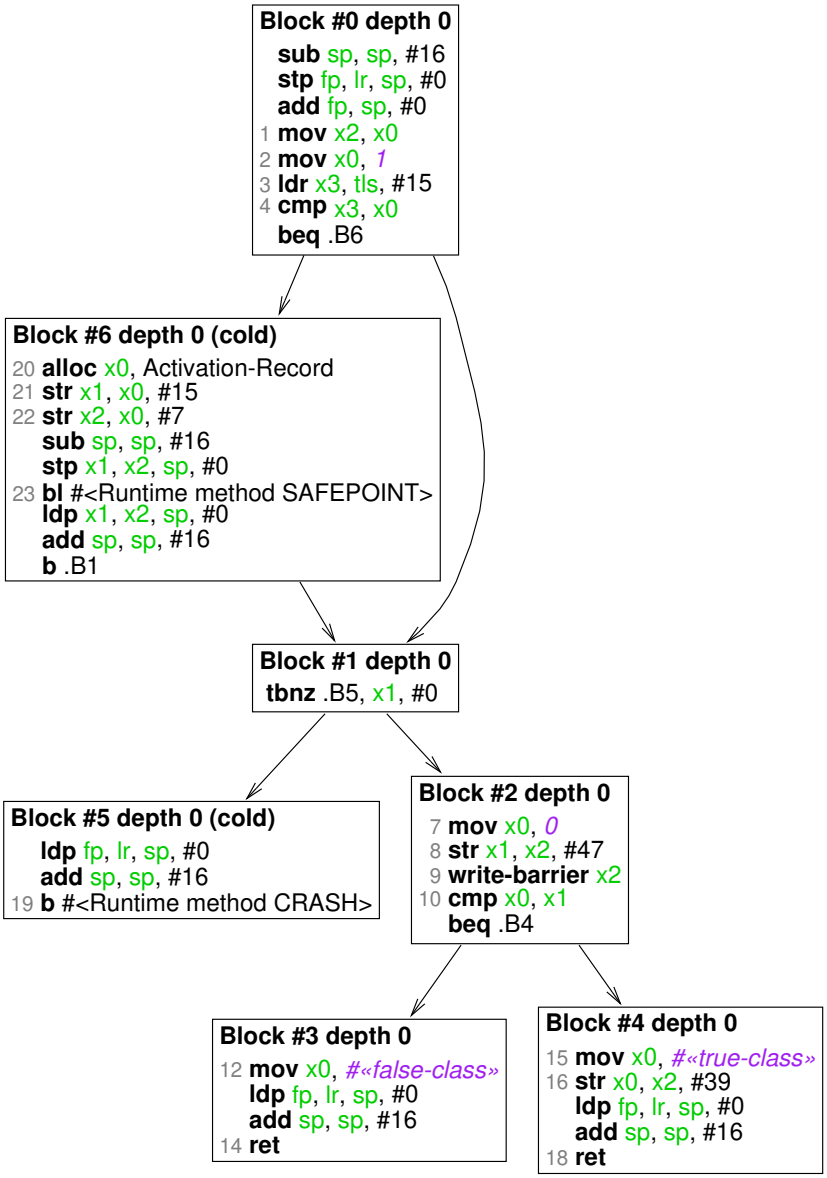
The results suggest that the additional precision of a flow analysis may not be worthwhile, and production runtimes have made a sensible tradeoff between implementation complexity and compiler speed on one hand, and the diminishing returns in precision on the other. The C2 compiler, for example, only checks whether the control-flow input to a write to a newly allocated object is the same as the control-flow input to the node which allocated the object; my compiler using this check eliminates 99% of the writes that the logged-before analysis eliminates. The gap in precision is unlikely to substantially affect performance: Zhao and Blackburn [2020] report an average 12% time overhead for the write barrier used in the garbage-first collector, for which eliminating 1% more write barriers could be estimated to have a miniscule 0.12% effect on run time. However, the inlining heuristic used by my compiler is crude and untested, and too little inlining would cause the compiler to generate functions with little code and little control flow, understating the effects of the flow analysis.

