Reasoning about Software in the Presence of Transient Faults

Frances Jane Perry

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

Advisor: David P. Walker

September 2008

Abstract

A transient fault occurs when an energetic particle strikes a chip and causes a change in state in the processor. Although there is no permanent damage, the current computation may become corrupt. Transient faults have been shown to be the cause of crashes at major companies, and current technology trends will make future processors more susceptible to them. Researchers have already developed a number of solutions using combinations of software and hardware where the general approach is to duplicate computations and check for consistency between the copies. Unfortunately, generating correct fault-tolerant code is difficult, and there has been little research on proving the correctness of these techniques. Reasoning formally about fault tolerance is challenging, because invariants that hold at compile time, such as standard type safety, may not actually hold at runtime. Previous work on formalizing fault tolerance has been at the high level, including proofs about fault-tolerant algorithms and the definition of a fault-tolerant lambda calculus.

This dissertation presents the first set of techniques for statically proving fault-tolerance properties of actual executable code. To address this challenge, we develop modifications to the general methodology for typed assembly languages. An assembly-level type system incorporates invariants about fault tolerance that are strong enough to prove that all well-typed programs have the desired behaviors. All that is required to guarantee that a specific piece of code is fault-tolerant is to type-check the code at the conclusion of compilation.

More specifically, we introduce a family of three type systems. TAL_{FT} is a core language with simple instructions that guarantees that well-typed programs implementing a hybrid fault-tolerance scheme will always detect a single fault before the fault can affect the observable behavior of the program. ETAL_{FT} extends TAL_{FT} with features necessary to support realistic compilation, including stack activation frames and dynamic memory allocation. The third language, TAL$_{\text{CF}}$, precisely captures the behavior of software solutions for control flow faults and can provably detect any fault that causes incorrect control transfers between basic blocks before control exits that first incorrect block. Although each typed assembly language is designed for a specific hardware context, the type systems and proof methods use similar designs, allowing us to demonstrate general approaches needed for reasoning in the presence of transient faults. For example, to statically reason about values that may be corrupted at runtime, we generalize the "color systems" of previous work into a framework for classifying values with related reliability properties.

As well as being the first to prove that executable code is fault-tolerant, this dissertation makes three additional contributions. By including a language of static expressions within the type system, we can verify low-level fault-tolerant code independently of the compilation process. We show to apply fault-tolerant typed assembly languages to two different fault models: a hybrid system to detect data faults and a software-only system to detect control-flow faults. Finally, we investigate how to generate fault-tolerant typed assembly language for a realistic compiler.

Acknowledgments

This dissertation owes the most to my advisor David Walker. He is an extremely knowledgeable and patient teacher who taught me to be clear and rigorous in my work and confident in myself. I clearly remember the day I first openly disagreed with him, and he responded, “That’s it! Now you are becoming a researcher.”

The members of my thesis committee were extremely helpful in improving the quality of this dissertation, as well as my job talk. I am very grateful for their time and feedback. Andrew Appel helped me develop clear explanations and pull out the high-level ideas. David Tarditi inspired me to work that bit harder to become a better writer. Margaret Martonosi and David August’s different backgrounds helped me target my work to the broader community.

Additionally, this dissertation benefited from the support of many others. George Reis and Neil Vachharajani were a great help with the architecture side of things. I am grateful for the wonderful administrative help and friendship of the staff at Princeton, especially Melissa Lawson and Donna O’Leary. This work was funded by the National Science Foundation award CNS-0627650 and a Microsoft fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are mine and do not necessarily reflect the views of the NSF or Microsoft.

During internships at Microsoft, I was lucky to work with a number of amazing people. Manuvir Das, Jason Yang, and the PAG group taught me that sometimes it’s okay to sacrifice soundness, as long as you can find real bugs in real code. Juan Chen and Chris Hawblitzel showed me the joy of straddling the boundary between theory and practice to design elegant, yet practical, systems.

My fellow students were always there to advise me, commiserate with me, and support me. (And, of course, procrastinate with me...) Limin Jia was both a fantastic colleague and a close friend, and our afternoon tea times were always a bright spot in my day. Ananya Misra's sarcastic emails and dry wit kept me laughing. Siggi Cherem and Sharon Betz kept me sane through the home stretch. In addition, my survival of grad school is due in part to Melissa Carroll, Dan Dantas, Haakon Larson, Guilherme Ottoni, Shirley Gaw, Henning Rohde, Zach Anderson, Akash Lal, Dan Wang, Jay Liggatti, the pizza crowd, my officemates, and many others.

My family's continued love and support means everything to me. My father Jeff, though he never did teach me how to throw a ball, encouraged me through my disgruntled phases and kept me focused on the final goal; my mother Liz reminded me to take care of myself and was always willing to proofread my papers for punctuation, even if the words themselves were gobbledygook. As we grew up, my sister Julia and I realized we loved each other after all, and our differences turned out to be quite complementary – I fix her computer and she picks out my interview suits. Erik, Isabelle, Becky, Ben, and Hunter have become my second family. They were always there, kept Dan and me well-fed, and rushed to our aid during the occasional emergency. I will greatly miss having them so close by. My "Aunt" Marilyn is always proud of me and reminds me to be proud of myself.

And finally, there is Dan. Although his continual jokes about "Frances the 12th year grad student" did get a bit old, he is, and always will be, my best friend. I am thrilled to have him by my side as we enter this next phase in our lives – moving cross country, becoming homeowners, making new friends, and who knows what else!

Contents

List of Figures

Chapter 1

Introduction and Background

1.1 Transient Faults

A *transient fault* occurs when an energetic particle strikes a transistor or wire in a processor and causes a change in state. These particles do not permanently damage the hardware, but they may corrupt the computation that is currently executing by depositing a charge that alters stored values and signals.

1.1.1 Issues Caused by Transient Faults

As one might expect, transient faults can cause a host of issues. In particular, as these anecdotes show, systems can crash due to transient faults.

- In 2000, Sun Microsystems acknowledged that transient faults interfered with cache memories and caused crashes in server systems at major customer sites, including AOL, eBay, and dozens of others [6].

- Cypress Semiconductor has stated “The wake-up call came in the end of 2001 with a major customer reporting havoc at a large telephone company. Technically it was found that a single soft fail...was causing an interleaved system farm to crash” [76].

- At Los Alamos in 2003, the ASC Q supercomputer crashed regularly due to soft errors [37].

Of course, and perhaps even worse, there is always the possibility that the result of a program is incorrect. Such an outcome is referred to as *silent data corruption*. Although these issues with crashing and correctness are bad enough, transient faults can also cause some more surprising problems.

Researchers [24] have shown how an attacker can exploit transient faults to take over a Java Virtual Machine. In a Java Virtual Machine, the virtual machine runs in the same address space as the untrusted program, relying on type safety to keep the untrusted program from touching its address space. Essentially, the attacker can craft a program that waits until a fault results in a pointer with a runtime type that does not match its static type and then uses this mismatch to execute arbitrary code.

Cryptographic protocols can be broken using transient faults [7, 8, 19, 56, 4]. For example, certain implementations of RSA based on the Chinese Remainder Theorem are vulnerable to a single fault [9]. RSA relies on the inability to factor a large number N into two prime numbers p and q. A signature computed from a message m and a private key d has the form $a * s_p + b * s_q$ where a and b are precomputed values and s_n is a function of m, d and n. If an attacker obtains two signatures (one correct, one faulty) of the same message, he can take the difference of the two signatures. One of the two terms cancels

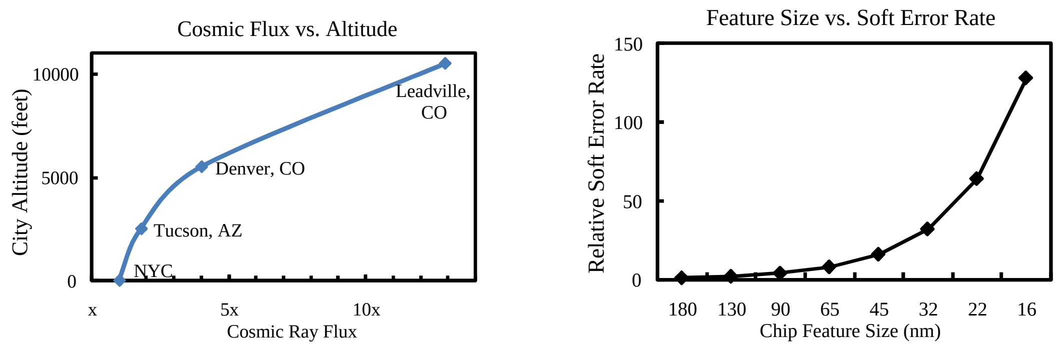


Figure 1.1: Effects of Altitude [76] and Feature Size [11] on Transient Faults

out, leaving the difference of the other. Then by factoring out the precomputed value and calculating a greatest common divisor, the attacker can determine p and q.

1.1.2 Transient Fault Trends

It is difficult to get precise numbers on transient fault rates, but as a benchmark, in 2004, a typical laptop with 1GB DRAM had approximately 1 soft fail per year [76].

The particles that cause transient faults are mostly high-energy neutrons in cosmic-ray radiation [76], and so fault rates increase with altitude. The graph in Figure 1.1 shows that the amount of cosmic ray flux in Denver, Colorado is about four times that of New York City. The data was gathered over three years beginning in 1986, with New York City averaging one failure every 45 days, and Leadville averaging one failure every two days.

These results are for transient faults that affect memory. Transient faults can also affect the latches in the processor itself, and these faults are much harder to detect. Unfortunately, trends in processor manufacturing are causing these fault rates to increase over time. Faster clock rates, increasing transistor density, decreasing voltages and smaller feature sizes are resulting in processors that are more susceptible to transient faults [5, 46, 63]. For example, decreasing voltages reduce the critical voltage of each transistor, making it more likely that adding the extra charge from a fault will change a value. On the other hand, decreasing feature size makes it less likely that a fault will affect a given transistor. As a result, the fault rate per bit is expected to remain roughly constant for the next few generations [26, 31]. However, the number of transistors per chip is increasing, and so the overall result is an increase in the fault rate per processor. Figure 1.1 shows how the transient fault rate increases as the feature size decreases in a processor. Because of the combination of these factors, fault rates are increasing approximately 8% per processor generation [28].

So although the average computer user may not be aware of transient faults today, transient faults are likely to become a significant problem in the near future.

1.2 Existing Transient Fault Solutions

The computer architecture and compiler communities are well aware of the issues surrounding transient faults, and dealing with transient faults has been an active area of research for many years. At the high level, all the solutions work by adding redundancy. Redundancy can be added in time (by computing a result and then computing it again), in space (by storing values in two locations), or in information (using techniques such as checksums or error-correcting codes). The many solutions can be divided into two main categories: hardware-based solutions that duplicate hardware structures in the processor or exploit multiple cores, and software-based solutions that insert additional instructions to verify that values are not corrupted.

Chapter 5 goes into more detail on the existing solutions, but as one would expect, there are trade-offs to each type. It is standard to evaluate a system in terms of how it performs along three axes: cost, performance, and power. Transient faults add a fourth axis to be considered – reliability. Hardware solutions are generally more efficient in terms of performance, but more expensive in terms of power and cost. In addition, once a hardware solution is deployed, it cannot be changed. Software solutions, on the other hand, are more flexible. Because they are controlled by a compiler, they can be deployed when, where, and to the degree necessary. Software solutions have no additional hardware cost, but they often have a noticeable performance overhead. The reliability added by each solution varies, but it is safe to say that although hardware solutions can achieve very high rates of fault detection and recovery, software-solutions are fundamentally limited without hardware support. More recently, researchers have begun proposing hybrid solutions that use small amounts of additional hardware that is controlled by the software with the hopes that such solutions can combine the best of both worlds.

1.2.1 An Example Solution: SWIFT

As an example, let us look more closely at one specific solution. SWIFT (SoftWare Implemented Fault Tolerance) [60] is a software-based solution to transient faults. During compilation, the SWIFT compiler duplicates the original computation, interleaves the two copies in some manner, and inserts comparisons before storing values to memory to ensure that the two versions of the computation agree. Because the two computations are completely independent, a single transient fault will likely result in a difference between the two computations. If this corruption spreads to a value that will be written to memory, the inserted comparison will fail, and control is transferred to a designated location containing error handling or recovery code. If a transient fault corrupts a value but does not result in a noticeable difference in the values written to memory, then it does not need to be detected.

The researchers who developed SWIFT evaluated it by randomly injecting faults and looking at the resulting performance and detection rate. At first, the detection rates were not nearly as good as they were expecting. After more investigation, they discovered that the compiler, having added the redundant computation, had been doing what compilers do best – optimizing the code to remove redundancy. Optimizations such as common subexpression elimination were resulting in a single computation, with additional code before store instructions that duplicated the value to be stored and then compared the value and its duplicate. Clearly, faults affecting the computation itself were not being detected. The solution was to turn off or reorder the optimizations that the compiler performed until the results were more in line with what was expected.[1]

This leaves an interesting question. How do we know that the code that SWIFT generates now is as reliable as intended? Any realistic optimizing compiler is extremely complicated, and it is very difficult to understand how all the phases may interact. The high-level intuition used in SWIFT is simple to understand, but how do we ensure that the compiled code correctly implements this intuition?

1.2.2 Do Existing Solutions Work?

Most other transient fault solutions are evaluated in similar ways: a promising idea is presented, a system is implemented, and experimental results show an increase in fault detection. None of these solutions provide rigorous proofs of their correctness. In fact, many do not even precisely define which faults they can handle.

[1]Personal communication with G.A. Reis, October 2006.

The goal of this thesis is provide an approach to formally reason about the behavior of transient fault solutions, focusing on those approaches that make use of software. The initial task is to determine the correct level of abstraction and best proof methodology for reasoning about these situations. As the example with SWIFT showed, it is not enough to simply reason about the *algorithm*; we need to reason about the *implementation* as well. Transient faults affect the hardware, so the reasoning technique needs to model primitive instructions, memory, registers, and any other relevant hardware structures. The addition of software redundancy occurs as a compiler phase, just like optimizations and register allocation. Because compiler phases may have unexpected interactions, we want to reason about the final assembly code generated by the compiler.

1.3 Proof-Carrying Code

Proof-Carrying Code [43, 42] is a technique for verifying properties of untrusted, low-level code. In a proof-carrying code system, the compiler is responsible for generating two things: the low-level code and a safety proof that the low-level code obeys a predefined set of properties. Then anyone who wishes to run the code can first verify the proof to ensure that the code will not behave unexpectedly. Traditionally, these safety proofs have included guarantees of memory safety and type safety.

1.3.1 Typed Assembly Languages

One standard way to represent these safety proofs is using typed assembly languages [39]. To generate a typed assembly language, the compiler begins with a type-safe source program. Instead of type checking this program and then discarding the typing information, the compiler preserves types through every level of intermediate representation. Each

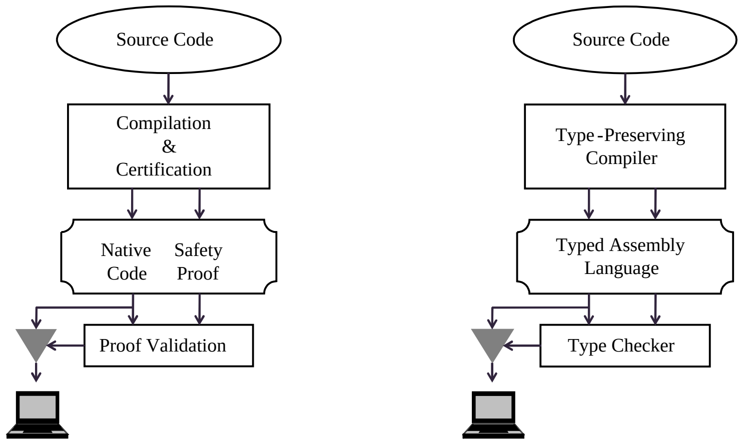


Figure 1.2: Proof-Carrying Code [42] and Typed Assembly Language [39].

intermediate representation has its own associated type system, and as the representations become closer to the machine level, the type systems become more complicated in order to maintain information about the program.

Once the compiler generates the final assembly code, type checking the code guarantees that the code obeys the properties encapsulated by the type system. Nothing is required to be known about the correctness of the compiler. If the generated code obeys the safety proof, then that proves that it behaves according to the properties captured by the type system.

1.3.2 Using Typed Assembly Languages

Typed assembly languages are an active area of research, and Chapter 5 gives some more background in this area. For now, we will explain the general methodology for using a typed assembly language.

1. **Model machine execution.** The goal of typed assembly languages is to reason about the execution behavior of a program, so the first step is to develop a model

of how programs execute. One common way to do this is by defining a small-step operational semantics. Imagine that we could pause execution and look at the current state of an abstract processor. We could inspect the values in each register and in memory, see which instruction will be executed next, and so on. The small-step operational semantics takes a snapshot of the machine like this and shows how the snapshot is modified by executing one single instruction. By sequencing many of these small steps together, we can reason about the execution of an entire program.

2. **Specify the desired execution behavior.** Given the model of execution, we can define which executions have the behavior that we desire.

3. **Design the type system.** The type system encapsulates the desired invariants that the program should maintain. Whereas high-level type systems track the types of program variables, assembly-level type systems track the types of the values in the machine state, including the register file and memory.

4. **Prove Soundness.** Once we have these three definitions, the next step is to prove that the type system is sound with respect to the machine model. In other words, any well-typed program is guaranteed to execute as desired on the machine model.

5. S**how that the type system is expressive.** Finally, the last step is to show that the typed assembly language is expressive. We want to rule out the possibility that we have designed an overly restrictive type system. If there are no interesting programs that type-check, then it means nothing to prove properties about the behavior of all well-typed programs. We can show that this is not the case by taking a high-level language and showing how any well-typed source program can be compiled into a well-typed assembly program.

1.4 Thesis Scope

This thesis investigates the use of typed assembly languages as a method for verifying software-based transient fault solutions.

Typed assembly languages are especially well-suited for this because of the sheer complexity of reasoning about transient faults. Just as the many possible interleavings of threads make it difficult to reason about concurrent programs, the many possible transient faults make it difficult to reason about fault tolerance. For this reason, testing is not sufficient. To truly test that the code a compiler generates can detect all possible faults, it would be necessary to test all combinations of features in the compiler in conjunction with all possible faults. The drastic explosion in the number of test cases makes this infeasible. By using a typed assembly language, we can statically guarantee that a program will achieve perfect fault coverage relative to the fault model.

In particular, this thesis defines two families of typed assembly languages, each designed for a different type of solution. TAL_{FT} and its close cousin ETAL_{FT} are designed for a hybrid solution that uses a mix of hardware and software to detect faults. TAL_{CF} is a separate language that takes the first steps towards reasoning about software-only solutions to control-flow faults. In order to use these typed assembly languages for verifying the behavior of programs in the presence of transient faults, we make some slight modifications to the usual methodology for typed assembly languages.

1.4.1 Modeling Transient Faults

The machine model needs to represent hardware that may be affected by transient faults.

We will assume that memory is protected by error-correcting codes (ECC) [25]. Unlike simple parity which only detects single-bit errors, ECC is capable of both correcting single-bit errors and detecting multi-bit errors. Each piece of data has a corresponding ECC code. The size of the code depends on the size of the data to be protected. Protecting 32 bits of data requires a 7 bit code, and protecting 64 bits of data requires an 8 bit code. When data is written to memory, the memory controller encodes the data using a device-specific algorithm and updates the corresponding ECC code with the result. When data is read from memory (or during periodic consistency checks), the read value is encoded and the result is compared to the stored code. If the two codes do not match, the memory controller corrects a single-bit error or reports a multi-bit error to the operating system. ECC has been shown to be extremely effective for protecting memory [12] and has been used since the 1950s [29, 64].

However, applying ECC to the register file is too costly in terms of both performance [68] and power [55] because of the frequency of accesses. We assume that the register file contents are vulnerable to transient faults and specifically model transient faults to registers in between the execution of two instructions. As the model machine executes its small-step semantics, it may nondeterministically insert an additional step rule that corrupts a random register value.

Transient faults may actually affect any part of the processor datapath or control. Many "inter-instruction" faults can be modeled as a correct instruction execution followed by a fault to the destination register. For example, a transient fault that strikes the ALU during the execution of an add instruction can be modeled as a correct add and then a register fault to the destination register. A fault that causes a multiply to be performed instead of add can be handled in the same way. However, this is not the case for all faults. The model does not account for faults that affect instructions with side effects beyond the register file, instruction decoding, the virtual memory page table, memory buses, and so on. (Section 2.1.2 gives a concrete example of a fault is not modeled.) This is not to say that solutions such as SWIFT would fail to detect all these unmodeled faults, in fact many such faults would likely result in a difference between the two computations and be detected, but only that all such faults are not guaranteed to be handled in such a system.

By specifying the machine with precise operational rules, we clearly identify which transient faults are under consideration and what hardware behavior we rely on. If a fault can modify the assumed hardware behavior and cannot be modeled as a register fault, then this identifies a vulnerability that needs to be addressed with additional hardware or software techniques.

As is standard in the literature [57, 61], we will work under the assumption that only one fault occurs during the execution of a program. This does not mean that multiple faults would not be caught, only that with multiple faults there is a minuscule chance that they may occur in a way that tricks the fault detection mechanisms. For example, the first fault may corrupt a value, while the second affects the checking code that would have otherwise detected the error. Multiple faults might also cause both computations to calculate the same incorrect result.

1.4.2 Defining Fault Tolerance

The goal is to prove that programs are fault-tolerant, but first we need to formally define what this means. Abstractly, a program is fault-tolerant if no fault can change the observable behavior of a program. The definition of “observable” may change, but one example would be to assume the system operates in the presence of a memory-mapped IO device. In this case, the program is fault-tolerant if a fault does not cause a change the sequence of values written to memory up until the point where a fault is detected.

We will focus only on formalizing *fault detection*, and not specify *fault recovery*. There are a number of known recovery techniques applicable to this problem domain.

The simplest way to recover from a transient error is to restart the computation. For example, Google's MapReduce implementation distributes work over thousands of machines. When a machine fails, its task is reassigned to another machine [18]. By running three copies of a computation, it is possible to use majority voting to recover from a single error [59, 73], though this requires consistently paying a high performance overhead to recover from a rare event. Checkpointing and rollback recovery are commonly used to recover from faults in databases and parallel systems [34, 67]. In our current setting, we already assume that memory is protected, so this can easily be used as stable storage for the checkpointing. Because transient faults are rare, it is not unreasonable to pay a high performance cost for rolling back to the most recent checkpoint.

1.4.3 Invariants of Fault Tolerance

The type systems are designed specifically to capture invariants about the fault-tolerance solution. For example, when using two computations to detect data faults, there are three main invariants that must be maintained. The copies of the computation need to be independent, so that a fault to one computation cannot affect the other computation. The computations need to be redundant, so that in the absence of faults they calculate equal values. In addition, any action that may affect the observable behavior of the program must be guarded by a comparison of the two computations.

1.4.4 Proving Fault Tolerance

We formalize the behavior of well-typed programs as a mathematical theorem that relates faulty and non-faulty executions of a program. First, we define a simulation relationship between a program snapshot and the equivalent faulty snapshot that essentially means that the uncorrupted computations are identical, although the corrupted computations may differ. Then we show that continuing to execute the two snapshots will always maintain the simulation relationship up until the point where the fault is detected. This is used to guarantee that a well-typed program behaves equivalently whether or not it is affected by a single fault. In addition, we show that well-typed code only claims to have detected a fault when a fault has actually occurred.

1.4.5 Compiling for Fault Tolerance

Finally, we show how to take high-level (unduplicated) programs and translate them into well-typed fault-tolerant assembly language programs. We include realistic low-level features such as memory management and discuss the effect optimizations may have on our typed assembly language.

1.5 Thesis Organization

The remainder of this dissertation is organized as follows. Chapters 2 and 3 reason about a hybrid transient fault solution modeled after the CRAFT system [61]. Chapter 2 begins the process with a core assembly language called TAL_{FT} and walks through the first four stages of using typed assembly languages. In order to show expressiveness, Chapter 3 defines ETAL_{FT}, an extension to TAL_{FT}, and discusses how to use ETAL_{FT} as the target language of an optimizing compiler. Chapter 4 takes the first steps toward reasoning about software-only solutions to control-flow faults using a language called TAL_{CF}. Chapter 5 goes into more detail on related work and ends with some final remarks.

Chapter 2

TAL$_{\mathrm{FT}}$: Fault-tolerant Typed Assembly Language

This chapter presents TAL$_{\mathrm{FT}}$, a typed assembly language designed for reasoning formally about a hybrid transient fault solution. Our system is abstractly modeled after CRAFT [61], a solution similar to SWIFT. Being a software-only solution, SWIFT has some fundamental limitations. For example, before storing a value to memory, SWIFT performs a comparison between the two computations. However, a fault that occurs after this comparison, but before the store instruction, will not be caught. CRAFT introduces a little bit of extra hardware to help close this window of vulnerability.

The rest of this chapter presents the details of the abstracted hybrid fault-tolerance technique. Section 2.1 presents the syntax and operational semantics of the machine model. It is a RISC-based architecture with special instructions to facilitate reliable communication with memory and to detect control-flow faults. Section 2.2 presents the

Material in this chapter is joint work with Lester Mackey, George A. Reis, Jay Ligatti, David I. August, and David Walker and has been previously published as [51] and [52].

key principles and formal definitions for the type system. Although the typing rules are specific to the particular setting, the underlying principles are more general; we believe many of these principles will apply to reasoning about related fault-tolerant systems. The combination of a TAL-like type-theory with concepts from classical Hoare Logics is a particularly general and important technical contribution. Section 2.3 describes the key theorems we have proven including Progress, Preservation, "No False Positives," and Fault Tolerance. Section 2.4 provides empirical evidence that our new hybrid solution to fault tolerance is feasible for many applications by measuring performance results on simulated hardware. Finally, Section 2.5 summarizes the contributions of TAL$_{\text{FT}}$.

2.1 The Fault-Tolerant Hardware

The first task is to develop the machine model so that we can reason about the execution of a program. The model hardware is based on a simple RISC architecture, extended with features to support detection of control-flow faults and safe interaction with memory-mapped output devices. Correct use of these features makes it possible to detect all faults that might change a program's observable behavior. Most practical systems also need a fault recovery mechanism of some kind. However, because recovery is largely a secondary issue to detection, we omit recovery for now.

The general strategy is to maintain two redundant and independent threads of computation, which we name the *green* (G) computation and the *blue* (B) computation. The green computation generally leads slightly, and the blue computation generally trails, though there is a fair amount of flexibility in how the instructions in each computation may be interleaved. Prior to writing data out to a memory-mapped output device, the results of the two computations are checked for equivalence. If the results are not equivalent, the machine will signal that a fault has been detected. The arguments to any control-flow transfer must also be checked for faults.

The execution of assembly programs is specified using a small-step operational semantics that maps *machine states* (Σ) to other machine states. These machine states are made up of a number of components. The first component is the machine's *register bank* R, which is a total function that maps register names to the values contained therein. The metavariable a ranges over all registers, and metavariable r ranges only over general-purpose registers (r_1, r_2, ...). In addition to general-purpose registers, there are two program counter registers (pc_G and pc_B), which contain the same value unless there has been the fault. There is one additional special register, the *green destination register*, gd. Its role in control-flow checking will be explained later.

To facilitate proofs of certain theorems, the value in each register is tagged with the color (either green or blue) of the computation to which it belongs. However, these tags have no effect on the run-time behavior of programs. Section 3.2.4 discusses how color tags can be removed by doing more work in the proofs. In contrast, the tags on instruction opcodes, to be introduced momentarily, *do* have an effect on evaluation.

In addition to a register bank, the machine state includes a *code memory* C, which we model as a function mapping integer addresses n to instructions. Address 0 is not considered a valid code address. The machine also has a *value memory* M, which maps addresses to integer values. In between the value memory and the processor is a special *store queue*, Q, which is used to detect faults before data is written to a memory-mapped output device. The store queue is a queue of address-value pairs. We will discuss the role of the queue in greater detail later.

Overall, an abstract machine state (Σ) may have the form *fault*, indicating the hardware has detected a transient fault, or the ordinary state (R, C, M, Q, ir), where the first

colors	c	$::=$	$G \mid B$
integers	n	$::=$	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$
colored values	v	$::=$	$c\ n$
general registers	r	$::=$	r_n
all regs	a	$::=$	$r \mid gd \mid pc_c$
register file	R	$::=$	$\cdot \mid R, a \rightarrow v$
code memory	C	$::=$	$\cdot \mid C, n \rightarrow i$
value memory	M	$::=$	$\cdot \mid M, n \rightarrow n$
store queue	Q	$::=$	$\overline{(n,n)}$
ALU ops	op	$::=$	$add \mid sub \mid mul$
instructions	i	$::=$	$op\ r_d, r_s, r_t \mid op\ r_d, r_s, v$
		$\mid$	$ld_c\ r_d, r_s \mid st_c\ r_d, r_s \mid mov\ \ r_d, v$
		$\mid$	$bz_c\ \ r_z, r_d \mid jmp_c\ \ r_d$
inst register	ir	$::=$	$i \mid \cdot$
state	Σ	$::=$	$(R, C, M, Q, ir) \mid fault$

Figure 2.1: Syntax of instructions and machine states.

four components are as discussed above, and *ir* is either an instruction *i* to be executed, or "·" indicating the next instruction should be fetched from code memory. Figure 2.1 summarizes the syntax of machine states. We use overbar notation to indicate a sequence of objects.

2.1.1 The Fault Model

The operational semantics is designed both to model proper execution of machine instructions and to make explicit, precise, and transparent all of the assumptions about when and where faults may occur. The central operational judgment has the form $\Sigma_1 \longrightarrow^s_k \Sigma_2$, which expresses a single step transition from state Σ_1 to state Σ_2 while incurring k faults and writing data s to a memory-mapped output device. We will work under the standard assumption of a single upset event [57, 61] and hence k will always be either 0 or 1. The data s is a (possibly empty) sequence of address-value pairs. While the operational semantics models the internal workings of the machine, the only externally observable behavior of the machine is the sequence of writes s to the output device or the signaling of a hardware-detected fault. If faults cause the processor to have drastically different internal behavior, but the externally observable sequence s is unchanged, we consider the program to have executed successfully.

Different fault-tolerance techniques protect different components of machines. In the literature, the protected areas are usually inside the *Sphere of Replication* (SoR) [57]. In our case, we target faults that may occur in data manipulated within the processor. We assume that both code memory C and value memory M are fully protected. This is often the case since error-correcting codes can very efficiently protect memory. To make these assumptions explicit, the following three operational rules specify exactly how faults may occur within our system.

$$\frac{R(a) = c\ n}{(R,C,M,Q,ir) \longrightarrow_1 (R[a \mapsto c\ n'],C,M,Q,ir)}\ (\textit{reg-zap})$$

$$\frac{\begin{array}{c} Q_1 = \overline{(n_1,n_1')},(m_1,m'),\overline{(n_2,n_2')} \\ Q_2 = \overline{(n_1,n_1')},(m_2,m'),\overline{(n_2,n_2')} \end{array}}{(R,C,M,Q_1,ir) \longrightarrow_1 (R,C,M,Q_2,ir)}\ (\textit{Q-zap1})$$

$$\frac{\begin{array}{c} Q_1 = \overline{(n_1,n_1')},(m,m_1'),\overline{(n_2,n_2')} \\ Q_2 = \overline{(n_1,n_1')},(m,m_2'),\overline{(n_2,n_2')} \end{array}}{(R,C,M,Q_1,ir) \longrightarrow_1 (R,C,M,Q_2,ir)}\ (\textit{Q-zap2})$$

Rule *reg-zap* nondeterministically introduces a fault into any register by replacing the value in that register with some other arbitrary value. There are no restrictions on how the underlying value might be changed. For instance, code pointers can be changed to arbitrary integer values; references may no longer be in bounds. However, the color tag is preserved to facilitate fault-tolerance proofs. Because the color tag is fictional (has no effect on run-time behavior), this poses no limitation on the fault model. Rules Q_1-*zap* and Q_2-*zap* alter the contents of the store queue in similar ways.

Formally, these are the only faults that can occur. However, notice that since the program counters and targets of indirect jumps are susceptible to the *reg-zap* rule, we effectively capture many forms of "control-flow faults" studied previously. Notice also that we do not explicitly consider faults that occur *during* execution of an instruction. However, many such faults may easily be shown to be equivalent to correct execution of an instruction composed with a fault either immediately before or afterward. For example, consider a simple register-to-register add instruction. Any fault within the adder hardware during execution of the add is equivalent to a correct add followed by a fault in the destination register. On the other hand, as we will see later, a fault during the st_B instruction cannot be modeled in our system.

An important benefit of our formal model is that there is actually a precise, concrete specification to analyze. Moreover, if a researcher wants to reason about the consequences of some fault that lives outside the formal model, this may be done by adding a new operational rule to the system and studying its semantic effect.

2.1.2 Instruction Semantics

The syntax of machine instructions was presented along with the rest of the components of our abstract machine in Figure 2.1. The semantics is described formally by the inference rules in Figures 2.2, 2.3, and 2.4, and explained informally below. The formal rules use several notational conventions. For instance, if R is a register file then $R(a)$ is the contents of register a and $R[a \mapsto v]$ is the updated register file with register a mapped to v. R++ is the register file that results from incrementing both pc_G and pc_B by 1. If $R(a)$ is the colored value $c\ n$, we write $R_{val}(a)$ to denote n and $R_{col}(a)$ to denote c. The function

Instruction Fetch:

$$\frac{R_{val}(pc_G) = R_{val}(pc_B) \qquad R_{val}(pc_G) \in \ Dom(C)}{(R,C,M,Q,\cdot) \longrightarrow_0 (R,C,M,Q,C(R_{val}(pc_G)))} \ (fetch)$$

$$\frac{R_{val}(pc_G) \neq R_{val}(pc_B)}{(R,C,M,Q,\cdot) \longrightarrow_0 \ fault} \ (fetch\text{-}fail)$$

Basic Instructions:

$$\frac{R' = R\text{++}[r_d \mapsto R_{col}(r_t) \ (R_{val}(r_s) \ op \ R_{val}(r_t))]}{(R,C,M,Q,op \ \ r_d,r_s,r_t) \longrightarrow_0 (R',C,M,Q,\cdot)} \ (op2r)$$

$$\frac{R' = R\text{++}[r_d \mapsto c \ (R_{val}(r_s) \ op \ n)]}{(R,C,M,Q,op \ \ r_d,r_s,c \ n) \longrightarrow_0 (R',C,M,Q,\cdot)} \ (op1r)$$

$$\frac{}{(R,C,M,Q,mov \ \ r_d,v) \longrightarrow_0 (R\text{++}[r_d \mapsto v],C,M,Q,\cdot)} \ (mov)$$

Figure 2.2: Operational rules for basic instructions.

find(Q,n) produces the first pair (n,n') that appears in Q, or $()$ if no pair (n,n') appears in Q.

Instruction Fetch. The machine operates by fetching an instruction from code memory and executing that instruction. The corresponding operational semantics rules are shown in Figure 2.2. When there is no current instruction to execute (i.e. $ir = \cdot$), the *fetch* rule should fire. This rule tests for equality of the two program counters to check for faults and loads the appropriate instruction from code memory. If pc_G and pc_B are the same but $R_{val}(pc_G)$ is not a valid address in code memory, execution "gets stuck" (no rule fires). Well-typed programs never get stuck, even when a single fault occurs. On the other hand, a fault can render the two program counters inequivalent. In this case, rule *fetch-fail* fires and causes a transition to the fault state. Abstractly, this transition represents hardware detection of a transient fault. Controlled program termination or perhaps recovery may follow. The fault model does not allow for the instruction itself to be corrupted.

Basic Instructions. The arithmetic and move instructions (rules *op2r*, *op1r*, and *mov*) shown in Figure 2.2 are completely standard. The first arithmetic operation $op\ \ r_d, r_s, r_t$ performs *op* on the values in r_s and r_t, storing the result in r_d. The second arithmetic operation uses a constant operand v in addition to r_s and r_d. All constants are annotated with the color of the computation they belong to. Likewise, the *mov* instruction loads an annotated constant into a register.

Memory Instructions. Transient faults are problematic only when they change the results of computations and those results are *observed* by an external user. In this model, the only way a result can be observed is for a program to write it to memory, where a memory-mapped output device may read and process it.

Without special hardware it appears *impossible* to guarantee that storage operations guard access to memory properly. No matter what sophisticated software checking is performed just before a conventional store instruction, it will be undone if a fault strikes between the check and execution of the store instruction.

To address this vulnerability, the machine possesses a modified store buffer (the queue Q), which is similar to the store buffer used in previous hardware [57] and hybrid [61] fault tolerant systems. In addition, there are two special storage instructions, each tagged with a color. The green store instruction $st_G\ r_d, r_s$ places the address-value pair $(R_{val}(r_d), R_{val}(r_s))$ on the front of the queue (rule *st*$_G$*-queue* in Figure 2.3). The blue store instruction $st_B\ r_d, r_s$ retrieves the pair (n_l, n_l') on the back of the queue, checks that it equals $(R_{val}(r_d), R_{val}(r_s))$, and then stores it in memory (rule *st*$_B$*-mem*). If the pairs are different, the hardware signals a fault (rule *st*$_B$*-mem-fail*). Because green stores must

$$\boxed{\Sigma \longrightarrow_k^s \Sigma'}$$

$$\frac{}{(R,C,M,Q,st_G\ \ r_d,r_s) \longrightarrow_0 (R{+}{+},C,M,((R_{val}(r_d),R_{val}(r_s)),Q),\cdot)}\ (st_G\text{-}queue)$$

$$\frac{R_{val}(r_d) = n_l \qquad R_{val}(r_s) = n'_l}{(R,C,M,(\overline{(n,n')},(n_l,n'_l)),st_B\ r_d,r_s) \longrightarrow_0^{(n_l,n'_l)} (R{+}{+},C,M[n_l \mapsto n'_l],\overline{(n,n')},\cdot)}\ (st_B\text{-}mem)$$

$$\frac{R_{val}(r_d) \neq\ n_l \text{ or } R_{val}(r_s) \neq\ n'_l}{(R,C,M,(\overline{(n,n')},(n_l,n'_l)),st_B\ \ r_d,r_s) \longrightarrow_0\ fault}\ (st_B\text{-}mem\text{-}fail)$$

$$\frac{find(Q,R_{val}(r_s)) = (R_{val}(r_s),n)}{(R,C,M,Q,ld_G\ \ r_d,r_s) \longrightarrow_0 (R{+}{+}[r_d \mapsto G\ n],C,M,Q,\cdot)}\ (ld_G\text{-}queue)$$

$$\frac{find(Q,R_{val}(r_s)) = () \qquad R_{val}(r_s) \in\ Dom(M)}{(R,C,M,Q,ld_G\ \ r_d,r_s) \longrightarrow_0 (R{+}{+}[r_d \mapsto G\ M(R_{val}(r_s))],C,M,Q,\cdot)}\ (ld_G\text{-}mem)$$

$$\frac{find(Q,R_{val}(r_s)) = () \qquad R_{val}(r_s) \notin\ Dom(M)}{(R,C,M,Q,ld_G\ \ r_d,r_s) \longrightarrow_0\ fault}\ (ld_G\text{-}fail)$$

$$\frac{R_{val}(r_s) \in\ Dom(M)}{(R,C,M,Q,ld_B\ \ r_d,r_s) \longrightarrow_0 (R{+}{+}[r_d \mapsto B\ M(R_{val}(r_s))],C,M,Q,\cdot)}\ (ld_B\text{-}mem)$$

$$\frac{R_{val}(r_s) \notin\ Dom(M)}{(R,C,M,Q,ld_B\ \ r_d,r_s) \longrightarrow_0\ fault}\ (ld_B\text{-}fail)$$

$$\frac{find(Q,R_{val}(r_s)) = () \qquad R_{val}(r_s) \notin\ Dom(M)}{(R,C,M,Q,ld_G\ \ r_d,r_s) \longrightarrow_0 (R{+}{+}[r_d \mapsto G\ n],C,M,Q,\cdot)}\ (ld_G\text{-}rand)$$

$$\frac{R_{val}(r_s) \notin\ Dom(M)}{(R,C,M,Q,ld_B\ \ r_d,r_s) \longrightarrow_0 (R{+}{+}[r_d \mapsto B\ n],C,M,Q,\cdot)}\ (ld_B\text{-}rand)$$

Figure 2.3: Selected operational rules for memory instructions.

always come before blue stores, instruction scheduling is somewhat constrained. As we will show later in Section 2.4, we have evaluated the performance with and without this scheduling constraint and show that its performance impact is negligible.

As an example, consider the following straight-line sequence:

```
1 mov $r_1$, G 5
2 mov $r_2$, G 256
3 st$_G$ $r_2$, $r_1$
4 mov $r_3$, B 5
5 mov $r_4$, B 256
6 st$_B$ $r_4$, $r_3$
```

These six instruction have the effect of storing 5 into memory address 256. Moreover, a fault at any point in execution, to either blue or green values or addresses, will be caught by the hardware when the blue store (instruction 6) compares its operands to those in the queue. In addition, our instruction set gives a compiler the freedom to allocate registers however it chooses (*e.g.*, reusing registers 1 and 2 in instructions 4-6 instead of registers 3 and 4) and to change the instruction schedule in various ways (*e.g.*, moving instruction 3 to a position between instructions 5 and 6).