5.4 Summary

Pulstar is able to validate the results of compiling much of the are-we-fast-yet benchmark suite, introducing a geomean 13% overhead on compile time. The implementation of Pulstar is about a quarter of the size of the compiler that Pulstar validates, although the compiler is missing optimisations which would not require additional complexity in Pulstar. The time spent type checking is quadratic to function size, as the number of visits of each basic block grows linearly with the number of basic blocks. The order in which basic blocks are re-visited by type inference drastically affects type checking speed, and a breadth-first traversal requires fewer re-visits than a depth-first traversal.

```
1  (define (root!) (set! root? #t))        ; Instruction #16
2  (define (set-nesting-level! level)
3    (set! %nesting-level level)           ; Instruction #8
4    (when (=: 0 level) (root!)))
```

(a) The methods `set-nesting-level!` and `root!` in Havlak.



(b) The assembly generated for `set-nesting-level!`.

Figure 5.3: The store in `set-nesting-level!` (instruction #8) requires a write barrier (instruction #9), but the second store in `root!` (instruction #16) does not.

Chapter 6

Conclusion

Memory safety is a crucial prerequisite for secure software, and memory safety can be attained through the use of garbage collection to automatically reclaim memory after it cannot be used by the mutator. The correct function of a garbage collector depends on the mutator following an interface which can include constraints on allocation, initialisation and when the mutator must run write barriers, and these constraints are often informal and not machine-checkable.

This thesis presented the typed assembly language Pulstar which formalises the invariants of the garbage collection interface, and can validate that compiled code for a mutator will not violate the invariants. The main contributions of Pulstar are that Pulstar can validate that allocation folding is performed correctly and that the compiler has emitted enough write barriers. The implementation of Pulstar is fast, introducing a 13% geomean overhead on compile time, which could be acceptable even when compilation speed is paramount as in just-in-time compiling runtimes. The implementation is only 829 lines of Common Lisp code, suggesting that use of a typed assembly language would drastically reduce the trusted computing base of a runtime which would otherwise depend on the compiler to generate memory-safe code.

Typed assembly languages provides a low-overhead approach to ensuring that a user cannot be subject to security exploits involving memory unsafety or type unsafety, and Pulstar illustrates how a typed assembly language may be designed to the requirements of modern production runtimes. A static analysis such as Pulstar can complement or outright replace time-consuming debugging checks. A program does not have to be run for Pulstar to detect miscompilations, and a static analysis considers all possible executions of the program at once, reducing the search space and thus time needed for fuzzing to find bugs. Finally, Pulstar represents a novel approach to designing typed assembly languages, by relying on the optimising compiler to eliminate complex types, and instead expending complexity on new features such as the model of the garbage collection interface.

6.1 Future Work

One remaining issue is how to eliminate coupling between a typed assembly language and its assumptions of the runtime system and source language. The typing rules of Pulstar are too tightly coupled to one particular object representation and the type inference algorithm. I have also presently only presented the type system in a somewhat informal manner, which cannot easily be used to prove that the type system is sound.

6.1.1 Decoupling Representations from the Typed Assembly Language

Whilst Pulstar supports one pointer tagging scheme (loosely based on the tagging scheme used in Steel Bank Common Lisp), other runtimes using pointer tagging may use different tags. For example, OCaml inverts the meanings of the tag bits, such that fixnums are tagged with a 1 and references to records are tagged with a 0. The typing rules around records and fixnums also assume the SBCL tagging scheme, by implicitly encoding how the tag bits and pointer alignment are affected by each operation. Applying the rule Fixnum-Tag-Preserving-Op for an `add` instruction for example would not work with the OCaml tagging scheme, as adding two `Fixnums` (odd integers) would result in a `Heap` reference (an even integer) contrary to the rule.