Interestingly, however, not all conventional optimizations are sound. This is why type checking generated code can be so helpful in detecting compiler errors. For example, common subexpression elimination might result in the following code:

```
1 mov $r_1$, G 5
2 mov $r_2$, G 256
3 st$_G$ $r_2$, $r_1$
4 st$_B$ $r_2$, $r_1$
```

In this case, a fault in r_1 after instruction 1, or a fault in r_2 after instruction 2 will cause both instructions 3 and 4 to manipulate the same, but incorrect, address-value pair. The result would be to store an incorrect value at the correct location or a correct value at an incorrect location. Fortunately, the TAL$_{\text{FT}}$ type system catches reliability errors like this one.

As mentioned in Section 2.1.1, many "intra-instruction" faults can be modeled by modifying the register file before or after the instruction. However, this is not the case for a fault that occurs during the execution of the st_B-*mem* rule in between the comparisons and the store. The hardware designer must implement structures that detect or mask any faults that occur here. If the hardware designer cannot meet the specification given by the operational semantics, he acknowledges there may be a vulnerability.

The load instructions also come in pairs: ld_B and ld_G. As Figure 2.3 shows, the only difference in their semantics is that ld_G checks for a pending store in the queue before loading its value from memory, whereas ld_B goes directly to memory, ignoring the queue. This wrinkle increases the freedom in instruction scheduling by allowing the green computation to load a value it may have recently stored before the blue computation has necessarily committed the store. Rules ld_G-*queue*, ld_G-*mem*, and ld_B-*mem* specify these behaviors.

Notice that there is no mechanism for verifying the address used in loads. Hence, a fault can result in an invalid address. In practice such a load might induce a hardware exception such as a segmentation fault (rules ld_G-*fail* and ld_B-*fail*) or might result in loading some arbitrary value (rules ld_G-*rand* and ld_B-*rand*).

Control-Flow Instructions. Any change in the control-flow of a program may cause a different sequence of values to be stored and observed by an external user. Consequently, the hardware contains mechanisms to detect faults in addresses that serve as jump targets. Intuitively, these mechanisms mirror the solution to faults in stored data in that execution of a control-flow transfer is accomplished through two instructions. Our solution uses a combination of software and hardware control-flow protection that is similar to watchdog processors [35], but that makes both versions of the control flow explicit as in software-only control flow protection [47, 60]. The operational rules that implement this solution are shown in Figure 2.4.

To achieve an unconditional jump, one executes a jmp_G instruction first and a related jmp_B instruction at some point in the future. A $jmp_G\ r_1$ moves the destination address from r_1 into the special destination register *gd* (rule jmp_G). Like the store queue, the destination register stores a programmer intention, initiated by the green computation. Later, when the blue computation attempts to commit the jump by executing a $jmp_B\ r_2$ instruction, the contents of r_2 are compared to the contents of the destination register and if they are equal, control jumps to that location (rule jmp_B). If the addresses are different, the hardware detects a fault (see rule jmp_B-*fail*). Similar to the constraint for the store queue, forcing green control flow instructions to be executed before the corresponding blue version constrains the instruction schedule. Section 2.4 will show that this scheduling constraint has only a minimal performance impact.

$\boxed{\Sigma \longrightarrow_k^s \Sigma'}$

$$\frac{R_{val}(gd) = 0}{(R, C, M, Q, jmp_G\ \ r_d) \longrightarrow_0 (R\text{++}[gd \mapsto R(r_d)], C, M, Q, \cdot)}\ (jmp_G)$$

$$\frac{R_{val}(gd) \neq\ 0}{(R, C, M, Q, jmp_G\ \ r_d) \longrightarrow_0\ \ fault}\ (jmp_G\text{-}fail)$$

$$\frac{\begin{array}{c} R_{val}(gd) \neq\ 0 \qquad R_{val}(r_d) = R_{val}(gd) \\ R' = R[pc_G \mapsto R(gd)][pc_B \mapsto R(r_d)][d \mapsto G\ 0] \end{array}}{(R, C, M, Q, jmp_B\ \ r_d) \longrightarrow_0 (R', C, M, Q, \cdot)}\ (jmp_B)$$

$$\frac{R_{val}(r_d) \neq R_{val}(gd)\ \ or\ R_{val}(gd) = 0}{(R, C, M, Q, jmp_B\ \ r_d) \longrightarrow_0\ \ fault}\ (jmp_B\text{-}fail)$$

$$\frac{R_{val}(gd) = 0 \qquad R_{val}(r_z) \neq 0}{(R, C, M, Q, bz_c\ \ r_z, r_d) \longrightarrow_0 (R\text{++}, C, M, Q, \cdot)}\ (bz\text{-}untaken)$$

$$\frac{R_{val}(gd) \neq\ 0 \qquad R_{val}(r_z) \neq\ 0}{(R, C, M, Q, bz_c\ \ r_z, r_d) \longrightarrow_0\ \ fault}\ (bz\text{-}untaken\text{-}fail)$$

$$\frac{R_{val}(gd) = 0 \qquad R_{val}(r_z) = 0}{(R, C, M, Q, bz_G\ \ r_z, r_d) \longrightarrow_0 (R\text{++}[gd \mapsto R(r_d)], C, M, Q, \cdot)}\ (bz_G\text{-}taken)$$

$$\frac{R_{val}(gd) \neq\ 0 \qquad R_{val}(r_z) = 0}{(R, C, M, Q, bz_G\ \ r_z, r_d) \longrightarrow_0\ \ fault}\ (bz_G\text{-}taken\text{-}fail)$$

$$\frac{\begin{array}{c} R_{val}(gd) \neq 0 \qquad R_{val}(r_z) = 0 \qquad R_{val}(r_d) = R_{val}(gd) \\ R' = R[pc_G \mapsto R(gd)][pc_B \mapsto R(r_d)][gd \mapsto G\ 0] \end{array}}{(R, C, M, Q, bz_B\ \ r_z, r_d) \longrightarrow_0 (R', C, M, Q, \cdot)}\ (bz_B\text{-}taken)$$

$$\frac{R_{val}(r_z) = 0 \qquad (R_{val}(r_d) \neq R_{val}(gd)\ \ or\ R_{val}(gd) = 0)}{(R, C, M, Q, bz_B\ \ r_z, r_d) \longrightarrow_0\ \ fault}\ (bz_B\text{-}taken\text{-}fail)$$

Figure 2.4: Operational rules for control flow instructions.

The following code illustrates a typical control-flow transfer.

$\texttt{ld}_G\ r_1\texttt{,}\ r_2$

$\texttt{jmp}_G\ r_1$

$\texttt{ld}_B\ r_3\texttt{,}\ r_4$

$\texttt{jmp}_B\ r_3$

Initially, registers r_2 and r_4 should point to the same memory location, which contains a code pointer to jump to. The example illustrates some of the flexibility in scheduling jump instructions.

Conditional jumps are more complex, but follow the same principles. The green conditional $bz_G\ r_z, r_d$ tests r_z and if it is 0, moves the contents of r_d into destination register gd (rules *bz-untaken* and bz_G*-taken*). No control-flow transfer occurs until a blue conditional $bz_B\ r'_z, r'_d$ tests the contents of its r'_z register. If r'_z is 0 then r'_d must equal the contents of gd, and if so, the control flow transfer occurs (rule bz_B*-taken*). If r'_z is not 0, it is not good enough merely to fall through — the contents of r'_z might be faulty. To avoid this possibility, the instruction examines the destination register. If it is 0 (and hence a prior bz_G instruction did not store an address), the fall-through occurs (rule *bz-untaken*). Our metatheory will show that this mechanism suffices to detect faults either in the green computation (registers r_z and r_d) or the blue computation (registers r'_z and r'_d).

2.2 Typing

The primary goal of the TAL$_{\text{FT}}$ type system is to ensure that well-typed programs exhibit fail-safe behavior in the presence of transient faults. In other words, well-typed programs must guarantee that even when a single fault occurs, a memory-mapped output device

$$\begin{array}{llcl}
\multicolumn{4}{l}{\text{Static Expressions}} \\
\textit{exp kinds} & \kappa & ::= & \kappa_{int} \mid \kappa_{mem} \\
\textit{exp contexts} & \Delta & ::= & \cdot \mid \Delta, x : \kappa \\
\textit{exps} & E & ::= & x \mid n \mid E \ op \ E \mid sel \ E_m \ E_n \mid emp \mid upd \ E_m \ E_{n_1} \ E_{n_2} \\
\textit{substitutions} & S & ::= & \cdot \mid S, E/x \\
\\
\multicolumn{4}{l}{\text{Types}} \\
\textit{zap tags} & Z & ::= & \cdot \mid c \\
\textit{basic types} & b & ::= & int \mid \Theta \rightarrow void \mid b \ ref \\
\textit{reg types} & t & ::= & \langle c, b, E \rangle \mid E' = 0 \Rightarrow \langle c, b, E \rangle \\
\textit{reg file types} & \Gamma & ::= & \cdot \mid \Gamma, a \rightarrow t \\
\textit{result types} & RT & ::= & \Theta \mid void \\
\\
\multicolumn{4}{l}{\text{Contexts}} \\
\textit{heap typing} & \Psi & ::= & \cdot \mid \Psi, n : b \\
\textit{static context} & \Theta & ::= & \Delta; \Gamma; \overline{(E_d, E_s)}; E_m
\end{array}$$

Figure 2.5: TAL$_{\text{FT}}$ type syntax.

will never read a corrupt value and make it visible to a user. We call this property "fault tolerance."

The syntax of the type system is presented in Figure 2.5. In the following sections, we explain the intuitions and principles behind the various elements of the type system. Throughout the discussion, the reader will notice that our typing rules are not syntax-directed. In other words, there may be situations when more than one rule can be applied at a given point. Of course, as with other sorts of typed assembly language or proof-carrying code, this presents no particular difficulty in practice — during the translation, the compiler knows the correct typing derivation. The compiler can either produce this full derivation or, for efficiency's sake, can produce sufficient "typing hints" to make type reconstruction trivial.

2.2.1 Static Expressions

Our “type system” is actually a combination of two theories, one being a relatively simple type theory for assembly language, inspired by previous work on TAL [40], and the second being a Hoare Logic, designed to enforce the more precise invariants required for fault tolerance. The latter component requires we define a language of *static expressions* for reasoning about values and storage.

The static expressions are drawn from the standard theory of arithmetic and arrays used in many classical Hoare Logics (*c.f.*, Necula’s thesis [45]). These static expressions are classified as either integers (kind κ_{int}) or memories (kind κ_{mem}). The integer expressions include variables, constants, simple arithmetic operations, and values from a memory ($sel\ E_m\ E_n$ is the integer located at address E_n in E_m). The memory expressions include variables, the empty memory (emp), and memory updates ($upd\ E_m\ E_{n_1}\ E_{n_2}$ is a memory E_m updated so that address E_{n_1} stores value E_{n_2}).

The context Δ is a mapping from variables to kinds and contains the free variables used in the static expressions at compile time. The judgment $\Delta \vdash E : \kappa$ classifies expression E as having kind κ. At runtime, these variables will have known values, so in order to reason about program execution we use a substitution S which maps expression variables to expressions. The judgment $\Delta \vdash S : \Delta'$ holds when the substitution S maps variables in $Dom(\Delta')$ to expressions well-formed in Δ with types in $Rng(\Delta')$. The judgment $\Delta \vdash E_1 = E_2$ is valid when E_1 and E_2 are equal objects in the standard model. The function $[\![E]\!]$ supplies the denotation of the closed static expression E as either an integer or a memory, depending on its kind. The judgments are defined in Figures 2.6 and 2.7, and the denotation function is defined in Figure 2.7.

$\boxed{\Delta \vdash E : \kappa}$

$$\frac{x \in Dom(\Delta)}{\Delta \vdash x : \Delta(x)} \; (E\text{-}var\text{-}t)$$

$$\frac{}{\Delta \vdash n : \kappa_{int}} \; (E\text{-}int\text{-}t) \qquad \frac{\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int}}{\Delta \vdash E_1 \; op \; E_2 : \kappa_{int}} \; (E\text{-}op\text{-}t)$$

$$\frac{}{\Delta \vdash emp : \kappa_{mem}} \; (E\text{-}emp\text{-}t) \qquad \frac{\Delta \vdash E_m : \kappa_{mem} \qquad \Delta \vdash E_n : \kappa_{int}}{\Delta \vdash sel \; E_m \; E_n : \kappa_{int}} \; (E\text{-}sel\text{-}t)$$

$$\frac{\Delta \vdash E_m : \kappa_{mem} \qquad \Delta \vdash E_{n_1} : \kappa_{int} \qquad \Delta \vdash E_{n_2} : \kappa_{int}}{\Delta \vdash upd \; E_m \; E_{n_1} \; E_{n_2} : \kappa_{mem}} \; (E\text{-}upd\text{-}t)$$

$\boxed{\Delta \vdash S : \Delta'}$

$$\frac{}{\Delta \vdash \cdot : \cdot} \; (sub\text{-}emp\text{-}t)$$

$$\frac{\Delta \vdash S' : \Delta'' \qquad \Delta \vdash E : \kappa \qquad x \notin Dom(\Delta) \cup Dom(\Delta'')}{\Delta \vdash S', E/x \; : \; \Delta'', x : \kappa} \; (sub\text{-}t)$$

Figure 2.6: Semantics of Static Expressions, Part 1.

$\boxed{[\![E]\!]}$

$$\begin{array}{lcl}
[\![n]\!] & = & n \\
[\![emp]\!] & = & \cdot \\
[\![E_1\ op\ E_2]\!] & = & [\![E_1]\!]\ op\ [\![E_2]\!] \\
[\![sel\ E_m\ E_n]\!] & = & [\![E_m]\!]([\![E_n]\!]) \\
[\![upd\ E_m\ E_1\ E_2]\!] & = & [\![E_m]\!][\ [\![E_1]\!] \mapsto [\![E_2]\!]\ ]
\end{array}$$

$\boxed{\Delta \vdash\ E_1 = E_2}$

$$\frac{\begin{array}{c}\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int} \\ \forall S.\ \cdot \vdash S : \Delta \implies [\![S(E_1)]\!] = [\![S(E_2)]\!]\end{array}}{\Delta \vdash\ E_1 = E_2}\ (\textit{E-eq})$$

$$\frac{\begin{array}{c}\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int} \\ \forall S.\ \cdot \vdash S : \Delta \implies [\![S(E_1)]\!] \neq [\![S(E_2)]\!]\end{array}}{\Delta \vdash\ E_1 \neq E_2}\ (\textit{E-neq})$$

$$\frac{\begin{array}{c}\Delta \vdash E_1 : \kappa_{mem} \qquad \Delta \vdash E_2 : \kappa_{mem} \\ \forall \ell \in\ Dom([\![S(E_1)]\!]) \cup Dom([\![S(E_2)]\!]).\ [\![S(E_1)]\!](\ell) = [\![S(E_2)]\!](\ell)\end{array}}{\Delta \vdash\ E_1 = E_2}\ (\textit{E-mem-eq})$$

Figure 2.7: Semantics of Static Expressions, Part 2.

2.2.2 Value Typing

Because faults strike values, corrupting their bit patterns in arbitrary ways, the subtleties of value typing are a key concern. Informally, the type system maintains three key pieces of information about every value:

1. *A color (green or blue).* The type system is organized to ensure that when a value is known to be green, its contents can only depend on the contents of other green values not blue ones, and likewise, blue can only depend upon blue. Hence, while a fault in a green value can eventually corrupt arbitrarily many other green values, it cannot corrupt any blue values, and vice versa.

2. *A "basic type".* When no fault has occurred in the value's color, the value's basic type describes its shape. Values with type *int* may have any bit pattern. Values with type $\Theta \rightarrow \mathit{void}$ are pointers to code (continuations). One must satisfy the precondition Θ before jumping to them. Values with type $b\ \mathit{ref}$ are pointers to values with type b.

3. *A static expression.* When there has been no fault in a value's color, the value exactly equals the static expression. Static expressions are used to guarantee that in the absence of faults, the green and blue computations produce equal values, and hence, dynamic fault detection checks always succeed.

To summarize, every value is typed using a triple $\langle c, b, E \rangle$, where c is a color, b is a basic type, and E is a static expression. The presence of the static expression makes this type a sort of singleton type.

Value Typing Judgment. The value typing judgment in Figure 2.8 has the form $\Psi; \Delta \vdash^{Z} v : t$, where Ψ maps heap addresses to basic types, and Δ contains the free expression

$\boxed{\Psi \vdash n : b}$

$$\frac{}{\Psi \vdash n : int}\ (int\text{-}t) \qquad \frac{}{\Psi \vdash n : \Psi(n)}\ (base\text{-}t)$$

$\boxed{\Psi; \Delta \vdash^Z v : t}$

$$\frac{\Psi \vdash n : b \qquad \Delta \vdash E = n}{\Psi; \Delta \vdash^Z c\ n : \langle c, b, E \rangle}\ (val\text{-}t)$$

$$\frac{n \neq 0 \qquad \Psi; \Delta \vdash^Z c\ n : \langle c, b, E \rangle \qquad \Delta \vdash E' = 0}{\Psi; \Delta \vdash^Z c\ n : E' = 0 \Rightarrow \langle c, b, E \rangle}\ (cond\text{-}t)$$

$$\frac{\Delta \vdash E' \neq 0}{\Psi; \Delta \vdash^Z c\ 0 : E' = 0 \Rightarrow \langle c, b, E \rangle}\ (cond\text{-}t\text{-}n0)$$

$$\frac{\Delta \vdash E : \kappa_{int}}{\Psi; \Delta \vdash^c c\ n : \langle c, b, E \rangle}\ (val\text{-}zap\text{-}t)$$

$$\frac{\Delta \vdash E : \kappa_{int}}{\Psi; \Delta \vdash^c c\ n : E' = 0 \Rightarrow \langle c, b, E \rangle}\ (val\text{-}zap\text{-}cond)$$

Figure 2.8: Value Typing.

variables. In the rule *val-t*, a colored value $c\ n$ is given the type $\langle c, b, E\rangle$ when the static expression E is equal to n, and $\Psi \vdash n : b$. The judgment $\Psi \vdash n : b$ allows n to be given either the basic type *int* or the type of the address n in memory.

The two rules *cond-t* and *cond-t-n0* are used to type the conditional type $(E' = 0 \Rightarrow \langle G, \Theta \to void, E'_r\rangle)$. When the static expression E' is equal to zero, values of this type also have type $\langle G, \Theta \to void, E'_r\rangle$. When E' is not equal to zero, values with this type must be 0.

The final two rules for $\Psi; \Delta \vdash^Z v : t$ make use of the *zap tag* Z, which is either empty or a color c. If the zap tag is a color c, then there may have been a fault affecting data of that color. Data colored the same as the zap tag can be given any type, as it may have been arbitrarily corrupted. The static expression used in this type may not contain any free expression variables.

Value Subtyping. There is also a subtyping relation $\Delta \vdash t \leq t'$ that allows all types $\langle c, b, E_1\rangle$ to be subtypes of $\langle c, int, E_2\rangle$ when $\Delta \vdash E_1 = E_2$. This relation is extended to register file subtyping $\Delta \vdash \Gamma_1 \leq \Gamma_2$, by requiring that the type of each general-purpose register in Γ_2 be a supertype of the corresponding register in Γ_1. Note that here is no required relationship between the special registers gd, pc_G, and pc_B. The rules for these judgments appear in Figure 2.9.

2.2.3 Instruction Typing

While many of the instruction typing rules are quite complex, the essential principles guiding their construction may be summarized as follows.

1. *In the absence of faults, standard type theoretic principles should be valid.* In order to guarantee basic safety properties, the type system checks standard properties in

$$\boxed{\Delta \vdash\ t \leq t'}$$

$$\frac{\Delta \vdash\ E_1 = E_2}{\Delta \vdash\ \langle c, b, E_1 \rangle \leq \langle c, b, E_2 \rangle}\ (\textit{subtp-triple})$$

$$\frac{\Delta \vdash\ E_1 = E_2}{\Delta \vdash\ \langle c, b, E_1 \rangle \leq \langle c, int, E_2 \rangle}\ (\textit{subtp-int})$$

$$\frac{\Delta \vdash\ t \leq t' \qquad \Delta \vdash\ E = E'}{\Delta \vdash\ (E = 0 \Rightarrow t) \leq (E' = 0 \Rightarrow t')}\ (\textit{subtp-cond})$$

$$\boxed{\Delta \vdash\ \Gamma_1 \leq \Gamma_2}$$

$$\frac{\forall r \in Dom(\Gamma_2).\ \Gamma_1(r) \leq \Gamma_2(r)}{\Delta \vdash\ \Gamma_1 \leq \Gamma_2}\ (\textit{reg-file-comp})$$

Figure 2.9: Subtyping.

much the same manner as previous typed assembly languages [40]. For example, jump targets must have code types, while loads and stores must operate over values with reference types.

2. *Green values only depend on other green values, and blue values only depend on blue values.* When this invariant is maintained, a fault in a blue value can never corrupt a green value and vice versa.

3. *Both green and blue computations have equal say in any dangerous actions.* Dangerous actions include storing values to memory-mapped output devices and executing control-flow operations. When both blue and green computations are involved, a fault in just one color is insufficient to deceive the hardware fault detection mechanisms.

4. *In the absence of faults, green and blue computations must compute identical values.* To be more precise, green and blue computations must store identical values to identical storage locations and must issue orders to transfer control to identical addresses. If not, the hardware will claim to detect faults when there have been none, or alternatively, might exhibit incorrect behaviors when there is a fault.

The first three principles are relatively straightforward to enforce. The fourth principle leads to the most technical challenges as it requires we check equality constraints between values. Moreover, since construction of these values depends on storage, the type system must maintain a relatively accurate static representation of storage. We accomplish this latter challenge using techniques drawn from Hoare Logics. The former challenge (testing values for equality) is achieved through the use of the singleton types described earlier.

The Instruction Typing Judgment. The judgment for typing instructions has the form $\Psi;\Theta \vdash ir \Rightarrow RT$. Unlike the context Ψ, which only contains invariant heap typing assumptions, Θ contains fine-grained context-sensitive information about the current state of memory and the register file. More specifically, Θ consists of the following subcontexts: (1) Δ, which describes the free expression variables appearing in the other context-sensitive objects, (2) Γ, which describes the mapping of register names to types for register values, (3) $\overline{(E_d, E_s)}$, which describes the values in the queue, and (4) E_m, which describes memory, as one does in Hoare Logic.

The "result" of checking an instruction is a result type RT. A result type may either be *void*, indicating control does not proceed past the instruction (it is a jump), or a postcondition Θ', which describes the state of memory and the register file after execution of the instruction.

The typing rules are defined using several notational abbreviations. The notation Γ++ adds one to the static expression associated with each program counter register in Γ. The expression $\overline{upd}\ E_m\ \overline{(E_d, E_s)}$ is $(upd\ (\ldots(upd\ E_m\ E_{d_k}\ E_{s_k})\ldots)\ E_{d_1}\ E_{s_1})$ when $\overline{(E_d, E_s)} = ((E_{d_1}, E_{s_1}), \ldots, (E_{d_k}, E_{s_k}))$. Figures 2.10, 2.11, and 2.12 presents the typing rules for instructions, and the following paragraphs explain the main points of interest.

Typing Basic Instructions. Basic arithmetic operations are not "dangerous" to execute, so the definitions of their typing rules are driven by principles 1 and 2, mentioned earlier. Consider, for example, rule *op2r-t* for an arithmetic operation op. This rule requires that the operand registers contain integers with the same color c in accordance with principal 2 (green depends on green, blue depends on blue). The result register r_d has a type colored c as well. In accordance with principle 1, the result has integer type. The rule also states that the static expression describing the result register is $E'_s\ op\ E'_t$ and that the state of the queue and memory are unchanged by evaluation of the instruction.

Typing Memory Instructions. Store operations are "dangerous" — they make computed values observable by the outside world — so we must be particularly careful in the formulation of their typing rules. In accordance with principle 1, both green and blue store instructions (rules st_G*-t* and st_B*-t*) require that the address register has the basic type *b ref* and the value register has the corresponding basic type b. Intuitively, the store queue is a green object, and in accordance with principle 2, the green store instruction may push an address-value pair onto the front the queue as long as the address and the value are green. In accordance with principle 4, the rule for the blue store checks that the address-value pair to be stored is exactly equal to the address-value pair at the end of the queue. Since the arguments to the blue store have a blue type and the queue always

$$\boxed{\Psi;\Theta \vdash\ ir\ \Rightarrow RT}$$

$$\frac{}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ \cdot\ \Rightarrow (\Delta;\Gamma;\overline{(E_d,E_s)};E_m)}\ (\cdot\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(r_s) = \langle c, int, E_s'\rangle \qquad \Gamma(r_t) = \langle c, int, E_t'\rangle \\ \Gamma' = \Gamma{+}{+}[r_d \mapsto\ \langle c, int, E_s'\ op\ E_t'\rangle]\end{array}}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ op\ \ r_d, r_s, r_t\ \Rightarrow (\Delta;\Gamma';\overline{(E_d,E_s)};E_m)}\ (op2r\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(r_s) = \langle c, int, E_s'\rangle \\ \Gamma' = \Gamma{+}{+}[r_d \mapsto\ \langle c, int, E_s'\ op\ n\rangle]\end{array}}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ op\ \ r_d, r_s, c\ n\ \Rightarrow (\Delta;\Gamma';\overline{(E_d,E_s)};E_m)}\ (op1r\text{-}t)$$

$$\frac{\Psi;\Delta \vdash\ v : t}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ mov\ \ r_d, v\ \Rightarrow (\Delta;\Gamma{+}{+}[r_d \mapsto\ t];\overline{(E_d,E_s)};E_m)}\ (mov\text{-}t)$$

$$\frac{\Gamma(r_s) = \langle G, b\ ref, E_s'\rangle \qquad E\ =\ sel\ (\overline{upd}\ E_m\ \overline{(E_d,E_s)})\ E_s'}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ ld_G\ r_d\ r_s\ \Rightarrow (\Delta;\Gamma{+}{+}[r_d \mapsto\ \langle G, b, E\rangle];\overline{(E_d,E_s)};E_m)}\ (ld_G\text{-}t)$$

$$\frac{\Gamma(r_s) = \langle B, b\ ref, E_s'\rangle \qquad E\ =\ sel\ E_m\ E_s'}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ ld_B\ r_d\ r_s\ \Rightarrow (\Delta;\Gamma{+}{+}[r_d \mapsto\ \langle B, b, E\rangle];\overline{(E_d,E_s)};E_m)}\ (ld_B\text{-}t)$$

$$\frac{\Gamma(r_d) = \langle G, b\ ref, E_d'\rangle \qquad \Gamma(r_s) = \langle G, b, E_s'\rangle}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ st_G\ r_d\ r_s\ \Rightarrow (\Delta;\Gamma{+}{+};(E_d',E_s'),\overline{(E_d,E_s)};E_m)}\ (st_G\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(r_d) = \langle B, b\ ref, E_d''\rangle \quad \Gamma(r_s) = \langle B, b, E_s''\rangle \\ \Delta \vdash\ E_d' = E_d'' \quad \Delta \vdash\ E_s' = E_s''\end{array}}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)},(E_d',E_s');E_m) \vdash\ st_B\ r_d\ r_s\ \Rightarrow (\Delta;\Gamma{+}{+};\overline{(E_d,E_s)};upd\ E_m\ E_d'\ E_s')}\ (st_B\text{-}t)$$

Figure 2.10: Instruction Typing Rules for Basic Instructions and Memory Instructions.

contains green objects, both blue and green computations contribute to the actual storage operation (in accordance with principle 3).

The load operations are somewhat simpler than the store instructions since they are not "dangerous" in our model. However, like the store instructions, the operands of blue loads must be blue and the operands of green loads must be green. Once again, in accordance with principle 2, the result of a blue load is value with a blue type and likewise for a green load. Other than the colors, the main difference between two load typing rules is the way they obtain the expression that described the loaded value. The blue load (rule ld_B-t) simply uses $sel\ E_m\ E'_s$, which is the expression that describes the contents of the address described by E'_s in the memory description E_m. However, the operational rule for ld_G first checks the queue for any pending stores before loading the value from memory, and so the typing rule ld_G-t uses the expression $sel\ (\overline{upd}\ E_m\ \overline{(E_d, E_s)})\ E'_s$. In other words, before taking the corresponding expression from E_m, all the pending changes in the queue (described by $\overline{(E_d, E_s)}$) are applied. This way, if the loaded value also has a pending update, the selected expression will describe the updated value.

Typing Control-Flow Instructions. The rules for the green jump and branch instructions are relatively simple, as they do not actually involve a control flow transfer.

The simplest situation involves the green jump (rule jmp_G-t). This instruction is just a move from register r_d to the special destination register gd. The type of register gd is updated to the type of r_d (obeying both principles 1 and 2). The rule contains constraints that gd must be equal to 0 in both Γ and Γ' since the hardware resets the destination register to 0 after a jump.

The typing of the green conditional branch (rule bz_G-t) is quite similar to that of the green jump. One difference is that the bz_G instruction is now a conditional move as

$\boxed{\Psi;\Theta \vdash\ ir\ \Rightarrow RT}$

$$\frac{\begin{array}{c}\Gamma(gd) = \langle G, int, 0\rangle \\ \Gamma(r_d) = \langle G, \Theta \rightarrow void, E_{rd'}\rangle \\ \Theta = (\Delta';\Gamma';\overline{(E'_d,E'_s)};E'_m) \\ \Gamma'(gd) = \langle G, int, 0\rangle\end{array}}{\begin{array}{l}\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ jmp_G\ r_d \\ \quad \Rightarrow (\Delta;\Gamma{+}{+}[gd \mapsto\ \langle G, \Theta \rightarrow void, E_{rd'}\rangle];\overline{(E_d,E_s)};E_m)\end{array}}\ (jmp_G\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(gd) = \langle G, (\Delta';\Gamma';\overline{(E'_d,E'_s)};E'_m) \rightarrow void, E'_r\rangle \\ \Gamma(r_d) = \langle B, (\Delta';\Gamma';\overline{(E'_d,E'_s)};E'_m) \rightarrow void, E_r\rangle \\ \Delta \vdash\ E_r = E'_r \\ \exists S.\Delta \vdash\ S : \Delta' \\ S(\Gamma')(gd) = \langle G, int, 0\rangle \\ S(\Gamma')(pc_G) = \langle G, int, E'_r\rangle \\ S(\Gamma')(pc_B) = \langle B, int, E_r\rangle \\ \Delta \vdash \Gamma \leq\ S(\Gamma') \\ \Delta \vdash\ \overline{(E_d,E_s)} = S(\overline{(E'_d,E'_s)}) \\ \Delta \vdash\ E_m = S(E'_m)\end{array}}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ jmp_B\ r_d\ \Rightarrow void}\ (jmp_B\text{-}t)$$

Figure 2.11: Instruction Typing Rules for Jump Instructions.

$\boxed{\Psi;\Theta \vdash\ ir\ \Rightarrow RT}$

$$\frac{\begin{array}{c}\Gamma(gd) = \langle G, int, 0\rangle \\ \Gamma(r_z) = \langle G, int, E_z\rangle \\ \Gamma(r_d) = \langle G, \Theta \to void, E'_d\rangle \\ \Theta = (\Delta';\Gamma';\overline{(E'_d,E'_s)};E'_m) \\ \Gamma'(gd) = \langle G, int, 0\rangle\end{array}}{\begin{array}{l}\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ bz_G\ r_z\ r_d \\ \quad\Rightarrow (\Delta;\Gamma{+}{+}[gd \mapsto\ E_z = 0 \Rightarrow \langle G, \Theta \to void, E'_d\rangle];\overline{(E_d,E_s)};E_m)\end{array}}\ (bz_G\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(r_z) = \langle B, int, E_z\rangle \\ \Gamma(r_d) = \langle B, (\Delta';\Gamma';\overline{(E'_d,E'_s)};E'_m) \to void, E_r\rangle \\ \Gamma(gd) = E'_z = 0 \Rightarrow \langle G, (\Delta';\Gamma';\overline{(E'_d,E'_s)};E'_m) \to void, E'_r\rangle \\ \Delta \vdash\ E_z = E'_z \\ \Delta \vdash\ E_r = E'_r \\ \exists S.\Delta \vdash\ S : \Delta' \\ S(\Gamma')(gd) = \langle G, int, 0\rangle \\ S(\Gamma')(pc_G) = \langle G, int, E'_r\rangle \\ S(\Gamma')(pc_B) = \langle B, int, E_r\rangle \\ \Delta \vdash \Gamma \leq\ S(\Gamma') \\ \Delta \vdash\ \overline{(E_d,E_s)} = S(\overline{(E'_d,E'_s)}) \\ \Delta \vdash\ E_m = S(E'_m)\end{array}}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ bz_B\ r_z\ r_d\ \Rightarrow (\Delta;\Gamma{+}{+};\overline{(E_d,E_s)};E_m)}\ (bz_B\text{-}t)$$

Figure 2.12: Instruction Typing Rules for Branch Instructions.

opposed to an unconditional move. Hence, to represent the result of the move (unknown at compile time) the conditional type $(E'_z = 0 \Rightarrow \langle G, \Theta \to \mathit{void}, E'_r \rangle)$ is used. In addition, since the conditional branch may fall through, the result of typing the bz_G instruction is a proper postcondition as opposed to *void*, like jmp_G.

The typing rules for the blue control-flow instructions continue to follow the same four principles as the other instructions. Much of the complexity is inherently due to principle 1, which mandates checking all the usual constraints associated with jumps in any typed assembly language. For comparison, a jump rule in a non-fault-tolerant typed assembly language requires that the target register contains a code pointer, and that the current register file is a subtype of that expected by the target code. (To be consistent with TAL$_{\mathrm{FT}}$, the subtyping relationship below does not include the program counter register, and the requirement for the program counter is written separately.)

$$\frac{\begin{array}{c}\Gamma(r_d) = \Gamma' \to \mathit{void} \\ \Gamma'(pc) = \mathit{int} \\ \Delta \vdash \Gamma \leq \Gamma'\end{array}}{\Psi; (\Delta; \Gamma) \vdash \ jmp\ r_d\ \Rightarrow \mathit{void}}\ (\mathit{conventional}\text{-}jmp\text{-}t)$$

The blue jump is a true jump. According to principle 1, the typing rule jmp_B-t checks the standard typing invariants needed to ensure safety in any typed assembly language, including (1) that both copies of the jump target have a code type (see the first two premises), and (2) that the current state, including register file, memory, and queue, matches the expected state at the jump target, modulo some substitution S of static expressions for universally quantified variables Δ from the code type (see the final seven premises).

The typing of the blue conditional branch (rule *bz*$_B$-*t*)is similar to the jump. The conditional type of the destination register *gd* must depend on an expression that is equivalent to the expression used in the blue branch. And again, the instruction may fall through, so the rule has a proper postcondition.

2.2.4 Machine State Typing

In order to prove various properties of the type system, we need to specify the invariants of machine states that are preserved during execution. The judgments for typing a machine state Σ are shown in Figures 2.13 and 2.14 and explained below.

Register File Typing. The judgment $\Psi \vdash^{Z} R : \Gamma$ states that the register file R has the register file type Γ under heap typing Ψ and a zap tag Z. The contents of each register must have the type given to that register by Γ. Each program counter must have the appropriate color, and the program counters must contain equal values. (In the case where one program counter is corrupted, the zap tag Z in the first premise allows its actual value to differ from the expected computed value.)

Code Typing. The judgment $\Psi \vdash C$ states that code memory C is well-formed with respect to heap typing Ψ. The address 0 is not a valid code address. Each address must have a code type, and the code type must contain the precondition for the instruction at that address. If the instruction typing results in a postcondition Θ' (meaning that control may fall through to the next instruction) then the subsequent instruction must be well typed using Θ' as its precondition.

Memory Typing. The judgment $\Psi \vdash M : E_m$ states that given heap typing Ψ the value memory M is well-formed and can be described by the static expression E_m. The static

$\boxed{\Psi \vdash^Z R : \Gamma}$

$$\frac{\begin{array}{c}\forall a.\ \Psi; \cdot \vdash^Z R(a) : \Gamma(a) \\ \cdot \vdash \Gamma(pc_G) \leq \langle G, int, E_G \rangle \qquad \cdot \vdash \Gamma(pc_B) \leq \langle B, int, E_B \rangle \qquad \cdot \vdash E_G = E_B\end{array}}{\Psi \vdash^Z R : \Gamma}\ (R\text{-}t)$$

$\boxed{\Psi \vdash C}$

$$\frac{\begin{array}{c}0 \notin Dom(C) \\ \forall n \in Dom(C).\quad \Psi(n) = \Theta \rightarrow void \ \wedge\ \Psi; \Theta \vdash C(n) \Rightarrow RT \\ \wedge\ (RT = \Theta' \ implies\ \Psi(n+1) = \Theta' \rightarrow void)\end{array}}{\Psi \vdash C}\ (C\text{-}t)$$

$\boxed{\Psi \vdash M : E_m}$

$$\frac{\cdot \vdash E_m : \kappa_{mem} \qquad [\![E_m]\!] = M \qquad \forall \ell \in Dom(M).\ \Psi \vdash \ell : b\ ref \ \wedge\ \Psi \vdash M(\ell) : b}{\Psi \vdash M : E_m}\ (M\text{-}t)$$

$\boxed{\Psi \vdash^Z Q : \overline{(E_d, E_s)}}$

$$\frac{}{\Psi \vdash^Z () : ()}\ (Q\text{-}emp\text{-}t)$$

$$\frac{\begin{array}{c}Z \neq G \\ \Psi \vdash n_1 : b\ ref \qquad \Psi \vdash n_2 : b \qquad \cdot \vdash E_d = n_1 \qquad \cdot \vdash E_s = n_2 \\ \Psi \vdash^Z \overline{(n_1', n_2')} : \overline{(E_d', E_s')}\end{array}}{\Psi \vdash^Z (n_1, n_2), \overline{(n_1', n_2')} : (E_d, E_s), \overline{(E_d', E_s')}}\ (Q\text{-}t)$$

$$\frac{\cdot \vdash E_d : \kappa_{int} \qquad \cdot \vdash E_s : \kappa_{int} \qquad \Psi \vdash^G \overline{(n_1', n_2')} : \overline{(E_d', E_s')}}{\Psi \vdash^G (n_1, n_2), \overline{(n_1', n_2')} : (E_d, E_s), \overline{(E_d', E_s')}}\ (Q\text{-}zap\text{-}t)$$

Figure 2.13: Machine State Element Typing.

$\boxed{\vdash^Z (R, C, M, Q, ir)}$

$$\frac{\begin{array}{c} Dom(\Psi) = Dom(C) \cup Dom(M) \\ Z \neq G \implies Dom(Q) \subseteq Dom(M) \\ \Psi \vdash C \\ \forall c \neq Z.\ ir \neq \cdot \implies C(R_{val}(pc_c)) = ir \\ \forall c \neq Z.\ \Psi(R_{val}(pc_c)) = (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \to void \\ \exists S.\ \cdot \vdash S : \Delta \\ \Psi \vdash M : S(E_m) \\ \Psi \vdash^Z Q : S\overline{(E_d, E_s)} \\ \Psi \vdash^Z R : S(\Gamma) \end{array}}{\vdash^Z (R, C, M, Q, ir)} \ (\Sigma\text{-}t)$$

Figure 2.14: Machine State Typing.

expression E_m must have kind κ_{mem}, and M must be the denotation of E_m. Each location in the domain of M must have a type $b\ ref$ and the contents of that location must have type b.

Queue Typing. The judgment $\Psi \vdash^Z Q : \overline{(E_d, E_s)}$ means that queue Q can be described by the sequence of static expressions $\overline{(E_d, E_s)}$ given heap typing Ψ and zap tag Z. When the queue is empty, it is described by the empty sequence. When the zap tag Z is not G, the first pair (n_1, n_2) must consist of an address n_1 with type $b\ ref$ and a value n_2 with type b. This pair is described by the static expression pair (E_d, E_s) when E_d evaluates to n_1 and E_s evaluates to n_2. The remainder of the queue must be described by the remainder of the static expression sequence. All values in the queue are considered to be green, so when the zap tag is G, these values may have been arbitrarily corrupted. Accordingly in this case, the only requirements are that each static expression must have kind κ_{int} and the length of the queue must be the same as the length of the static expression sequence.

Machine State Typing. The judgment $\vdash^Z \Sigma$ states that a machine state Σ is well-typed under zap tag Z. This judgment holds when Σ is a five-tuple (R, C, M, Q, ir), and these elements are each well-typed and consistent with each other. Note that Σ is not well-typed when it is the fault state *fault*.

The domain of the heap typing Ψ must be the union of the domains of the code memory C and the value memory M. When Z is not equal to G, the queue has not been corrupted, and so each address in Q is also a valid address in M. The code memory C must be well-formed with respect to the heap typing Ψ.

The zap tag Z must be either empty or colored blue or green. At least one of the program counters, and possibly both, will not be colored by Z, and therefore must not be corrupted. If ir is an instruction, these correct program counters must point to an address that contains ir. The heap typing Ψ gives this address a code type $(\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \to$ *void*.

The context Δ contains the free variables in Γ, $\overline{(E_d, E_s)}$, and E_m. There must be some substitution S that gives values to each of the variables in Δ. Value memory M must be well-formed and described by the static expression $S(E_m)$. The queue Q must be described by $S\overline{(E_d, E_s)}$. The register file R must have type $S(\Gamma)$.

2.3 Formal Results

Now that we have defined the machine model and the type system, we need to prove that the type system is sound with respect to the machine model. In other words, if a program is well-typed, we can guarantee certain properties about all possible executions of that program. This section provides a high-level overview of the formal results for TAL$_{\mathrm{FT}}$. A more in-depth discussion appears in Appendix A.

In order to prove properties about program execution, we extend our single-step transition $\Sigma_1 \longrightarrow^s_k \Sigma_2$ from Section 2.1 to a sequence of n transitions containing exactly k faults $\Sigma_1 \overset{n}{\longrightarrow}{}^s_k \Sigma_2$, where n is greater than or equal to zero, and k is still either 0 or 1.

2.3.1 Type Safety

Progress states that well-typed states can take a step. In particular, a machine state that is well-typed under the empty zap tag can take a step to another machine state. A machine state that is well-typed under a zap tag of color c can take a step, but the result of that step may either be another machine state or the *fault* state. (Recall that $\vdash^Z \Sigma$ only holds when Σ is a five-tuple (R, C, M, Q, ir) and not the fault state *fault*.)

Theorem 1 (Progress)

1. *If* $\vdash \Sigma$ *then* $\Sigma \longrightarrow^s_0 \Sigma'$ *and* $\Sigma' \neq \mathit{fault}$.

2. *If* $\vdash^c \Sigma$ *then* $\Sigma \longrightarrow^s_0 \Sigma$.

According to Preservation, if a machine state is well-typed under a zap tag Z, and it takes a non-faulty step to another machine state, then that resulting state will also be well-typed under Z. Additionally, if a state is well-typed under the empty zap tag, and it takes a faulty step, then there is some color c such that the resulting state is well-typed under c.

Theorem 2 (Preservation)

1. *If* $\vdash^Z \Sigma$ *and* $\Sigma \longrightarrow^s_0 \Sigma'$ *and* $\Sigma' \neq \mathit{fault}$ *then* $\vdash^Z \Sigma'$.

2. *If* $\vdash \Sigma$ *and* $\Sigma \longrightarrow^s_1 \Sigma'$ *then* $\exists\, c.\ \vdash^c \Sigma'$.

Progress and Preservation define the usual notion of type safety. In addition, part one of Progress, together with part one of Preservation entail the following important corollary: The hardware never claims to have detected a fault when no fault has occurred during execution of a well-typed program.

Corollary 3 (No False Positives)
If $\vdash \Sigma$ *then* $\forall\, n.\ \Sigma \xrightarrow{n}{}_{0}^{s} \Sigma'$ *and* $\vdash \Sigma'$.

2.3.2 Fault Tolerance

A program is fault tolerant when all the faulty executions of that program *simulate* fault-free executions of the program. More precisely, the sequence of outputs from the faulty execution is required either to be identical to the fault-free execution or, in the case the hardware detects the fault, a prefix of the fault-free execution.

In order to reason about pairs of faulty and fault-free executions, we define similarity relations between values, register files, queues and machine states. Each of these relations is defined relative to the zap tag Z. Intuitively, if Z is empty, the related objects must be identical. If Z is a color c, the objects must be identical modulo values colored c. In the latter case, values colored c may be corrupted, and there is no hope they satisfy any particular relation. The formal definitions of these relations are shown in Figure 2.15.

Using the simulation relation, we can state and prove the fault tolerance theorem for well-typed programs precisely. Assume that machine state Σ is well-typed under the empty zap tag, and non-faulty execution of Σ for n steps results in a state Σ' and outputs a sequence of value-address pairs s. If somewhere during that execution a single fault is encountered, the faulty execution will either run for $n+1$ steps or terminate in the fault state during that time. If the faulty execution takes $n+1$ steps and reaches the non-faulty state Σ'_f, then Σ' simulates Σ'_f and the sequence of output pairs is identical the original execution. Alternatively, if the faulty execution reaches the fault state then the output pairs will be a prefix of the non-faulty output pairs.

$\boxed{v_1 \; sim^Z \; v_2}$

$$\frac{}{C\ n\ sim^Z\ C\ n}\ (sim\text{-}val) \qquad \frac{}{C\ n\ sim^C\ C\ n'}\ (sim\text{-}val\text{-}zap)$$

$\boxed{R \; sim^Z \; R'}$

$$\frac{\forall a.\ R(a)\ sim^Z\ R'(a)}{R\ sim^Z\ R'}\ (sim\text{-}R)$$

$\boxed{Q \; sim^Z \; Q'}$

$$\frac{}{\cdot\ sim^Z\ \cdot}\ (sim\text{-}Q\text{-}empty)$$

$$\frac{G\ n_1\ sim^Z\ G\ n_1' \qquad G\ n_2\ sim^Z\ G\ n_2' \qquad Q\ sim^Z\ Q'}{((n_1,n_2),Q)\ \ sim^Z\ ((n_1',n_2'),Q')}\ (sim\text{-}Q)$$

$\boxed{\Sigma_1 \; sim^Z \; \Sigma_2}$

$$\frac{R\ sim^Z\ R' \qquad Q\ sim^Z\ Q'}{(R,C,M,Q,ir)\ sim^Z\ (R',C,M,Q',ir)}\ (sim\text{-}\Sigma)$$

Figure 2.15: Similarity of Machine States.

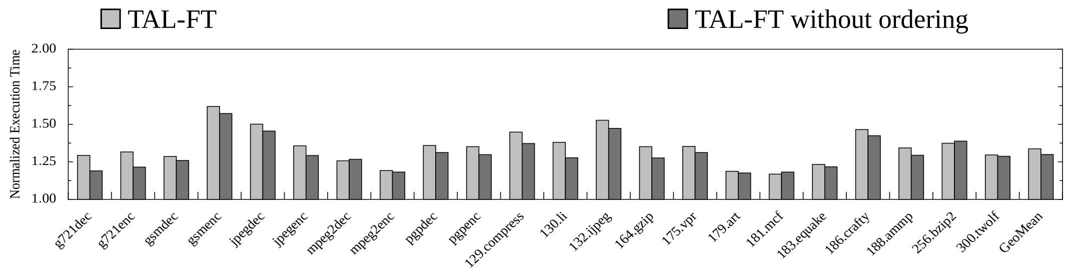


Figure 2.16: Performance Normalized to Unprotected Version.

Theorem 4 (Fault Tolerance)

If $\vdash \Sigma$ *and* $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$ *then either* $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$
or $\exists m \leq (n+1)\ .\ \Sigma \xrightarrow{m}{}^{s'}_{1} \mathit{fault}$*, and*

1. *For all derivations* $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$ *where* $\Sigma'_f \neq \mathit{fault}$*.*

   $s' = s$ *and* $\exists\ c.\ \Sigma'\ \mathit{sim}^c\ \Sigma'_f$*.*

2. *For all derivations* $\Sigma \xrightarrow{m}{}^{s'}_{1} \mathit{fault}$ *where* $m \leq (n+1)$*.*

   s' *is a prefix of* s*.*

2.4 Performance[1]

To better understand how TAL$_{\text{FT}}$ can be applied to real world situations, Reis [51] simulated the TAL$_{\text{FT}}$ hardware in the framework of a current computer architecture, the Intel Itanium 2 ISA. The instruction set of the Itanium 2 contains many more types of instructions than those specified in TAL$_{\text{FT}}$. While not an exact representation of the performance of TAL$_{\text{FT}}$, simulating the performance of TAL$_{\text{FT}}$ applied to this architecture gives guidance as to the feasibility of this system in a real architecture.

[1]This subsection is the work of G. A. Reis [51] and is included in this thesis for completeness.

To evaluate the performance impact of the techniques, a version of the VELOCITY compiler [69] was modified to add the reliability techniques of TAL$_{FT}$ and was used to compile the SPEC CINT2000 and MediaBench benchmark suites. These executions were compared against binaries generated by the original VELOCITY compiler, which have no fault detection. The reliability transformation was compiled into the low-level code immediately before register allocation and scheduling. To simulate the new hardware structures of TAL$_{FT}$, extra instructions were inserted to emulate the timing and dependences of the hardware structure accesses. Only optimizations that were TAL$_{FT}$ aware were used on the reliable code.

Performance metrics were obtained by running the resulting binaries with reference inputs on an HP workstation zx6000 with 2 900Mhz Intel Itanium 2 processors running Redhat Advanced Workstation 2.1 with 4Gb of memory. The `perfmon` utility was used to measure the CPU time.

Figure 2.16 presents the execution time of the fault-tolerant code relative to baseline binaries with no fault detection. Naïvely, one might expect the fault-tolerant code to run twice as slowly as the fault intolerant code since the number of instructions is essentially doubled. However, smart instruction scheduling and efficient allocation of resources reduces the execution time to only 34% more than the fault-intolerant baseline average. These simulations are in line with previously published software-only reliability performance experiments [60] that show the degradation due to redundant code to be less than double.

As alluded to in Section 2.1.2, Figure 2.16 compares the performance degradation with and without the scheduling constraint that green memory and control flow instructions must be executed before the corresponding blue versions. In order to perform the second set of experiments, the compiler was modified to produce code that had more flexibility in the scheduling of the green and blue versions. A more aggressive hardware implementation that could correlate the original and redundant memory operations regardless of the executed order was then simulated. As expected, this version has better performance (in most cases) than the unconstrained code. Although the colored ordering restriction of TAL$_{\text{FT}}$ may seem costly, removing this restriction provides only a small improvement. Comparing both to the unprotected code, the version without the ordering constraint increases execution time by 30% while the version with the ordering increases execution time by 34%.

2.5 Summary

This chapter introduced TAL$_{\text{FT}}$, a typed assembly language designed to verify a hybrid fault-tolerance technique. TAL$_{\text{FT}}$ is the first technique for reasoning about fault-tolerance properties of executable code. In addition, we identify four general principles for verifying correctness of fault tolerant systems and capture these in the TAL$_{\text{FT}}$ type system. The two main formal results show that a single fault affecting observable behavior in a well-typed TAL$_{\text{FT}}$ program will always be detected, and that the system will not claim to have detected a fault when none has occurred. Despite the fact that well-typed programs essentially duplicate all computation, we provide simulation results showing a performance overhead of 1.34x. The next chapter investigates using a similar language as the target of a realistic optimizing compiler.

Chapter 3

ETAL$_{\text{FT}}$: Generating fault-tolerant assembly code

The previous chapter introduced the TAL$_{\text{FT}}$ language and formally proved that all well-typed programs have certain behaviors, namely that they detect any single fault before it can affect the observable behavior of the program. TAL$_{\text{FT}}$ forms the core of a fault-tolerant typed assembly language, but is only a starting point for developing a realistic, usable language. This chapter shows how to extend TAL$_{\text{FT}}$ with additional features to serve as the target of a type-directed compiler.

In addition, these changes allow us to investigate the interactions between fault-tolerant typed assembly languages and other current research areas in typed assembly languages, namely typing the control stack and typing memory allocation and initialization.

The rest of the chapter is organized as follows. Section 3.1 defines a simple source-level imperative language called MiniC. Section 3.2 introduces the extended typed assembly language ETAL$_{\text{FT}}$ and summarizes its differences from TAL$_{\text{FT}}$. Section 3.3

```
int double(x:int ref) {
  int v = 0;
  v = !x;
  v = v + v;
  x := v;
  return 0;
}

int ref y = ref 3;
int result = 0;

result = double(y);
return result;
```

Figure 3.1: MiniC Example Program.

updates the formal results from the previous chapter for ETAL$_{\mathrm{FT}}$. Section 3.4 gives a naive translation from MiniC into ETAL$_{\mathrm{FT}}$ and sketches a proof that the result is well-typed. Section 3.5 gives an example of the translation. Section 3.6 discusses how standard compiler optimizations can be applied to well-typed ETAL$_{\mathrm{FT}}$ programs. Finally, Section 3.7 summarizes the contributions of ETAL$_{\mathrm{FT}}$.

3.1 MiniC

MiniC is a simple imperative language with basic arithmetic operations, references, and functions. An example program is shown in Figure 3.1, and the complete abstract syntax is shown in Figure 3.2. The notation *ref v* allocations a new heap location and initializes it with value *v*. Heap updates and accesses are notated by := and ! respectively. We will assume the existence of a garbage collector to run in the background and handle deallocation. Though the syntax is extremely simplified in order to facilitate the proofs later in this chapter, it is still quite expressive.

variables	x, f		
type	τ	$::=$	$int \mid \tau\ ref \mid X \to \tau$
typing context	X, F, A, L	$::=$	$\cdot \mid x : \tau,\ X$
value	v	$::=$	$n \mid x \mid ref\ v$
value list	vs	$::=$	$\cdot \mid v,\ vs$
binaryop	op	$::=$	$+ \mid - \mid *$
statement	s	$::=$	$x = v \mid x = v\ op\ v$
			$\mid x = !v \mid v := v$
			$\mid x = f(vs)$
			$\mid \texttt{if}\ v\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ v\ \texttt{do}\ s$
			$\mid s;\ s$
function declarations	fds	$::=$	$\cdot \mid \tau\ f(X)\ \{lds;\ s;\ \texttt{return}\ v\}\ fds$
local declarations	lds	$::=$	$\cdot \mid \tau\ x = v;\ lds$
program	p	$::=$	$fds;\ lds;\ s;\ \texttt{return}\ v;$

Figure 3.2: MiniC Syntax.

The main typing judgments for MiniC are summarized below, and the complete typing rules are provided in Appendix B. The typing rules are completely standard. Essentially they enforce that variables are defined before their use and that variables of a given type (integers, references, and functions) are used appropriately. We use the metavariable X to refer to a generic typing context and metavariables F, A, and L when referring to typing contexts for function variables, argument variables, and local variables respectively.

$X \vdash v : \tau$	*Value v has type τ in typing context X.*
$X \vdash vs : X'$	*Value list vs has type X' in typing context X.*
$F; A; L \vdash s\ wf$	*Statement s is well-formed given function context F, argument context A, and local variable context L.*

$F;A;L \vdash lds : L'$	*Given function context F, argument context A, and current local declaration L, after adding local declarations lds, the new set of local declarations is L'*
$F \vdash fds : F'$	*Given function context F, after adding the well-formed functions defined in fds, the new function context is F'.*
$\vdash p$ *wf*	*Program p is well-typed.*

We have defined MiniC in order to show how to compile MiniC programs into TAL$_{\text{FT}}$ programs. However, there are two things that make TAL$_{\text{FT}}$ unsuitable to be the target language for MiniC: (1) TAL$_{\text{FT}}$ has only a limited number of registers so it can't handle the unlimited number of arguments or local variables allowed by MiniC, and (2) TAL$_{\text{FT}}$ has no support for dynamic memory allocation.

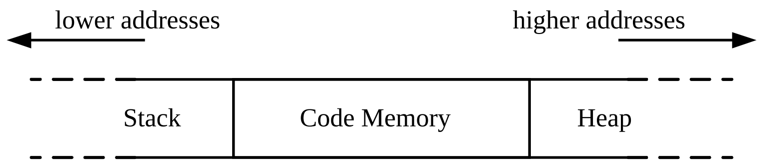
3.2 Extending TAL$_{\text{FT}}$ to ETAL$_{\text{FT}}$

In order to compile MiniC into fault-tolerant assembly code, we need to make a couple of extensions to TAL$_{\text{FT}}$ to address the issues with limited registers and lack of dynamic memory allocation. This new language is called ETAL$_{\text{FT}}$, and this section focuses on only on how it differs from TAL$_{\text{FT}}$. The complete specification of ETAL$_{\text{FT}}$ is provided as a reference in Appendix C.

3.2.1 Memory Layout

In TAL$_{\text{FT}}$, memory is divided into two pieces: a code memory C and a value memory M. Both of these are of a fixed size throughout the execution of a program. In addition, the abstract machine is able to tell the difference between these two memories, and many of the operational rules test to see if a location is only one of these two. (For example, rule *fetch-fail* applies when the address is not in C, even if it is in M.) It is reasonable to assume that the abstracted hardware can make this distinction, as real machines are often able to do the same using an execute bit.

In ETAL$_{FT}$ we will conceptually further divide the value memory M into two pieces: a heap and a stack. The goal is to use the stack as scratch space and consider the heap to be the portion of memory that is ”observable”. Both the heap and stack may grow during execution, and the stack may shrink as well. To avoid dealing with the possibility of running out of memory, we will assume an infinite sized address space with code memory C in the middle, the stack in the lower addresses growing downward, and the heap in the higher addresses growing upwards.



We will continue to assume that the machine can differentiate between code memory and the value memory, but we do not assume it can distinguish between the locations in M that represent the heap and those that represent the stack.

The following judgment is used to break a memory into two pieces, by using a set of addresses $\mathcal{L}$ to define the division. By using the domain of C as the set of addresses, we can divide the value memory M into the stack and the heap.

$$\boxed{M = M_1 \overset{\mathcal{L}}{\#} M_2}$$

$$\frac{\begin{array}{c} Dom(M) = Dom(M_1) \cup Dom(M_2) \\ Dom(M_1) \cap Dom(M_2) = \emptyset \\ \forall \ell_1 \in Dom(M_1).\ \forall \ell_{\mathcal{L}} \in \mathcal{L}.\ \forall \ell_2 \in Dom(M_2).\ \ell_1 < \ell_{\mathcal{L}} < \ell_2 \end{array}}{M = M_1 \overset{\mathcal{L}}{\#} M_2} \ (\#\text{-}def)$$

3.2.2 Stacks

Real machines use the stack to pass arguments and return values, as temporary space when the number of live values exceeds the number of registers, for local variables whose address may be taken, and to allocate local storage without the overhead of the garbage collector.

For simplicity, we will only investigate the first two of these uses. Even so, we need to immediately think about how the stack will interact with our goal of fault tolerance. In TAL$_{\text{FT}}$, values are only committed to memory when both the green and blue computations agree. However, the goal of the ETAL$_{\text{FT}}$ stack is to act as temporary spill space that is not externally visible. We do not want to require that the two computations synchronize when spilling values to the stack. For example, consider a program which needs many registers to perform some arithmetic computation and then stores the final result to memory. In this case, it may well make sense for the blue computation to spill all its values to the stack, reducing the register pressure for the green computation. Once the green computation has completed, it can spill the result and allow the blue computation to use all registers for its version of the task. Therefore the stack may contain a mix of green and blue values. It will be up to our type system to track which is which and to enforce the values of one color continue to only depend on the values of the same color.

The stack grows down towards lower addresses, and we will refer to the lowest location on the stack as the "top" of the stack and the highest location as the "base" of the stack.

New Instruction Syntax and Operational Semantics

For manipulating the stack, there are two new registers and four new instructions. Each of these new instructions has corresponding operational rules, shown in Figure 3.3.

$$\begin{array}{lcll} \textit{general regs} & a & ::= & \ldots \mid sp_c \\ \textit{instructions} & i & ::= & \ldots \mid salloc\ n \mid sfree\ n \mid sld_c\ r_d\ n \mid sst\ n\ r_v \end{array}$$

$$\frac{\begin{array}{c} R' = R{+}{+}[sp_G \mapsto R(sp_G) - n][sp_B \mapsto R(sp_B) - n] \\ m = min(Dom(M)) \\ M' = (M, m-1 \mapsto 0, \ldots, m-n \mapsto 0) \end{array}}{(R,C,M,Q,salloc\ n) \longrightarrow_0 (R',C,M',Q,\cdot)}\ (salloc)$$

$$\frac{\begin{array}{c} R' = R{+}{+}[sp_G \mapsto R(sp_G) + n][sp_B \mapsto R(sp_B) + n] \\ m = min(Dom(M)) \\ M = M', m \mapsto v_m, \ldots, (m+n-1) \mapsto v_1 \end{array}}{(R,C,M,Q,sfree\ n) \longrightarrow_0 (R',C,M',Q,\cdot)}\ (sfree)$$

$$\frac{\begin{array}{c} R(sp_c) + n \in Dom(M) \\ R' = R{+}{+}[r_d \mapsto M(R(sp_c) + n)] \end{array}}{(R,C,M,Q,sld_c\ r_d,\ n) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (sld_c)$$

$$\frac{R(sp_c) + n \notin Dom(M)}{(R,C,M,Q,sld_c\ r_d,\ n) \longrightarrow_0 fault}\ (sld_c\text{-}fail)$$

$$\frac{R(sp_B) = R(sp_G) \qquad R(sp_G) + n \in Dom(M)}{(R,C,M,Q,sst\ n,\ r_v) \longrightarrow_0^{(R(sp_G)+n, R(r_v))} (R{+}{+},C,M[R(sp_G) + n \mapsto R(r_v)],Q,\cdot)}\ (sst)$$

$$\frac{R(sp_G) \neq R(sp_B)\ or\ R(sp_B) + n \notin Dom(M)}{(R,C,M,Q,sst\ n,\ r_v) \longrightarrow_0\ fault}\ (sst\text{-}fail)$$

Figure 3.3: New Instruction Syntax and Operation Semantics to Support Stacks.

Just as there is a program counter for each computation, there is a green stack pointer sp_G and a blue stack pointer sp_B. Both stack pointers should point to the location on the top of the single stack.

The instructions $salloc\ n$ and $sfree\ n$ are used to allocate and deallocate space on the stack. The rules for $salloc$ and $sfree$ do this by either subtracting or adding n to both stack pointers. Our abstract semantics for these instructions also modify the set of locations in memory. Stacks grow downwards, so the lowest address in M is currently the top of the stack. When allocating, n new locations are added below the original top of the stack. For convenience, we assume these locations contain the value 0, but do consider them initialized yet. The type system will prevent a programmer from loading from uninitialized locations. When deallocating, the bottom n addresses are removed.

The instructions $sld_c\ r_d\ n$ and $sst\ n\ r_v$ are used to load from and store to the stack at offset n above the stack pointers. The rules for $sld_c\ n\ r_v$ are very similar to the rules for the basic load instructions. Instead of taking the address as an argument, sld_c takes an offset n and adds that to the stack pointer colored c. If that address is not in M, the machine detects a fault, otherwise the value is loaded. If the stack pointer colored c is corrupted, the loaded value may actually be a heap value or a value of the other color from the stack. But since the computation is already corrupted, no harm is done. As we will see, stores to the stack commit immediately, so there is no reason to check the queue for pending stores when loading from the stack.

In TAL$_{\text{FT}}$, the goal was to detect a fault before allowing a store to corrupt M (through either a corrupt address or a corrupt value). In ETAL$_{\text{FT}}$, that restriction is loosened slightly. If there has been a fault to the blue computation, that corruption may affect blue registers which are later spilled to the stack. We do not need to detect this unless it will affect the heap portion of memory. So when the zap tag is c, we will allow corrupt values in the locations in M corresponding to the c-colored stack locations. However, we still need to be extremely careful about the addresses used in storing to the stack. Like the program counters, the stack pointers can be corrupted. Because there is only one stack containing both green and blue values, we need to make sure that we do not store a blue value at an offset off of a corrupt stack pointer, as that might actually store into a green stack location or into the heap. The following ill-typed sequence has exactly the behavior we desire:

$$\texttt{add}\ \texttt{r}_{\texttt{G}}\ \texttt{sp}_{\texttt{G}}\ \texttt{n};\ \ \texttt{add}\ \texttt{r}_{\texttt{B}}\ \texttt{sp}_{\texttt{B}}\ \texttt{n};\ \ \texttt{st}_{\texttt{G}}\ \texttt{sp}_{\texttt{G}}\ \texttt{r}_{\texttt{v}};\ \ \texttt{st}_{\texttt{B}}\ \texttt{sp}_{\texttt{B}}\ \texttt{r}_{\texttt{v}}$$

It uses an equivalent address from each computation, so the store will only proceed if both computations agree on the address. However, both store instructions store the same (potentially corrupt) value r_v. The instruction $sst\ n\ r_v$ can be thought of as a macro for this sequence. Accordingly, it adds address-value pair written to M to the sequence of observable values, just as st_B does.

Typing the Stack

Typing program stacks is an active area of research in typed assembly languages [39, 30, 50]. Since stack locations are reused during the life of a program to hold values of different types, the type system needs to carefully track these changes. If stack locations may be aliased, tracking these changes soundly becomes difficult. Luckily, we do not require the ability to alias stack locations in ETAL$_{\mathrm{FT}}$, and so we can use a simple stack typing method.

In ETAL$_{\mathrm{FT}}$, a stack is conceptually an ordered list of location labels and type triples. We extend a number of definitions from TAL$_{\mathrm{FT}}$ and add two mutually recursive definitions for the stack type itself.

$$
\begin{array}{llcl}
\textit{exp kinds} & \kappa & ::= & \ldots \mid \kappa_\sigma \\
\textit{base types} & b & ::= & \ldots \mid \mathit{sptr} \\
\textit{reg types} & t & ::= & \ldots\ldots \mid \mathit{ns} \\
\textit{unlabeled stack} & \sigma & ::= & \mathit{sbase} \mid \rho \mid t :: \varsigma \\
\textit{labeled stack} & \varsigma & ::= & E : \sigma \\
\textit{static context} & \Theta & ::= & \Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma
\end{array}
$$

The kinds for expression variables are extended to include a kind κ_σ for the unlabeled stack type. Stack pointers are given base type *sptr* and their static expression is used to refer to the exact location on the stack. The nonsense type *ns* is given to newly allocated stack locations. A labeled stack type ς consists of a static expression E that represents the address on the top of unlabeled stack σ. An unlabeled stack type σ is either the empty stack *sbase*, a stack variable ρ or a type t on top of a labeled stack ς. Stack variables like ρ have kind κ_σ and are used to abstract parts of the caller's stack from the callee. The stack type needs to be tracked at each program point, so it is added to context Θ along with the existing register file type, queue description, and so on. The snippet below contains an example of a stack type:

$$
\begin{array}{lcl}
\varsigma & = & E_t : \langle G, int, E \rangle \ :: \ E_t + 1 : \langle B, int, E \rangle \ :: \ E_t + 2 : \rho \\
\Gamma(sp_G) & = & \langle G, sptr, E_t \rangle
\end{array}
$$

The stack type ς says that the top location on the stack is E_t and that location contains a green integer. The next location contains a blue integer. The rest of the stack starting at location $E_t + 2$ is unknown and abstracted into variable ρ. In addition, register sp_G points to the top of the stack.

$\boxed{\Psi \vdash n : b}$

$$\ldots \qquad \frac{}{\Psi \vdash n : sptr}\ (\textit{addr-stack-t})$$

$\boxed{\Psi; \Delta \vdash^{Z} n : t}$

$$\ldots \qquad \frac{}{\Psi; \Delta \vdash^{Z} n : ns}\ (\textit{ns-t})$$

$\boxed{\Delta \vdash t \leq t'}$

$$\ldots \qquad \frac{}{\Delta \vdash t \leq ns}\ (\textit{subtp-t-ns})$$

$\boxed{\Delta \vdash \varsigma \leq \varsigma'}$

$$\frac{\Delta \vdash E = E'}{\Delta \vdash E : sbase \leq E' : sbase}\ (\textit{subtp-}\varsigma\textit{-base})$$

$$\frac{\Delta \vdash E = E'}{\Delta \vdash E : \rho \leq E' : \rho}\ (\textit{subtp-}\varsigma\textit{-var})$$

$$\frac{\Delta \vdash E = E' \qquad \Delta \vdash t \leq t' \qquad \Delta \vdash \varsigma \leq \varsigma'}{\Delta \vdash E : (t :: \varsigma) \leq E' : (t' :: \varsigma')}\ (\textit{subtp-}\varsigma\textit{-cons})$$

$\boxed{\Delta \vdash \varsigma \ wf}$

$$\frac{\Delta \vdash E : \kappa_{int}}{\Delta \vdash E : sbase \ wf}\ (\varsigma\textit{- wf-base}) \qquad \frac{\Delta(\rho) = \kappa_{\sigma} \qquad \Delta \vdash E : \kappa_{int}}{\Delta \vdash E : \rho \ wf}\ (\varsigma\textit{- wf-var})$$

$$\frac{\Delta \vdash E + 1 = E' \qquad \Delta \vdash t \ wf \qquad \Delta \vdash (E' : \sigma') \ wf}{\Delta \vdash E : (t :: (E' : \sigma') \ wf}\ (\varsigma\textit{- wf-cons})$$

Figure 3.4: Modifications to Value Typing, Subtyping, and Well-Formed Judgments.

$\boxed{\Delta; \varsigma \vdash E : t}$

$$\frac{\Delta \vdash E_s = E}{\Delta;\ E_s : (t :: \varsigma')\ \vdash E : t}\ (\varsigma\text{-}\mathit{lookup\text{-}top}) \qquad \frac{\Delta \vdash E_s \neq E \qquad \Delta; \varsigma' \vdash E : t}{\Delta;\ E_s : (t :: \varsigma')\ \vdash E : t}\ (\varsigma\text{-}\mathit{lookup\text{-}tail})$$

$\boxed{\Delta \vdash \varsigma[E \mapsto t] = \varsigma'}$

$$\frac{\Delta \vdash E_s = E}{\Delta \vdash (E_s : (t_s :: \varsigma))[E \mapsto t] = E_s : (t :: \varsigma)}\ (\varsigma\text{-}\mathit{update\text{-}top})$$

$$\frac{\Delta \vdash E_s \neq E \qquad \Delta \vdash \varsigma[E \mapsto t] = \varsigma'}{\Delta \vdash (E_s : (t_s :: \varsigma))[E \mapsto t] = E_s : (t_s :: \varsigma')}\ (\varsigma\text{-}\mathit{update\text{-}tail})$$

Figure 3.5: Stack Lookup and Update.

Figure 3.4 shows the minor changes required to the value typing, subtyping judgments, and well-formed expression judgments. Any integer can be given the base type $sptr$. In other words, dangling pointers into the stack may exist, as long as we do not dereference them. Any integer can also given the type *ns*. Since values of type *ns* have no color, neither computation can manipulate them. Instead, all that can be done is to overwrite the value. The subtyping judgments are extended so that all types are subtypes of *ns* and one stack is a subtype of another if the locations are equivalent and corresponding locations have types which are subtypes of each other. In order for a stack type to be well formed, the unabstracted portion of the stack must describe sequential locations.

In addition, there are two new judgments for looking up the type of a stack location and updating the type of a stack location shown in Figure 3.5. Judgment $\Delta; \varsigma \vdash E : t$ states that location E in stack type ς has type t. Judgment $\Delta \vdash \varsigma[E \mapsto t] = \varsigma'$ Says that the result of updating location E in stack type ς to contain type t is a new stack type ς'. All stack updates are strong updates. In other words, it does not matter what type is currently in a location when we go to update it.

The instruction typing rules for the four new stack instructions are shown in Figure 3.6. When allocating or deallocating space on the stack, we require that both stack pointers and the location on the top of the stack be described by equivalent expressions. When allocating space, new locations with the nonsense type *ns* are added to the top of the stack. When deallocating, the stack must contain at least as many locations as are being freed. Both instructions update the types of the two stack pointers to refer to the new top of the stack.

To load from the stack at offset n, we must be able to look up the type of the location at that offset. The destination register is updated to contain that type. When storing to the stack, the two stack pointers must be equivalent, and the stack type is updated with the type of the value being stored. Notice that the type of the value being overwritten is irrelevant.

The rules for all the other instructions are modified to propagate the stack type as part of Θ. The rules for jmp_B and brz_B are extended to require that the current stack type is a subtype of the target's stack type.

In terms of top-level judgments, the major change is a new judgment $\Psi \vdash^Z M : \varsigma$ shown in Figure 3.7. The stack typing judgment states that a memory M can be described by a stack type ς under a zap tag Z. Unlike the judgment about memory in TAL$_{\text{FT}}$, this judgment is parameterized by a zap tag which is used to call the standard value typing judgment on each location and its corresponding type.

The machine state type judgment $\vdash^Z (R, C, M, Q, ir)$ is updated to divide memory M into a stack portion M_s and a heap portion M_h. The stack is typed with $\Psi \vdash^Z M_s : \varsigma$, and M_h is typed by a new heap typing judgment described next.

$$\boxed{\Psi;\Theta \vdash \ ir \ \Rightarrow RT}$$

$$\ldots$$

$$\frac{\begin{array}{c}\Gamma(sp_G) = \langle G, sptr, E_g\rangle \qquad \Gamma(sp_B) = \langle B, sptr, E_b\rangle \\ \Delta \vdash E_g = E_b \qquad \Delta \vdash E_g = E_t \\ \Gamma' = \Gamma{+}{+}[sp_G \mapsto \langle G, sptr, (E_g{-}n)\rangle][sp_B \mapsto \langle B, sptr, (E_b{-}n)\rangle] \\ \varsigma' = (E_t{-}n) : ns :: (E_t{-}(n{+}1)) : ns :: \ldots :: E_t : \sigma\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; (E_t : \sigma)) \vdash \ salloc\ n \ \Rightarrow (\Delta; \Gamma'; \overline{(E_d, E_s)}; E_m; \varsigma')} \ (salloc\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(sp_G) = \langle G, sptr, E_g\rangle \qquad \Gamma(sp_B) = \langle B, sptr, E_b\rangle \\ \Delta \vdash E_g = E_b \qquad \Delta \vdash E_g = E_t \\ \varsigma = E_t : t :: \ldots :: E_f : \sigma \qquad \Delta \vdash E_f = E_g + n \\ \Gamma' = \Gamma{+}{+}[sp_G \mapsto \langle G, sptr, (E_g{+}n)\rangle][sp_B \mapsto \langle B, sptr, (E_b{+}n)\rangle]\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ sfree\ n \ \Rightarrow (\Delta; \Gamma'; \overline{(E_d, E_s)}; E_m; E_f : \sigma)} \ (sfree\text{-}t)$$

$$\frac{\Gamma(sp_c) = \langle c, sptr, E_c\rangle \qquad \Delta \vdash E_c + n = E_n \qquad \Delta; \varsigma \vdash E_n : \langle c, b, E\rangle}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ sld_c\ r_d\ n \ \Rightarrow (\Delta; \Gamma{+}{+}[r_d \mapsto \langle c, b, E\rangle]; \overline{(E_d, E_s)}; E_m; \varsigma)} \ (sld_c\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(sp_G) = \langle G, sptr, E_g\rangle \qquad \Gamma(sp_B) = \langle B, sptr, E_b\rangle \qquad \Delta \vdash E_g = E_b \\ \Delta \vdash E_g + n = E_n \qquad \Gamma(r_v) = \langle c, b, E_v\rangle \qquad \Delta \vdash \varsigma[E_n \mapsto \langle c, b, E_v\rangle] = \varsigma'\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ sst\ n\ r_v \ \Rightarrow (\Delta; \Gamma{+}{+}; \overline{(E_d, E_s)}; E_m; \varsigma')} \ (sst\text{-}t)$$

Figure 3.6: Typing Rules for the New Stack Instructions.

$\boxed{\Psi \vdash^{Z} M : \varsigma}$

$$\frac{\cdot \vdash E = \ell \qquad Dom(M) = \{\ell\}}{\Psi \vdash^{Z} M : (E : sbase)} \ (\varsigma\text{-}t\text{-}base)$$

$$\frac{\begin{array}{l} \cdot \vdash (E : t :: \varsigma')\ wf \qquad \cdot \vdash E = \ell \\ M = \{\ell \to n\} \dot{\#} M' \\ \Psi; \cdot \vdash^{Z} n : t \qquad \Psi \vdash^{Z} M' : \varsigma' \end{array}}{\Psi \vdash^{Z} M : (E : t :: \varsigma')} \ (\varsigma\text{-}t\text{-}cons)$$

Figure 3.7: Stack Typing.

3.2.3 Memory Allocation and Initialization

When designing type systems for assembly languages with dynamic allocation, we have to be able to separate allocation and initialization. One common approach is to use initialization flags to track whether a location has been initialized [39]. Newly allocated locations are flagged with a 0, which is changed to 1 the first time something is written to the location. Values can only be loaded from initialized locations.

We will use a similar idea, but must make some modifications for this to work in the context of fault tolerance. The issue occurs because the green and blue computations may have different views of the same location. The store queue allows a lot of flexibility in instruction scheduling, including situations where the green computation has initialized a newly allocated location, but the blue computation has not. The stored value is pending in the queue, and so a green load, which always checks the queue first, may load from the location, but a blue load may not. To solve this, ETAL$_{\text{FT}}$ has three initialization flags instead of the standard two. The added flag $\frac{1}{2}$ means that only the green computation has done the initialization.

New Instruction Syntax and Operational Semantics

To allocate new locations, we add a new instruction $malloc[b]\ r_g\ r_b$ shown in Figure 3.8. It allocates a new location in the heap portion of memory and puts the address of that location in a green register r_g and a blue register r_b. In addition, memory M is extended with the new location. The instruction is also annotated with the basic type b that the location will contain, but this information has no operational effect and is only used by the type system.

The operational rule currently allocates memory sequentially, but one can imagine modifying this rule to implement other allocation algorithms. In all likelihood, *malloc*

$$\textit{instructions} \quad i \quad ::= \quad \ldots \mid malloc[b]\ r_g\ r_b$$

$$\frac{\begin{array}{c} n = max(Dom(M)) + 1 \\ R' = R{+}{+}[r_g \mapsto n][r_b \mapsto n] \end{array}}{(R, C, M, Q, malloc[b]\ r_g\ r_b) \longrightarrow_0 (R', C, (M, n \mapsto 0), Q, \cdot)}\ (malloc)$$

Figure 3.8: New Operation Semantics to Support Dynamic Memory Allocation.

would actually be implemented with a sequence of instructions or a function call with corresponding behavior.

As in MiniC, we will assume the existence of a garbage collector to handle heap deallocation.

Typing Memory Allocation

In order to track the initialization state of heap locations, reference types are updated to contain an initialization flag. As mentioned earlier, 0 means completely uninitialized, and 1 means initialized, and $\frac{1}{2}$ means that only the green computation has performed its initialization.

$$\begin{array}{llll} \textit{initialization flags} & \varphi & ::= & 1 \mid \frac{1}{2} \mid 0 \\ \textit{base types} & b & ::= & \ldots \mid b\ ref^{\varphi} \end{array}$$

There is a subtyping relationship between the different initialization flags. A half initialized location containing a value of type b is a subtype of an uninitialized location with a value of type b. Similarly, a fully initialized location is a subtype of the corresponding half initialized location. Figure 3.9 shows the additional rules added to support these relationships.

$$\boxed{\Psi \vdash n : b}$$

$$\ldots \qquad \frac{\Psi \vdash n : b\ \mathit{ref}^{\varphi} \qquad \varphi \leq \varphi'}{\Psi \vdash n : b\ \mathit{ref}^{\varphi'}}\ (\mathit{addr\text{-}subtp\text{-}t})$$

$$\boxed{\Psi; \Delta \vdash^{Z} n : t}$$

$$\ldots \qquad \frac{\Psi; \Delta \vdash^{Z} n : t' \qquad \Delta \vdash t' \leq t}{\Psi; \Delta \vdash^{Z} n : t}\ (\mathit{val\text{-}subtp\text{-}t})$$

$$\boxed{\varphi \leq \varphi'}$$

$$1 \leq \frac{1}{2} \leq 0 \qquad \varphi \leq \varphi$$

$$\boxed{b \leq b'}$$

$$\ldots \qquad \frac{\varphi \leq \varphi'}{b\ \mathit{ref}^{\varphi} \leq b\ \mathit{ref}^{\varphi'}}\ (\mathit{subtp\text{-}b\text{-}ref})$$

Figure 3.9: Additions to Subtyping to Support Reference Initialization.

Figure 3.10 shows the modifications to the instruction typing judgment. When a new location is allocated to contain a value of type b, it is given type $b\ ref^{0}$, and a fresh expression variable x is selected to represent the newly allocated address. The two destination registers are updated to contain green and blue type triples with the base type b and expression x. The expression E_m that describes memory is updated to map x to 0. In addition to the new rule for typing *malloc*, the typing rules for the existing load and store instructions are also modified. When the green computation wishes to load a value, the location must be at least half initialized. For the blue computation, the location must be fully initialized. The typing rules for the two store instructions are modified to update the initialization flags. When the green computation stores to a location, it uses the function $\varphi\uparrow$ to update the flag. This function changes the uninitialized flag to be half initialized, but does not affect the other flags. The blue computation modifies the initialization flag to be fully initialized. In addition, when one of the store instructions changes the initialization flag on a value, it needs to update the type for that location used by the corresponding computation.

Now that we have initialization flags, we can no longer type of heap portion of memory by itself. We also need information about the queue to check for consistency in the initialization states of each location. The new and modified rules are shown in Figure 3.11.

Judgment $\Psi;M;Q \vdash^{Z} \ell : b\ ref^{\varphi}$ holds for location ℓ when Q and M are consistent with the initialization flag φ. If the initialization flag is 1, then $M(\ell)$ needs to have type b. If the initialization flag is $\frac{1}{2}$, and the zap tag is not green, there must be a pending store to ℓ in the queue. If the initialization flag is 0, nothing is required of M or Q.