One way to decouple Pulstar from any object representation is have the type system only have hard-coded rules for determining which bits in each location are known to have particular values (a *known-bit analysis*). Then the type system would consult a user-provided specification of the object representation to turn the bit patterns into higher-level types. Such specifications of object representations have appeared already appeared for the purpose of introducing custom representations to high-level languages [Teo, 2024; Baudon et al., 2023]. The specifications would also be required by the garbage collector for the garbage collector to trace objects with the custom representations. The combination of typed assembly language and garbage collector would allow multiple runtimes with different object representations to share a heap, and could enable interoperability with higher-level data structures and automatic memory management.

6.1.2 Verifying Type System Soundness

The complex parts of Pulstar are very different to the complex parts of most type systems: Pulstar only uses monomorphic types and does not have first-class functions. Pulstar however tracks aliasing in order to apply downcasting precisely, and its use of abstract interpretation might complicate proving termination of type inference. Formalising Pulstar in a proof assistant language is a prerequisite to a machine-checked proof, and thus even just a formalisation would be a valuable first step. One could use the *extraction* feature of some proof assistants to use a verified type checker in a production runtime, whilst avoiding introducing bugs in translation to another programming language.

6.1.3 Separate Inference and Checking

Separating type inference and type checking would further simplify verifying the type checker, as suggested in iTalX, and we could only trust the type checker. The type inference would perform abstract interpretation solely to produce annotations for the expected state at the start of per basic block. Then the checker would only compute types following the annotations, identically to the original stack-based typed assembly language of Morrisett et al. [2002], which would take linear time as the checker only has to compute types for each basic block once. We could also cache the inferred annotations alongside compiled code, and then only run the type checker against the annotations to quickly ensure that previously compiled code, from a cache on disk or downloaded from the Internet, is at least memory safe.

6.1.4 Removing Magic Instructions

The use of magic instructions in Pulstar may also be problematic for adoption in production runtimes. I introduced the `branch-instance-of` and `branch-not-instance-of` instructions, as the existing solution to representing downcast checks is slightly complicated: Coolaid tracks which values contain the record headers (which Chang et al. call "tags", unrelated to my use of the term) of other values. Coolaid then interprets comparisons to such values containing headers as downcasts. The `write-barrier` instruction stands in for the write barrier used by the garbage collector, which is specific to the garbage collector.

The `alloc` instruction and the treatment of allocation folding could be replaced with a more general model however. Many garbage collectors have the mutator allocate memory by *bump allocation*, wherein the mutator allocates objects contiguously out of a range (an *allocation buffer*) of unallocated memory, until the mutator detects that it cannot fit another object into that range and then obtains another range from the heap. The mutator essentially splits the range of unallocated memory into a range to be used for the new object and another range of the remaining unallocated memory. The original range cannot be reused after being split, else it would be possible to split the same range twice to produce two objects with the same location in memory; ranges of unallocated memory are linearly-typed resources [Girard, 1987]. With such restrictions we should be able to replace `alloc` by the memory-splitting operation. Implementing allocation folding only requires that both ranges produced from a splitting operation can be split again, so that the result of one bump allocation can be subdivided into multiple objects. Baker [1995] curiously also suggests the use of linear types in constructing objects, but still treats allocation as a primitive operation. Baker however mentions that objects under construction in his scheme are not first-class objects, like uninitialised objects in Pulstar, and like unallocated memory in this memory-splitting model. His model explicitly converts constructed objects to a non-linear type, whereas Pulstar implicitly treats fully-initialised records as first-class objects.