$\boxed{\varphi\uparrow}$

$$0\uparrow \;=\; \tfrac{1}{2} \qquad \tfrac{1}{2}\uparrow \;=\; \tfrac{1}{2} \qquad 1\uparrow \;=\; 1$$

$\boxed{\Psi;\Theta \vdash\; ir \;\Rightarrow RT}$

$$\cdots$$

$$\frac{\begin{array}{c} x \notin \Delta \\ \Gamma' = \Gamma{+}{+}[r_g \mapsto \langle G, b\ ref^0, x\rangle][r_b \mapsto \langle B, b\ ref^0, x\rangle] \\ E'_m = upd\ E_m\ x\ 0 \end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\; malloc[b]\ r_g\ r_b \;\Rightarrow (\Delta, x : \kappa_{int}; \Gamma'; \overline{(E_d, E_s)}; E'_m; \varsigma)}\ (malloc\text{-}t)$$

$$\frac{\Delta \vdash \Gamma(r_s) \leq \langle G, b\ ref^{\frac{1}{2}}, E'_s\rangle \qquad E \;=\; sel\ (\overline{upd}\ E_m\ \overline{(E_d, E_s)})\ E'_s}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\; ld_G\ r_d\ r_s \;\Rightarrow (\Delta; \Gamma{+}{+}[r_d \mapsto \langle G, b, E\rangle]; \overline{(E_d, E_s)}; E_m; \varsigma)}\ (ld_G\text{-}t)$$

$$\frac{\Gamma(r_s) = \langle B, b\ ref^1, E'_s\rangle \qquad E \;=\; sel\ E_m\ E'_s}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\; ld_B\ r_d\ r_s \;\Rightarrow (\Delta; \Gamma{+}{+}[r_d \mapsto \langle B, b, E\rangle]; \overline{(E_d, E_s)}; E_m; \varsigma)}\ (ld_B\text{-}t)$$

$$\frac{\begin{array}{c} \Gamma(r_d) = \langle G, b\ ref^{\varphi}, E'_d\rangle \qquad \Gamma(r_s) = \langle G, b, E'_s\rangle \\ \Gamma' = \Gamma{+}{+}\ except\ \forall\ r\ where\ \Gamma(r) = \langle c_r, b\ ref^{\varphi}, E_r\rangle\ and\ \Delta \vdash E_r = E'_d\ . \\ \Gamma'(r) = \langle c_r, b\ ref^{\varphi\uparrow}, E_r\rangle \end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\; st_G\ r_d\ r_s \;\Rightarrow (\Delta; \Gamma'; (E'_d, E'_s), \overline{(E_d, E_s)}; E_m; \varsigma)}\ (st_G\text{-}t)$$

$$\frac{\begin{array}{c} \Delta \vdash \Gamma(r_d) \leq \langle B, b\ ref^{\frac{1}{2}}, E''_d\rangle \qquad \Gamma(r_s) = \langle B, b, E''_s\rangle \\ \Delta \vdash\; E'_s = E''_s \qquad \Delta \vdash\; E'_d = E''_d \\ \Gamma' = \Gamma{+}{+}\ except\ \forall\ r\ where\ \Gamma(r) = \langle c_r, b\ ref^{\frac{1}{2}}, E_r\rangle\ and\ \Delta \vdash E_r = E'_d\ . \\ \Gamma'(r) = \langle c_r, b\ ref^1, E_r\rangle \end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}, (E'_d, E'_s); E_m; \varsigma) \vdash\; st_B\ r_d\ r_s \;\Rightarrow (\Delta; \Gamma'; \overline{(E_d, E_s)}; upd\ E_m\ E'_d\ E'_s; \varsigma)}\ (st_B\text{-}t)$$

Figure 3.10: Instruction Typing Rules to Support Dynamic Memory Allocation.

The queue typing judgment $\Psi \vdash^{Z} Q : \overline{(E_d, E_s)}$ in TAL$_{\text{FT}}$ requires that all addresses in the queue have type *b ref* for some *b* unless the zap tag is green. In addition in ETAL$_{\text{FT}}$, the base type of these locations must be at least half initialized.

The heap typing judgment $\Psi \vdash^{Z} (M, Q) : (E_m, \overline{(E_d, E_s)})$ requires all locations to be consistent with M and Q. In addition, $\overline{(E_d, E_s)}$ describes Q, and E_m describes M.

Finally, the machine state typing judgment is modified so that it divides memory M into the stack portion M_s and the heap portion M_m and calls the new stack typing judgment on M_s and the new heap typing judgment on M_m and Q. The final judgment is shown in Figure 3.12.

3.2.4 Removal of Color Tags

In TAL$_{\text{FT}}$, register values are tagged with the color of the computation to which they belong. The main use of this color tag is in the relation $\Sigma\ sim^{Z}\ \Sigma'$. Because each value is tagged with a color, we can compare two machine states directly without relying on any extra information. This simplifies the judgments quite a bit. However in ETAL$_{\text{FT}}$, we don't have that luxury. Values on the stack are conceptually colored, but those in the heap are not. Adding color tags to memory doesn't make sense for all locations, and in any case, the colors already appear in the typing information. Instead we change the simulation relationship to depend on color information extracted from the typing information. We go ahead and make this change throughout the language, and remove the color tags from all values. This also enforces our earlier claim that the color tags in TAL$_{\text{FT}}$ were only for convenience.

This change requires removing the color tags from all values in the existing rules. The dynamic semantics only propagated the tags, so their removal has no effect. In the static semantics, the color tags are duplicated in the typing information, and so again their

$$\boxed{\Psi; M; Q \vdash^{Z} \ell : b\ \mathit{ref}^{\varphi}}$$

$$\frac{\Psi(\ell) = b\ \mathit{ref}^{1} \qquad \Psi \vdash M(\ell) : b}{\Psi; M; Q \vdash^{Z} \ell : b\ \mathit{ref}^{1}}\ (\mathit{init\text{-}t}) \qquad \frac{\Psi(\ell) = b\ \mathit{ref}^{0}}{\Psi; M; Q \vdash^{Z} \ell : b\ \mathit{ref}^{0}}\ (\mathit{uninit\text{-}t})$$

$$\frac{\begin{array}{c}\Psi(\ell) = b\ \mathit{ref}^{\frac{1}{2}} \\ Z \neq G \implies \exists n.\ (\ell, n) \in Q\end{array}}{\Psi; M; Q \vdash^{Z} \ell : b\ \mathit{ref}^{\frac{1}{2}}}\ (\mathit{halfinit\text{-}t})$$

$$\boxed{\Psi \vdash^{Z}\ Q : \overline{(E_d, E_s)}}$$

$$\ldots$$

$$\frac{\begin{array}{c} Z \neq G \\ \Psi \vdash^{Z}\ \overline{(n'_1, n'_2)} : \overline{(E'_d, E'_s)} \\ \cdot \vdash E_d = n_1 \qquad \cdot \vdash E_s = n_2 \\ \Psi \vdash n1 : b\ \mathit{ref}^{\varphi} \qquad \varphi \leq \frac{1}{2} \qquad \Psi \vdash n2 : b \end{array}}{\Psi \vdash^{Z}\ (n_1, n_2), \overline{(n'_1, n'_2)} : (E_d, E_s), \overline{(E'_d, E'_s)}}\ (\mathit{Q\text{-}t})$$

$$\boxed{\Psi \vdash^{Z}\ (M, Q) : (E_m, \overline{(E_d, E_s)})}$$

$$\frac{\begin{array}{l} \forall \ell \in \mathit{Dom}(M).\ \exists \varphi.\ \Psi; M; Q \vdash^{Z} \ell : b\ \mathit{ref}^{\varphi} \\ [\![E_m]\!] = M \qquad \Psi \vdash^{Z} Q : \overline{(E_d, E_s)} \end{array}}{\Psi \vdash^{Z}\ (M, Q) : (E_m, \overline{(E'_d, E'_s)})}\ (\mathit{heap\text{-}t})$$

Figure 3.11: Changes to Memory Typing Judgments.

$\boxed{\vdash^Z (R, C, M, Q, ir)}$

$$\frac{\begin{array}{l} Dom(\Psi) = Dom(C) \cup Dom(M_m) \\ M = M_s \overset{Dom(C)}{\#} M_m \\ \Psi \vdash C \\ \forall c \neq Z.\ ir \neq \cdot \implies C(R(pc_c)) = ir \\ \forall c \neq Z.\ \Psi(R(pc_c)) = (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \rightarrow void \\ \exists S.\ \cdot \vdash S : \Delta \\ \Psi \vdash^Z M_s : S(\varsigma) \\ \Psi \vdash^Z (M_m, Q) : (S(E_m), S(\overline{(E_d, E_s)})) \\ \Psi \vdash^Z R : S(\Gamma) \end{array}}{\vdash^Z (R, C, M, Q, ir)} \ (\Sigma\text{-}t)$$

Figure 3.12: Changes to the Machine State Typing Judgment.

removal has no effect. Section 3.3.2 will discuss how the simulation relation and formal results are modified.

3.2.5 Other Changes

In addition to the changes above, we added a new move instruction to move the contents of one register into another register. This instruction is not technically necessary, but is helpful to have when generating assembly code.

$$instructions \quad i \quad ::= \quad \ldots \mid mov\ r_d, r_s$$

The TAL$_{\text{FT}}$ rule for Register File Typing required specifying a type for every register. Again, for simplicity of the translation, we drop this requirement. A register file R has type Γ when all types in Γ hold true for R.

$\boxed{\Psi \vdash^{Z} R : \Gamma}$

$$\frac{\begin{array}{c}\forall a \in Dom(\Gamma).\ \Psi; \cdot \vdash^{Z} R(a) : \Gamma(a) \\ \cdot \vdash \Gamma(pc_G) \leq \langle G, int, E_G \rangle \\ \cdot \vdash \Gamma(pc_B) \leq \langle B, int, E_B \rangle \\ \cdot \vdash E_G = E_B\end{array}}{\Psi \vdash^{Z} R : \Gamma}\ (\textit{reg-file-t})$$

The complete specification of ETAL$_{\mathrm{FT}}$ is provided as a reference in Appendix C. Next, we discuss the formal results for these modified rules.

3.3 ETAL$_{\mathrm{FT}}$ Formal Results

Because ETAL$_{\mathrm{FT}}$ contains modifications to TAL$_{\mathrm{FT}}$, all the formal results need to be repeated. The results are summarized below, and Appendix D contains proof sketches of the main lemmas and theorems.

3.3.1 Type Safety

The statements of Progress and Preservation do not change, although many of the cases must be modified. A number of the lemmas used in these proofs are modified, and there are also a number of new lemmas for reasoning about the stack and dynamic memory allocation.

Theorem 5 (Progress)

1. *If* $\vdash \Sigma$ *then* $\Sigma \longrightarrow_0^s \Sigma'$ *and* $\Sigma' \neq \textit{fault}$.

2. *If* $\vdash^{c} \Sigma$ *then* $\Sigma \longrightarrow_0^s \Sigma$.

Theorem 6 (Preservation)

1. *If* $\vdash^Z \Sigma$ *and* $\Sigma \longrightarrow^s_0 \Sigma'$ *and* $\Sigma' \neq$ *fault then* $\vdash^Z \Sigma'$.

2. *If* $\vdash \Sigma$ *and* $\Sigma \longrightarrow^s_1 \Sigma'$ *then* $\exists\, c.\ \vdash^c \Sigma'$.

3.3.2 Fault Tolerance

Due to the removal of color tags, there are more significant changes needed in the formal Fault Tolerance results.

Similarity of Machine States

Now that values do not contain color tags, we need to extract the color of values from the typing information. In order to make as few changes as possible, we divide the judgments into two phases.

First we extract the color of the value in each register and memory location from the typing information and to store this information in a mapping K. The range of K is an *extended color* k which can either be a color c, GB, or *none*. Registers that do not have a corresponding type in Γ are mapped to *none*. Locations in the heap are considered to be GB, as both the green and blue computations have agreed on their values.

$$\begin{array}{llcl} \textit{extended color} & k & ::= & c \mid GB \mid \textit{none} \\ \textit{coloring} & K & ::= & \cdot \mid a \mapsto k \mid \ell \mapsto k \end{array}$$

Extracting the color of a type t is simple. Type triples and conditional types clearly specify the color, and the *ns* type has no color.

$$\begin{array}{lcl} extractK_t(\langle c,b,E\rangle) & = & c \\ extractK_t(E_z = 0 \Rightarrow \langle c,b,E\rangle) & = & c \\ extractK_t(ns) & = & none \end{array}$$

For the register file, those registers given a type by Γ extract the color from that type, and those registers with no typing information have no color.

$$extractK(R,\Gamma) = \forall a \in R.\ a \in Dom(\Gamma)\ \ ?\ \ a \mapsto extractK_t(\Gamma(a))\ \ :\ \ a \mapsto none$$

Similarly for the stack, the color for each location is extracted from the type of that location.

$$extractK_s(M_s,\varsigma) = \forall \ell \in Dom(M_s).\ \ .;\varsigma \vdash \ell : t\ \Rightarrow\ \ell \mapsto extractK_t(t)$$

All locations in the heap have extended color *GB*.

$$extractK_h(M_h) = \forall \ell \in Dom(M_h).\ \ell \mapsto GB$$

The top-level typing judgment $\vdash^Z\ \Sigma$ is modified slightly to $\vdash^Z\ \Sigma : K$, which generates the coloring calculated using the typing information in the premises of the judgment.

Given the coloring information, we modify the TAL$_{\text{FT}}$ simulation relations. Figure 3.13 gives the new judgments. The base judgment $k\ n_1\ sim^Z\ k\ n_2$ is very similar to the simulation of colored values in TAL$_{\text{FT}}$. The main difference is the addition of a rule *sim-val-no-color* that allows any two values with no color to simulate each other regardless

$\boxed{k\ n_1\ sim^Z\ k\ n_2}$

$$\frac{}{k\ n\ sim^Z\ k\ n}\ (sim\text{-}val) \qquad \frac{}{c\ n\ sim^c\ c\ n'}\ (sim\text{-}val\text{-}zap)$$

$$\frac{}{none\ n\ sim^Z\ none\ n'}\ (sim\text{-}val\text{-}no\text{-}color)$$

$\boxed{K \vdash R\ sim^Z\ R'}$

$$\frac{\forall a.\ \ K(a)\ R(a)\ sim^Z\ K(a)\ R'(a)}{K \vdash R\ sim^Z\ R'}\ (sim\text{-}R)$$

$\boxed{K \vdash M\ sim\ ^Z M'}$

$$\frac{\begin{array}{c} Dom(M) = Dom(M') \\ \forall \ell \in Dom(M).\ \ K(\ell)\ M(\ell)\ \ sim^Z\ \ K(\ell)\ M'(\ell) \end{array}}{K \vdash M\ sim^Z\ M'}\ (sim\text{-}M)$$

$\boxed{Q\ sim^Z\ Q'}$

$$\frac{}{\cdot\ sim^Z\ \cdot}\ (sim\text{-}Q\text{-}empty)$$

$$\frac{G\ n_1\ sim^Z\ G\ n_1' \qquad G\ n_2\ sim^Z\ G\ n_2' \qquad Q\ sim^Z\ Q'}{((n_1, n_2), Q)\ \ sim^Z\ ((n_1', n_2'), Q')}\ (sim\text{-}Q)$$

$\boxed{\Sigma_1\ sim^Z\ \Sigma_2}$

$$\frac{\begin{array}{c} \vdash (R, C, M, Q, ir) : K \\ \vdash^Z (R', C, M', Q', ir) : K \\ K \vdash R\ sim^Z\ R' \\ K \vdash M\ sim^Z\ M' \\ Q\ sim^Z\ Q' \end{array}}{(R, C, M, Q, ir)\ sim^Z\ (R', C, M', Q', ir)}\ (sim\text{-}\Sigma)$$

Figure 3.13: Similarity of Machine States.

of the zap tag. This rule is used when a fault affects a register that has no associated type in Γ.

The judgment for register files is now parameterized by a coloring which it uses to determine the color for each register. In addition there is a new rule for memories. Since the coloring is defined differently for stack and heap locations, two memories simulate each other when their heap locations are identical and their stack locations simulate each other under the zap tag. The simulation relationship for queues is unmodified.

Finally, two machine states simulate each other under Z if the first is well-typed under the empty zap tag and the second is well-typed under Z. Both typing judgments must generate the same coloring K, and all elements must simulate each other given K.

Fault Tolerance Theorem

Using these modified similarity relations, we can state and prove the fault tolerance theorem for well-typed programs.

The main difference comes from the modified definition of "observable" behavior that requires the heap to be identical, but allows different values in the stack. We formalize this with the new relationships $s' \overset{\ell}{\simeq} s$ and $s' \overset{\ell}{\preceq} s$ that are used to relate two output sequences of address-value pairs. Because the faulty computation may store faulty values into the stack portion of memory, we can no longer use simple equality to compare address-value pairs. The judgment $\overset{\ell}{\simeq}$ says that the addresses in the two sequences are equal, and for all addresses greater than ℓ, the values are also equal. The judgment $s' \overset{\ell}{\preceq} s$ is similar, but only requires that the locations in the first sequence are a subsequence of those in the second sequence. In other words, when given $max(Dom(C))$ as ℓ, these judgments check that the stores committed to the heap are identical and the stores committed to the stack are to the same locations, though the values may differ.

Theorem 7 (Fault Tolerance)
If $\vdash \Sigma$ *and* $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$ *then either* $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$
or $\exists m \leq (n+1)\ .\ \Sigma \xrightarrow{m}{}^{s'}_{1}$ *fault, and*

1. *For all derivations* $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$ *where* $\Sigma'_f \neq fault$.
   $s' \overset{max(Dom(C))}{\simeq} s$ *and* $\exists\, c.\ \Sigma'\ sim^c\ \Sigma'_f$.

2. *For all derivations* $\Sigma \xrightarrow{m}{}^{s'}_{1}$ *fault where* $m \leq (n+1)$.
   $s' \overset{max(Dom(C))}{\preceq} s$.

3.4 Translation from MiniC to ETAL$_{\mathrm{FT}}$

When designing a sound type system, we need to be careful that the restrictions imposed by the type system are not so stringent as to rule out all interesting programs. This section gives an algorithm for translating all well-typed MiniC programs into ETAL$_{\mathrm{FT}}$. By doing so, we show that the ETAL$_{\mathrm{FT}}$ type system is expressive enough to be of interest.

Optimizing compilers generally generate code using a simple translation that may produce inefficient code, and then they apply optimizations to this code. Similarly, this section gives only a naive translation from MiniC to ETAL$_{\mathrm{FT}}$, and Section 3.6 will sketch the interactions between ETAL$_{\mathrm{FT}}$ and some common optimizations. The rest of this chapter gives an overview of the translation. Appendix E provides the complete rules for the translation and a proof sketch of the Translation Theorem.

3.4.1 Translation Introduction

For simplicity, the translation uses the designated registers pc_G, pc_B, sp_G, sp_B, and gd and then as many fresh temporary registers $t_1, t_2, \ldots$ as needed. Many of these temporary registers can be easily removed by coalescing move instructions. Section 3.6 discusses how to support register allocation if the number of temporary registers is greater than the number of actual registers.

The translation uses a simplistic calling convention. In order to support fault tolerance, all function arguments and return values need to be duplicated. Arguments are passed on the stack, with the last argument pushed on first. There will be two assembly-level arguments for each MiniC argument, and the green argument of each pair is always below the corresponding blue argument on the stack. Below the arguments are the blue and green copies of the return address. When a function returns, it pops the return address and all the arguments and pushes two copies of the return value.

On function entry, each function loads all the arguments into temporary registers. When making a function call, all temporary registers that correspond to local variables are spilled to the stack before the arguments and return address are pushed. After the call returns, the registers for the local variables are restored and then the return values are moved to their destinations.

3.4.2 Translation Details

At a high level, the translation works by passing around a code memory C and continually accumulating new instructions onto the end. In addition, many judgments track the following additional information:

n	*the number of temporary registers required so far.*
V	*a mapping from MiniC variables x to pairs of registers* (r_g, r_b).
B	*a mapping from function variables to addresses in code memory.*

$[\![X \vdash v : \tau]\!]\ C\ n\ V = C'\ n'\ r\ r'$	*Given a value v, generate code to move the translation of v into registers r and r′.*
$[\![X \vdash vs : X]\!]\ C\ n\ V\ n_p = C'\ n'$	*Given a list of values vs to be passed to a function, generate code to push them onto the stack in reverse order.*
$[\![F;A;L \vdash s\ \mathrm{wf}]\!]_f\ C\ n\ V\ B = C'\ n'$	*Given a statement s in function f, generate code for the translation of s.*
$[\![F;A;L \vdash\ lds : L']\!]\ C\ n\ V = C'\ n'\ V'$	*Given a list of local declarations lds, generate code to initialize them and modify the variable map V to contain the new mappings.*
$[\![F \vdash\ fds : F']\!]\ C\ B\ =\ C'\ B'\ n$	*Given a list of function declarations fds, generate code for each function and modify the function mapping B to contain the starting address for each function. n is the maximum number of temporary registers used by any one function.*
$[\![\vdash\ p\ \ \mathrm{wf}]\!] =\ C\ n\ l$	*Given a program p, generate code for p, the maximum number of registers n required by p, and the starting address l.*

Figure 3.14: Summary of Translation Judgments.

The translation is defined over the typing rules for MiniC. Figure 3.14 summarizes the program translation judgments. The eventual goal is to show that all well-typed MiniC programs can be translated into well-typed ETAL$_{\text{FT}}$ programs, so the translation uses MiniC typing information to generate the corresponding ETAL$_{\text{FT}}$ typing information. The assorted type translation judgments are shown in Figure 3.15.

$[\![\tau]\!] = b$	*The translation of MiniC type* τ *is* ETAL$_{\mathrm{FT}}$ *basic type b.*
$[\![A]\!] = \varsigma$	*The MiniC arguments A correspond to a stack type* ς.
$[\![A;\varsigma]\!]_V = \Gamma$	*Given MiniC arguments A and their translation into stack type* ς*, calculate the register file type* Γ *that results after loading each argument into its corresponding register.*
$[\![X]\!]_V = \Gamma$	*Given a variable context X and variable mapping V, generate the corresponding register file type* Γ.
$[\![A \to \tau; L]\!]_V = \Theta$	*Given a function type* $A \to \tau$*, local variables L, and variable mapping V, generate the current precondition* Θ.

Figure 3.15: Summary of Type Translation Judgments.

3.4.3 Translation Formal Results

The Translation Theorem states that the result of translating a well-typed program p can be used to create a well-typed ETAL$_{\mathrm{FT}}$ machine state. The code memory in this machine state is just the code memory returned by the translation. Because ETAL$_{\mathrm{FT}}$ has no true notion of termination, programs "complete" by jumping to a designated address l_{halt} which contains a short code snippet for an infinite loop. The heap is empty, and the stack contains two pointers to l_{halt}. The register file is built by calling the function $buildR(n)$ to generate a blank register with n temporary registers. The two program counters are set to the start address generated by the translation, and the stack pointers are set to the top of the stack.

Theorem 8 (Translation)

If $[\![\vdash\ p\ \ \mathrm{wf}]\!] = C\ n\ l_{start}$ *then* $\vdash (R, C, M, (), \cdot)$

$$\begin{array}{lrcl} where & l_\varsigma & = & min(Dom(C) - 3) \\ & R & = & buildR(n),\ pc_G \mapsto l_{start},\ pc_B \mapsto l_{start},\ sp_G \mapsto l_\varsigma,\ sp_B \mapsto l_\varsigma,\ gd \mapsto 0 \\ & M & = & l_\varsigma \mapsto l_{halt},\ l_\varsigma + 1 \mapsto l_{halt},\ l_\varsigma + 1 \mapsto 0 \end{array}$$

Again, the complete translation and corresponding proof sketches are provided in Appendix E.

3.5 Translation Example

Figure 3.16 shows the result of applying the naive translation to the MiniC example program from the beginning of this chapter.

Clearly, this translation is not efficient. But again, that is not yet the point. The generated code is well-typed. Let us walk through the translation of the `double()` function to get an idea how this works.

We use the type translation $[\![\tau]\!]$ (defined in Appendix E.2.1) to generate the type for the first instruction in the translation of a function. Function `double()` has type $x : \mathit{int\ ref} \rightarrow \mathit{int}$, and $[\![x : \mathit{int\ ref} \rightarrow \mathit{int}]\!]$ gives us the type below. Though it looks complicated, it is just laying out the calling convention discussed earlier. When entering this function, the stack should have two copies of the argument and then two copies of the return address. The rest of the stack is unknown. The stack pointers should point to the top of the stack, the program counters are equal, and the special destination register *gd* must be zero (meaning that no control flow transfers are in progress). When the function returns, it should leave things in a state satisfying Θ_r. In other words, it needs to remove the return address and arguments from the stack and push two copies of the return value.

```
// original MiniC program
int double(x:int ref) {
  int v = 0;
  v = !x;
  v = v + v;
  x := v;
  return 0;
}

int ref y = ref 3;
int result = 0;

result = double(y);
return result;

// infinite loop
50  mov t_0 1
51  mov t_1 1
52  jmp_G t_1
53  jmp_B t_1

// double()
54  sld_G 2 t_0     // prologue
55  sld_B 3 t_1
56  mov t_2 0       // int v = 0
57  mov t_3 0
58  mov t_4 t_2
59  mov t_5 t_3
60  ld_G t_4 t_0    // v = !x
61  ld_B t_5 t_1
62  add t_4 t_4 t_4 // v = v + v
63  add t_5 t_5 t_5
64  st_G t_0 t_4    // x := v
65  st_B t_1 t_5
66  mov t_6 0       // get ret val
67  mov t_7 0
68  sld_G 0 t_8     // epilogue
69  sld_B 1 t_9
70  sfree 4
71  salloc 2
72  sst 0 t_6
73  sst 1 t_7
74  jmp_G t_8
75  jmp_B t_9
```

```
// main()
76   mov t_0 3    // int ref y = ref 3
77   mov t_1 3
78   malloc[int] t_2 t_3
79   st_G t_2 t_0
80   st_B t_3 t_1
81   mov t_4 t_2
82   mov t_5 t_3
83   mov t_6 0    // int result = 0
84   mov t_7 0
85   mov t_8 t_6
86   mov t_9 t_7
87   salloc 8     // result = double(y)
88   sst 4 t_4       - st local vars/args
89   sst 5 t_5
90   sst 6 t_8
91   sst 7 t_9
92   sst 2 t_4       - st args
93   sst 3 t_5
94   mov t_10 50     - st ret addr
95   sst 0 t_10
96   mov t_11 50
97   sst 1 t_11
98   mov t_12 54     - jmp to double()
99   mov t_13 54
100  jmp_G t_12
101  jmp_B t_13
102  sld_G 2 t_4     - reload local vars
103  sld_B 3 t_5
104  sld_G 4 t_6
105  sld_B 5 t_7
106  sld_G 0 t_8     - set result
107  sld_B 1 t_9
108  sfree 6         - free call space
109  mov t_14 0   // get ret val
110  mov t_15 0
111  sld_G 0 t_16 // epilogue
112  sld_B 1 t_17
113  sfree 2
114  salloc 2
115  sst 0 t_14
116  sst 0 t_15
117  jmp_G t_16
118  jmp_B t_17
```

Figure 3.16: Example Translation from MiniC to ETAL$_{\mathrm{FT}}$.

$$\begin{array}{llll}
\multicolumn{4}{l}{\Psi(54) = (\Delta_d, \Gamma_d, (), \alpha_m, \varsigma_d) \rightarrow void} \\
\text{where:} & \Delta_d & = & \alpha_p : \kappa_{int}, \alpha_m : \kappa_{mem}, \alpha_\ell : \kappa_{int}, \alpha_\sigma : \kappa_\sigma, \alpha_r : \kappa_{int}, \alpha_x : \kappa_{int} \\
& \Gamma_d & = & pc_G \mapsto \langle G, int, \alpha_p \rangle, pc_B \mapsto \langle B, int, \alpha_p \rangle, \\
& & & sp_G \mapsto \langle G, sptr, \alpha_\ell - 4 \rangle, sp_B \mapsto \langle B, sptr, \alpha_\ell - 4 \rangle, \\
& & & gd \mapsto \langle G, int, 0 \rangle \\
& \varsigma_d & = & \alpha_\ell - 4 : \langle G, \Theta_r \rightarrow void, \alpha_r \rangle \ :: \ \alpha_\ell - 3 : \langle B, \Theta_r \rightarrow void, \alpha_r \rangle \\
& & & \ :: \ \alpha_\ell - 2 : \langle G, int\ ref^1, \alpha_x \rangle \ :: \ \alpha_\ell - 1 : \langle B, int\ ref^1, \alpha_x \rangle \\
& & & \ :: \ \alpha_\ell : \alpha_\sigma \\
\\
\text{and:} & \Theta_r & = & (\Delta_r, \Gamma_r, (), \alpha'_m, \varsigma_r) \\
& \Delta_r & = & \alpha'_m : \kappa_{mem}, \alpha_\tau : \kappa_{int} \\
& \Gamma_r & = & pc_G \mapsto \langle G, int, \alpha_r \rangle, pc_B \mapsto \langle B, int, \alpha_r \rangle, \\
& & & sp_G \mapsto \langle G, sptr, \alpha_\ell - 2 \rangle, sp_B \mapsto \langle B, sptr, \alpha_\ell - 2 \rangle, \\
& & & gd \mapsto \langle G, int, 0 \rangle \\
& \varsigma_d & = & \alpha_\ell - 2 : \langle G, int, \alpha_\tau \rangle \ :: \ \alpha_\ell - 1 : \langle B, int, \alpha_\tau \rangle \\
& & & \ :: \ \alpha_\ell : \alpha_\sigma
\end{array}$$

```
54  sld_G 2 t_0
```

After executing the first stack load, the next instruction has a similar type except that the type of the destination register used in the load is updated to contain the same type as the stack slots that the value was loaded from.

$$\Psi(55) = (\Delta_d, \Gamma_d[t_0 \mapsto \langle G, int\ ref^1, \alpha_x \rangle], (), \alpha_m, \varsigma_d) \rightarrow void$$

```
55  sld_B 3 t_1
```

Executing the second load has a similar effect.

$$\Psi(56) = (\Delta_d, \Gamma_d[t_0 \mapsto \langle G, int\ ref^1, \alpha_x \rangle][t_1 \mapsto \langle B, int\ ref^1, \alpha_x \rangle], (), \alpha_m, \varsigma_d) \rightarrow void$$

```
56  mov t_2 0
57  mov t_3 0
58  mov t_4 t_2
59  mov t_5 t_3
```

This sequence of move instructions adds typing information about the temporary registers to the register file type.

$$\begin{array}{l} \Psi(60) = (\Delta_d, \Gamma_{60}, (), \alpha_m, \varsigma_d) \rightarrow \mathit{void} \\ \quad \text{where:} \quad \Gamma_{60} \;=\; \Gamma_d \; \begin{array}[t]{l} [\, t_0 \mapsto \langle G, \mathit{int\ ref}^1, \alpha_x \rangle \,] \\ [\, t_1 \mapsto \langle B, \mathit{int\ ref}^1, \alpha_x \rangle \,] \\ [\, t_2 \mapsto \langle G, \mathit{int}, 0 \rangle \,] \\ [\, t_3 \mapsto \langle B, \mathit{int}, 0 \rangle \,] \\ [\, t_4 \mapsto \langle G, \mathit{int}, 0 \rangle \,] \\ [\, t_5 \mapsto \langle B, \mathit{int}, 0 \rangle \,] \end{array} \end{array}$$

```
60  ld_G t_4 t_0
61  ld_B t_5 t_1
```

Each of these load instructions loads from some address α_x in memory α_m, so the type of the loaded values can be described by the static expression $\mathit{sel}\ \alpha_m\ \alpha_x$.

$$\begin{array}{l} \Psi(62) = (\Delta_d, \Gamma_{62}, (), \alpha_m, \varsigma_d) \rightarrow \mathit{void} \\ \quad \text{where:} \quad \Gamma_{62} \;=\; \Gamma_{60} \; \begin{array}[t]{l} [\, t_4 \mapsto \langle G, \mathit{int}, \ \mathit{sel}\ \alpha_m\ \alpha_x \rangle \,] \\ [\, t_5 \mapsto \langle B, \mathit{int}, \ \mathit{sel}\ \alpha_m\ \alpha_x \rangle \,] \end{array} \end{array}$$

```
62  add t_4 t_4 t_4
63  add t_5 t_5 t_5
```

When the value is added to itself, the static expressions are modified accordingly. At this point, we know that t_4 and t_5 both contain double the value in address α_x, whatever that value may be.

$$\begin{array}{l} \Psi(64) = (\Delta_d, \Gamma_{64}, (), \alpha_m, \varsigma_d) \rightarrow \mathit{void} \\ \quad \text{where:} \quad \Gamma_{64} \;=\; \Gamma_{62} \; \begin{array}[t]{l} [\, t_4 \mapsto \langle G, \mathit{int}, \ (\mathit{sel}\ \alpha_m\ \alpha_x) + (\mathit{sel}\ \alpha_m\ \alpha_x) \rangle \,] \\ [\, t_5 \mapsto \langle B, \mathit{int}, \ (\mathit{sel}\ \alpha_m\ \alpha_x) + (\mathit{sel}\ \alpha_m\ \alpha_x) \rangle \,] \end{array} \end{array}$$

```
64  st_G t_0 t_4
65  st_B t_1 t_5
```

The green store typing rule will add the pair $(\alpha_x, (\mathit{sel}\ \alpha_m\ \alpha_x) + (\mathit{sel}\ \alpha_m\ \alpha_x))$ to the queue. The blue store typing rule removes this pair from the queue description, enforces that it is equal to the expressions describing the operands, and updates the type of memory.

$$\begin{array}{l} \Psi(66) = (\Delta_d, \Gamma_{64}, (), E'_m, \varsigma_d) \rightarrow \mathit{void} \\ \quad \text{where:} \quad E'_m \;=\; \mathit{upd}\ \alpha_m\ \alpha_x\ ((\mathit{sel}\ \alpha_m\ \alpha_x) + (\mathit{sel}\ \alpha_m\ \alpha_x)) \end{array}$$

```
66  mov t6 0
67  mov t7 0
68  sldG 0 t8
69  sldB 1 t9
```

Again, these instructions add more temporary register typing information to the register file type.

$$\Psi(70) = (\Delta_d, \Gamma_{70}, (), E'_m, \varsigma_d) \rightarrow \textit{void}$$
$$\begin{array}{llll} \text{where:} & \Gamma_{70} = & \Gamma_{64} & [\, t_6 \mapsto \langle G, int, 0\rangle \,] \\ & & & [\, t_7 \mapsto \langle B, int, 0\rangle \,] \\ & & & [\, t_8 \mapsto \langle G, \Theta_r \rightarrow void, \alpha_r\rangle \,] \\ & & & [\, t_9 \mapsto \langle B, \Theta_r \rightarrow void, \alpha_r\rangle \,] \end{array}$$

```
70  sfree 4
71  salloc 2
```

The $sfree$ instruction modifies the stack type to be $\alpha_\ell : \alpha_\sigma$ and updates the stack pointer types appropriately. Then the *salloc* instruction pushes two nonsense types onto the stack type and again updates the stack pointer types.

$$\Psi(72) = (\Delta_d, \Gamma_{72}, (), E'_m, \varsigma_{72}) \rightarrow \textit{void}$$
$$\begin{array}{llll} \text{where:} & \varsigma_{72} = & \multicolumn{2}{l}{\alpha_\ell - 2 : ns \;::\; \alpha_\ell - 1 : ns \;::\; \alpha_\ell : \alpha_\sigma} \\ & \Gamma_{72} = & \Gamma_{70} & [\, sp_G \mapsto \langle G, sptr, \alpha_\ell - 2\rangle \,] \\ & & & [\, sp_B \mapsto \langle B, sptr, \alpha_\ell - 2\rangle \,] \end{array}$$

```
72  sst 0 t6
73  sst 1 t7
```

The two stack store instructions overwrite the nonsense types in the stack type with the types of registers t_6 and t_7.

$$\Psi(74) = (\Delta_d, \Gamma_{70}, (), E'_m, \varsigma_{74}) \rightarrow \textit{void}$$
$$\text{where:} \quad \varsigma_{74} = \alpha_\ell - 2 : \langle G, int, 0\rangle \;::\; \alpha_\ell - 1 : \langle B, int, 0\rangle \;::\; \alpha_\ell : \alpha_\sigma$$

```
74  jmpG t8
```

The green jump instruction updates the type of the destination register *gd*. For clarity, the type of location 75 is fully expanded. The subscript *c* appears on the current contexts, and the subscript *r* appears on those describing the type of the return address.

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\Psi(75) = (\Delta_c, \Gamma_c, (), E_{mc}, \varsigma_c) \to void} \\
\text{where:} & \Delta_c & = & \alpha_p : \kappa_{int}, \alpha_m : \kappa_{mem}, \alpha_\ell : \kappa_{int}, \alpha_\sigma : \kappa_\sigma, \alpha_r : \kappa_{int}, \alpha_x : \kappa_{int} \\
 & \Gamma_c & = & pc_G \mapsto \langle G, int, \alpha_p \rangle,\ pc_B \mapsto \langle B, int, \alpha_p \rangle, \\
 & & & sp_G \mapsto \langle G, sptr, \alpha_\ell - 2 \rangle,\ sp_B \mapsto \langle B, sptr, \alpha_\ell - 2 \rangle, \\
 & & & gd \mapsto \langle G, \Theta_r \to void, \alpha_r \rangle, \\
 & & & t_0 \mapsto \langle G, int\ ref^1, \alpha_x \rangle,\ t_1 \mapsto \langle B, int\ ref^1, \alpha_x \rangle, \\
 & & & t_2 \mapsto \langle G, int, 0 \rangle,\ t_3 \mapsto \langle B, int, 0 \rangle, \\
 & & & t_4 \mapsto \langle G, int, (sel\ \alpha_m\ \alpha_x) + (sel\ \alpha_m\ \alpha_x) \rangle, \\
 & & & t_5 \mapsto \langle B, int, (sel\ \alpha_m\ \alpha_x) + (sel\ \alpha_m\ \alpha_x) \rangle, \\
 & & & t_6 \mapsto \langle G, int, 0 \rangle,\ t_7 \mapsto \langle B, int, 0 \rangle, \\
 & & & t_8 \mapsto \langle G, \Theta_r \to void, \alpha_r \rangle,\ t_9 \mapsto \langle B, \Theta_r \to void, \alpha_r \rangle, \\
 & \varsigma_c & = & \alpha_\ell - 2 : \langle G, int, 0 \rangle\ ::\ \alpha_\ell - 1 : \langle B, int, 0 \rangle\ ::\ \alpha_\ell : \alpha_\sigma \\
 & E_{mc} & = & upd\ \alpha_m\ \alpha_x\ ((sel\ \alpha_m\ \alpha_x) + (sel\ \alpha_m\ \alpha_x)) \\
 & & & \\
\text{and:} & \Theta_r & = & (\Delta_r, \Gamma_r, (), \alpha'_m, \varsigma_r) \\
 & \Delta_r & = & \alpha'_m : \kappa_{mem}, \alpha_\tau : \kappa_{int} \\
 & \Gamma_r & = & pc_G \mapsto \langle G, int, \alpha_r \rangle, pc_B \mapsto \langle B, int, \alpha_r \rangle, \\
 & & & sp_G \mapsto \langle G, sptr, \alpha_\ell - 2 \rangle, sp_B \mapsto \langle B, sptr, \alpha_\ell - 2 \rangle, \\
 & & & gd \mapsto \langle G, int, 0 \rangle \\
 & \varsigma_d & = & \alpha_\ell - 2 : \langle G, int, \alpha_\tau \rangle\ ::\ \alpha_\ell - 1 : \langle B, int, \alpha_\tau \rangle \\
 & & & \quad ::\ \alpha_\ell : \alpha_\sigma
\end{array}
$$

```
75  jmpB t9
```

Type checking the blue jump instruction is more involved. There are 11 premises which must be satisfied. The first three require that the destination register and the jump target both have the same code type and are described by equivalent expressions.

1. $\Gamma_c(gd) = \langle G, (\Delta_r; \Gamma_r; (); \alpha'_m, \varsigma_r) \to void, \alpha_r \rangle$
2. $\Gamma_c(r_d) = \langle B, (\Delta_r; \Gamma_r; (); \alpha'_m, \varsigma_r) \to void, \alpha_r \rangle$
3. $\Delta_c \vdash\ \alpha_r = \alpha_r$

There exists a substitution S that gives expressions to substitute in for the expression variables in Δ_r. These expressions may contain free variables from Δ_c. We can construct such a substitution by inspecting the current typing information and the typing information required by the return address.

4. $\exists S.\ \Delta_c \vdash\ S : \Delta_r \qquad (\ let\ S = E_{mc}/\alpha'_m,\ 0/\alpha_\tau\ )$

The return address has appropriate requirements for the designated registers gd, pc_G and pc_B.

5. $S(\Gamma_r)(gd) = \langle G, int, 0\rangle$
6. $S(\Gamma_r)(pc_G) = \langle G, int, \alpha_r\rangle$
7. $S(\Gamma_r)(pc_B) = \langle B, int, \alpha_r\rangle$

And finally, the current register file type, queue description, memory description, and stack type are subtypes of the substituted equivalents for the return address.

8. $\Delta_c \vdash \Gamma_c \leq\ S(\Gamma_r)$
9. $\Delta_c \vdash\ () = S(\ ()\ )$
10. $\Delta_c \vdash\ E_{mc} = S(\alpha'_m)$
11. $\Delta_c \vdash\ \varsigma_c \leq S(\varsigma_r)$

At this point, we have shown that each instruction in the translation of the `double()` function results in a type that can be used to type check the next instruction. In other words, this sequence of instructions can be added to a well-typed code memory, and the resulting code memory will also be well-typed.

3.6 Type-preserving Optimizations

In this section, we discuss the effects of the type system on various common optimizations. These are the sorts of optimizations a compiler may apply to a low-level intermediate representation, which is very close to executable code. So by discussing how to support these optimizations, we are arguing that ETAL$_{\text{FT}}$ can be used for the final result of compilation.

Previous work [66, 62, 14] has demonstrated how to implement many common optimizations for a non-fault-tolerant typed assembly language. Below, we will sketch the effects of the additional typing infrastructure for supporting fault tolerance on some common optimizations. However, a full analysis of the impact of the type system on various optimizations is beyond the scope of this thesis.

3.6.1 General Considerations

Unfortunately for optimizations, ETAL$_{\text{FT}}$ works with actual code addresses instead of symbolic labels. Therefore inserting or removing instructions requires incrementing or decrementing all uses of subsequent code addresses. Though cumbersome, it is possible to deal with this. Alternatively, one could make minor modifications to ETAL$_{\text{FT}}$ in order to use a different code representation more similar to a control flow graph.