Bibliography

Aiken, M.; Fahndrich, M.; Hawblitzel, C.; Hunt, G.; and Larus, J., 2006. Deconstructing process isolation. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 1–10. ACM. https://www.microsoft.com/en-us/research/publication/deconstructing-process-isolation/. (cited on page 9)

Applied Language, 2023. Utena: Draft specification of a maximalist computing system. https://cal-coop.gitlab.io/utena/utena-specification/main.pdf. (cited on page 33)

Arm Limited, 2024. Procedure call standard for the Arm 64-bit architecture (AArch64). https://github.com/ARM-software/abi-aa/blob/main/aapcs64/aapcs64.rst. (cited on page 29)

Baker, H. G., 1995. "Use-once" variables and linear objects: storage management, reflection and multi-threading. *SIGPLAN Not.*, 30, 1 (Jan. 1995), 45–52. doi:10.1145/199818.199860. https://doi.org/10.1145/199818.199860. (cited on page 51)

Bartlett, J. F., 1988. Compacting garbage collection with ambiguous roots. *SIGPLAN Lisp Pointers*, 1, 6 (Apr. 1988), 3–12. doi:10.1145/1317224.1317225. (cited on page 5)

Bastien, J., 2015. No sane compiler would optimize atomics. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4455.html. (cited on page 10)

Baudon, T.; Radanne, G.; and Gonnord, L., 2023. Bit-stealing made legal: Compilation for custom memory representations of algebraic data types. *Proc. ACM Program. Lang.*, 7, ICFP (Aug. 2023). doi:10.1145/3607858. https://doi.org/10.1145/3607858. (cited on page 50)

Blackburn, S. M., 2020. Garbage collection: implementation, innovation, performance, and security (keynote). In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, MPLR '20 (Virtual, UK, 2020), 1. Association for Computing Machinery, New York, NY, USA. https://youtu.be/3L6XEVaYAmU?t=2337. (cited on page 33)

Blackburn, S. M. and McKinley, K. S., 2002. In or out? Putting write barriers in their place. *SIGPLAN Not.*, 38, 2 supplement (6 2002), 175–184. doi:10.1145/773039.512452. https://doi.org/10.1145/773039.512452. (cited on page 9)

Blackburn, S. M. and McKinley, K. S., 2003. Ulterior reference counting: fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM*

SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA '03 (Anaheim, California, USA, 2003), 344–358. Association for Computing Machinery, New York, NY, USA. doi:10.1145/949305.949336. https://doi.org/10.1145/949305.949336. (cited on page 26)

Boehm, H.-J. and Weiser, M., 1988. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18, 9 (Sep. 1988), 807–820. doi:10.1002/spe.4380180902. https://doi.org/10.1002/spe.4380180902. (cited on page 5)

Bracha, G.; von der Ahé, P.; Bykov, V.; Kashai, Y.; Maddox, W.; and Miranda, E., 2010. Modules as objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming*, 405–428. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 33)

Chambers, C. and Ungar, D., 1990. Iterative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90 (White Plains, New York, USA, 1990), 150–164. Association for Computing Machinery, New York, NY, USA. doi:10.1145/93542.93562. https://doi.org/10.1145/93542.93562. (cited on page 41)

Chang, B.-Y. E.; Chlipala, A.; Necula, G. C.; and Schneck, R. R., 2005. Type-based verification of assembly language for compiler debugging. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '05 (Long Beach, California, USA, 2005), 91–102. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1040294.1040303. https://doi.org/10.1145/1040294.1040303. (cited on page 8)

Chen, J.; Hawblitzel, C.; Perry, F.; Emmi, M.; Condit, J.; Coetzee, D.; and Pratikaki, P., 2008. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08 (Tucson, AZ, USA, 2008), 183–192. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1375581.1375604. https://doi.org/10.1145/1375581.1375604. (cited on page 9)

Click, C., 2016. Bits of advice for the VM writer. https://youtu.be/Hqw57GJSrac?si=zg2MgHonBVISb8-e&t=4299. (cited on page 31)