One important invariant that must be maintained is the equivalence of related static expressions. This equivalence is abstracted in the typing rules by the judgement $\Delta \vdash E_1 = E_2$. In the translation in Section 3.4, the two computations are exactly the same, and so syntactic equality is all that is required. Using a more powerful theorem prover to compare expressions allows flexibility in how an optimization is applied. For example, assume we have code to multiply a value by two. It is possible to optimize this code to use addition instead of costly multiplication. If this optimization is only applied to the green computation, the end result will be two values with type $\langle G, int, E+E\rangle$ and $\langle B, int, 2*E\rangle$. The type checker will allow this as long as the theorem prover used can determine that $E+E = 2*E$.

3.6.2 Removal of Redundant Moves

If a move exists between two registers in well-typed code, then we know the registers must belong to the same computation. Therefore redundant moves will only occur within a single computation, and so removing them will continue to maintain the separation between colored computations.

3.6.3 Register Allocation

The most obvious issue with the ETAL$_{\text{FT}}$ translation is that it is constantly generating fresh temporary registers. Removing redundant moves will decrease the number of temporary registers, but for some programs the number of live temporary registers will exceed the number of actual registers.

We have already shown how to add a stack into ETAL$_{\text{FT}}$. Once other optimizations have been completed, and compiler can determine a graph coloring and modify the code to spill the selected values to the stack.

In the overly simplistic translation, the stack is only accessed upon entering and exiting functions and before and after function calls. However, there is nothing in the typing rules to prevent stack operations at any point in the code, so the implementation of register allocation should be able to proceed in the normal way.

3.6.4 Common Subexpression Elimination

Common subexpressions elimination is one of the optimizations that caused issues in the initial implementation of the SWIFT compiler. Naively performing CSE on ETAL$_{\text{FT}}$ code would likely cause dependencies between the two computations, resulting in code that would fail to type check. However, it is simple to solve this by modifying CSE to be aware of the color tags. As long as the optimization only applies within a computation of a single color, there appear to be no additional issues.

3.6.5 Dead Code Elimination

In general, dead code elimination is not affected by the addition of fault tolerance to the type system.

However, if dead stores are removed, it is imperative that these same modifications are made to each computation. Otherwise, the queue will be inconsistent, and so the code will fail to type check. (The same holds for the removal of silent stores, which are stores that overwrite a value with the same value.)

3.6.6 Constant Folding and Propagation

In constant folding and propagation, the constant has the same type as the expression it is replacing, and so these optimizations do not cause any issues in ETAL$_{\text{FT}}$ as long as the theorem prover can prove the equivalence. It is worth noting that a number of the instructions only take registers as operands (for example $add\ r_1\ r_2\ r_3$), but adding other addressing modes for instructions would only require simple changes.

3.6.7 Stack Packing

In a stack packing optimization, the compiler reuses the same stack slot for two values with disjoint lifetimes. Because ETAL$_{\text{FT}}$ uses strong updates when storing to the stack, this can be supported. The information about where a stack pointer points is separate from the information about what each stack location contains. So even if one register is used to perform a strong update on a stack pointer, any other registers that also point to this location will be aware of the change.

3.6.8 Instruction Scheduling

As with TAL$_{\text{FT}}$, ETAL$_{\text{FT}}$ allows a lot of flexibility in the instruction scheduling. Within a computation, all the usual reorderings are allowed, given the standard constraints about instruction dependencies.

There is a great deal of flexibility between the green and blue computations as well. Essentially, computations are synchronized at the block level. In other words, the green computation may not execute past a control flow transfer until the blue computation has also executed the transfer. Within a block, the main restriction is that each green store instruction comes before the corresponding blue store instruction. Though our machine model assumes an infinite length store queue, in reality this would be implemented as a fixed-length buffer. This will require the compiler to limit the number of green store instructions that can come before the first of the corresponding blue instructions.

The new instructions in ETAL$_{\text{FT}}$ such as $malloc$ and *sst* would likely be implemented with sequences of instructions, which may cause unanticipated interactions with scheduling. For example, if *sst* is implemented using st_G and st_B, then it cannot occur while there is a green store pending in the queue. Previous work on LTAL [16] develops a standard typed assembly language without macros to avoid interference with scheduling. Investigating the interaction of equivalent ideas with fault tolerance is left as future work.

3.7 Summary

This chapter has presented ETAL$_{\text{FT}}$, a version of TAL$_{\text{FT}}$ extended with support for stacks and dynamic memory allocation. We show how to compile a simple imperative source language into fault-tolerant assembly code. This demonstrates that the restrictions imposed by the type system do not affect our ability to express interesting programs. In addition, we discuss how ETAL$_{\text{FT}}$ can support common low-level optimizations, thereby showing that it can be used as the final language in a realistic compiler.

Chapter 4

$\mathrm{TAL}_{\mathrm{CF}}$: Reasoning about Control Flow

The previous two chapters focused on a *hybrid* transient fault solution that included specialized hardware to help detect faults. Although this type of the solution is extremely promising, there are many cases in which additional hardware is not available. Fortunately, techniques based only on software can detect a large number of faults on off-the-shelf processors.

Chapter 5 covers related work on existing software-only techniques, but the general methodology is similar to what we have already seen, though the comparisons are performed by additional instructions in software. Doing so sometimes leaves a *window of vulnerability* in which faults may not be detected before affecting the observable behavior of the program.

In particular, this chapter focuses on reasoning about a software solution for detecting control flow faults, which occur when a transient fault causes control to jump to an

Material in this chapter is joint work with David Walker and has been previously published as [54] and [53].

unexpected code address. TAL$_{FT}$ avoids the issue by using a hardware comparison in the fetch stage and control-flow instructions to detect such faults.

Researchers [47, 60, 10] have developed techniques for detecting many, but not all, classes of control-flow errors entirely in software. Though these techniques are promising, they have not been proven sound and many theoretical questions remain. In particular, is it possible to characterize the effectiveness of these techniques *analytically* as opposed to empirically? In other words, can we prove that such techniques are sound with respect to an interesting and nontrivial, though incomplete, fault model? One of the key benefits of such an analysis is that it would guarantee that an important fragment of the problem has been thoroughly solved and thereby allow researchers to study auxiliary instrumentation techniques that address the remaining incompleteness. Perhaps more importantly, a formal fault model and proof of soundness would define an important hardware/software interface: The software has been proven to handle faults that lie within the model; future hardware designers need only provide mechanisms to catch those faults that lie outside the model. Such results would show how to shift a substantial portion of the control-flow checking burden from the hardware to software. This may lead to simpler hardware designs as well as the opportunity to trade performance for reliability at *compile time* as opposed to *hardware design time*.

This chapter presents TAL$_{CF}$, a type system for reasoning about a software-only technique for detecting a specific class of control-flow faults. The technique instruments each basic block with a certain instructions that implement a fault tolerance protocol. From a technical perspective, the type system introduces a novel way of classifying the reliability properties of program values and entire machine states, generalizing the earlier "color systems" used by λ_{zap} [73] and TAL$_{FT}$. The type system is also of interest for the way it uses a collection of abstract types to track the state of the fault tolerance protocol. The key technical challenge in reasoning about such a solution is the fact that after a control-flow fault has occurred, it is impossible to count on almost *any* standard program invariant. So, how can one carry out a proof of type preservation under such circumstances?

The rest of the chapter is organized as follows. First, Section 4.1 gives additional intuition about the problem and solution by explaining a simple assembly-language protocol for detecting control-flow errors. This protocol is a simplified version of the protocols used by Oh [47] and Reis [60]. Section 4.2 begins the more formal work by defining the syntax and operational semantics of an idealized assembly language that includes rules to model transient faults. Section 4.3 defines the type system that guarantees that assembly code follows the simple protocol outlined earlier in Section 4.1. This type system, particularly the special value and machine state typing rules, codifies the major invariants needed to prove the subsequent type safety and strong reliability properties. Section 4.4 sketches the major components of the type safety and fault-tolerance proofs. Section 4.5 shows that our typed assembly language is sufficiently expressive to translate basic "while" programs into well-typed, fault-tolerant code. Finally, Section 4.6 summarizes.

4.1 Informal Overview

When a transient fault causes the actual sequence of control flow blocks visited by a program to deviate from the expected sequence, we say a control-flow error has occurred. In our model, control-flow errors arise in three different ways: (1) there may be a fault to the target address of a jump instruction; (2) there may be a fault to the target address of a conditional jump instruction; or (3) there may be a fault to the boolean used to decide whether to jump or fall through a conditional. Such faults may occur immediately prior to attempting the control-flow transfer or at any other time during the computation. However, whenever a control-flow operation is executed, we assume execution is either transferred to the beginning of some valid block, or to some invalid block or illegal instruction. In the latter case, we assume the hardware immediately catches an attempt to execute the illegal instruction. We do not consider the possibility that a fault causes a control flow transfer to a legal instruction in the middle of some valid block. We discuss this limitation in Section 4.6.

As in previous chapters, we adhere to the *Single Event Upset* model, which states that only one fault may occur during an execution. However, even though just one fault occurs, faulty values may be copied, propagated and used in any way an ordinary value may be used. Hence, a single fault can lead to arbitrarily many corrupted values if not caught soon after it occurs.

The goal of this work is to develop and prove correct a software protocol that guarantees such control-flow errors can never go undetected. The central challenge in this endeavor is to overcome the problem that *no single value can ever be trusted to be correct* — a transient fault may strike any value in any register. Consequently, as is usual in fault tolerance, the solution is to avoid relying on any single value by replicating the critical state and checking replicas against one another. In this case, the critical state is the value of the program counter. Checking the correctness of a control-flow transfer involves creating a replica of the intended control-flow destination and then checking the replica against the real program counter to detect any difference.

To be more specific, compiled code creates the replica prior to any control-flow transfer by moving the intended destination into a designated register. We refer to this register as the *intention register* `ri`. This intention register is part of the global "calling convention" for fault-tolerant control flow transfers. We fix the register so that all jump targets know where to find the intended destination, even when there has been a control-flow fault.

As an example, to jump to address `L2`, one might use the following code sequence. In this code, we leave ellipsis in between instructions to emphasize our system allows flexible scheduling of instructions — ordinary instructions may be interleaved with the instructions used to guarantee fault tolerance.

```
L1: ...; movi ri, L2; ...; movi r2, L2; ...; jmp r2;
```

Because the intention register `ri` plays a special role in the protocol for detecting control-flow errors, we will need to type-check the move instruction that loads this register in a special way. To designate the move as special, we henceforth write it `intend L2` rather than `movi ri, L2` as in the following example code.

```
L1: ...; intend L2; ...; movi r2, L2; ...; jmp r2
```

If the intention register has been set properly prior to all jump instructions, the jump targets are able to catch control flow errors. To be specific, all jump targets should be instrumented with the following code.

```
Lk: movi r2, Lk; ...; sub r2, r2, ri; ...; brnz r2, lrecover; ...
```

Here, the current block address `Lk` is loaded into some register `r2`. That register is then compared with the contents of `ri` and if there is any difference, control is transferred to `lrecover`, an address containing recovery code. Once again, since the branch to the recovery code plays a special role in the fault-tolerance protocol, we give it the special syntax `recovernz r2`. Thus, our detection code will henceforth be written as follows.

```
L2: movi r2, L2; ...; sub r2, r2, ri; ...; recovernz r2; ...
```

As an example of how a transient fault might be caught using our protocol, suppose register `r2` is corrupted just prior to attempting to execute the jump to `L2` in block `L1`. Upon arrival at some erroneous control flow block, say `L3`, the intended destination `L2` remains safely untouched in register `ri`, though, unnervingly, all other program invariants may be disrupted. The target code compares the contents of `ri` (*i.e.*, `L2`) with `L3`, which it loaded into `r2` after arriving at the current block. It detects a difference and jumps to the recovery code.

One must also consider what happens if faults strike at different times or in different places. For instance, the jump target might have been corrupted much earlier than we suggested above, perhaps just after being initially loaded into `r2`, instead of just prior to the jump. Will that make a difference? In this case, no. Likewise, `ri` might be corrupted, either before or after jumping. In this case, we reach the correct destination, but it appears as though there was a fault because `ri` differs from the current block label (assuming the fault occurs prior to the subtraction). Unable to tell the difference between a fault in the intention register and a fault in the control-flow transfer itself, we jump to recovery code. A number of other scenarios must also be analyzed — in order to have confidence in the solution, one must do so in a principled, disciplined fashion.

It is important to observe that similar, but subtly different code sequences do not adequately protect against faults. In particular, optimizations like copy propagation, common subexpression elimination and some code motion transformations are not always semantics-preserving in the context of transient faults. For instance, the following simple change to the way block `L1` was written above leads to a vulnerability.

```
L1: movi r2, L2(*); ...; movi ri, r2(**); ...; jmp r2
```

Here, a single transient fault to `r2` anywhere between execution of instructions `(*)` and `(**)` results in an uncaught control-flow fault as both the jump target and the intention register will simultaneously be incorrect.

Likewise, the code motion transformation illustrated below shifts the move from a target block into the jumping block and creates a vulnerability.

```
L1: movi r2, L2; intend L2; movi r3, L2; jmp r2;
Lk: sub r3, r3, ri(***); recovernz r3; ...
```

Above, a fault to `r2` causes a control-flow error, but testing `r3` against `ri` at instruction `(***)` will not help detect the fault. The conclusions to draw from these examples are that the correctness properties of this code are indeed subtle and that verifying fault tolerance properties *after* the compiler has completed its suite of performance optimizations may help detect errors in code generation.

Conditional Branches. The protocol for handling conditional branches is slightly more involved than the case for jumps, but follows a similar pattern. We begin by assuming that the condition for the jump is held in registers `r4` and `r4′`. These two registers must be *independent replicas* of one another. In other words, in the absence of faults, they should contain the same boolean value, and moreover, a fault to one should have no impact on the value of the other. Given this assumption (which will be verified by our type system), the following code sequence sets up a conditional branch, which may fall through to `L2` or may jump to `L3`. The code uses a conditional branch `brz r4, r3`, which jumps to `r3` if `r4` is zero and otherwise falls through to `L2`. It also uses a conditional move `cmovz` `r4′, ri, r3′`, which moves the contents of `r3′` into `ri` if `r4′` is zero, and otherwise does nothing. [1]

```
L1:  ...; movi r3, L3; movi r3′, L3; ...; intend L2;
     cmovz r4′, ri, r3′; brz r4, r3;
L2:  ...
L3:  ...
```

Again, to indicate the special role of `ri` and simplify the presentation, we will henceforth write the conditional move `cmovz r4′, ri, r3′` as `intendz r4′, r3′`. Intuitively, the `intend` instruction unconditionally sets the intention register, whereas the `intendz` instruction conditionally sets the intention register. The error-detection code in blocks labeled `L2` and `L3` is identical to the error-detection code discussed earlier for jumps, as it must be.

Summary With just a few, well-thought-through instructions, it is possible to create a redundant copy of the intended destination of any control flow transfer prior to initiating the transfer itself. Moreover, at any control-flow target, it is possible to use that redundant copy to check that control has actually arrived at the proper place. However, as our examples illustrated, it is also easy to make slight errors in the process. In addition, because transient faults can occur at so many different places in the protocol and influence so many different bits of state, one needs a proof to believe such a protocol will work. Hence, in the following sections, we make the machine's operational semantics and fault model precise and develop a sound type system strong enough to verify that the "good" instruction sequences we have discussed in this section are indeed fault tolerant.

[1]Many architectures including the IA-32 following the Pentium Pro, the Sparc V-9 and the IA-64 have conditional moves. If the architecture does not have a a conditional move, a conditional branch and a move instruction can be used instead, but this branch will not be protected against faults.

$$
\begin{array}{llcl}
\textit{colors} & c & ::= & G \mid B \mid O \\
\textit{integers} & n & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \\
\textit{colored values} & v & ::= & c\ n \\
\\
\textit{code memory} & C & ::= & \cdot \mid C, \ell \rightarrow b \\
\\
\textit{registers} & r & ::= & r_i \mid r_1 \mid \ldots \mid r_n \\
\textit{register file} & R & ::= & \cdot \mid R, r \rightarrow v \\
\\
\textit{history} & h & ::= & \ell_1, \ldots, \ell_n \\
\textit{instructions} & i & ::= & \texttt{movi}\ r_d\ v \mid \texttt{sub}\ r_d\ r_s\ r_s \\
 & & & \mid \texttt{intend}\ r_t \mid \texttt{intendz}\ r_z\ r_t \\
 & & & \mid \texttt{recovernz}\ r_z \\
\textit{blocks} & b & ::= & i; b \mid \texttt{jmp}\ r_t \mid \texttt{brz}\ r_z\ r_t \\
\\
\textit{state} & \Sigma & ::= & (C, h, R, b) \\
\textit{final states} & \mathcal{F} & ::= & \Sigma \mid \texttt{recover}(h) \mid \texttt{hwerror}(h)
\end{array}
$$

Figure 4.1: Machine State Syntax.

4.2 The Control-Flow Machine

Figure 4.1 summarizes the syntax of the assembly language and machine states. For clarity and elegance, we will work with a minimal assembly instruction set involving move (`movi`), subtraction (`sub`), jump (`jmp`) and conditional branch if zero (`brz`) instructions as well as the special macros `intend`, `intendz` and `recovernz`. Instruction operands include constant values v and registers r. Similar to TAL$_{\text{FT}}$, we annotate every value with a *color*, although TAL$_{\text{CF}}$ has three colors: green G, blue B and orange O. Again, these colors have no operational significance, but will be used by the simulation relation. The only kind of value is an integer. In general, meta-variable n ranges over integers, but when we wish to emphasize that an integer will be used as an address, we use the meta-variable ℓ.

Instructions are grouped together in code blocks b. These blocks are always terminated by either a jump or a conditional branch instruction. Code memory C is a partial map from addresses to valid code blocks b. Addresses are ordered, and we use the notation $\ell+1$ to refer to the address of the block following the block at ℓ. If a block at ℓ ends with a conditional branch, we assume $\ell+1$ inhabits the domain of C — in other words, conditional branches always have a block to fall through to.

As before, the register file R is a mapping from registers to the colored values they contain. The registers include the intention register r_i and a number of general-purpose registers r_1 through r_n. We use the notation $R(r)$ to denote the contents of r in R. We use the notation $R[r \mapsto v]$ to denote a new register file R' created by updating R so it maps r to v. When we wish to refer to the unannotated integer n as opposed to the colored value $c\ n$ in a register r in R, we use the notation $R_{val}(r)$. Similarly, $R_{col}(r)$ refers to the color annotating the value in r.

An ordinary abstract machine state Σ is a tuple containing code C, history h, register file R and code block to be executed b. The history h is a sequence of labels. It records the code blocks visited during the current execution. As TAL$_{\mathrm{CF}}$ has no concept of memory, we will define "observable behavior" and the sequence of blocks visited. In addition to ordinary abstract machine states, there are two special "final states." The state $\mathtt{recover}(h)$ represents a state in which a transient fault has occurred and has been caught. The labels in history h were visited during the execution. The state $\mathtt{hwerror}(h)$ represents a state in which a transient fault causes transition to an invalid address.

4.2.1 Dynamic Semantics

We model the dynamic semantics of the assembly language using a small-step operational semantics. In general, the single-step operational judgments have the form $\Sigma \longrightarrow_k \mathcal{F}$

where k, which is either zero or one, records the number of faults that occur during the step.

The Fault Model. The most interesting rules in the system are the rules modeling faults. The primary rule (*zap-reg*) is familiar and again states that the value in any register may be corrupted arbitrarily, though its color tag (which has no operational significance) remains unchanged.

$$\frac{R(r) = c\ n}{(C,h,R,b) \longrightarrow_1 (C,h,R[r \mapsto c\ n'],b} \ (\textit{zap-reg})$$

The rule above may fire at any time. In particular, it may fire just prior to execution of a jump (`jmp` r_t) or a branch (`brz` r_z r_t), corrupting the jump target in register r_t. Such a fault models a control-flow error. Of course, it is equally possible that any other register is corrupted.

For uniformity in our fault model, we also consider errors in the execution of the `recovernz` r_z instruction. Recall, this instruction is merely a macro for the conditional branch `brz` r_z $\ell_{recover}$. However, since $\ell_{recover}$ is a constant, it is unaffected by faults in registers modeled by the *zap-reg* rule (our other branching instructions take arguments in registers). To simulate a fault that causes control to jump somewhere other than the $\ell_{recover}$ label when the r_z register contains a non-zero value, we add the following rules.

$$\frac{R_{val}(r_z) \neq 0}{(C,h,R,\texttt{recovernz}\ r_z;b) \longrightarrow_1 (C,h,R,C(\ell))} \ (\textit{zap-recovernz1})$$

$$\frac{R_{val}(r_z) \neq 0}{(C,h,R,\texttt{recovernz}\ r_z;b) \longrightarrow_1 \texttt{hwerror}(h)} \ (\textit{zap-recovernz2})$$

The *zap-recovernz1* rule expresses the possibility that a fault causes execution to jump to some random block labeled ℓ rather than the recovery code block. The *zap-recovernz2* rule expresses the possibility that a fault causes control to jump to an illegal address. Attempted execution of code at this address results in immediate transition to the final state $\texttt{hwerror}(h)$, where h represents the sequence of blocks visited not including the illegal address.

Other Operational Rules. All other operational rules are presented in Figure 4.2. The majority of these rules are quite unsurprising. For instance, the *movi* rule implements the move by updating the register file. Notice that the index on the arrow is “0” indicating no fault occurs during this transition. Naturally, the *intend* rule is similar to *movi* as `intend` is just a macro for a move into r_i.

Skipping to the bottom of the figure, it is important to notice there are two rules for expressing the semantics of a `jmp` r_t instruction. The first rule *jmp* fires whenever r_t contains the address of a valid block. Of course, due to a fault earlier in execution, the address in r_t may not be the intended destination for this jump. In addition to transferring control to the new block, this instruction does some bookkeeping. In particular, it extends the current history with the destination address and it changes the color of r_i to be orange. The latter effect facilitates the proof of correctness and will be explained in detail in Section 4.3. The second rule *jmp-hw-error* fires whenever r_t does *not* contain the address of a valid block. In this case, there is an attempt to transfer control to an illegal address, which is caught by the hardware. The rules for conditional branches follow a similar pattern to those for the unconditional jumps.

$$\frac{}{(C,h,R,\texttt{movi}\ r_d\ v;b) \longrightarrow_0 (C,h,R[r_d \mapsto v],b)}\ (\mathit{movi})$$

$$\frac{v' = R_{col}(r_a)\ (R_{val}(r_a) - R_{val}(r_b))}{(C,h,R,\texttt{sub}\ r_d\ r_a\ r_b;b) \longrightarrow_0 (C,h,R[r_d \mapsto v'],b)}\ (\mathit{sub})$$

$$\frac{}{(C,h,R,\texttt{intend}\ r_t;b) \longrightarrow_0 (C,h,R[r_i \mapsto R(r_t)],b)}\ (\mathit{intend})$$

$$\frac{R_{val}(r_z) = 0}{(C,h,R,\texttt{intendz}\ r_z\ r_t;b) \longrightarrow_0 (C,h,R[r_i \mapsto R(r_t)],b)}\ (\mathit{intendz\text{-}set})$$

$$\frac{R_{val}(r_z) \neq 0}{(C,h,R,\texttt{intendz}\ r_z\ r_t;b) \longrightarrow_0 (C,h,R,b)}\ (\mathit{intendz\text{-}unset})$$

$$\frac{R_{val}(r_z) = 0}{(C,h,R,\texttt{recovernz}\ r_z;b) \longrightarrow_0 (C,h,R,b)}\ (\mathit{recovernz\text{-}ok})$$

$$\frac{R_{val}(r_z) \neq 0}{(C,h,R,\texttt{recovernz}\ r_z;b) \longrightarrow_0 \texttt{recover}(h)}\ (\mathit{recovernz\text{-}halt})$$

$$\frac{R_{val}(r_z) = 0 \qquad R_{val}(r_t) \in \mathit{Dom}(C)}{(C,h,R,\texttt{brz}\ r_z\ r_t) \longrightarrow_0 (C,(h,R_{val}(r_t)),R[r_i \mapsto O\ R_{val}(r_i)],C(R_{val}(r_t)))}\ (\mathit{brz\text{-}taken})$$

$$\frac{R_{val}(r_z) \neq 0 \qquad \ell{+}1 \in \mathit{Dom}(C)}{(C,h,R,\texttt{brz}\ r_z\ r_t) \longrightarrow_0 (C,(h,\ell{+}1),R[r_i \mapsto O\ R_{val}(r_i)],C(\ell{+}1))}\ (\mathit{brz\text{-}untaken})$$

$$\frac{R_{val}(r_z) = 0 \qquad R_{val}(r_t) \notin \mathit{Dom}(C)}{(C,h,R,\texttt{brz}\ r_z\ r_t) \longrightarrow_0 \texttt{hwerror}(h)}\ (\mathit{brz\text{-}hw\text{-}error})$$

$$\frac{R_{val}(r_t) \in \mathit{Dom}(C)}{(C,h,R,\texttt{jmp}\ r_t) \longrightarrow_0 (C,(h,R_{val}(r_t)),R[r_i \mapsto O\ R_{val}(r_i)],C(R_{val}(r_t)))}\ (\mathit{jmp})$$

$$\frac{R_{val}(r_t) \notin \mathit{Dom}(C)}{(C,h,R,\texttt{jmp}\ r_t) \longrightarrow_0 \texttt{hwerror}(h)}\ (\mathit{jmp\text{-}hw\text{-}error})$$

Figure 4.2: Operational Semantics.

4.3 Typing

The design of the type system is based on three main concepts:

- Classifying the reliability properties of values.
- Using abstract types to make sure that the fault tolerance protocol proceeds in the correct order, with no steps omitted or inappropriate steps inserted.
- Equivalence checking to ensure that redundant values act as proper backups to the original.

The following paragraphs explain the main intuitions behind each concept. Later subsections will give precise details.

Classifying the Reliability Properties of Values. Because faults occur completely unpredictably and at run time, it is not possible for the type system to know which values have incurred faults or to track the propagation of presumed faulty values precisely. It is not possible to know exactly values may or may not be trusted. The type system will have to approximate these properties somehow. It does so by assigning each value to one of several compile-time "groups" and ensuring that each member of a group has related reliability properties. As a mnemonic, each group has an associated color c, which may be either *green*, *blue* or *orange*. This is a generalized version of the color scheme used by TAL$_{\text{FT}}$.

As we saw in Section 4.1, the protocol for detecting faults in software involves keeping redundant copies of the values used in control flow transfers and using these to check for correct control flow. We will refer to the main computation as the *green* computation, and the redundant copies as the *blue* (or "backup") computation. Most values either belong to the green group or to the blue group. As in TAL$_{\text{FT}}$, these two groups have the property that they are *redundant* and *independent*. In other words, a fault in a green value can never percolate to a blue value and vice versa. Consequently, when corresponding green and blue values are compared at least one of them must be correct, even when a fault has occurred. This mutual independence property is ensured by a series of simple checks in the type system that guarantee that green values are not used to construct blue values and vice versa.

But what if a control-flow fault *has* occurred? In that case, almost all program invariants are invalidated, including any properties of either blue or green values. Fortunately, though, the defining characteristic of *orange* values is preservation of their properties in just this situation.

There are two general mechanisms by which one can guarantee orange values maintain their expected properties in the face of a control-flow fault. The first mechanism is to ensure that the orange value in question is not live across the control-flow transfer: If the value has been constructed in the current block and does not depend upon values in previous blocks, a control-flow error will not influence its properties. This first mechanism is used in the checking code at the beginning of each program block. In particular, the operation that moves a label into a register at the beginning of a block may label its results orange:

```
Lk: movi r2, Lk;          // r2 is orange
    ...;
    sub r2, r2, ri;
    ...;
    recovernz r2;
    ...
```

The second mechanism involves ensuring that every possible control-flow transfer maintains the invariant in question. If the invariant is true across *every* control-flow transfer, then it is true no matter where control winds up. This second mechanism is used to classify the contents of r_i as orange across every control-flow transfer. Just as the type system isolates green values from blue and blue from green, orange is also isolated from the other two. Again, the purpose is to avoid having a fault in one color influence the others.

Although values are classified using colors, we again use the idea of *zap tags* to classify machine states. Intuitively, each zap tag specifies which colors may no longer be trusted. For example, if zap tag Z is empty (written "$\cdot$"), then there have been no faults during the computation, and all values, no matter what their color, satisfy the standard invariants associated with their compile-time type. On the other hand, if Z is a color c, then there has been a fault to a value colored c and, moreover, the corruption may have spread to any other value colored c. Consequently, values colored c will not necessarily satisfy any particular properties associated with their compile-time type.

The final zap tag *CF* classifies machine states after a control-flow error has occurred. In this case, control may have transferred somewhere totally unexpected, and so we know nothing about green *or* blue values. Fortunately, though, the properties of orange values remain valid.

Figure 4.3 summarizes the properties that hold under each zap tag while in block ℓ. We say a value is *trusted* if it satisfies standard canonical forms properties (*e.g.,* a value with code type is actually a pointer to valid code). We say a value is *untrusted* when we cannot guarantee standard canonical forms properties hold.

We say a zap tag Z is a subtype of another Z', written $Z \leq Z'$, when the values in machine states classified by Z are more trusted than the values in machine states classified

Zap Tag	G **values**	B **values**	O **values**	ℓ **correct**
$\cdot$	trusted	trusted	trusted	yes
G	*untrusted*	trusted	trusted	yes
B	trusted	*untrusted*	trusted	yes
O	trusted	trusted	*untrusted*	yes
CF	*untrusted*	*untrusted*	trusted	*no*

Figure 4.3: Properties of colored values and zap tags.

by Z'. Hence the empty zap tag is a subtype of all other zap tags, and both B and G zap tags are a supertype of CF. We use this relationship in the Preservation Theorem. Well-typed states will remain well-typed, although the zap tag may escalate to a subtype. For example, a program may start out well-typed under the empty zap tag. After a fault affects a green value, the program is well-typed under the G zap tag. If a corrupt green value is eventually used in a control flow transfer, then the program becomes well-typed under the CF zap tag. Each time the zap tag changes, more values become untrusted.

Typing Protocol Stages. The instructions in each block can be thought of as being divided into three distinct stages – the *checking code*, the *block body*, and the *exit code*. Each of these stages has its own distinct invariants. The type of intention register r_i encodes the current stage and ensures that the stages occur in the correct order. It also guarantees no part of the protocol can be omitted or any inappropriate instruction added. These stages may be summarized as follows.

1. The checking code compares the intended target with the current location to determine if there has been a control flow fault. In this region, r_i must be colored orange and have basic type *check*.

```
L1:  movi r1, L1;        }
     ...                 }
     sub r1, r1, ri;     } checking code
     ...                 }
     recovernz r1;       }
     .                   }
     .                   } block body
     .                   }
     intend L2;          }
     ...                 }
     movi r2, L2;        } exit code
     ...                 }
     jmp r2;             }
```

Figure 4.4: Example: Protocol Stages.

2. In the block body, we already know the control flow correctly transferred to this block. At the end of this sequence, there is some green register that holds the target label for the next control flow transfer and some blue register that holds the duplicate copy of this label. In the absence of faults, these two values are equal. In this region, r_i must have basic type *ok*.

3. The exit code sequence sets the intended target and transfers control to the new block. In the exit code sequence, r_i is colored blue and has type *go* when an intention has been set, and type *goz* when a conditional intention has been set. As we saw in Section 4.2.1, r_i is recolored orange during the execution of the control flow transfer.

For example, consider the example code sequences from Section 4.1 shown in Figure 4.4. On entry, each block first checks that control has reached this block correctly, and sets its intention before transferring control to another block.

Static Expressions			
exp kinds	κ	$::=$	$\kappa_{int} \mid \kappa_{hist}$
exp contexts	Δ	$::=$	$\cdot \mid \Delta, x : \kappa$
exps	E	$::=$	$x \mid n \mid E - E \mid E?E : E$
substitutions	S	$::=$	$\cdot \mid S, E/x$
Types			
stage description	ρ	$::=$	$check \mid ok \mid go \mid goz$
basic types	τ	$::=$	$int \mid \rho \mid \forall[\Delta](\Gamma, \sigma)$
value types	t	$::=$	$\langle c, \tau, E \rangle$
type option	$\tau\ opt$	$::=$	$\tau \mid \mathrm{undef}$
Context Typing			
heap typing	Ψ	$::=$	$\cdot \mid \Psi, \ell \to \tau$
reg file types	Γ	$::=$	$\cdot \mid \Gamma, r \to t$
history typing	σ	$::=$	$\varepsilon \mid x \mid \sigma \circ E$
zap tags	Z	$::=$	$\cdot \mid c \mid CF$

Figure 4.5: Typing Syntax.

Testing Value Equivalence. There are many places in the fault tolerance protocol where we require a blue value to be an independent and redundant copy of a green value. To ensure that blue and green values are equal in the absence of faults, we use the same technique using static expressions as in previous chapters, though the language of static expressions is different.

Now that we have given the intuition behind the type system design, we will move on to the technical details. The syntax for the type system is presented in Figure 4.5, and the next few subsections explain the elements and the corresponding judgments in detail.

4.3.1 Value Typing

The type of a value is a triple $\langle c, \tau, E\rangle$. The color c is assigned according to the intuitions expressed in the previous subsection. A basic type τ is either an integer, a code type, or a special type that indicates the state of the fault tolerance protocol.

The third component E is a static expression that describes the value in more detail. These expressions are used to require that blue and green computations compute identical results in the absence of faults. In TAL$_{\text{CF}}$, expressions include variables x, integers n, subtraction $E_1 - E_2$ and conditional expressions $E_1?E_2 : E_3$ which equal E_2 when E_1 is non-zero and E_3 when E_1 is zero.

Expression judgments are shown in Figures 4.6 and 4.7. The kinding judgment $\Delta \vdash E : \kappa$ holds when all the free variables in E are contained in Δ. Expression E has kind κ_{int} when it describes an integer and kind κ_{hist} when it describes a history typing. Expression variables x are the only expressions that can have type κ_{hist}. Judgments $\Delta \vdash \sigma$ wf, $\Delta \vdash \Gamma$ wf, and $\Delta \vdash \tau$ wf hold when all expressions used in these constructs are well-kinded. The judgment $\Delta \vdash S : \Delta'$ holds when S provides substitutions for all variables in Δ', and the substituted expressions are well-formed in Δ.

The function $[\![E]\!]$ supplies the denotation of the closed static expression E as an integer. The judgments $\Delta \vdash E_1 = E_2$ and $\Delta \vdash E_1 \neq E_2$ hold when the relation holds for all substitutions of the variables in Δ. $\Delta \vdash \sigma_1 = \sigma_2$ simply extends this relationship to all expressions in the two sequences.

Value Typing Judgment. The value typing judgment has the form $\Delta; \Psi \vdash^{Z} v : t$ and is shown in Figure 4.8. The context Δ contains free expression variables, and the heap type Ψ maps integer addresses to basic types. The zap tag Z characterizes the current state of the machine as explained earlier. Z is always the empty tag when a user checks a program

$\boxed{\Delta \vdash E : \kappa}$

$$\frac{x \in Dom(\Delta)}{\Delta \vdash x : \Delta(x)}\ (\textit{wf-var}) \qquad \frac{\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash\ E_2 : \kappa_{int}}{\Delta \vdash E_1 - E_2 : \kappa_{int}}\ (\textit{wf-sub})$$

$$\frac{}{\Delta \vdash n : \kappa_{int}}\ (\textit{wf-int}) \qquad \frac{\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa \qquad \Delta \vdash E_3 : \kappa}{\Delta \vdash E_1\ ?\ E_2 : E_3\ \ : \kappa}\ (\textit{wf-ifexp})$$

$\boxed{\Delta \vdash \sigma \text{ wf}}$

$$\frac{}{\Delta \vdash \varepsilon \text{ wf}}\ (\textit{wf-}\sigma\textit{-}\varepsilon) \qquad \frac{\Delta \vdash x : \kappa_{hist}}{\Delta \vdash x \text{ wf}}\ (\textit{wf-}\sigma\textit{-var}) \qquad \frac{\Delta \vdash \sigma \text{ wf} \qquad \Delta \vdash E : \kappa_{int}}{\Delta \vdash \sigma \circ E \text{ wf}}\ (\textit{wf-}\sigma)$$

$\boxed{\Delta \vdash \Gamma \text{ wf}}$

$$\frac{\forall r.\ \ \Gamma(r) = \langle c, \tau, E \rangle \ \ \wedge\ \ \Delta \vdash \tau \text{ wf} \ \ \wedge\ \ \Delta \vdash E : \kappa_{int}}{\Delta \vdash \Gamma \text{ wf}}\ (\textit{wf-R})$$

$\boxed{\Delta \vdash \tau \text{ wf}}$

$$\frac{}{\Delta \vdash \textit{int} \text{ wf}}\ (\textit{wf-int}) \qquad \frac{}{\Delta \vdash \rho \text{ wf}}\ (\textit{wf-}\rho)$$

$$\frac{(\Delta \cup \Delta') \vdash \Gamma' \text{ wf} \qquad (\Delta \cup \Delta') \vdash \sigma' \text{ wf}}{\Delta \vdash \forall[\Delta'](\Gamma', \sigma') \text{ wf}}\ (\textit{wf-}\forall[\Delta'](\Gamma', \sigma'))$$

$\boxed{\Delta \vdash S : \Delta'}$

$$\frac{}{\Delta \vdash \cdot : \cdot}\ (\textit{subst-emp-t}) \qquad \frac{\Delta \vdash S' : \Delta'' \qquad \Delta \vdash E : \kappa \qquad x \notin (\Delta \cup \Delta'')}{\Delta \vdash\ S', E/x\ :\ (\Delta'', x : \kappa)}\ (\textit{subst-t})$$

Figure 4.6: Static Expression Judgments, Part 1.

$\boxed{[\![E]\!]}$

$$\begin{array}{lcl} [\![n]\!] & = & n \\ [\![E_1 - E_2]\!] & = & [\![E_1]\!] - [\![E_2]\!] \\ [\![E_b \mathbin{?} E_t : E_f]\!] & = & if\ [\![E_b]\!]\ then\ [\![E_t]\!]\ else\ [\![E_f]\!] \end{array}$$

$\boxed{\Delta \vdash E = E}$

$$\frac{\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int} \qquad \forall S.\ \cdot \vdash S : \Delta \Longrightarrow [[S(E_1)]] = [[S(E_2)]]}{\Delta \vdash E_1 = E_2}\ (E\text{-}eq)$$

$$\frac{\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int} \qquad \forall S.\ \cdot \vdash S : \Delta \Longrightarrow [[S(E_1)]] \neq [[S(E_2)]]}{\Delta \vdash E_1 \neq E_2}\ (E\text{-}neq)$$

$\boxed{\Delta \vdash \sigma = \sigma}$

$$\frac{}{\Delta \vdash \varepsilon = \varepsilon}\ (\varepsilon\text{-}eq) \qquad \frac{}{\Delta \vdash x = x}\ (\sigma\text{-}var)$$

$$\frac{\Delta \vdash E_1 = E_2 \qquad \Delta \vdash \sigma_1 = \sigma_2}{\Delta \vdash \sigma_1 \circ E_1 = \sigma_2 \circ E_2}\ (\sigma\text{-}eq)$$

Figure 4.7: Static Expression Judgments, Part 2.

$\boxed{\Psi \vdash n : \tau}$

$$\frac{}{\Psi \vdash n : int}\ (int\text{-}t) \qquad \frac{}{\Psi \vdash n : \Psi(n)}\ (address\text{-}t) \qquad \frac{}{\Psi \vdash n : \rho}\ (\rho\text{-}t)$$

$\boxed{\Delta; \Psi \vdash^{Z} v : t}$

$$\frac{\Psi \vdash n : \tau \qquad \Delta \vdash E = n}{\Delta; \Psi \vdash^{Z}\ c\ n :\ \langle c, \tau, E \rangle}\ (val\text{-}t)$$

$$\frac{\Delta \vdash E : \kappa_{int}}{\Delta; \Psi \vdash^{c}\ c\ n :\ \langle c, \tau, E \rangle}\ (val\text{-}zap\text{-}c\text{-}t) \qquad \frac{\Delta \vdash E : \kappa_{int} \qquad c' = B \text{ or } c' = G}{\Delta; \Psi \vdash^{CF}\ c\ n :\ \langle c', \tau, E \rangle}\ (val\text{-}zap\text{-}CF\text{-}t)$$

$\boxed{\Delta \vdash t \leq t'}$

$$\frac{\Delta \vdash E_1 = E_2}{\Delta \vdash\ \langle c, \tau, E_1 \rangle\ \leq\ \langle c, \tau, E_2 \rangle}\ (subtp\text{-}reflex) \qquad \frac{\Delta \vdash E_1 = E_2}{\Delta \vdash\ \langle c, \tau, E_1 \rangle\ \leq\ \langle c, int, E_2 \rangle}\ (subtp\text{-}int)$$

$\boxed{\Delta \vdash \Gamma \leq \Gamma'}$

$$\frac{\forall r.\ \Gamma_1(r) \leq \Gamma_2(r)}{\Delta \vdash \Gamma_1 \leq \Gamma_2}\ (\Gamma\text{-}subtp)$$

Figure 4.8: Value Typing Judgment and Subtyping Judgment.

at compile time. It only takes on other values at run time for the purposes of the proof of type preservation.

The main value typing judgment depends upon an auxiliary judgment with the form $\Psi \vdash n : \tau$. This auxiliary judgment allows integer n to be given either a basic *int* type, a stage description type ρ, or a code type $\Psi(n)$. If E is equal to n and $\Psi \vdash n : \tau$, then $c\ n$ can always be given the type $\langle c, \tau, E \rangle$. However, if the zap tag Z is a color c, then all values $c\ n$ can also be typed using any basic type and any well-formed expression — such a general rule reflects the fact that we can make no guarantees about such values. When the zap tag is *CF*, then *any* green and blue value can be given *any* type, including giving green values blue types and vice versa. In other words, as mentioned earlier, when there has been a control-flow fault, we can make no assumptions about either green or blue values.

Value Subtyping. There is also a subtyping relationship $\Delta \vdash t \leq t'$ shown in Figure 4.8. This judgment allows type $\langle c, \tau, E \rangle$ to be subtype of $\langle c, int, E' \rangle$ whenever $\Delta \vdash E = E'$. Register File subtyping is a basic extension of value subtyping that requires every register in the first register file type to be a subtype of the corresponding register in the second. The subtyping judgment is used to type check control flow transfers.

4.3.2 Instruction Typing

Figure 4.9 presents the instruction typing judgment, which has the form $\Delta;\Psi;\Gamma \vdash i : \Gamma'$. As before, Δ contains free expression variables and Ψ types heap addresses. Γ acts as the precondition for the instruction, mapping registers to their corresponding types prior to execution of the instruction. Γ' acts as the postcondition for the instruction, mapping registers to types guaranteed after execution of the instruction.

$\boxed{\Delta;\Psi;\Gamma \vdash i : \Gamma'}$

$$\frac{r_d \neq r_i}{\Delta;\Psi;\Gamma \vdash \texttt{movi}\ r_d\ c\ n : \Gamma[r_d \mapsto \langle c, int, n\rangle]}\ (\textit{movi-t})$$

$$\frac{r_d \neq r_i \qquad \Gamma(r_a) = \langle c, int, E_a\rangle \qquad \Gamma(r_b) = \langle c, int, E_b\rangle}{\Delta;\Psi;\Gamma \vdash \texttt{sub}\ r_d\ r_a\ r_b : \Gamma[r_d \mapsto \langle c, int, E_a - E_b\rangle]}\ (\textit{sub-t})$$

$$\frac{\Gamma(r_i) = \langle c_i, ok, E_i\rangle \qquad \Gamma(r_t) = \langle B, \forall[\Delta_t](\Gamma_t, \sigma_t), E_t\rangle}{\Delta;\Psi;\Gamma \vdash \texttt{intend}\ r_t : \Gamma[r_i \mapsto \langle B, go, E_t\rangle]}\ (\textit{intend-t})$$

$$\frac{\begin{array}{ll}\Gamma(r_i) = \langle B, go, E_i\rangle & \Gamma(r_t) = \langle B, \forall[\Delta_t](\Gamma_t, \sigma_t), E_t\rangle \\ \Gamma(r_z) = \langle B, int, E_z\rangle & t' = \langle B, goz, E_z ? E_i : E_t\rangle\end{array}}{\Delta;\Psi;\Gamma; \vdash \texttt{intendz}\ r_z\ r_t : \Gamma[r_i \mapsto t']}\ (\textit{intendz-t})$$

Figure 4.9: Instruction Typing Judgment.

The simplest instruction to type check is the `movi` r_d c n instruction. It merely updates the type of r_d to be $\langle c, int, n\rangle$. The subtraction instruction `sub` r_d r_a r_b requires that the values being subtracted are integers. Notice it also requires the integer arguments have the same color as the result – this restriction prevents faults in values with one color from influencing another. These two instructions place no restrictions on the type of r_i, so they can occur during any stage of a block.

The unconditional intention instruction `intend` r_t requires that r_i has basic type *ok*. This restriction guarantees any new intend will occur after the checking code has been completed. Intentions are part of the blue computation, so the register that is used to set the intention must contain a blue value with code type. The type of r_i is updated to reflect the new static expression and the new stage *go*.

The conditional intention instruction `intendz` r_z r_t is similar, although it must occur after an unconditional intention. In other words, to set the intention for a conditional branch, first use `intend` to set r_i to contain the address of the fall through block, and then conditionally set it to contain the branch target. The resulting type of r_i has basic type *goz* and a conditional expression guarded by the expression E_z describing r_z. If E_z is nonzero, then r_i will be described by E_i, which describes the fall through branch. Otherwise, it is described by E_t, which describes the branch target.

Despite the fact that `recovernz` is syntactically an instruction, it is type-checked using the block typing judgment because it affects the set of free expression variables.

4.3.3 Block Typing

Figure 4.10 presents the block typing judgment, which has the form $\Delta;\Psi;\Gamma;\sigma;E_i;\tau\ opt \vdash b$. In addition to Δ, Ψ, and Γ, the block typing judgment is parameterized by a sequence σ, an expression E_i, and a type option $\tau\ opt$.

The sequence σ contains a list of expressions that describe the locations in the current history h. While typing a block at location ℓ, σ has the form $x_h \circ \ell$ meaning that the program has already visited some unknown sequence of locations (x_h) leading up to this point and that the label of the current block is ℓ. The expression E_i describes the intended target when the transfer occurred to the current label ℓ. If control flow correctly transferred to ℓ, then $\Delta \vdash E_i = \ell$. The option type $\tau\ opt$ contains the type of the label $\ell + 1$ if such a label exists. It is used when a branch falls through to the subsequent block to determine the type of that block.

The first rule, *sequence-t*, is used when the first instruction in a block is one of the basic instructions described previously. Descriptions of the other rules follow.

$$\boxed{\Delta;\Psi;\Gamma;\sigma;E_i;\tau\ opt \vdash b}$$

$$\frac{\Delta;\Psi;\Gamma \vdash i : \Gamma' \qquad \Delta;\Psi;\Gamma';\sigma;E_i;\tau\ opt \vdash b}{\Delta;\Psi;\Gamma;\sigma;E_i;\tau\ opt \vdash i;b}\ (\textit{sequence-t})$$

$$\frac{\begin{array}{l}\Gamma(r_i) = \langle O, check, x_i\rangle \\ \Gamma(r_z) = \langle O, int, E_z\rangle \\ \Delta, x:\kappa_{int} \vdash E_z = E_\ell - x_i \\ \Delta \vdash \Gamma/r_i/r_z \text{ wf} \qquad \Delta \vdash \sigma \text{ wf} \qquad \Delta \vdash E_\ell : \kappa_{int} \\ \Gamma' = \Gamma[r_z \mapsto \langle O, int, 0\rangle][r_i \mapsto \langle B, ok, E_\ell\rangle] \\ \Delta;\Psi;\Gamma';\sigma \circ E_\ell;E_\ell;\tau\ opt \vdash b\end{array}}{(\Delta, x:\kappa_{int});\Psi;\Gamma;\sigma \circ E_\ell;x_i;\tau\ opt \vdash \texttt{recovernz}\ r_z;\ b}\ (\textit{recovernz-t})$$

$$\frac{\begin{array}{ll}\Gamma(r_z) = \langle O, int, E_z\rangle & \cdot \vdash E_z = E_i - E_\ell \\ \Gamma(r_i) = \langle O, check, E_i\rangle & \cdot \vdash E_i = E_\ell \\ \multicolumn{2}{l}{\cdot;\Psi;\Gamma[r_i \mapsto \langle O, ok, E_i\rangle];\sigma \circ E_\ell;E_i;\tau\ opt \vdash b}\end{array}}{.;\Psi;\Gamma;\sigma \circ E_\ell;E_i;\tau\ opt \vdash \texttt{recovernz}\ r_z;\ b}\ (\textit{recovernz-eq-t})$$

$$\frac{\begin{array}{ll}\Gamma(r_z) = \langle O, int, E_z\rangle & \cdot \vdash E_z = E_i - E_\ell \\ \Gamma(r_i) = \langle O, check, E_i\rangle & \cdot \vdash E_i \neq E_\ell\end{array}}{.;\Psi;\Gamma;\sigma \circ E_\ell;E_i;\tau\ opt \vdash \texttt{recovernz}\ r_z;\ b}\ (\textit{recovernz-neq-t})$$

$$\frac{\begin{array}{ll}\Gamma(r_i) = \langle B, goz, E'_z?E'_f : E'_t\rangle & \Delta \vdash E'_f = E_\ell + 1 \\ \Gamma(r_z) = \langle G, int, E_z\rangle & \Delta \vdash E_z = E'_z \\ \Gamma(r_t) = \langle G, \forall[\Delta_t](\Gamma_t, \sigma_t), E_t\rangle & \Delta \vdash E_t = E'_t \\ \Delta \vdash \Gamma[r_i \mapsto \langle O, check, E'_t\rangle] \leq S_t(\Gamma_t) & \exists S_t\ .\ \Delta \vdash S_t : \Delta_t \\ \Delta \vdash \Gamma[r_i \mapsto \langle O, check, E'_f\rangle] \leq S_f(\Gamma_f) & \exists S_f\ .\ \Delta \vdash S_f : \Delta_f \\ \Delta \vdash \sigma \circ E_\ell \circ E'_t = S_t(\sigma_t) & \Delta \vdash \sigma \circ E_\ell \circ E'_f = S_f(\sigma_f)\end{array}}{\Delta;\Psi;\Gamma;\sigma \circ E_\ell;E_i;\forall[\Delta_f](\Gamma_f, \sigma_f) \vdash \texttt{brz}\ r_z\ r_t}\ (\textit{brz-t})$$

$$\frac{\begin{array}{ll}\Gamma(r_i) = \langle B, go, E'_t\rangle & \Gamma(r_t) = \langle G, \forall[\Delta_t](\Gamma_t, \sigma_t), E_t\rangle \\ \Delta \vdash E_t = E'_t & \exists S_t\ .\ \Delta \vdash S_t : \Delta_t \\ \Delta \vdash \Gamma[r_i \mapsto \langle O, check, E'_t\rangle] \leq S_t(\Gamma_t) & \Delta \vdash \sigma \circ E_\ell \circ E_t = S_t(\sigma_t)\end{array}}{\Delta;\Psi;\Gamma;\sigma \circ E_\ell;E_i;t \vdash \texttt{jmp}\ r_t}\ (\textit{jmp-t})$$

Figure 4.10: Block Typing Judgment.

Recovery. There are three distinct rules for checking `recovernz` r_z. All of them require the instruction to occur in the first stage of the block when r_i contains an orange value with basic type *check*. The operand register r_z compares this value to the current label.

The first rule *recovernz-t* applies when r_i is described by variable x_i. This is the rule used by a programmer to check correctness of their program at compile time. Control only proceeds past this point in the block if x_i is equal to the expression E_ℓ, which describes the current location, so the remainder of the block is typed by substituting E_ℓ for x_i. The types of r_i and r_z are updated to reflect the deletion of x_i. Judgment $\Delta \vdash \Gamma / r_i / r_z$ wf and $\Delta \vdash \sigma$ wf hold when all variables used in registers other than r_i and r_z as well as the expressions in σ are all contained in Δ. Because none of these pieces of state contain x_i, they do not need to be modified.

The other two rules *recovernz-eq-t* and *recovernz-neq-t* are needed to carry out the proof of type preservation (particularly the substitution lemma), but would never be used to type check programs prior to execution. In these situations, x_i has already been replaced with a closed expression E_i that describes the intention register at block entry. Here, it is evident that either $\cdot \vdash E_i = E_\ell$ or not, so there is one typing rule for each situation. The rule *recovernz-neq-t* does not place any requirements on the remainder of the block since control does not proceed past this point.

Control Flow Transfers. In order to verify unexpected transfers from the end of one block to the beginning of any other, code blocks must have the same basic precondition. To be specific, each block must expect that the intention register r_i contains an orange value with basic type *check* that is described by a variable x_i. This variable does not occur anywhere else in the function precondition. This condition entails every target block can accept any orange value in r_i.

The rule *jmp-t* requires that r_i has type $\langle B, go, E_t' \rangle$ specifying that the intention must already have been set before the jump. Also, the current jump target has a code type and is described by an expression E_t that is equal to E_t'. This enforces that in the absence of faults, the duplicate target is equal to the target.

The target label precondition contains a set of expression variables Δ_t and requires a register file described by Γ_t and a history described by σ_t. There is some substitution S_t for the variables in Δ_t so that the current register file type and sequence are subtypes of those required by the target.

The `jmp` r_t and `brz` r_z r_t instructions recolor the blue intention register to be orange when control is transferred to a new block. At first, this seems to contradict the rule that faults to a value of one color should never corrupt values of other colors. However, because the target block does not place any restrictions on the expression describing r_i, the variable x_i that describes the value can be instantiated with the value itself. Because of this, a blue value that is not trusted can become a trusted orange value during a control flow transfer, continuing to leave only the blue values untrusted.

The rule *brz-t* is similar, but adds in the conditional register r_z and specifies both the fall through and the branch cases.

4.3.4 Machine State Typing

Code Memory Typing. The judgment $\vdash C : \Psi$ describes the invariants for code memory. As described previously, all blocks must have the same basic precondition. The register r_i is described by the type $\langle O, check, x_i \rangle$. The other registers are colored either blue or green, and their static expressions do not contain the variable x_i. If a label ℓ has type $\forall[\Delta](\Gamma, x_h \circ \ell)$, then code at that label must be well-typed given Ψ, Δ, Γ, $x_h \circ \ell$, the intention expression x_i, and the fall-through label type $\Psi(\ell + 1)$.

$\boxed{\vdash C : \Psi}$

$$\frac{\begin{array}{l}\forall \ell \in Dom(C) \cup Dom(\Psi)\ . \\ \quad \Psi(\ell) = \forall[\Delta](\Gamma, x_h \circ \ell) \\ \quad \Delta = \Delta',\ x_i : \kappa_{int},\ x_h : \kappa_{hist} \\ \quad \Gamma = \Gamma',\ r_i \mapsto \langle O, check, x_i \rangle \\ \quad \forall r' \in Dom(\Gamma')\ .\ \ \Gamma'(r') \neq \langle O, b', E' \rangle \\ \quad \Delta' \vdash \Psi(\ell+1)\ \mathrm{wf} \qquad \Delta' \vdash \Gamma'\ \mathrm{wf} \\ \quad \Delta; \Psi; \Gamma; x_h \circ \ell; x_i; \Psi(\ell+1) \vdash C(\ell)\end{array}}{\vdash C : \Psi}\ (C\text{-}t)$$

$\boxed{\Psi \vdash R : \Gamma}$

$$\frac{\forall r.\ \cdot; \Psi \vdash^{Z} R(r) : \Gamma(r)}{\Psi \vdash^{Z} R : \Gamma}\ (R\text{-}t)$$

$\boxed{\vdash h : \sigma}$

$$\frac{}{\vdash () : \varepsilon}\ (h\text{-}empty\text{-}t) \qquad \frac{\vdash h : \sigma \qquad \cdot \vdash E = n}{\vdash (h, n) : \sigma \circ E}\ (h\text{-}app\text{-}t)$$

$\boxed{\vdash^{Z} (C, h, R, b)}$

$$\frac{\begin{array}{l}\vdash C : \Psi \\ \Psi \vdash^{Z}\ R : \Gamma \\ \vdash (h, \ell) : \sigma \\ (Z = CF)\ ?\ \ (\cdot \vdash E_i \neq \ell)\ \ :\ \ (\cdot \vdash E_i = \ell) \\ \Gamma(r_i) \neq\ \langle O, check, E_i \rangle\ \Longrightarrow\ . \vdash E_i = \ell \\ \cdot; \Psi; \Gamma; \sigma; E_i; \Psi(\ell+1) \vdash b\end{array}}{\vdash^{Z}\ \ (C, (h, \ell), R, b)}\ (\Sigma\text{-}t)$$

Figure 4.11: Machine State Typing.

Register File Typing. The judgment $\Psi \vdash^{Z} R : \Gamma$ states that register file R has type Γ under zap tag Z given heap typing Ψ. It holds when each register in R has the corresponding type in Γ under Z. Again, values with colors that are affected by Z are not trusted to have their given types.

History Typing. A history h is described by sequence σ when each location is equal to the corresponding expression.

Machine State Typing. A machine state Σ is well-typed under zap tag Z when each of its elements is well-typed, and two additional invariants hold. (1) If Z is CF then the current location ℓ is not equal to the intended location E_i. Otherwise, if Z is not CF, then these two are equal. (2) If the current block b has proceeded past the checking stage, then it must be the case that ℓ is equal to E_i. These two invariants together imply it is not possible for code past the checking stage of a block to be well-typed under the CF zap tag. Consequently, a proof of type preservation will imply that any control-flow error will be caught in the checking stage of the next block.

4.4 Formal Results

We have proven a number of properties of our type system including variants of the standard Progress, Preservation and Type Safety theorems. The statement of the Progress Theorem is standard. The statement of the Preservation Lemma relies on the subtyping relationship between zap tags. As the program executes, it remains well-typed, although the zap tag used to type machine states may elevate to a supertype. In other words, the program may start out well-typed under the empty zap tag, become well-typed under a colored zap tag after a fault, and then become well typed under CF if a control-flow fault occurs. Our most important result is a Fault Tolerance theorem, which we sketch briefly below. More detailed proof sketches appear in Appendix F.

In order to explain the theorem, we require a couple of additional concepts which should be familiar. First, we say a machine state Σ is well-formed (written $\vdash^{Z} \Sigma$) when all code and state are well-typed relative to the zap tag Z. Second, we say a faulty machine state Σ_f simulates a fault-free state Σ under color c (written $\Sigma_f \;\; sim^c \;\; \Sigma$) whenever the two states are identical except for values colored c. In other words, values colored c may be completely different from one another, but otherwise the two states are identical.

The judgment $\Sigma \Longrightarrow_k^h \mathcal{F}$ states that machine state Σ executes through a sequence of blocks h to reach state $\mathcal{F}$ while incurring k faulty transitions. So if $\Sigma = (C, h_1, R, b)$, then $\mathcal{F}$ is either $(C, (h_1, h), R', b')$, $\mathtt{hwerror}(h_1, h)$, or $\mathtt{recover}(h_1, h)$.

We say a program is fault-tolerant if any execution of the program with a single fault behaves in one of four possible ways with regards to the original, non-faulty computation: (1) The faulty computation visits the same sequence of blocks as the original, and the final faulty state simulates the original result state under some color c. (2) The faulty computation attempts to transfer control to an invalid address outside the domain of code memory and triggers a hardware fault. Prior to the occurrence of the hardware fault, the faulty computation visited the same blocks as the original computation. (3) A fault affecting the intention register or checking code cause the faulty computation to detect a fault in software and jump to recovery code. (4) The faulty computation veers off course to a block that does not match the corresponding block in the original computation. In this case, the checking code in the invalid block catches the error and transfers control to the recovery code.

Theorem 9 (Fault Tolerance)

If $\vdash \Sigma$ *and* $\Sigma \Longrightarrow_0^h \Sigma'$ *then at least one of the following cases applies and all derivations* $\Sigma \Longrightarrow_1^{h_f} \mathcal{F}$ *where* $length(h_f) \leq length(h)$ *fit one of these cases:*

1. $\Sigma \Longrightarrow_1^h \Sigma'_f$ *and* $\exists c \,.\, \Sigma'_f$ *sim*^{c} Σ'

2. $\Sigma \Longrightarrow_1^{h_f} \texttt{hwerror}(h', h_f)$ *and* h_f *is a prefix of* h

3. $\Sigma \Longrightarrow_1^{h_f} \texttt{recover}(h', h_f)$ *and* h_f *is a prefix of* h

4. $\Sigma \Longrightarrow_1^{h_f} \texttt{recover}(h', h_f)$ *and* $h_f = (h_1, l')$ *and* $h = (h_1, l, h_2)$

4.5 Translation

In order to show that TAL$_{\text{CF}}$ is sufficiently expressive to be of interest, we define a simple language of while loops and show how to compile statements in this language into well-typed TAL$_{\text{CF}}$ programs. We only sketch the results here, but more details are available in Appendix G.

The while loop language statements consist of simple assignment, subtraction, if statements, while loops, and sequences of statements. As all the variables in this language contain integers, the well-formedness judgment $V \vdash s$ simply enforces that all variables v in s exist in the variable context V.

$$\begin{array}{rcl} s & ::= & v := n \mid v_d := v_a - v_b \\ & & \mid \texttt{if0}\ v_z\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ v_z \neq 0\ \texttt{do}\ s \\ & & \mid s_1; s_2 \end{array}$$

To translate a statement s as a stand-alone program, it is translated with 1 as the starting label. The result of the translation is a code memory C, the instructions $\vec{i}$ from the body of the final block of the program, and the label ℓ of that last block. Because there is no halt instruction in TAL$_{\text{CF}}$, code is added to the last block in the translation to create an infinite loop at label ℓ_{halt}. The function InitRegFile(V) creates an initial register file that maps each register used to translate V to 0. The assembly language program corresponding to s is the TAL$_{\text{CF}}$ state consisting of the generated code memory, a history with only the first location, an initial register file, and code to jump to the first label in code memory. If the original statement is well-formed, then the translation is well-typed.

Theorem 10 (Translation)

If $[\![V \vdash s]\!](.,.,.,1) = (.,C,\vec{i},\ell)$ *then*

$\vdash (\ C',\ \ 0,\ \ \text{InitRegFile}(V),\ \ \text{intendjmp}\ 1\ )$

where $C' = [\ell \mapsto \text{check}\ \ell;\vec{i};\text{intendjmp}\ \ell_{halt}][\ell_{halt} \mapsto \text{check}\ \ell_{halt};\text{intendjmp}\ \ell_{halt}]$

4.6 Summary and Future Work

This chapter presents TAL$_{\text{CF}}$, a typed assembly language design for reasoning about software-only transient fault solutions. Well-typed TAL$_{\text{CF}}$ programs will always detect a single transient fault that causes control to incorrectly transfer between basic blocks before control has reached more than one incorrect block. A translation from a simple language of while programs into TAL$_{\text{CF}}$ gives evidence that it is sufficiently expressive to serve as a target of compilation.

We acknowledge that the fault model used in this chapter is simplistic. By assuming hardware support to catch control transfers into the middle of blocks, we avoid dealing with many interesting and likely situations. This assumption is required because stating intentions involves resetting r_i, so an incorrect transfer into a block before the `intend` r_t instruction may not be caught.

A sequence of existing work discussed in Section 5.2.2 handles increasing classes of erroneous transfers. By ensuring that the intention register is a function of the entire control-flow path, not just the current block, they can detect most jumps into the middle of blocks. The classification scheme of values and reliability properties from this chapter does not transfer directly to these more complex solutions, but we believe we can develop a similar classification to capture the necessary invariants. In doing so, the Fault Tolerance Theorem becomes more difficult to state and prove due to the increase of possible scenarios a single fault may cause. (For example, a fault may cause control to transfer from the middle of one block to the middle of a second block. This second block may transfer control to a third block before the error is finally detected.) In essence, the current work and proof strategy are an important building block for reasoning about more complex solutions.

Chapter 5

Related Work and Conclusion

This dissertation applies techniques for reasoning about low-level software to the problem of verifying fault tolerance techniques. As such, the related work draws from diverse topics in programming languages and computer architecture.

5.1 PCC and TAL

Proof-carrying Code (PCC) [44, 43, 3] is a method for guaranteeing properties of low-level code. In PCC, the compiler generates a safety proof along with the code binary, and the end user can automatically verify the safety proof before executing the binary.

The most popular way of implementing PCC is using a *type-preserving* compiler. Like ordinary compilers, type-preserving compilers are organized as a series of translations between intermediate languages. Unlike in ordinary compilers, each of these intermediate languages has a corresponding type system and may be type checked. In the final compilation phase, a fully type-preserving compiler will output an assembly or native code binary as well as sufficient type information for a type checker to reconstruct a typing derivation for the assembly code. The first such language was called TAL (Typed Assembly Language) [40]. Over the years, many variants have been developed [66, 62, 17, 16, 14].

Working at such a low level leads to a number of technical challenges not seen in high-level type systems. For example, creating a new object actually consists of allocating the memory and then initializing the contents. TAL [40] handled this by using *initialization flags* to track which locations had been initialized. In addition, representing class and object encodings requires new technology [15, 13]. The stack is a concept that is not part of most high-level languages, and it turns out to be quite tricky. Unlike memory allocated in the heap, the type of the value stored in a given stack slot changes during the execution of the program. As the stack grows and shrinks, a location may hold a code pointer in one stack frame and an integer in some later stack frame. Assembly-level type systems need to track these changing types carefully to avoid unsafe behavior. Things get even more complicated when allocating program data on the stack because stack locations may be aliased. At this point, it becomes very difficult to develop a type system that remains sound and yet is expressive enough to handle advanced stack allocation optimizations with. A sequence of existing work [38, 30, 50] has investigated this issue. Other current areas of interest include handling interrupts [22], interfacing with the garbage collector [27, 33], and dealing with concurrency [21].

Recently, the Bartok Compiler [14] was modified to be fully type-preserving. Bartok is an optimizing, object-oriented compiler of approximately 200,000 lines. Not only does the type-preserving version generate well-typed assembly code and support almost all the original compiler's optimizations, the additional compilation overhead is only 42%[1] and

[1]Personal communication with J. Chen. August 2008. Compilation overhead published in June 2008 [14] was 83% but has since been reduced by performance tuning.

the generated code is only 2.3% slower. In other words, type-preserving compilers are reaching the point of being practical for use in real-world situations.

However, one assumption made by all of these languages is that the hardware will faithfully execute programs. No guarantees are provided about the behavior of programs in the presence of transient faults.

5.2 Research in Transient Fault Tolerance

Researchers in the computer architecture and compiler communities have been working on solutions to transient faults for a long time. This section is not meant to be a full summary of the research in this area. Instead, it will provide a high-level overview of some of the differing types of solutions and focus on those techniques most similar to the ideas used in this dissertation.

5.2.1 Hardware-Based Solutions

Hardware-based techniques add hardware to the processor to detect transient faults. These include fine-grained techniques such as error-correcting codes and parity bits, as well as more course-grained solutions that duplicate entire structures within the processor.

For example, the 777 primary flight computer triplicates all hardware resources [75]. The Compaq Nonstop Himalaya [74] has two identical processors that run in lockstep executing their own copies of the same program. The external pins of the processors are compared on every cycle to detect faults, but recovery is left to the software. The IBM S/390 [65] duplicates units within the processor and compares their outputs on each cycle, invoking hardware recovery if necessary. It also uses parity and ECC to protect the L1 and L2 caches.

One of the main drawbacks of executing in lockstep is that it may reduce performance because processor resources are partitioned statically. Simultaneous Multithreading (SMT) [70] is a technique that allows two independent threads to run at the same time while issuing instructions to different functional units. Researchers exploited SMT to develop fault detection techniques that allow two copies of the same program to run simultaneously while still allowing dynamic scheduling of the hardware resources [57]. Others have extended this idea to handle recovery [71]. Similarly, two copies of a program can be run independently on separate cores of a processor [41], and also deal with recovery [23].

Watchdog coprocessors [35] involve a second simple processor which monitors the inputs and outputs of the main processor in order to detect faults. Despite requiring less hardware than full replication, this technique can detect a large number of faults by looking only at control flow and memory access behavior.

5.2.2 Software-Based Solutions

Just as the hardware-based solutions work by adding hardware redundancy, the software-based solutions add redundant instructions. The techniques can be divided into two main camps: protecting data and protecting control flow.

Protecting Data

An very general method for protecting data is to duplicate the entire computation and insert extra instructions to compare the two copies and check for consistency [48]. SWIFT [60] is one solution that uses this technique. Comparisons are inserted before each store instruction to ensure that the two copies of the address and the two copies of the value agree. Once the comparison succeeds, a single store instruction commits the change to memory. However, there is a *window of vulnerability* between the comparison and single store instruction where faults may go undetected and cause memory corruption. It appears impossible to address this vulnerability without relying on additional, nonstandard hardware support.

Protecting Control Flow

One particular challenge for software-based solutions involves the detection of control-flow faults. Our language TAL_{CF} in Chapter 4 keeps an *approximate program counter* and use an *intentions register* across control flow transfers to ensure that the location reached was the same as the intended location. Of course, as we discussed, this solution has the immediate drawback of being unable to detect faults into or out of the middle of code blocks.

A sequence of existing work [48, 60, 10] uses a more sophisticated version of this idea to handle increasing classes of erroneous transfers. By ensuring that the intentions register is a function of the entire control-flow path, not just the current block, they can detect most jumps into the middle of blocks.

CFCSS [48] uses an approximate program counter and assigns each block a static signature. After a control flow transfer, the approximate pc is updated to contain the xor of its old value and a precalculated value containing the xor of the current signature and the predecessors' signature. (A correct transfer from block A to block B will result in a new approximate program counter of $A \otimes (A \otimes B)$, which is equal to B.) Because the predecessor is clearly defined, there is no need for this technique to duplicate both computations. However, since the true block and the false block of a branch have the same predecessor block, this technique cannot detect a fault that causes control to take the incorrect arm of a branch. The authors present experimental results for CFCSS show that only 3.1% of the injected faults were undetected and resulted in incorrect outputs. The authors include high-level arguments about the ability to catch certain classes of faults but do not clearly specify what classes of faults where resulting in that undetected 3.1%.

SWIFT [60] uses a slightly different technique that adds a *transition register* to specify the intended transition. Before each control-flow transfer, the current block and the intended target block are xored together and the result is put in a designated transition register. At the beginning of each block, the transition register is xored with the approximate program counter to give the new approximation. Again, without faults, this should result in $A \otimes (A \otimes B) = B$ However, since $A \otimes B$ is computed at run time, this is more precise than the static method used by CFCSS. Because the intended targets cannot be determined statically, SWIFT must duplicate all the values used in control flow, but this is already being done to protect data. Again, the authors present experimental results show an impressive reduction in uncaught errors, and though the work includes itemized lists of classes of faults that are believed to be caught and not caught, there is no attempt to prove their fault tolerance scheme is correct.

One class of faults that SWIFT cannot always detect is faults from the end of a block to an instruction within the body of that same block. Borin et al. [10] introduce two additional techniques to handle this. The first technique zeros out the approximate program counter within the middle of block to detect situations where the program has incorrectly entered the middle of the block. The second technique, RCF (Region Based Control Flow) Checking, protects the branches that must be used to set the transition register. TAL_{CF} assumed the existence of a conditional move instruction to avoid unprotected branches for setting the intentions register. The additional branch is given its own signature, allowing faults there to be detected. The authors give a clear diagram-based explanation of the types of faults handled by each solution. In addition, they give a strong argument that the high-level algorithm they provide is correct. However, that does not guarantee that the generated code correctly implements the algorithm.

Despite the improvements made in this area, there are still situations (such as jumping back two instructions within a block) that cannot be handled.

Dealing with Performance

Even though many software-based solutions more than double the number of instructions, the performance overhead is generally around 40% [60, 10]. Superscaler processors are rarely able to make full use of their resources, so by adding an additional independent computation, the processor is able to achieve higher throughput.

Even so, 40% overhead is still substantial. There may be times when this overhead is acceptable, and times when it is not. But luckily, being software-based, these kinds of solutions can be applied *selectively*. Reis [58] shows that often protecting only a small number of functions can obtain nearly as much benefit as protecting the entire program, while incurring a much smaller overhead.

Adding Recovery

The software techniques discussed so far have focused on error *detection*, but being able to *recover* from errors is also extremely important.

Two copies of the program are enough to detect errors, but three copies can be used detect *and* recover by using majority voting. SWIFT-R [59] is a version of SWIFT extend to triple redundancy. Though it can recover from faults, the performance overhead is higher, and there are still windows of vulnerability. The authors also introduce a different recovery technique that only requires two copies of a computation. The difference is that in the second copy, all values are multiplied by three. Single-bit errors can be detected by checking that the original value multiplied by three is the same as the duplicate. If an error is detected, the program can determine which one is corrupt by using the modulo operation on the duplicate value. Because a single-bit error will manifest as a difference of 2^n for some n, the duplicate is only evenly divisible by three when it has not been corrupted.

5.2.3 Hybrid Solutions

There are clear trade-offs between hardware-based solutions and software-based solutions. *Hybrid* solutions use additional hardware and additional software to try to exploit the best of each type.

CRAFT [61], the system TAL_{FT} is modeled after, is a hybrid version of SWIFT. It duplicates the computation in software, but uses an additional hardware buffer to close the window of vulnerability around store instructions.

5.3 Formal Reasoning about Faults

There are a number of efforts that look at formalizing algorithms for fault tolerance, both for systems affected by transient faults and concurrent systems. A couple of the software-based systems for control flow that we have already discussed [10, 48] include formal reasoning about the high-level algorithms involved.

Di Vito and Bulter verify the system design for a fault-tolerant system for flight control [72] that votes using results from replicated processors. The system is specified at a high-level using an abstract model of real-time computation, and the proofs are completed using algebraic techniques and verified by an automated proof assistant.

Lamport and Merz [32] give a proof of correctness for a solution to the Byzantine generals problem in which the states and behaviors of the concurrent system are specified using logic, and the desired properties are checked using automated tools. However, as they themselves admit, this is very far from being able to reason at the level of executable code.

Argus [36] is a hardware-based solution with the goal of providing low-cost, low-power reliability using simple cores. It dynamically checks four invariants about control flow, computations, dataflow, and memory correctness. The authors model the system as an abstract von Neumann processor, similar to the operational semantics used in this dissertation. They prove that all executions that satisfy the four invariants will output the same values as a correct execution by using induction on the length of the execution. However, the actual algorithms for checking the invariants require hardware support and are specified at a high level.

Walker et al. [73] were the first to use a type system to reason about fault-tolerant code. They defined a typed λ-calculus called λ_{zap} in which computation are triplicated and compared at control flow transfers and before output instructions using a special atomic *vote* operation. Later work by Elsman [20] showed how to break the *vote* operation down into a sequence of conditional statements while still being able to prove correctness. There are two fundamental differences between these results and the research presented here. First, by working at such a high level of abstraction, they are able to avoid dealing with the low-level hardware structures. Second, their type systems are much weaker than those we have presented in previous chapters. For example, types in λ_{zap} consist of a color and a base type, but there are no static expressions. This means that all the formal results depend on the correctness of the translation into λ_{zap} which adds the duplication. As we have discussed, trusting the compiler can be dangerous.

Closely related to our work on $\mathrm{TAL}_{\mathrm{CF}}$ is work by Abadi et al. [1, 2] on secure control flow. Their system CFI (Control Flow Integrity) prevents attackers from manipulating the control flow of a program. The goal is to guarantee that an executing program will always follow a path through a statically determined control flow graph. However, an essential difference is the fault model. CFI assumes that an attacker can make arbitrary changes to data memory and most registers, but that three designated registers are protected. So while CFI must worry about arbitrary amounts of data corruption but can rely on three values to be correct, $\mathrm{TAL}_{\mathrm{CF}}$ only has to worry about a single fault, but can never trust a single value. In addition, the formal results for CFI state that the attacked program will follow *some* path through the existing control graph, while $\mathrm{TAL}_{\mathrm{CF}}$ guarantees that a faulty program will follow the *same* path as the original program.

5.4 Concluding Remarks

Transient faults are caused when energetic particles strike a processor and deposit charge. They may corrupt executing programs and have caused crashes at major companies. Processor technology trends, including decreasing feature size, decreasing voltages, and increasing clock rates, will result in future processors that are even more susceptible. Existing solutions that involve additional software often have a simple intuition but are implemented as just one of the many phases in a large, optimizing compiler. At the same time, recent work has shown that full-scale, type-preserving compilers are a practical way for reasoning about the behavior of low-level code.

This dissertation develops the first set of techniques for verifying fault-tolerant executable code and gives solid evidence that typed assembly languages are a promising direction for continued research. Specifically, we introduce three new typed assembly languages. TAL$_{\text{FT}}$ is used to reason about a hybrid fault detection scheme. Well-typed programs are guaranteed to detect a single fault before that fault causes a change in observable behavior. ETAL$_{\text{FT}}$, an extended version of TAL$_{\text{FT}}$, is expressive enough to be used as the target language in a compiler and also allows us to investigate the interactions between fault tolerance and other current research areas in typed assembly languages. In a slightly different direction, TAL$_{\text{CF}}$ takes a first step towards formalizing the guarantees that can be provided by a purely software-based solution.

The general approach used by all three languages includes some modifications to the standard typed assembly language methodology and can be thought of in five stages. First, in order to reason about machine execution, we define an operational semantics that includes specific rules to model faults. One example is the rule *reg-zap* that can occur at any point during execution and corrupt the contents of a random register in the register file. Second, it is necessary to formally define what "fault tolerance" means in each specific situation. For TAL$_{\text{FT}}$ and ETAL$_{\text{FT}}$, we assume that memory is externally visible and so faults may never corrupt memory writes. Since TAL$_{\text{FT}}$ focuses only on control flow, it only allows faulty programs to visit one incorrect block before detecting the error. Next, the type system is designed to combine standard type safety invariants with invariants about the fault tolerance solution, particularly specifying the relationship between different copies of a computation. Once we have these definitions, we show that the type system is sound – well-typed programs will always be fault-tolerant when executed. The main theorem for each language relates an execution and a faulty version of the same execution and shows that they will be indistinguishable to an observer. The final step is to show a language can be used as the target language of a compiler.

In addition to being the first to verify low-level code, these languages make three important contributions. The type systems combine basic typing information, computation colors, and a language of static expressions to enforce that the two different computations perform equivalent tasks. This allows us to verify low-level code without relying on how it was created. By capturing different fault models, we show that our methodology is generally applicable. TAL_{FT} and ETAL_{FT} focus on a hybrid solution, where additional hardware is used to address the vulnerability between comparing two values and acting on the result of the comparison. TAL_{FT} is purely software-based, which leads to a number of challenges, including the need to reason about situations where control ends up in an unexpected location. Finally, the translations into typed assembly, particular that of ETAL_{FT}, show that these languages can be used as the final result of a realistic, optimizing compiler.

Appendix A

TAL$_{\text{FT}}$ Proof Details

This appendix gives expanded details on the formal results in Section 2.3, including sketches of the corresponding lemmas and proofs. Working notes for the complete proofs appear in the companion technical report [49].

Section A.1 discusses useful lemmas used throughout the proofs. Section A.2 proves the standard notions of type safety. Section A.3 defines a multistep operation semantics and some associated lemmas, including the No False Positives Corollary. Section A.4 contains the Fault Tolerance Theorem and its associated lemmas. Each lemma and theorem is preceeded by a brief English explanation of its role in the proof, and followed by details on how the proof was constructed.

A.1 Lemmas

The proofs of the theorems in the remainder of this section rely on the lemmas discussed below.

A.1.1 Properties of Static Expressions

When an expression is closed, applying substitutions to that expression results in a syntactically equivalent expression.

Lemma 11 (Substituting Closed Expressions)

1. *If* $\cdot \vdash E : \kappa$ *then* $\forall S.\ S(E) = E$

Proof By induction on the structure of $\cdot \vdash E : \kappa$. ■

Even though $[\![E]\!]$ is a partial function, it is always defined over well-kinded closed expressions.

Lemma 12 (Expression Denotation)

1. *If* $\cdot \vdash E : \kappa_{int}$ *then* $\exists\, n.\ [\![E]\!] = n$.

2. *If* $\cdot \vdash E : \kappa_{mem}$ *then* $\exists\, M.\ [\![E]\!] = M$.

Proof By induction on the structure of $\cdot \vdash E : \kappa$. ■

The equality of expressions is transitive.

Lemma 13 (Expression Equality Transitivity)

If $\Delta \vdash E_1 = E_2$ *and* $\Delta \vdash E_2 = E_3$ *then* $\Delta \vdash E_1 = E_3$.

Proof By inspection of the definition of $\Delta \vdash E_1 = E_2$. ■

Substituting an expression of kind κ for a free variable of type κ preserves typing.

Lemma 14 (Substitution Lemma)

1. *If* $\Delta, x : \kappa \vdash E' : \kappa'$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E'[E/x] : \kappa'$.

2. *If* $\Delta, x : \kappa \vdash E_1 = E_2$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E_1[E/x] = E_2[E/x]$.

3. *If* $\Delta, x:\kappa \vdash E_1 \neq E_2$ *and* $\Delta \vdash E:\kappa$ *then* $\Delta \vdash E_1[E/x] \neq E_2[E/x]$.

4. *If* $\Psi;\Delta, x:\kappa \vdash^Z v:t$ *and* $\Delta \vdash E:\kappa$ *then* $\Psi;\Delta \vdash^Z v:t[E/x]$.

5. *If* $\Psi;(\Delta, x:\kappa;\Gamma;\overline{(E_d,E_s)};E_m) \vdash^Z ir \Rightarrow RT$ *and* $\Delta \vdash E:\kappa$
   then $\Psi;(\Delta;\Gamma[E/x];\overline{(E_d,E_s)}[E/x];E_m[E/x]) \vdash^Z ir \Rightarrow RT[E/x]$.

6. *If* $\cdot \vdash S:\Delta$ *and* $\Psi;\Delta \vdash^Z v:t$ *then* $\Psi;\cdot \vdash^Z v:S(t)$.

7. *If* $\cdot \vdash S:\Delta$ *and* $\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash^Z ir \Rightarrow (\Delta;\Gamma';\overline{(E'_d,E'_s)};E'_m)$
   then $\Psi;(\cdot;S(\Gamma);S(\overline{(E_d,E_s)});S(E_m)) \vdash^Z ir \Rightarrow (\cdot;S(\Gamma');S(\overline{(E'_d,E'_s)});S(E'_m))$.

Proof Parts 1, 4, and 5 – by induction on the respective typing derivation. Parts 2 and 3 – by inspection of the equality judgment definition. Parts 6 and 7 – by induction on the size of Δ, using parts 4 and 5 respectively. ■

A.1.2 Properties of Well-Typed Values

Well-typed values are described by static expressions denoting integers and not memories.

Lemma 15 (Value Kinding)

1. *If* $\Psi;\Delta \vdash^Z v:\langle c,b,E\rangle$ *then* $\Delta \vdash E:\kappa_{int}$.

2. *If* $\Psi;\Delta \vdash^Z v:(E'=0) \Rightarrow \langle c,b,E\rangle$ *then* $\Delta \vdash E:\kappa_{int}$.