Click, C. and Paleczny, M., 1995. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95 (San Francisco, California, USA, 1995), 35–49. Association for Computing Machinery, New York, NY, USA. doi:10.1145/202529.202534. https://doi.org/10.1145/202529.202534. (cited on page 34)

Clifford, D.; Payer, H.; Starzinger, M.; and Titzer, B. L., 2014. Allocation folding based on dominance. In *Proceedings of the 2014 International Symposium on Memory Management*. New York, NY, USA. (cited on page 7)

Cousot, P. and Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77 (Los Angeles, California, 1977), 238–252. Association for Computing Machinery, New York, NY, USA. doi:10.1145/512950.512973. https://doi.org/10.1145/512950.512973. (cited on page 17)

Dean, J.; Chambers, C.; and Grove, D., 1995. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95 (La Jolla, California, USA, 1995), 93–102. Association for Computing Machinery, New York, NY, USA. doi:10.1145/207110.207119. https://doi.org/10.1145/207110.207119. (cited on page 34)

Dijkstra, E. W.; Lamport, L.; Martin, A. J.; Scholten, C. S.; and Steffens, E. F. M., 1978. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21, 11 (Nov. 1978), 966–975. doi:10.1145/359642.359655. https://doi.org/10.1145/359642.359655. (cited on page 6)

Diwan, A.; McKinley, K. S.; and Moss, J. E. B., 1998. Type-based alias analysis. *SIGPLAN Not.*, 33, 5 (May 1998), 106–117. doi:10.1145/277652.277670. https://doi.org/10.1145/277652.277670. (cited on page 34)

Dolan, S.; Sivaramakrishnan, K.; and Madhavapeddy, A., 2018. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018 (Philadelphia, PA, USA, 2018), 242–255. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3192366.3192421. https://doi.org/10.1145/3192366.3192421. (cited on page 25)

Ericsson AB, 2024. Erlang garbage collector. https://www.erlang.org/doc/apps/erts/garbagecollection.html. (cited on page 7)

Fallin, C., 2021. Cranelift, part 3: Correctness in register allocation. https://cfallin.org/blog/2021/03/15/cranelift-isel-3/. (cited on page 10)

Fenichel, R. R. and Yochelson, J. C., 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12, 11 (Nov. 1969), 611–612. doi:10.1145/363269.363280. https://doi.org/10.1145/363269.363280. (cited on page 38)

Gammie, P.; Hosking, A. L.; and Engelhardt, K., 2015. Relaxing safely: verified on-the-fly garbage collection for x86-tso. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15 (Portland, OR, USA, 2015), 99–109. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2737924.2738006. https://doi.org/10.1145/2737924.2738006. (cited on page 1)

Girard, J.-Y., 1987. Linear logic. *Theoretical Computer Science*, 50, 1 (1987), 1–101. doi: https://doi.org/10.1016/0304-3975(87)90045-4. https://www.sciencedirect.com/science/article/pii/0304397587900454. (cited on page 51)

Goldberg, A. and Robson, D., 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., USA. ISBN 0201113716. (cited on page 6)

Google, 2017. Security: V8: JIT: Simplified-lowererer IrOpcode::kStoreField, IrOpcode::kStoreElement optimization bug. https://issues.chromium.org/issues/40089761. (cited on pages 2 and 7)

Gross, S., 2024. The V8 sandbox. https://v8.dev/blog/sandbox. (cited on page 9)

Haupt, M.; Hirschfeld, R.; Pape, T.; Gabrysiak, G.; Marr, S.; Bergmann, A.; Heise, A.; Kleine, M.; and Krahn, R., 2010. The som family: virtual machines for teaching and research. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10 (Bilkent, Ankara, Turkey, 2010), 18–22. Association for Computing Machinery, New York, NY, USA. doi: 10.1145/1822090.1822098. https://doi.org/10.1145/1822090.1822098. (cited on page 42)