Proof By case analysis on the value typing judgment. ■

The type of a value gives us information about the shape of the value.

Lemma 16 (Canonical Forms)

If $\Psi;\cdot \vdash^Z c\ n:t$ *and* $Dom(\Psi) = Dom(C) \cup Dom(M)$ *and* $\Psi \vdash M:E_m$ *and* $\Psi \vdash C$*, then*

1. *If $t = \langle c, b, E\rangle$ or $t = (E' = 0) \Rightarrow \langle c, b, E\rangle$ and $c = Z$ then no particular properties of n are known.*

2. *If $t = \langle c, int, E\rangle$ and $c \neq Z$ then $\cdot \vdash E = n$.*

3. *If $t = \langle c, \Theta \rightarrow void, E\rangle$ and $c \neq Z$ then $\Psi(n) = \Theta \rightarrow void$ and $n \in Dom(C)$ and $\cdot \vdash E = n$ and $n \neq 0$.*

4. *If $t = \langle c, b\ ref, E\rangle$ and $c \neq Z$ then $\Psi(n) = b\ ref$ and $n \in Dom(M)$ and $\cdot \vdash E = n$.*

5. *If $t = (E' = 0) \Rightarrow t'$ and $c \neq Z$ and $\cdot \vdash E' = 0$ then $n \neq 0$.*

6. *If $t = (E' = 0) \Rightarrow t'$ and $c \neq Z$ and $\cdot \vdash E' \neq 0$ then $n = 0$.*

Proof By induction on the structure of $\Psi; \cdot \vdash^{Z} c\ n : t$. ■

If a value has a type, and this type has a supertype, then the value also has the supertype.

Lemma 17 (Subtyping)

If $\Psi; \Delta \vdash^{Z} v : t$ and $\Delta \vdash t \leq t'$ then $\Psi; \Delta \vdash^{Z} v : t'$.

Proof By induction on the derivation of $\Psi; \Delta \vdash^{Z} v : t$. ■

If a value is well-typed under the empty zap tag, then that value is well-typed under all colors.

Lemma 18 (Color Weakening)

If $\Psi; \cdot \vdash v : t$ then $\forall c.\ \Psi; \cdot \vdash^{c} v : t$.

Proof By induction on the value typing judgment. ■

Color weakening extends to register files.

Lemma 19 (Register File Color Weakening)

If $\Psi \vdash R : \Gamma$ *then* $\forall c.\ \Psi \vdash^{c} R : \Gamma$.

Proof By inversion of the register file typing rule *R-t* and the Color Weakening Lemma. ■

A.1.3 Properties of Well-typed Memories

Well-typed programs with no faults only load values from valid locations.

Lemma 20 (Well-typed Domain)

If $\vdash (R, C, M, Q,\ ld_G\ r_d,\ r_s)$ *then* $R_{val}(r_d) \in Dom(M)$.

Proof By inversion of $\vdash (R, C, M, Q, ir)$, inversion of the ld_G-*t* typing rule, and the Canonical Forms lemma. ■

When looking up a value in a memory with an update, if the updated location is not the requested location, then this is equivalent to looking up the location in the memory without the update.

Lemma 21 (Irrelevant Update)

If $E = sel\ (upd\ E_m\ E_d\ E_s)\ E_n$ *and* $\cdot \vdash E_d \neq E_n$ *then* $E = sel\ E_m\ E_n$.

Proof By inspection of the denotation of *sel* and *upd*. ■

A.1.4 Properties of Well-typed Queues

The length of the queue is the same as the length of the sequence that describes it. When the zap tag is not green, then each item in the queue is described by the corresponding expression. In addition, the first element of each pair has type *b ref* and its value has type *b*.

Lemma 22 (Queue)

1. *If* $\Psi \vdash^Z Q : \overline{(E_d, E_s)}$ *then* $length(Q) = length(\overline{(E_d, E_s)})$.

2. *If* $\Psi \vdash^Z \overline{(n_1, n_2)} : \overline{(E_d, E_s)}$ *and* $Z \neq G$ *then for all* k *from 1 to* $length(\overline{(n_1, n_2)})$, $\cdot \vdash E_{dk} = n_{1k}$ *and* $\cdot \vdash E_{sk} = n_{2k}$ *and there is some base type* b *such that* $\Psi \vdash n_{1k} : b$ *ref and* $\Psi \vdash n_{2k} : b$.

Proof By induction on the structure of $\Psi \vdash^Z Q : \overline{(E_d, E_s)}$. ■

If the $find$ function returns no match on an address n_1, and the queue is described by the sequence of address-value pairs $\overline{(E_d, E_s)}$, then none of the address expressions are equal to n_1.

Lemma 23 (Find)

If $find(Q, n_1) = ()$ *and* $\Psi \vdash Q : \overline{(E_d, E_s)}$ *then for* k *from 1 to* $length(Q)$, $\cdot \vdash E_{dk} \neq n_1$.

Proof By definition of the $find$ function and the queue typing judgment. ■

If a queue is well-typed under the empty zap tag, then it is also well-typed under any colored zap tag.

Lemma 24 (Queue Color Weakening)

If $\Psi \vdash Q : \overline{(E_d, E_s)}$ *then* $\forall c.\ \Psi \vdash^c Q : \overline{(E_d, E_s)}$.

Proof By induction on the queue typing judgment. ∎

A.2 Type Safety

Progress states that well-typed states can take a step. In particular, a machine state that is well-typed under the empty zap tag can take a non-faulty step to another ordinary, non-faulty machine state. A machine state that is well-typed under a zap tag of color *c* can take a step, but the result of that step may either be another ordinary machine state or the *fault* state.

Theorem 25 (Progress)

1. *If* $\vdash \Sigma$ *then* $\Sigma \longrightarrow_0^s \Sigma'$ *and* $\Sigma' \neq \textit{fault}$.

2. *If* $\vdash^c \Sigma$ *then* $\Sigma \longrightarrow_0^s \Sigma$.

Proof By case analysis on the instruction *ir* in state Σ.

Part 1 uses inversion on the typing rules to determine that the preconditions hold for the appropriate operational rule. For example, Figure A.1 shows the case for st_B. By inverting various typing rules, we gather enough information to conclude that the last pair in the queue is equal to the contents of the two registers. Notice that step 9 inverts the *val-t* rule. This can only be done because we know that $Z = \cdot$, and so the *val-zap-t* rule cannot apply instead. (And similarly, step 12 inverts rule *Q-t*.) Once we have these equalities, we can apply the operational rule st_B*-mem*. Other cases are similar, although the cases for ld_G and bz_B subdivide further based on the result of the *find* function and the value in the branch register.

Part 2 is simpler than part 1. Instead of using typing rules to gather as much information, we just further subdivided based on properties needed to apply the rules.

These subcases take either the normal operation rule, or the corresponding failure rule as necessary. Some inversion may be done on the typing rules to show that the case does not get stuck. Again, Figure A.1 shows the case for st_B. From the typing information, we know that the Q is not empty. Then either the last pair in the queue is equal to the registers (rule st_B applies) or it is not (rule st_B-*fail* applies). ■

According to Preservation, if a machine state is well-typed under a zap tag Z, and it takes a non-faulty step to another machine state, then that resulting state will also be well-typed under Z. Additionally, if a state is well-typed under the empty zap tag, and it takes a faulty step, then there is some color c such that the resulting state is well-typed under c.

Theorem 26 (Preservation)

1. *If* $\vdash^Z \Sigma$ *and* $\Sigma \longrightarrow^s_0 \Sigma'$ *and* $\Sigma' \neq$ *fault then* $\vdash^Z \Sigma'$.

2. *If* $\vdash \Sigma$ *and* $\Sigma \longrightarrow^s_1 \Sigma'$ *then* $\exists\, c.\ \vdash^c \Sigma'$.

Proof By case analysis of the structure of the derivation $\Sigma \longrightarrow^s_k \Sigma'$.

Part 1 only applies to cases where Σ' is not *fault*. We use inversion to take apart the judgment for $\vdash^Z \Sigma$, modify it as necessary, and build the judgment $\vdash^Z \Sigma'$. Each case is subdivided based on the actual value of the zap tag. In subcases where the zap tag is not the same as the color of a value, we can invert rule *val-t* on the typing of a value and use that information to construct a typing derivation for Σ'. In cases where the zap tag has the same color as the value, modified values in Σ' can be trivially typed using rules *val-zap-t* and *Q-zap-t*. Figures A.2 and A.3 show an example case for rule st_B-*mem*.

Part 2 chooses c to be the color of the zapped value. It uses rules *val-zap-t*, *val-zap-cond* or *Q-zap-t* to show that the zapped value can be typed under zap tag c. The remainder of the state can be typed as before using the Color Weakening Lemmas 18, 19,

Part 1 Example Case: st$_B$

a1.	$\vdash (R,C,M,Q,\ st_B\ \ r_d,r_s)$	[assumption]
1.	$\Psi \vdash C$	[Inversion of Σ-*t*, a1]
2.	$\forall c.\ C(R_{val}(pc_c)) = st_B\ \ r_d,r_s$	[Inversion of Σ-*t*, a1]
3.	$\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)},(E'_d,E'_s);E_m) \vdash\ st_B\ r_d\ r_s$	[Inversion of *C*-*t*, 1, 2, inspection of *st$_B$*-*t*]
	$\Rightarrow (\Delta;\Gamma{+}{+};\overline{(E_d,E_s)};upd\ E_m\ E'_d\ E'_s)$	
4.	$\exists S.\ \cdot \vdash S:\Delta$	[Inversion of Σ-*t*, a1]
5.	$\Psi;(\cdot;S(\Gamma);S(\overline{(E_d,E_s)},(E'_d,E'_s));S(E_m)) \vdash\ st_B\ r_d\ r_s$	[Lemma 14 (Substitution), 4, 3]
	$\Rightarrow (\cdot;S(S(\Gamma){+}{+};S(\overline{(E_d,E_s)});upd\ S(E_m\ E'_d\ E'_s))$	
6.	$\Psi \vdash R:S(\Gamma)$	[Inversion of Σ-*t*, a1]
7.	$S(\Gamma)(r_d) = \langle B,b\ ref,E''_d\rangle$	[Inversion of *st$_B$*-*t*, 5]
	$S(\Gamma)(r_s) = \langle B,b,E''_s\rangle$	
8.	$\Psi;\cdot \vdash R(r_d):\langle B,b\ ref,E''_d\rangle$	[Inversion of *R*-*t*, 6, 7]
	$\Psi;\cdot \vdash R(r_s):\langle B,b,E''_s\rangle$	
9.	$\cdot \vdash Rval(r_d) = E''_d$	[Inversion of *val*-*t*, 8]
	$\cdot \vdash Rval(r_s) = E''_s$	
10.	$\cdot \vdash S(E'_d) = E''_d$	[Inversion of *st$_B$*-*t*, 5]
	$\cdot \vdash S(E'_s) = E''_s$	
11.	$\Psi \vdash Q:S(\overline{(E_d,E_s)},(E'_d,E'_s))$	[Inversion of Σ-*t*, a1]
12.	$Q = (\overline{(n,n')},(n_l,n'_l))$	[Lemma 22 (Queue), 11]
	$\cdot \vdash S(E'_d) = n_l$ and $\cdot \vdash S(E'_s) = n'_l$	
13.	$R_{val}(r_d) = n_l$ and $R_{val}(r_s) = n'_l$	[Lemma 13 (Exp Eq Transitivity), 9, 10, 12]
14.	$(R,C,M,(\overline{(n,n')},(n_l,n'_l)),st_B\ \ r_d,r_s) \xrightarrow{(n_l,n'_l)}_0$	[*st$_B$*-*mem*, 14]
	$(R{+}{+},C,M[n_l \mapsto n'_l],\overline{(n,n')},\cdot)$	

Case complete.

Part 2 Example Case: st$_B$

a1.	$\vdash^c (R,C,M,Q,\ st_B\ \ r_d,r_s)$	[assumption]
1.	$\Psi \vdash^c Q:S(\overline{(E_d,E_s)},(E'_d,E'_s))$	[Inversion of Σ-*t*, a1]
2.	$Q = (\overline{(n,n')},(n_l,n'_l))$	[Lemma 22 (Queue), 1]

Case on whether $R_{val}(r_d) \overset{?}{=} n_l$ and $R_{val}(r_s) \overset{?}{=} n'_l$

subcase a: r_d and r_s are the same as the last pair in the queue

a4a.	$R_{val}(r_d) = n_l$ and $R_{val}(r_s) = n'_l$	[subcase assumption]
3a.	$(R,C,M,Q,st_B\ \ r_d,r_s) \xrightarrow{(n_l,n'_l)}_0 (R{+}{+},C,M[n_l \mapsto n'_l],\overline{(n,n')},\cdot)$	[*st$_B$*-*mem*, a4a]

Subcase complete.

subcase b: either r_s or r_d or the last pair in the queue has been corrupted

a4b.	$R_{val}(r_d) \neq n_l$ or $R_{val}(r_s) \neq n'_l$	[subcase assumption]
3b.	$(R,C,M,Q,st_B\ \ r_d,r_s) \longrightarrow_0 fault$	[*st$_B$*-*mem*-*fail*, a4b]

Subcase complete.

Case complete.

Figure A.1: Example Cases of Theorem 25 (Progress).

and 24. Figure A.4 shows an example case for rule *reg-zap*. ■

The Progress and Preservation Theorems define the usual notion of type safety.

A.3 Multistep Transitions

In order to prove properties of our type system, we extend our single-step transition $\Sigma_1 \longrightarrow_k^s \Sigma_2$ from Section 2.1 to a sequence of n transitions containing exactly k faults $\Sigma_1 \xrightarrow{n}{}_k^s \Sigma_2$, where n is greater than or equal to zero, and k is still either 0 or 1.

$$\frac{}{\Sigma \xrightarrow{0}{}_0^{()} \Sigma}\ \textit{(multi-base)} \qquad \frac{\Sigma \longrightarrow_{k_1}^{s_1} \Sigma'' \quad \Sigma'' \xrightarrow{(n-1)}{}_{k_2}^{s_2} \Sigma'}{\Sigma \xrightarrow{n}{}_{k_1+k_2}^{(s_1,s_2)} \Sigma'}\ \textit{(multi-compose)}$$

A.3.1 No False Positives

By combining part one of Progress with part one of Preservation, we get the following important corollary: The hardware never claims to have detected a fault when no fault has occurred during execution of a well-typed program.

Corollary 27 (No False Positives)

If $\vdash \Sigma$ *then* $\forall\, n.\ \Sigma \xrightarrow{n}{}_0^s \Sigma'$ *and* $\vdash \Sigma'$.

Proof By Progress Part 1, Preservation Part 1, and induction on the derivation of $\Sigma \xrightarrow{n}{}_0^s \Sigma'$. ■

Example Case: st$_B$-mem

$$\frac{R_{val}(r_d) = n_l \qquad R_{val}(r_s) = n_l'}{(R, C, M, (\overline{(n, n')}, (n_l, n_l')), st_B\ \ r_d, r_s) \longrightarrow_0^{(n_l, n_l')} (R{++}, C, M[n_l \mapsto n_l'], \overline{(n, n')}, \cdot)}\ (st_B\text{-}mem)$$

a1.	$\vdash^Z (R, C, M, Q,\ st_B\ \ r_d, r_s)$	[assumption]
p1.	$R_{val}(r_d) = n_l$	[premise]
p2.	$R_{val}(r_s) = n_l'$	[premise]
1.	$Dom(\Psi) = Dom(C) \cup Dom(M)$	[Inversion of Σ-t, a1]
2.	$Z \neq G \implies Dom(\overline{(n, n')}, (n_l, n_l')) \subseteq Dom(M)$	[Inversion of Σ-t, a1]
3.	$\Psi \vdash C$	[Inversion of Σ-t, a1]
4.	$\forall c \neq Z.\ C(R_{val}(pc_c)) =\ st_B\ \ r_d, r_s$	[Inversion of Σ-t, a1]
5.	$\forall c \neq Z.\ \Psi(R_{val}(pc_c)) = (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \rightarrow void$	[Inversion of Σ-t, a1]
6.	$\exists S.\ \cdot \vdash S : \Delta$	[Inversion of Σ-t, a1]
7.	$\Psi \vdash M : S(E_m)$	[Inversion of Σ-t, a1]
8.	$\Psi \vdash^Z (\overline{(n, n')}, (n_l, n_l')) : S(\overline{(E_d, E_s)}, (E_d', E_s'))$	[Inversion of Σ-t, a1]
9.	$\Psi \vdash^Z R : S(\Gamma)$	[Inversion of Σ-t, a1]
10.	$\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}, (E_d', E_s'); E_m) \vdash\ st_B\ r_d\ r_s \Rightarrow \Theta'$	[Inversion of C-t, 3, 4]
11.	$\Theta' = (\Delta; \Gamma{++}; \overline{(E_d, E_s)}; upd\ E_m\ E_d'\ E_s')$	[10, inspection of st_B-t]
12.	$\forall c \neq Z.\ C(R_{val}(pc_c) + 1) = \Theta' \mapsto void$	[Inversion of C-t, 3, 4, 10, 11]
5'.	$\forall c \neq Z.\ C(R{++}_{val}(pc_c)) = \Theta' \mapsto void$	[12, def of $R{++}$]
9'.	$\Psi \vdash^Z R{++} : S(\Gamma){++}$	[9, def of $R{++}$ and $\Gamma{++}$]
13.	$\Psi; (\cdot; S(\Gamma); S(\overline{(E_d, E_s)}, (E_d', E_s')); S(E_m)) \vdash\ st_B\ r_d\ r_s$ $\Rightarrow (\cdot; S(S(\Gamma){++}; S(\overline{(E_d, E_s)}); upd\ S(E_m\ E_d'\ E_s'))$	[Lemma 14 (Substitution), 10, 11, 6]
14.	$S(\Gamma)(r_d) = \langle B, b\ ref, E_d'' \rangle$	[Inversion of st_B-t, 5]
15.	$S(\Gamma)(r_s) = \langle B, b, E_s'' \rangle$	[Inversion of st_B-t, 5]
16.	$\cdot \vdash S(E_d') = E_d''$	[Inversion of st_B-t, 5]
17.	$\cdot \vdash S(E_s') = E_s''$	[Inversion of st_B-t, 5]

Case on Z

Subcase a1: $Z = G$

a2a.	$Z = G$	[subcase assumption]
18a.	$\Psi \vdash^Z \overline{(n, n')} : S(\overline{(E_d, E_s)})$	[Repeated Inversion of Q-zap-t, a2a, 8, Q-zap-t]

Subcase b1: $Z = \cdot$ or $Z = B$

a2b.	$Z = \cdot$ or $Z = B$	[subcase assumption]
18b.	$\Psi \vdash^Z \overline{(n, n')} : S(\overline{(E_d, E_s)})$	[Repeated Inversion of Q-t, a2a, 8, Q-t]

Merge subcases a1 and b1:

8'.	$\Psi \vdash^Z \overline{(n, n')} : S(\overline{(E_d, E_s)})$	[18a/18b]
2'.	$Z \neq G \implies Dom(\overline{(n, n')}) \subseteq Dom(M)$	[2]

Continued in Figure A.3...

Figure A.2: Example Case from Theorem 26 (Preservation) Part 1.

Example Case: st$_B$-mem (...continued from Figure A.2)

Case on Z

Subcase a2: $Z = B$ (the queue is correct)

a3a.	$Z = B$	[subcase assumption]
19a.	$\cdot \vdash S(E'_d) = n_l$	[Inversion of *Q-t*, a3a, 8]
20a.	$\Psi \vdash n_l : b\ ref$	[Inversion of *Q-t*, a3a, 8]
21a.	$\Psi; \cdot \vdash^B n'_l : \langle B, b\ ref, E''_d \rangle$	[*val-t*, 19a, 20a]
22a.	$n_l \in Dom(M)$	[Lemma 16 (Canonical Forms), 7, 3, 21a
23a.	$\cdot \vdash S(E'_s) = n'_l$	[Inversion of *Q-t*, a3a, 8]
24a.	$\Psi \vdash n'_l : b$	[Inversion of *Q-t*, a3a, 8]
25a.	$[\![S(E'_d)]\!] = n_l$ and $[\![S(E'_s)]\!] = n'_l$	[Inversion of *E-eq*, 19a, 23a]

Subcase b2: $Z = \cdot$ or $Z = G$ (r_d and r_s are correct)

a3b.	$Z = \cdot$ or $Z = G$	[subcase assumption]
19b.	$\Psi; \cdot \vdash^Z R_{val}(r_d) : \langle B, b\ ref, E''_d \rangle$	[Inversion of *R-t*, 9, 14]
20b.	$\Psi; \cdot \vdash^Z n_l : \langle B, b\ ref, S(E'_d) \rangle$	[19b, p1, Lemma 13 (Exp Eq Transitivity), 16]
21b.	$n_l \in Dom(M)$	[Lemma 16 (Canonical Forms), 7, 3, 20b
22b.	$\Psi \vdash n_l : b\ ref$	[Inversion of *val-t*, a3b, 20b]
23b.	$\Psi; \cdot \vdash^Z R_{val}(r_s) : \langle B, b, E''_s \rangle$	[Inversion of *R-t*, 9, 15]
24b.	$\Psi; \cdot \vdash^Z n'_l : \langle B, b, S(E'_s) \rangle$	[23b, p2, Lemma 13 (Exp Eq Transitivity), 17]
25b.	$\Psi \vdash n'_l : b$	[Inversion of *val-t*, a3b, 24b]
26b.	$\cdot \vdash S(E'_d) = n_l$ and $\cdot \vdash S(E'_s) = n'_l$	[Inversion of *val-t*, a3b, 20b, 24b]
27b.	$[\![S(E'_d)]\!] = n_l$ and $[\![S(E'_s)]\!] = n'_l$	[Inversion of *E-eq*, 26b]

Merge subcases a2 and b2:

30.	$\forall \ell \in Dom(M).\ \Psi \vdash \ell : b\ ref \wedge \Psi \vdash M(\ell) : b$	[Inversion of *M-t*, 7]
31.	$\forall \ell \in Dom(M[n_l \mapsto n'_l]).\ \Psi \vdash \ell : b\ ref \wedge \Psi \vdash M(\ell) : b$	[30, 20a/22b, 24a/25b]
32.	$[\![S(E_m)]\!] = M$	[Inversion of *M-t*, 7]
33.	$[\![upd\ S(E_m)\ S(E'_d)\ S(E'_s)]\!] = [\![S(E_m)]\!][\ [\![S(E'_d)]\!] \mapsto [\![S(E'_s)]\!]\ ]$	[def of $[\![E]\!]$]
34.	$[\![upd\ S(E_m)\ S(E'_d)\ S(E'_s)]\!] = M[n_l \mapsto n'_l]$	[33, 32, 25a/27b]
35.	$\cdot \vdash S(E_m) : \kappa_{mem}$	[Inversion of *M-t*, 7]
36.	$\cdot \vdash S(E'_d) : \kappa_{int}$ and $\cdot \vdash S(E'_s) : \kappa_{int}$	[Lemma 15 (Value Kinding), Inversion of *R-t*, 9, 14, 15]
37.	$\cdot \vdash upd\ S(E_m)\ S(E'_d)\ S(E'_s) : \kappa_{mem}$	[*E-upd-t*, 35, 36]
7'.	$\Psi \vdash M[n_l \mapsto n'_l] : S(E_m\ E'_d\ E'_s)$	[*M-t*, 31, 34, 37]
1'.	$Dom(\Psi) = Dom(C) \cup Dom(M[n_l \mapsto n'_l])$	[1, 22a/21b]
38.	$\vdash^Z (R\text{++}, C, M[n_l \mapsto n'_l], \overline{(n, n')}, \cdot)$	[Σ-*t*, 1', 2', 3, *ir* = ., 5', 6, 7', 8', 9'

Figure A.3: Example Case from Theorem 26 (Preservation) Part 1.

Example Case: reg-zap

$$\frac{R(a) = c\ n}{(R,C,M,Q,ir) \longrightarrow_1 (R[a \mapsto c\ n'],C,M,Q,ir)}\ (\textit{reg-zap})$$

a1.	$\vdash (R,C,M,Q,ir)$	[assumption]
1.	$Dom(\Psi) = Dom(C) \cup Dom(M)$	[Inversion of Σ-*t*, a1]
2.	$Dom(Q) \subseteq Dom(M)$	[Inversion of Σ-*t*, a1]
3.	$\Psi \vdash C$	[Inversion of Σ-*t*, a1]
4.	$\forall c.\ ir \neq \cdot \implies C(R_{val}(pc_c)) = ir$	[Inversion of Σ-*t*, a1]
5.	$\forall c.\ \Psi(R_{val}(pc_c)) = (\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \rightarrow void$	[Inversion of Σ-*t*, a1]
6.	$\exists S.\ \cdot \vdash S : \Delta$	[Inversion of Σ-*t*, a1]
7.	$\Psi \vdash M : S(E_m)$	[Inversion of Σ-*t*, a1]
8.	$\Psi \vdash Q : S\overline{(E_d,E_s)}$	[Inversion of Σ-*t*, a1]
9.	$\Psi \vdash R : S(\Gamma)$	[Inversion of Σ-*t*, a1]

Case on the shape of $S(\Gamma)(a)$

Subcase a: $S(\Gamma)(a)$ is a triple $\langle c,b,E \rangle$

a2a.	Let $\langle c,b,E \rangle = S(\Gamma)(a)$	[subcase assumption]
10a.	$\cdot \vdash E : \kappa_{int}$	[Lemma 15 (Value Kinding), Inversion of *R-t*, 9, a2a]
11a.	$\Psi;\cdot \vdash^c c\ n' : \langle c,b,E \rangle$	[*val-zap-t*, 10a]
12a.	$\Psi;\cdot \vdash^c R[a \mapsto c\ n'](a) : S(\Gamma)(a)$	[11a, a2a, def of R[ ]]

Subcase b: $S(\Gamma)(a)$ is a conditional type $E' = 0 \Rightarrow \langle c,b,E \rangle$

a2b.	Let $(E' = 0 \Rightarrow \langle c,b,E \rangle) = S(\Gamma)(a)$	[subcase assumption]
10b.	$\cdot \vdash E : \kappa_{int}$	[Lemma 15, Inversion of *R-t*, 9, a2b]
11b.	$\Psi;\cdot \vdash^c c\ n' : E' = 0 \Rightarrow \langle c,b,E \rangle$	[*val-zap-cond*, 10b]
12b.	$\Psi;\cdot \vdash^c R[a \mapsto c\ n'](a) : S(\Gamma)(a)$	[11b, a2b, def of R[ ]]

Merge subcases a and b:

9'.	$\Psi \vdash^c R[a \mapsto c\ n'] : S(\Gamma)$	[*R-t*, 9, Lemma 19 (Reg File Color Weakening), 12a/12b]
8'.	$\Psi \vdash^c Q : S\overline{(E_d,E_s)}$	[Lemma 24 (Queue Color Weakening), 8]
2'.	$Z \neq G \implies Dom(Q) \subseteq Dom(M)$	[2]
4'.	$\forall c \neq Z.\ ir \neq \cdot \implies C(R_{val}(pc_c)) = ir$	[4]
5'.	$\forall c \neq Z.\ \Psi(R_{val}(pc_c)) = (\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \rightarrow void$	[5]
13.	$\vdash^c (R[a \mapsto c\ n'],C,M,Q,ir)$	[Σ-*t*, 1, 2', 3, 4', 5', 6, 7, 8', 9']

Case complete.

Figure A.4: Example Case of Theorem 26 (Preservation) Part 2.

A.3.2 Multistep Split and Combine

The following two lemmas about the multistep relation let us take apart and put together different sequences of steps.

If a machine state evaluates in a sequence of steps with no faults to a final state, then this computation can be divided into a sequence of non-faulty steps reaching an intermediate state, and a sequence of non-faulty steps from this intermediate state to the final state.

Lemma 28 (Multistep Split)

If $\Sigma \xrightarrow{n}{}_0^{s} \Sigma'$ *then there exists* n_1, n_2, Σ'', s_1, *and* s_2 *such that* $n = n_1 + n_2$ *and* $s = (s_1, s_2)$ *and* $\Sigma \xrightarrow{n_1}{}_0^{s_1} \Sigma''$ *and* $\Sigma'' \xrightarrow{n_2}{}_0^{s_2} \Sigma'$*.*

Proof By induction on the structure of $\Sigma \xrightarrow{n}{}_0^{s} \Sigma'$. ■

If a machine state evaluates in a sequence of n_1 non-faulty steps to another state, that state faults to a third state, and the third state evaluates in n_2 non-faulty steps to a final state, then the original state can reach the faulty state in a sequence of $n_1 + 1 + n_2$ steps including exactly one fault step.

Lemma 29 (Multistep Combine)

If $\Sigma \xrightarrow{n_1}{}_0^{s_1} \Sigma'$ *and* $\Sigma' \longrightarrow_1 \Sigma'_f$ *and* $\Sigma'_f \xrightarrow{n_2}{}_0^{s_2} \Sigma''$ *then* $\Sigma \xrightarrow{n'}{}_1^{(s_1,s_2)} \Sigma''$ *where* $n' = n_1 + 1 + n_2$*.*

Proof By induction on the structure of $\Sigma \xrightarrow{n_1}{}_0^{s_1} \Sigma'$. ■

A.4 Fault Tolerance

A program is fault tolerant when all the faulty executions of that program *simulate* fault-free executions of the program. More precisely, the sequence of outputs from the faulty

$\boxed{v_1 \ sim^Z \ v_2}$

$$\frac{}{C\ n\ sim^Z\ C\ n}\ (sim\text{-}val) \qquad \frac{}{C\ n\ sim^C\ C\ n'}\ (sim\text{-}val\text{-}zap)$$

$\boxed{R\ sim^Z\ R'}$

$$\frac{\forall a.\ R(a)\ sim^Z\ R'(a)}{R\ sim^Z\ R'}\ (sim\text{-}R)$$

$\boxed{Q\ sim^Z\ Q'}$

$$\frac{}{\cdot\ sim^Z\ \cdot}\ (sim\text{-}Q\text{-}empty)$$

$$\frac{G\ n_1\ sim^Z\ G\ n_1' \qquad G\ n_2\ sim^Z\ G\ n_2' \qquad Q\ sim^Z\ Q'}{((n_1,n_2),Q)\ \ sim^Z\ ((n_1',n_2'),Q')}\ (sim\text{-}Q)$$

$\boxed{\Sigma_1\ sim^Z\ \Sigma_2}$

$$\frac{R\ sim^Z\ R' \qquad Q\ sim^Z\ Q'}{(R,C,M,Q,ir)\ sim^Z\ (R',C,M,Q',ir)}\ (sim\text{-}\Sigma)$$

Figure A.5: Similarity of Machine States.

executions are required either to be identical to the fault-free execution or, in the case the hardware detects the fault, a prefix of the fault-free execution.

A.4.1 Simulation Relation

In order to reason about pairs of faulty and fault-free executions, we define similarity relations between values, register files, queues and machine states. Each of these relations is defined relative to the zap tag Z. Intuitively, if Z is empty, the related objects must be identical. If Z is a color c, the objects must be identical modulo values colored c. In the latter case, values colored c may be corrupted, and there is no hope they satisfy any particular relation. The formal definitions of these relations are shown in Figure A.5.

A.4.2 Singlestep Fault Detection

We begin by defining fault detection for a single step in the execution of a program. It essentially says that if we have two similar computations, one with a fault and one without, then either the faulty computation can take a step indistinguishable from that of the non-faulty version, or the faulty computation reaches the fault state.

Lemma 30 (Singlestep Fault Detection)

If $\vdash \Sigma$ *and* $\Sigma\ sim^c\ \Sigma_f$ *and* $\Sigma \longrightarrow_0^s \Sigma'$ *then* $\Sigma_f \longrightarrow_0^{s_f} \Sigma'_f$ *and either*

1. $\Sigma'\ sim^c\ \Sigma'_f$ *and* $s = s_f$*, or*

2. $\Sigma'_f = fault$ *and* $s_f = ()$*.*

Proof By case analysis of $\Sigma \longrightarrow_0^s \Sigma'$. Each case is handled in one of three ways.

Failure Cases. Rules where Σ steps to *fault* are not applicable because Σ is well typed under the empty zap tag, and so Progress 1 tells us that Σ' is not *fault*.

Random Cases. Rules *ld$_B$-rand* and *ld$_G$-rand* only apply when loading from an invalid address that is not in the domain of memory. According to Lemma 20 (Well-typed Domain), states that are well-typed under the empty zap tag only load from valid addresses, so these cases can be ruled out.

Standard Cases. The remaining rules are all handled in approximately the same way.

Each rule has a handful of premises. These relationships may or may not hold in the faulty computation. It will depend on what exactly has been corrupted. The proof divides into subcases based on these relationships. Some subcases may further divide based on whether c is G or B.

Figure A.6 shows the case for *st$_B$-mem*. There are two premises relating the two registers to the last pair in the queue. If one of these premises does not hold in the faulty computation, then it will step to *fault* using the *st$_B$-fail* rule with no output.

However, if both equalities hold then the proof further subdivides based on the color c. If c is B, then we know that the faulty and non-faulty queues are equal because they contain green values and simulate each other under color B. From there, we can determine that the faulty computation also steps using *st$_B$*, the resulting state is similar to Σ', and the outputs are equal.

If c is G, then we use the fact that Σ is well-typed to show that the registers r_d and r_s are colored blue, and so both the faulty and non-faulty registers are correct and equal to each other. And since the registers are assumed to be equal to the ends of the queues, we also know that the ends of the queues are equal. From there, we continue as in the previous subcase.

■

A.4.3 Multistep Fault Detection

The Multistep Fault Detection Lemma extends the Singlestep Fault Detection Lemma for n steps. If a fault has occurred and the non-faulty computation takes n steps to a state Σ', then the faulty computation with either take n steps to a state that simulates Σ', or it will terminate in the fault state *fault* during this time.

Lemma 31 (Multistep Fault Detection)

If $\vdash \Sigma$ *and* $\Sigma\ sim^c\ \Sigma_f$ *and* $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$ *then either*

1. $\Sigma_f \xrightarrow{n}{}^{s_f}_{0} \Sigma'_f$ *and* $\Sigma'\ sim^c\ \Sigma'_f$ *and* $s_f = s$*, or*

Example Case: st$_B$-mem

$$\frac{R_{val}(r_d) = n_l \qquad R_{val}(r_s) = n'_l}{(R, C, M, (\overline{(n,n')}, (n_l, n'_l)), st_B\ \ r_d, r_s) \xrightarrow{(n_l, n'_l)}_0 (R{+}{+}, C, M[n_l \mapsto n'_l], \overline{(n,n')}, \cdot)}\ (st_B\text{-}mem)$$

a1.	$\vdash (R, C, M, (\overline{(n,n')}, (n_l, n'_l)),\ st_B\ \ r_d, r_s)$	[assumption]
a2.	$(R, C, M, (\overline{(n,n')}, (n_l, n'_l)),\ st_B\ \ r_d, r_s)\ \ sim^c\ \Sigma_f$	[assumption]
a3.	$(R, C, M, (\overline{(n,n')}, (n_l, n'_l)),\ st_B\ \ r_d, r_s) \xrightarrow{(n_l, n'_l)}_0 (R{+}{+}, C, M[n_l \mapsto n'_l], \overline{(n,n')}, \cdot)$	[assumption]
p1.	$R_{val}(r_d) = n_l$	[premise]
p2.	$R_{val}(r_s) = n'_l$	[premise]
1.	$\Sigma_f = (R_f, C, M, Q_f, ir)$	[a2, definition of *sim*-Σ]
2.	$R\ sim^c\ R_f$	[a2, 1, inversion of *sim*-Σ]
3.	$(\overline{(n,n')}, (n_l, n'_l))\ sim^c\ Q_f$	[a2, 1, inversion of *sim*-Σ]
4.	$Q_f = \overline{(n_f, n'_f)}, (n_{lf}, n'_{lf})$	[3, definition of *sim-Q*]

Case on whether $R_{f\,val}(r_d) \overset{?}{=} n_{lf}$ and $R_{f\,val}(r_s) \overset{?}{=} n'_{lf}$ and the color c

subcase a: either r_s or r_d or the last pair in the queue has been corrupted

a4a.	$R_{f\,val}(r_d) \neq n_{lf}$ or $R_{f\,val}(r_s) \neq n'_{lf}$	[subcase assumption]
6a.	$(R_f, C, M, Q_f, st_B\ \ r_d, r_s) \longrightarrow_0 fault$	[st_B-*fail*, 4, a4a]
7a.	$\Sigma_f \xrightarrow{s_f}_0 fault$ and $s_f = ()$	[6a]

subcase complete.

subcase b: blue values are corrupted, but not r_s or r_d

a4b.	$R_{f\,val}(r_d) = n_{lf}$ and $R_{f\,val}(r_s) = n'_{lf}$	[subcase assumption]
a5b.	$c = B$	[subcase assumption]
6b.	$G\ n_l\ \ sim^B\ \ G\ n_{lf}$	[3, inversion of *sim-Q*]
7b.	$n_l = n_{lf}$	[6b, inversion of *sim-val*, a5b]
8b.	$G\ n'_l\ \ sim^B\ \ G\ n'_{lf}$	[3, inversion of *sim-Q*]
9b.	$n'_l = n'_{lf}$	[8b, inversion of *sim-val*, a5b]
10b.	$M[n_l \mapsto n'_l] = M[n_{lf} \mapsto n'_{lf}]$	[7b, 9b, a5b]
11b.	$R{+}{+}\ sim^B\ R_f{+}{+}$	[2, definition of R++, *sim-val*, *sim-R*, a5b]
12b.	$\overline{(n,n')}\ sim^B\ \overline{(n_f, n'_f)}$	[3, 4, inversion of *sim-Q*, a5b]
13b.	$(R{+}{+}, C, M[n_l \mapsto n'_l], \overline{(n,n')}, \cdot)\ sim^B\ (R_f{+}{+}, C, M[n_{lf} \mapsto n'_{lf}], \overline{(n_f, n'_f)}, \cdot)$	[10b, 11b, 12b]
14b.	$\Sigma_f \xrightarrow{(n_{lf}, n'_{lf})}_0 (R_f{+}{+}, C, M[n_{lf} \mapsto n'_{lf}], \overline{(n_f, n'_f)}, \cdot)$	[st_B-*mem*, a4b, 1, 4]
15b.	$(n_{lf}, n'_{lf}) = (n_l, n'_l)$	[7b, 9b]
16b.	$\Sigma_f \xrightarrow{s_f}_0 \Sigma'_f$ and $\Sigma'\ sim^c\ \Sigma'_f$ and $s_f = s$	[14b, 13b, 15b]

subcase complete.

subcase c: green values are corrupted, but not the last pair in the queue

a4c.	$R_{f\,val}(r_d) = n_{lf}$ and $R_{f\,val}(r_s) = n'_{lf}$	[subcase assumption]
a5c.	$c = G$	[subcase assumption]
6c.	$R_{col}(r_d) = R_{col}(r_s) = B$	[a1, inversion of Σ-*t*, inversion of st_B-*t*]
7c.	$R(r_d)\ sim^G\ R_f(r_d)$	[2, inversion of *sim-R*, a5c]
8c.	$R(r_s)\ sim^G\ R_f(r_s)$	[2, inversion of *sim-R*, a5c]
9c.	$R_{val}(r_d) = R_{f\,val}(r_d)$	[7c, 6c, inversion of *sim-val*]
10c.	$R_{val}(r_s) = R_{f\,val}(r_s)$	[8c, 6c, inversion of *sim-val*]
11c.	$n_l = n_{lf}$	[9c, a4c, p1]
12c.	$n'_l = n'_{lf}$	[10c, a4c, p2]
13c.	$M[n_l \mapsto n'_l] = M[n_{lf} \mapsto n'_{lf}]$	[11c, 12c]
14c.	$R{+}{+}\ sim^G\ R_f{+}{+}$	[2, definition of R++, *sim-val*, *sim-R*, a5c]
15c.	$\overline{(n,n')}\ sim^G\ \overline{(n_f, n'_f)}$	[3, 4, inversion of *sim-Q*, a5c]
16c.	$(R{+}{+}, C, M[n_l \mapsto n'_l], \overline{(n,n')}, \cdot)\ sim^G\ (R_f{+}{+}, C, M[n_{lf} \mapsto n'_{lf}], \overline{(n_f, n'_f)}, \cdot)$	[13c, 14c, 15c]
17c.	$\Sigma_f \xrightarrow{(n_{lf}, n'_{lf})}_0 (R_f{+}{+}, C, M[n_{lf} \mapsto n'_{lf}], \overline{(n_f, n'_f)}, \cdot)$	[st_B-*mem*, a4c]
18c.	$(n_{lf}, n'_{lf}) = (n_l, n'_l)$	[11c, 12c]
19c.	$\Sigma_f \xrightarrow{s_f}_0 \Sigma'_f$ and $\Sigma'\ sim^c\ \Sigma'_f$ and $s_f = s$	[17c, 16c, 18c]

subcase complete.

case complete.

Figure A.6: Example Case of Lemma 30.

2. *Exists* $m \leq n$. $\Sigma_f \xrightarrow{m}{}_0^{s_f} fault$ *and* s_f *is a prefix of* s.

Proof By induction on the structure of $\Sigma \xrightarrow{n}{}_0^{s} \Sigma'$. The base case for *multi-base* is immediate.

In the case for *multi-compose*, we know from the premises that Σ takes a single step to some Σ''. Using this and the Singlestep Fault Detection Lemma, we know that Σ_f either takes a single step with no output to fault or takes a single step to some state Σ''_f that simulates Σ'' while generating equal output.

In the former case, we can immediately prove the second possibility with $m = 1$ and $()$ as a prefix of s.