Hunt, S. and Sands, D., 2006. On flow-sensitive security types. *SIGPLAN Not.*, 41, 1 (jan 2006), 79–90. doi:10.1145/1111320.1111045. https://doi.org/10.1145/1111320.1111045. (cited on page 15)

Jimenez, J. and Rao, V., 2024. Google Chrome V8 CVE-2024-0517 out-of-bounds write code execution. https://blog.exodusintel.com/2024/01/19/google-chrome-v8-cve-2024-0517-out-of-bounds-write-code-execution/. (cited on page 2)

Kwon, S.; Kwon, J.; Kang, W.; Lee, J.; and Heo, K., 2024. Translation validation for JIT compiler in the V8 JavaScript engine. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24 (Lisbon, Portugal, 2024). Association for Computing Machinery, New York, NY, USA. doi:10.1145/3597503.3639189. https://doi.org/10.1145/3597503.3639189. (cited on page 10)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26, 6 (Jun. 1983), 419–429. doi:10.1145/358141.358147. https://doi.org/10.1145/358141.358147. (cited on page 6)

Lopes, N. P.; Lee, J.; Hur, C.-K.; Liu, Z.; and Regehr, J., 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021 (Virtual, Canada, 2021), 65–79. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3453483.3454030. https://doi.org/10.1145/3453483.3454030. (cited on page 10)

Lozano, R. C. and Österlund, E., 2023. Late G1 barrier expansion. https://openjdk.org/jeps/8322295. (cited on page 2)

Lulamoon, G., 2023. Utena: A system to fulfil my desire for maximalist computing. https://applied-langua.ge/~gnuxie/posts/utena-introduction.html. (cited on page 33)

Marr, S.; Daloze, B.; and Mössenböck, H., 2016. Cross-language compiler benchmarking: are we fast yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016 (Amsterdam, Netherlands, 2016), 120–131. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2989225.2989232. https://doi.org/10.1145/2989225.2989232. (cited on page 41)

McCreight, A.; Shao, Z.; Lin, C.; and Li, L., 2007. A general framework for certifying garbage collectors and their mutators. *SIGPLAN Not.*, 42, 6 (6 2007), 468–479. doi:10.1145/1273442.1250788. https://doi.org/10.1145/1273442.1250788. (cited on page 1)

McIlroy, R.; Sevcik, J.; Tebbi, T.; Titzer, B. L.; and Verwaest, T., 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. https://arxiv.org/abs/1902.05178. (cited on page 9)

Melis, G., 2024. Adaptive hashing. In *Proceeedings of the 17th European Lisp Symposium (ELS'24)*. European Lisp Symposium. doi:10.5281/zenodo.11091296. https://doi.org/10.5281/zenodo.11091296. (cited on page 36)

Morrisett, G.; Crary, K.; Glew, N.; and Walker, D., 2002. Stack-based typed assembly language. *Journal of Functional Programming*, 12, 1 (2002), 43–88. doi:10.1017/S0956796801004178. (cited on pages 8 and 51)

Morrisett, G.; Walker, D.; Crary, K.; and Glew, N., 1999. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21, 3 (5 1999), 527–568. doi:10.1145/319301.319345. https://doi.org/10.1145/319301.319345. (cited on pages 7, 17, and 24)

Myreen, M. O.; Fox, A. C. J.; and Gordon, M. J. C., 2007. Hoare logic for ARM machine code. In *International Symposium on Fundamentals of Software Engineering*, 272–286. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 17)

Österlund, E., 2020. C2 SATB barriers are not safepoint-safe. https://bugs.openjdk.org/browse/JDK-8242115. (cited on pages 1 and 11)

Patton, H., 2023. Parallel garbage collection for SBCL. In *Proceedings of the 16th European Lisp Symposium (ELS'23)*. European Lisp Symposium. doi:10.5281/zenodo.7816398. https://zenodo.org/records/7816398. (cited on page 5)

Pnueli, A.; Siegel, M.; and Singerman, E., 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, 151–166. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 10)

Poletto, M. and Sarkar, V., 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21, 5 (sep 1999), 895–913. doi:10.1145/330249.330250. https://doi.org/10.1145/330249.330250. (cited on page 34)

Rees, J. A., 1996. A security kernel based on the lambda-calculus. Technical Report AIM-1564, Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/5944. (cited on page 9)

Reynolds, J. C., 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, ACM '72 (Boston, Massachusetts, USA, 1972), 717–740. Association for Computing Machinery, New York, NY, USA. doi:10.1145/800194.805852. https://doi.org/10.1145/800194.805852. (cited on page 18)

Rideau, S. and Leroy, X., 2010. Validating register allocation and spilling. In *CC 2010: Compiler Construction*, no. 6011 in LNCS, 224–243. Springer. doi:10.1007/978-3-642-11970-5_13. (cited on page 10)

Saha, A., 2009. Disable ZapUnusedHeapArea to reduce GC execution times of debug JVMs. https://bugs.openjdk.org/browse/JDK-2174848. (cited on page 12)

Sandberg Ericsson, A.; Myreen, M. O.; and Åman Pohjola, J., 2017. A verified generational garbage collector for CakeML. In *Interactive Theorem Proving*, 444–461. Springer International Publishing, Cham. (cited on page 1)

Shahriyar, R.; Blackburn, S. M.; and McKinley, K. S., 2014. Fast conservative garbage collection. *SIGPLAN Not.*, 49, 10 (Oct. 2014), 121–139. doi:10.1145/2714064.2660198. (cited on page 5)

Steele, G. L., 1976. Lambda: The ultimate declarative. Technical report, Massachusetts Institute of Technology, USA. (cited on page 8)

Tate, R.; Chen, J.; and Hawblitzel, C., 2010. Inferable object-oriented typed assembly language. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10 (Toronto, Ontario, Canada, 2010), 424–435. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1806596.1806644. https://doi.org/10.1145/1806596.1806644. (cited on pages 9 and 42)

Teo, B., 2024. *Algebraic Data Type Representation in Virgil*. Honors thesis, Carnegie Mellon University. http://ra.adm.cs.cmu.edu/anon/usr/ftp/srthesis/SeniorThesis24-TEO-Bradley.pdf. (cited on page 50)

Tiszka, B., 2021. CVE-2021-4102: Chrome incorrect node elision in Turbofan leads to unexpected WriteBarrier elision. https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2021/CVE-2021-4102.html. (cited on pages 2 and 7)

Traub, O.; Holloway, G.; and Smith, M. D., 1998. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98 (Montreal, Quebec, Canada, 1998), 142–151. Association for Computing Machinery, New York, NY, USA. doi:10.1145/277650.277714. https://doi.org/10.1145/277650.277714. (cited on page 41)

Ungar, D. and Smith, R. B., 1987. Self: The power of simplicity. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '87 (Orlando, Florida, USA, 1987), 227–242. Association for Computing Machinery, New York, NY, USA. doi:10.1145/38765.38828. https://doi.org/10.1145/38765.38828. (cited on page 33)

X3J13, 1994. *Draft proposed American National Standard for Information Systems – Programming Language – Common Lisp*. ANSI. https://franz.com/support/documentation/cl-ansi-standard-draft-w-sidebar.pdf. (cited on page 6)

Yang, X.; Blackburn, S. M.; Frampton, D.; Sartor, J. B.; and McKinley, K. S., 2011. Why nothing matters: the impact of zeroing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11 (Portland, Oregon, USA, 2011), 307–324. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2048066.2048092. https://doi.org/10.1145/2048066.2048092. (cited on page 6)

Zhao, W. and Blackburn, S. M., 2020. Deconstructing the garbage-first collector. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20 (Lausanne, Switzerland, 2020), 15–29. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3381052.3381320. https://doi.org/10.1145/3381052.3381320. (cited on page 46)