In the latter case, we call the Induction Hypothesis, which tells us that either Σ''_f takes $n-1$ steps to a state that simulates Σ' generating equal output, or it reaches *fault* in no more than $n-1$ steps with a prefix of the output. In the first case, where *fault* is never reached, we use *multi-compose* to conclude that Σ_f takes n steps to Σ'_f and the total output is equal. In the second case, where a *fault* is reached later in execution, we use *multi-compose* to show that Σ_f reaches *fault* in no more than n steps, and its output is a prefix of the non-faulty output. ■

A.4.4 Fault Similarity

The Fault Similarity Lemma states that if a non-faulty machine state takes a single faulty step, then the resulting machine state is similar to the original state under some color c.

Lemma 32 (Fault Similarity)

If $\Sigma \longrightarrow_1 \Sigma_f$, *then* $\exists\, c.\ \Sigma\ sim^c\ \Sigma'$.

Proof By case analysis on the definition of $\Sigma \longrightarrow_1 \Sigma'$. In each case, c is assigned the color of the value that is zapped. The zapped value is similar to the original value by *sim-val-zap*. The remainder of the state is equal, and is similar using *sim-val*. ■

A.4.5 Fault Tolerance Theorem

By using the three previous lemmas, we can state and prove the fault tolerance theorem for well-typed programs. Assume that machine state Σ is well-typed under the empty zap tag, and non-faulty execution of Σ for n steps results in a state Σ' and outputs a sequence of value-address pairs s. If somewhere during that execution a single fault is encountered, the faulty execution will either run for $n+1$ steps or terminate in the fault state during that time. If the faulty execution takes $n+1$ steps and reaches the non-faulty state Σ'_f, then Σ' simulates Σ'_f and the sequence of output pairs is identical the original execution. Alternatively, if the faulty execution reaches the fault state then the output pairs will be a prefix of the non-faulty output pairs.

Theorem 33 (Fault Tolerance)
If $\vdash \Sigma$ *and* $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$ *then either* $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$ *or* $\exists m \leq (n+1)\ .\ \Sigma \xrightarrow{m}{}^{s'}_{1} \mathit{fault}$*, and*

1. *For all derivations* $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$ *where* $\Sigma'_f \neq \mathit{fault}.\ s' = s$ *and* $\exists\, c.\ \Sigma'\ \mathit{sim}^c\ \Sigma'_f$.
2. *For all derivations* $\Sigma \xrightarrow{m}{}^{s'}_{1} \mathit{fault}$ *where* $m \leq (n+1).\ s'$ *is a prefix of* s.

Proof By case analysis on the definition of $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$.

In the base case for *multi-base*, we can easily prove result 1 by having Σ take a faulty step to Σ_f and using the Fault Similarity Lemma to show that $\exists\, c.\ \Sigma\ \mathit{sim}^c\ \Sigma_f$.

In the recursive case *multi-compose*, we use the Multistep Split Lemma to divide the computation into two pieces with an intermediate state Σ''. Σ'' can take a faulty step to Σ_f, and we can again use the Fault Similarity Lemma to show that they are similar. We then call the Multistep Fault Detection Lemma with the execution from Σ'' to Σ' and the similarity between Σ'' and Σ_f. This tells us that Σ_f can either step to a state Σ'_f that is similar to Σ' or reaches a fault before that point. Finally, we use the Multistep Combine Lemma to combine the first part of the non faulty execution, the fault step, and the resulting execution from the Multistep Fault Detection Lemma, to show that either $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$ or $\Sigma \xrightarrow{m}{}^{s'}_{1} \mathit{fault}$. The length of the faulty computation is at most $n+1$ because of the addition of the single fault step. Since the Multistep Split Lemma chooses some unspecified division of the original computation, the result hold regardless of exactly where the fault is injected. ■

Appendix B

MiniC Typing Rules

This appendix contains the typing rules for the MiniC Language defined in Section 3.1.

type	τ	$::=$	$int \mid \tau\ ref \mid X \rightarrow \tau$
variable context	X	$::=$	$\cdot \mid x : \tau,\ X$
value	v	$::=$	$n \mid x \mid ref\ v$
value list	vs	$::=$	$\cdot \mid v,\ vs$
binaryop	op	$::=$	$+ \mid - \mid *$
statement	s	$::=$	$x = v \mid x = v\ op\ v$
			$\mid x = !v \mid v := v$
			$\mid x = f(vs)$
			$\mid \texttt{if}\ v\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ v\ \texttt{do}\ s$
			$\mid s;\ s$
function declarations	fds	$::=$	$\cdot \mid \tau\ f(X)\ \{lds;\ s;\ \texttt{return}\ v\}\ fds$
local declarations	lds	$::=$	$\cdot \mid \tau\ x = v;\ lds$
program	p	$::=$	$fds;\ lds;\ s;\ \texttt{return}\ v;$

$\boxed{X \vdash v : \tau}$

$$\frac{}{X \vdash\ n : int}\ (n\text{-}t) \qquad \frac{}{X \vdash x : X(x)}\ (var\text{-}t) \qquad \frac{}{X \vdash ref\ v : \tau\ ref}\ (ref\text{-}t)$$

$\boxed{X \vdash vs : X}$

$$\frac{}{X \vdash \cdot : \cdot}\ (\textit{vs-}\cdot\textit{-t}) \qquad \frac{X \vdash v : \tau \qquad X \vdash vs : X}{X \vdash (v,\ vs)\ :\ (x : \tau,\ X)}\ (\textit{vs-t})$$

$\boxed{F;A;L \vdash s\ wf}$

$$\frac{L \vdash x : \tau \qquad (A \cup L) \vdash v : \tau}{F;A;L \vdash\ x = v\ \ wf}\ (\textit{s-assign-wf})$$

$$\frac{L \vdash x : int \qquad (A \cup L) \vdash v_1 : int \qquad (A \cup L) \vdash v_2 : int}{F;A;L \vdash\ x = v_1\ op\ v_2\ \ wf}\ (\textit{s-op-wf})$$

$$\frac{L \vdash x : \tau \qquad (A \cup L) \vdash v : \tau\ ref}{F;A;L \vdash\ x = !v\ \ wf}\ (\textit{s-deref-wf})$$

$$\frac{(A \cup L) \vdash v_1 : \tau\ ref \qquad (A \cup L) \vdash v_2 : \tau}{F;A;L \vdash\ v_1 := v_2\ \ wf}\ (\textit{s-update-wf})$$

$$\frac{(A \cup L) \vdash v : int \qquad F;A;L \vdash s_1\ wf \qquad F;A;L \vdash s_2\ wf}{F;A;L \vdash\ \texttt{if}\ v\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \ wf}\ (\textit{s-if-wf})$$

$$\frac{(A \cup L) \vdash v : int \qquad F;A;L \vdash s\ wf}{F;A;L \vdash\ \texttt{while}\ v\ \texttt{do}\ s\ \ wf}\ (\textit{s-while-wf})$$

$$\frac{L \vdash x : \tau \qquad F \vdash f : A' \to \tau \qquad (A \cup L) \vdash vs : A'}{F;A;L \vdash x = f(vs)\ wf}\ (\textit{s-call-wf})$$

$$\frac{F;A;L \vdash s_1\ wf \qquad F;A;L \vdash s_2\ wf}{F;A;L \vdash\ s_1 ; s_2\ \ wf}\ (\textit{s-seq-wf})$$

$$\boxed{F;A;L \vdash lds : L'}$$

$$\frac{}{F;A;L \vdash \cdot : L}\ (lds\text{-}\cdot\text{-}t)$$

$$\frac{\begin{array}{c} x \notin Dom(F \cup A \cup L) \\ (A \cup L) \vdash v : \tau \qquad F;A;L[x:\tau] \vdash lds : L' \end{array}}{F;A;L \vdash \ \tau\, x = v;\ lds\ : L'}\ (lds\text{-}t)$$

$$\boxed{F \vdash fds : F'}$$

$$\frac{}{F \vdash \cdot : F}\ (fds\text{-}\cdot\text{-}t)$$

$$\frac{\begin{array}{c} f \notin Dom(F) \\ F[f:(A \to \tau)];A;\cdot \vdash lds : L \\ F[f:(A \to \tau)];A;L \vdash s\ wf \\ (A \cup L) \vdash v : \tau \\ F[f:(A \to \tau)] \vdash fds : F' \end{array}}{F \vdash \ \tau\ f(A)\ \{lds;\ s;\ \texttt{return}\, v\}\ fds\ : F'}\ (fds\text{-}t)$$

$$\boxed{\vdash p\ wf}$$

$$\frac{\begin{array}{c} \cdot \vdash fds : F \\ F;\cdot \vdash lds : L \\ F;\cdot;L \vdash s\ wf \\ L \vdash v : int \end{array}}{\vdash \ fds;\ lds;\ s;\ \texttt{return}\, v\ \ \mathrm{wf}}\ (p\text{-}wf)$$

Appendix C

Complete Rules for $\mathrm{ETAL}_{\mathrm{FT}}$

This appendix contains the complete dynamic and static semantics for $\mathrm{ETAL}_{\mathrm{FT}}$. Section 3.2 discusses the main differences between $\mathrm{ETAL}_{\mathrm{FT}}$ and $\mathrm{TAL}_{\mathrm{FT}}$.

C.1 Syntax of Machine States

$$
\begin{array}{llcl}
colors & c & ::= & G \mid B \\
colored\ values & v & ::= & n \\
registers & r & ::= & r_n \\
general\ regs & a & ::= & r \mid gd \mid pc_c \mid sp_c \\
register\ file & R & ::= & \cdot \mid R, a \rightarrow n \\
code\ memory & C & ::= & \cdot \mid C, n \rightarrow i \\
value\ memory & M & ::= & \cdot \mid M, n \rightarrow n \\
store\ queue & Q & ::= & \overline{(n,n)} \\
ALU\ ops & op & ::= & add \mid sub \mid mul
\end{array}
$$

$$
\begin{array}{llcl}
instructions & i & ::= & op\ r_d, r_s, r_t \mid op\ r_d, r_s, n \mid mov\ r_d, n \mid mov\ r_d, r_s \\
 & & \mid & ld_c\ r_d, r_s \mid sld_c\ r_d\ n \mid st_c\ r_d, r_s \mid sst\ n\ r_v \\
 & & \mid & bz_c\ r_z, r_d \mid jmp_c\ r_d \\
 & & \mid & malloc[b]\ r_g,\ r_b \mid salloc\ n \mid sfree\ n \\
inst\ register & ir & ::= & i \mid \cdot \\
state & \Sigma & ::= & (R, C, M, Q, ir) \mid fault
\end{array}
$$

C.2 Dynamic Semantics

C.2.1 Fault Rules

$\boxed{\Sigma \longrightarrow_1^s \Sigma'}$

$$\frac{R(a) = n}{(R, C, M, Q, ir) \longrightarrow_1 (R[a \mapsto n'], C, M, Q, ir)} \; (\textit{reg-zap})$$

$$\frac{\begin{array}{c} Q_1 = \overline{(n_1, n_1')}, (m_1, m'), \overline{(n_2, n_2')} \\ Q_2 = \overline{(n_1, n_1')}, (m_2, m'), \overline{(n_2, n_2')} \end{array}}{(R, C, M, Q_1, ir) \longrightarrow_1 (R, C, M, Q_2, ir)} \; (Q_1\textit{-zap})$$

$$\frac{\begin{array}{c} Q_1 = \overline{(n_1, n_1')}, (m, m_1'), \overline{(n_2, n_2')} \\ Q_2 = \overline{(n_1, n_1')}, (m, m_2'), \overline{(n_2, n_2')} \end{array}}{(R, C, M, Q_1, ir) \longrightarrow_1 (R, C, M, Q_2, ir)} \; (Q_2\textit{-zap})$$

C.2.2 Normal Execution Rules

$\boxed{\Sigma \longrightarrow_0^s \Sigma'}$

$$\frac{R(pc_G) = R(pc_B) \quad R(pc_G) \in \mathit{Dom}(C)}{(R, C, M, Q, \cdot) \longrightarrow_0 (R, C, M, Q, C(R(pc_G)))} \; (\textit{fetch})$$

$$\frac{R(pc_G) \neq R(pc_B)}{(R, C, M, Q, \cdot) \longrightarrow_0 \; \mathit{fault}} \; (\textit{fetch-fail})$$

$$\frac{R' = R{+}{+}[r_d \mapsto R(r_s) \; op \; R(r_t))]}{(R, C, M, Q, op \;\; r_d, r_s, r_t) \longrightarrow_0 (R', C, M, Q, \cdot)} \; (\textit{op2r})$$

$$\frac{R' = R{+}{+}[r_d \mapsto R(r_s) \; op \; n]}{(R, C, M, Q, op \;\; r_d, r_s, n) \longrightarrow_0 (R', C, M, Q, \cdot)} \; (\textit{op1r})$$

$$\frac{R' = R{+}{+}[r_d \mapsto n]}{(R, C, M, Q, mov \; r_d \; n) \longrightarrow_0 (R', C, M', Q, \cdot)} \; (\textit{mov-n})$$

$$\frac{R' = R{+}{+}[r_d \mapsto R(r_s)]}{(R,C,M,Q,mov\ r_d\ r_s) \longrightarrow_0 (R',C,M',Q,\cdot)}\ (mov\text{-}reg)$$

$$\frac{\begin{array}{c} n = max(Dom(M)) + 1 \\ R' = R{+}{+}[r_g \mapsto n][r_b \mapsto n] \end{array}}{(R,C,M,Q,malloc[b]\ r_g\ r_b) \longrightarrow_0 (R',C,(M,n \mapsto 0),Q,\cdot)}\ (malloc)$$

$$\frac{\begin{array}{c} R' = R{+}{+}[sp_G \mapsto R(sp_G) - n][sp_B \mapsto R(sp_B) - n] \\ m = min(Dom(M)) \\ M' = (M, m-1 \mapsto 0, \ldots, m-n \mapsto 0) \end{array}}{(R,C,M,Q,salloc\ n) \longrightarrow_0 (R',C,M',Q,\cdot)}\ (salloc)$$

$$\frac{\begin{array}{c} R' = R{+}{+}[sp_G \mapsto R(sp_G) + n][sp_B \mapsto R(sp_B) + n] \\ m = min(Dom(M)) \\ M = M', m \mapsto v_m, \ldots, (m+n-1) \mapsto v_1 \end{array}}{(R,C,M,Q,sfree\ n) \longrightarrow_0 (R',C,M',Q,\cdot)}\ (sfree)$$

$$\frac{Q' = ((R(r_d),R(r_s)),Q)}{(R,C,M,Q,st_G\ \ r_d,r_s) \longrightarrow_0 (R{+}{+},C,M,Q',\cdot)}\ (st_G\text{-}queue)$$

$$\frac{R(rd) = n_1 \qquad R(r_s) = n'_1}{(R,C,M,(\overline{(n,n')},(n_l,n'_l)),st_B\ r_d,r_s) \longrightarrow_0^{(n_l,n'_l)} (R{+}{+},C,M[n_l \mapsto n'_l],\overline{(n,n')},\cdot)}\ (st_B\text{-}mem)$$

$$\frac{Q = (\overline{(n,n')},(n_l,n'_l)) \qquad R(r_d) \neq\ n_l\ or\ R(r_s) \neq\ n'_l}{(R,C,M,Q,st_B\ \ r_d,r_s) \longrightarrow_0\ fault}\ (st_B\text{-}mem\text{-}fail)$$

$$\frac{R(sp_B) = R(sp_G) \qquad R(sp_G) + n \in Dom(M)}{(R,C,M,Q,sst\ n,\ r_v) \longrightarrow_0^{(R(sp_G)+n,R(r_v))} (R{+}{+},C,M[R(sp_G)+n \mapsto R(r_v)],Q,\cdot)}\ (sst)$$

$$\frac{R(sp_G) \neq R(sp_B)\ or\ R(sp_B) + n \notin Dom(M)}{(R,C,M,Q,sst\ n,\ r_v) \longrightarrow_0\ fault}\ (sst\text{-}fail)$$

$$\frac{\begin{array}{c} find(Q,R(r_s)) = (R(r_s),n) \\ R' = R\text{++}[r_d \mapsto n] \end{array}}{(R,C,M,Q,ld_G\ \ r_d,r_s) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (ld_G\text{-}queue)$$

$$\frac{\begin{array}{c} find(Q,R(r_s)) = () \\ R(r_s) \in\ Dom(M) \\ R' = R\text{++}[r_d \mapsto M(R(r_s))] \end{array}}{(R,C,M,Q,ld_G\ \ r_d,r_s) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (ld_G\text{-}mem)$$

$$\frac{\begin{array}{c} R(r_s) \in\ Dom(M) \\ R' = R\text{++}[r_d \mapsto M(R(r_s))] \end{array}}{(R,C,M,Q,ld_B\ \ r_d,r_s) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (ld_B\text{-}mem)$$

$$\frac{\begin{array}{c} find(Q,R(r_s)) = () \\ R(r_s) \notin\ Dom(M) \end{array}}{(R,C,M,Q,ld_G\ \ r_d,r_s) \longrightarrow_0\ fault}\ (ld_G\text{-}fail)$$

$$\frac{R(r_s) \notin\ Dom(M)}{(R,C,M,Q,ld_B\ \ r_d,r_s) \longrightarrow_0\ fault}\ (ld_B\text{-}fail)$$

$$\frac{\begin{array}{c} find(Q,R(r_s)) = () \\ R(r_s) \notin\ Dom(M) \\ R' = R\text{++}[r_d \mapsto n] \end{array}}{(R,C,M,Q,ld_G\ \ r_d,r_s) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (ld_G\text{-}rand)$$

$$\frac{\begin{array}{c} R(r_s) \notin\ Dom(M) \\ R' = R\text{++}[r_d \mapsto n] \end{array}}{(R,C,M,Q,ld_B\ \ r_d,r_s) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (ld_B\text{-}rand)$$

$$\frac{\begin{array}{c} R(sp_c) + n \in Dom(M) \\ R' = R\text{++}[r_d \mapsto M(R(sp_c) + n)] \end{array}}{(R,C,M,Q,sld_c\, r_d,\ n) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (sld_c)$$

$$\frac{R(sp_c) + n \notin Dom(M)}{(R,C,M,Q,sld_c\, r_d,\ n) \longrightarrow_0 fault}\ (sld_c\text{-}fail)$$

$$\frac{R(gd) = 0 \qquad R' = R{+}{+}[gd \mapsto R(r_d)]}{(R,C,M,Q,jmp_G\ \ r_d) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (jmp_G)$$

$$\frac{R(gd) \neq\ 0}{(R,C,M,Q,jmp_G\ \ r_d) \longrightarrow_0\ \ fault}\ (\,jmp_G\text{-}fail)$$

$$\frac{\begin{array}{c} R(gd) \neq\ 0 \qquad R(r_d) = R(gd) \\ R' = R[pc_G \mapsto R(gd)][pc_B \mapsto R(r_d)][d \mapsto 0] \end{array}}{(R,C,M,Q,jmp_B\ \ r_d) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (jmp_B)$$

$$\frac{R(r_d) \neq R(gd)\ \ or\ R(gd) = 0}{(R,C,M,Q,jmp_B\ \ r_d) \longrightarrow_0\ \ fault}\ (\,jmp_B\text{-}fail)$$

$$\frac{R(gd) = 0 \qquad R(r_z) \neq 0}{(R,C,M,Q,bz_c\ \ r_z,r_d) \longrightarrow_0 (R{+}{+},C,M,Q,\cdot)}\ (bz\text{-}untaken)$$

$$\frac{\begin{array}{c} R(r_z) \neq\ 0 \\ R(gd) \neq\ 0 \end{array}}{(R,C,M,Q,bz_c\ \ r_z,r_d) \longrightarrow_0\ \ fault}\ (bz\text{-}untaken\text{-}fail)$$

$$\frac{\begin{array}{c} R(gd) = 0 \qquad R(r_z) = 0 \\ R' = R{+}{+}[gd \mapsto R(r_d)] \end{array}}{(R,C,M,Q,bz_G\ \ r_z,r_d) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (bz_G\text{-}taken)$$

$$\frac{\begin{array}{c} R(r_z) = 0 \\ R(gd) \neq\ 0 \end{array}}{(R,C,M,Q,bz_G\ \ r_z,r_d) \longrightarrow_0\ \ fault}\ (bz_G\text{-}taken\text{-}fail)$$

$$\frac{\begin{array}{c} R(gd) \neq 0 \qquad R(r_z) = 0 \\ R(r_d) = R(gd) \\ R' = R[pc_G \mapsto R(gd)][pc_B \mapsto R(r_d)][gd \mapsto 0] \end{array}}{(R,C,M,Q,bz_B\ \ r_z,r_d) \longrightarrow_0 (R',C,M,Q,\cdot)}\ (bz_B\text{-}taken)$$

$$\frac{\begin{array}{c} R(r_z) = 0 \\ R(r_d) \neq R(gd)\ \ or\ R(gd) = 0 \end{array}}{(R,C,M,Q,bz_B\ \ r_z,r_d) \longrightarrow_0\ \ fault}\ (bz_B\text{-}taken\text{-}fail)$$

C.3 Static Semantics

C.3.1 Syntax

Static Expressions

exp kinds	κ	$::=$	$\kappa_{int} \mid \kappa_{mem} \mid \kappa_{\sigma}$
exp contexts	Δ	$::=$	$\cdot \mid \Delta, x : \kappa$
exps	E	$::=$	$x \mid n \mid E\ op\ E \mid sel\ E_m\ E_n$
			$\mid emp \mid upd\ E_m\ E_{n_1}\ E_{n_2}$
substitutions	S	$::=$	$\cdot \mid S, E/x$

Types

zap tags	Z	$::=$	$\cdot \mid c$
initialization flags	φ	$::=$	$1 \mid \frac{1}{2} \mid 0$
base types	b	$::=$	$int \mid \Theta \to void \mid b\ ref^{\varphi} \mid sptr$
reg types	t	$::=$	$\langle c, b, E\rangle \mid E' = 0 \Rightarrow \langle c, b, E\rangle \mid ns$
reg file types	Γ	$::=$	$\cdot \mid \Gamma, a \to t$
unlabeled stack	σ	$::=$	$sbase \mid \rho \mid t :: \varsigma$
labeled stack	ς	$::=$	$E : \sigma$
result types	RT	$::=$	$\Theta \mid void$

Contexts

heap typing	Ψ	$::=$	$\cdot \mid \Psi, n : b$
static context	Θ	$::=$	$\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma$

C.3.2 Properties of Static Expressions

$\boxed{\Delta \vdash E : \kappa}$

$$\frac{x \in Dom(\Delta)}{\Delta \vdash x : \Delta(x)}\ (E\text{-}var\text{-}t)$$

$$\frac{}{\Delta \vdash n : \kappa_{int}}\ (E\text{-}int\text{-}t)$$

$$\frac{\begin{array}{c}\Delta \vdash E_1 : \kappa_{int} \\ \Delta \vdash E_2 : \kappa_{int}\end{array}}{\Delta \vdash E_1\ op\ E_2 : \kappa_{int}}\ (E\text{-}op\text{-}t)$$

$$\frac{\Delta \vdash E_m : \kappa_{mem} \quad \Delta \vdash E_n : \kappa_{int}}{\Delta \vdash sel\ E_m\ E_n : \kappa_{int}}\ (E\text{-}sel\text{-}t)$$

$$\frac{\Delta \vdash E_m : \kappa_{mem} \quad \Delta \vdash E_{n_1} : \kappa_{int} \quad \Delta \vdash E_{n_2} : \kappa_{int}}{\Delta \vdash upd\ E_m\ E_{n_1}\ E_{n_2} : \kappa_{mem}}\ (E\text{-}upd\text{-}t)$$

$$\frac{}{\Delta \vdash emp : \kappa_{mem}}\ (E\text{-}emp\text{-}t)$$

$\boxed{\Delta \vdash S : \Delta'}$

$$\frac{}{\Delta \vdash \cdot : \cdot}\ (sub\text{-}emp\text{-}t)$$

$$\frac{\Delta \vdash S : \Delta' \quad \Delta \vdash E : \kappa \quad x \notin Dom(\Delta) \cup Dom(\Delta')}{\Delta \vdash S, E/x : \Delta', x : \kappa}\ (sub\text{-}t)$$

$\boxed{[\![E]\!]}$

$$\begin{array}{lcl}
[\![n]\!] & = & n \\
[\![emp]\!] & = & \cdot \\
[\![E_1\ op\ E_2]\!] & = & [\![E_1]\!]\ op\ [\![E_2]\!] \\
[\![sel\ E_m\ E_n]\!] & = & [\![E_m]\!]([\![E_n]\!]) \\
[\![upd\ E_m\ E_1\ E_2]\!] & = & [\![E_m]\!][\ [\![E_1]\!] \mapsto [\![E_2]\!]\ ]
\end{array}$$

$\boxed{\Delta \vdash E_1\ op\ E_2}$

$$\frac{\begin{array}{c}\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int} \\ \forall S.\ \cdot \vdash S : \Delta \implies [\![S(E_1)]\!] = [\![S(E_2)]\!]\end{array}}{\Delta \vdash E_1 = E_2}\ (E\text{-}eq)$$

$$\frac{\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int} \qquad \forall S.\ \cdot \vdash S : \Delta \implies [\![S(E_1)]\!] \neq [\![S(E_2)]\!]}{\Delta \vdash\ E_1 \neq E_2}\ (\textit{E-neq})$$

$$\frac{\Delta \vdash E_1 : \kappa_{mem} \qquad \Delta \vdash E_2 : \kappa_{mem} \qquad \forall \ell \in\ Dom([\![S(E_1)]\!]) \cup Dom([\![S(E_2)]\!]).\ [\![S(E_1)]\!](\ell) = [\![S(E_2)]\!](\ell)}{\Delta \vdash\ E_1 = E_2}\ (\textit{E-mem-eq})$$

C.3.3 Value Typing

$\boxed{\Psi \vdash\ n : b}$

$$\frac{}{\Psi \vdash\ n : int}\ (\textit{int-t}) \qquad \frac{}{\Psi \vdash\ n : \Psi(n)}\ (\textit{addr-heap-t})$$

$$\frac{}{\Psi \vdash\ n : sptr}\ (\textit{addr-stack-t}) \qquad \frac{\Psi \vdash\ n : b\ ref^{\varphi} \qquad \varphi \leq \varphi'}{\Psi \vdash\ n : b\ ref^{\varphi'}}\ (\textit{addr-subtp-t})$$

$\boxed{\Psi; \Delta \vdash^{Z}\ n : t}$

$$\frac{\Psi \vdash\ n : b \qquad \Delta \vdash\ E = n}{\Psi; \Delta \vdash^{Z}\ n : \langle c, b, E \rangle}\ (\textit{val-t})$$

$$\frac{\Psi; \Delta \vdash^{Z}\ n : t' \qquad \Delta \vdash t' \leq t}{\Psi; \Delta \vdash^{Z}\ n : t}\ (\textit{val-subtp-t})$$

$$\frac{n \neq 0 \qquad \Psi; \Delta \vdash^{Z}\ n : \langle c, b, E \rangle \qquad \Delta \vdash\ E' = 0}{\Psi; \Delta \vdash^{Z}\ n : E' = 0 \Rightarrow \langle c, b, E \rangle}\ (\textit{cond-t})$$

$$\frac{\Delta \vdash\ E' \neq 0}{\Psi; \Delta \vdash^{Z}\ 0 : E' = 0 \Rightarrow \langle c, b, E \rangle}\ (\textit{cond-n0-t})$$

$$\frac{\Delta \vdash E : \kappa_{int}}{\Psi;\Delta \vdash^{c} n : \langle c, b, E\rangle}\ (\textit{val-zap-t})$$

$$\frac{\Delta \vdash E' : \kappa_{int} \qquad \Delta \vdash E : \kappa_{int}}{\Psi;\Delta \vdash^{c} n : E' = 0 \Rightarrow \langle c, b, E\rangle}\ (\textit{val-zap-cond-t})$$

$$\frac{}{\Psi;\Delta \vdash^{Z} n : ns}\ (\textit{ns-t})$$

C.3.4 Subtyping

$\boxed{\varphi\uparrow}$

$$\begin{array}{rcl} 0\uparrow & = & \tfrac{1}{2} \\ \tfrac{1}{2}\uparrow & = & \tfrac{1}{2} \\ 1\uparrow & = & 1 \end{array}$$

$\boxed{\varphi \le \varphi'}$

$$1 \le \frac{1}{2} \le 0 \qquad \varphi \le \varphi$$

$\boxed{b \le b'}$

$$\frac{}{b \le b}\ (\textit{subtp-b-reflex}) \qquad \frac{\varphi \le \varphi'}{b\ ref^{\varphi} \le b\ ref^{\varphi'}}\ (\textit{subtp-b-ref}) \qquad \frac{}{b \le int}\ (\textit{subtp-b-int})$$

$\boxed{\Delta \vdash t \le t'}$

$$\frac{\Delta \vdash E_1 = E_2 \qquad b_1 \le b_2}{\Delta \vdash \langle c, b_1, E_1\rangle \le \langle c, b_2, E_2\rangle}\ (\textit{subtp-t-triple})$$

$$\frac{\Delta \vdash\ t \leq t' \qquad \Delta \vdash\ E = E'}{\Delta \vdash\ (E = 0 \Rightarrow t) \leq (E' = 0 \Rightarrow t')}\ (\textit{subtp-t-cond})$$

$$\frac{}{\Delta \vdash\ t \leq ns}\ (\textit{subtp-t-ns})$$

$\boxed{\Delta \vdash\ \Gamma_1 \leq \Gamma_2}$

$$\frac{\forall r \in Dom(\Gamma_2).\ \Gamma_1(r) \leq \Gamma_2(r)}{\Delta \vdash\ \Gamma_1 \leq \Gamma_2}\ (\textit{reg-file-comp})$$

$\boxed{\Delta \vdash\ \varsigma \leq \varsigma'}$

$$\frac{\Delta \vdash E = E'}{\Delta \vdash\ E : sbase \leq E' : sbase}\ (\textit{subtp-}\varsigma\textit{-base})$$

$$\frac{\Delta \vdash E = E'}{\Delta \vdash\ E : \rho \leq E' : \rho}\ (\textit{subtp-}\varsigma\textit{-var})$$

$$\frac{\Delta \vdash E = E' \qquad \Delta \vdash t \leq t' \qquad \Delta \vdash \varsigma \leq \varsigma'}{\Delta \vdash\ E : (t :: \varsigma) \leq E' : (t' :: \varsigma')}\ (\textit{subtp-}\varsigma\textit{-cons})$$

C.3.5 Stack Typing Judgments

$\boxed{\Delta \vdash \varsigma\ \ wf}$

$$\frac{\Delta \vdash E : \kappa_{int}}{\Delta \vdash E : sbase\ \ wf}\ (\varsigma\text{-}\ wf\textit{-base}) \qquad \frac{\Delta(\rho) = \kappa_\sigma \qquad \Delta \vdash E : \kappa_{int}}{\Delta \vdash E : \rho\ \ wf}\ (\varsigma\text{-}\ wf\textit{-var})$$

$$\frac{\Delta \vdash E + 1 = E' \qquad \Delta \vdash (E' : \sigma')\ \ wf}{\Delta \vdash E : (t :: (E' : \sigma')\ \ wf}\ (\varsigma\text{-}\ wf\textit{-cons})$$

$\boxed{\Delta;\varsigma \vdash E : t}$

$$\frac{\Delta \vdash E_s = E}{\Delta;\ E_s : (t :: \varsigma')\ \vdash E : t}\ (\varsigma\textit{-lookup-top}) \qquad \frac{\Delta \vdash E_s \neq E \qquad \Delta;\varsigma' \vdash E : t}{\Delta;\ E_s : (t :: \varsigma')\ \vdash E : t}\ (\varsigma\textit{-lookup-tail})$$

$\boxed{\Delta \vdash \varsigma[E \mapsto t] = \varsigma'}$

$$\frac{\Delta \vdash E_s = E}{\Delta \vdash (E_s : (t_s :: \varsigma))[E \mapsto t] = E_s : (t :: \varsigma)}\ (\varsigma\textit{-update-top})$$

$$\frac{\Delta \vdash E_s \neq E \qquad \Delta \vdash \varsigma[E \mapsto t] = \varsigma'}{\Delta \vdash (E_s : (t_s :: \varsigma))[E \mapsto t] = E_s : (t_s :: \varsigma')}\ (\varsigma\textit{-update-tail})$$

C.3.6 Instruction Typing Rules

$\boxed{\Psi;\Theta \vdash\ ir\ \Rightarrow RT}$

$$\frac{}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m;\varsigma) \vdash\ \cdot\ \Rightarrow (\Delta;\Gamma;\overline{(E_d,E_s)};E_m;\varsigma)}\ (\cdot\textit{-t})$$

$$\frac{\begin{array}{c}\Gamma(r_s) = \langle c, int, E_s' \rangle \qquad \Gamma(r_t) = \langle c, int, E_t' \rangle \\ \Gamma' = \Gamma\text{++}[r_d \mapsto \langle c, int, E_s'\ op\ E_t' \rangle]\end{array}}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m;\varsigma) \vdash\ op\ \ r_d, r_s, r_t\ \Rightarrow (\Delta;\Gamma';\overline{(E_d,E_s)};E_m;\varsigma)}\ (\textit{op2r-t})$$

$$\frac{\Gamma(r_s) = \langle c, int, E_s' \rangle \qquad \Gamma' = \Gamma\text{++}[r_d \mapsto \langle c, int, E_s'\ op\ n \rangle]}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m;\varsigma) \vdash\ op\ \ r_d, r_s, n\ \Rightarrow (\Delta;\Gamma';\overline{(E_d,E_s)};E_m;\varsigma)}\ (\textit{op1r-t})$$

$$\frac{\Psi;\Delta \vdash\ n : t}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m;\varsigma) \vdash\ mov\ \ r_d, n\ \Rightarrow (\Delta;\Gamma\text{++}[r_d \mapsto t];\overline{(E_d,E_s)};E_m;\varsigma)}\ (\textit{mov-n-t})$$

$$\frac{\Gamma(r_s) = t}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ mov\ \ r_d, r_s\ \Rightarrow (\Delta; \Gamma{+}{+}[r_d \mapsto t]; \overline{(E_d, E_s)}; E_m; \varsigma)}\ (\textit{mov-reg-t})$$

$$\frac{\begin{array}{c} x \notin \Delta \\ \Gamma' = \Gamma{+}{+}[r_g \mapsto \langle G, b\ ref^0, x\rangle][r_b \mapsto \langle B, b\ ref^0, x\rangle] \\ E'_m = upd\ E_m\ x\ 0 \end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ malloc[b]\ r_g\ r_b\ \Rightarrow (\Delta, x : \kappa_{int}; \Gamma'; \overline{(E_d, E_s)}; E'_m; \varsigma)}\ (\textit{malloc-t})$$

$$\frac{\begin{array}{c} \Gamma(sp_G) = \langle G, sptr, E_g\rangle \qquad \Gamma(sp_B) = \langle B, sptr, E_b\rangle \\ \Delta \vdash E_g = E_b \qquad \Delta \vdash E_g = E_t \\ \Gamma' = \Gamma{+}{+}[sp_G \mapsto \langle G, sptr, (E_g{-}n)\rangle][sp_B \mapsto \langle B, sptr, (E_b{-}n)\rangle] \\ \varsigma' = (E_t{-}n) : ns :: (E_t{-}(n{+}1)) : ns :: \ldots :: E_t : \sigma \end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; (E_t : \sigma)) \vdash \ salloc\ n\ \Rightarrow (\Delta; \Gamma'; \overline{(E_d, E_s)}; E_m; \varsigma')}\ (\textit{salloc-t})$$

$$\frac{\begin{array}{c} \Gamma(sp_G) = \langle G, sptr, E_g\rangle \qquad \Gamma(sp_B) = \langle B, sptr, E_b\rangle \\ \Delta \vdash E_g = E_b \qquad \Delta \vdash E_g = E_t \\ \varsigma = E_t : t :: \ldots :: E_f : \sigma \qquad \Delta \vdash E_f = E_g + n \\ \Gamma' = \Gamma{+}{+}[sp_G \mapsto \langle G, sptr, (E_g{+}n)\rangle][sp_B \mapsto \langle B, sptr, (E_b{+}n)\rangle] \end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ sfree\ n\ \Rightarrow (\Delta; \Gamma'; \overline{(E_d, E_s)}; E_m; E_f : \sigma)}\ (\textit{sfree-t})$$

$$\frac{\Delta \vdash \Gamma(r_s) \le \langle G, b\ ref^{\frac{1}{2}}, E'_s\rangle \qquad E\ =\ sel\ (\overline{upd}\ E_m\ \overline{(E_d, E_s)})\ E'_s}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ ld_G\ r_d\ r_s\ \Rightarrow (\Delta; \Gamma{+}{+}[r_d \mapsto \langle G, b, E\rangle]; \overline{(E_d, E_s)}; E_m; \varsigma)}\ (\textit{ld}_G\textit{-t})$$

$$\frac{\Gamma(r_s) = \langle B, b\ ref^1, E'_s\rangle \qquad E\ =\ sel\ E_m\ E'_s}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ ld_B\ r_d\ r_s\ \Rightarrow (\Delta; \Gamma{+}{+}[r_d \mapsto \langle B, b, E\rangle]; \overline{(E_d, E_s)}; E_m; \varsigma)}\ (\textit{ld}_B\textit{-t})$$

$$\frac{\Gamma(sp_c) = \langle c, sptr, E_c\rangle \qquad \Delta \vdash E_c + n = E_n \qquad \Delta; \varsigma \vdash E_n : \langle c, b, E\rangle}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash \ sld_c\ r_d\ n\ \Rightarrow (\Delta; \Gamma{+}{+}[r_d \mapsto \langle c, b, E\rangle]; \overline{(E_d, E_s)}; E_m; \varsigma)}\ (\textit{sld}_c\textit{-t})$$

$$\frac{\begin{array}{c}\Gamma(r_d) = \langle G, b\ ref^{\varphi}, E'_d\rangle \qquad \Gamma(r_s) = \langle G, b, E'_s\rangle \\ \Gamma' = \Gamma\texttt{++}\ except\ \forall\ r\ where\ \Gamma(r) = \langle c_r, b\ ref^{\varphi}, E_r\rangle\ and\ \Delta \vdash E_r = E'_d\ . \\ \Gamma'(r) = \langle c_r, b\ ref^{\varphi\uparrow}, E_r\rangle\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\ st_G\ r_d\ r_s\ \Rightarrow (\Delta; \Gamma'; (E'_d, E'_s), \overline{(E_d, E_s)}; E_m; \varsigma)}\ (st_G\text{-}t)$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma(r_d) \le \langle B, b\ ref^{\frac{1}{2}}, E''_d\rangle \qquad \Gamma(r_s) = \langle B, b, E''_s\rangle \\ \Delta \vdash\ E'_s = E''_s \qquad \Delta \vdash\ E'_d = E''_d \\ \Gamma' = \Gamma\texttt{++}\ except\ \forall\ r\ where\ \Gamma(r) = \langle c_r, b\ ref^{\frac{1}{2}}, E_r\rangle\ and\ \Delta \vdash E_r = E'_d\ . \\ \Gamma'(r) = \langle c_r, b\ ref^{1}, E_r\rangle\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}, (E'_d, E'_s); E_m; \varsigma) \vdash\ st_B\ r_d\ r_s\ \Rightarrow (\Delta; \Gamma'; \overline{(E_d, E_s)}; upd\ E_m\ E'_d\ E'_s; \varsigma)}\ (st_B\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(sp_G) = \langle G, sptr, E_g\rangle \qquad \Gamma(sp_B) = \langle B, sptr, E_b\rangle \qquad \Delta \vdash E_g = E_b \\ \Delta \vdash E_g + n = E_n \qquad \Gamma(r_v) = \langle c, b, E_v\rangle \qquad \Delta \vdash \varsigma[E_n \mapsto \langle c, b, E_v\rangle] = \varsigma'\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\ sst\ n\ r_v\ \Rightarrow (\Delta; \Gamma\texttt{++}; \overline{(E_d, E_s)}; E_m; \varsigma')}\ (sst\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(gd) = \langle G, int, 0\rangle \qquad \Gamma(r_z) = \langle G, int, E_z\rangle \\ \Theta = (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m) \qquad \Gamma(r_d) = \langle G, \Theta \to void, E'_d\rangle \\ \Gamma'(gd) = \langle G, int, 0\rangle \qquad \Gamma'' = \Gamma\texttt{++}[gd \mapsto\ E_z = 0 \Rightarrow \langle G, \Theta \to void, E'_d\rangle]\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\ bz_G\ r_z\ r_d\ \Rightarrow (\Delta; \Gamma''; \overline{(E_d, E_s)}; E_m; \varsigma)}\ (bz_G\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(r_d) = \langle G, \Theta \to void, E_{rd'}\rangle \qquad \Theta = (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m) \\ \Gamma(gd) = \langle G, int, 0\rangle \qquad \Gamma'(gd) = \langle G, int, 0\rangle \\ \Gamma'' = \Gamma\texttt{++}[gd \mapsto\ \langle G, \Theta \to void, E_{rd'}\rangle]\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\ jmp_G\ r_d\ \Rightarrow (\Delta; \Gamma''; \overline{(E_d, E_s)}; E_m; \varsigma)}\ (jmp_G\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(r_z) = \langle B, int, E_z\rangle \\ \Gamma(r_d) = \langle B, (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m; \varsigma') \rightarrow void, E_r\rangle \\ \Gamma(gd) = E'_z = 0 \Rightarrow \langle G, (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m; \varsigma') \rightarrow void, E'_r\rangle \\ \Delta \vdash\ E_z = E'_z \\ \Delta \vdash\ E_r = E'_r \\ \exists S. \Delta \vdash\ S : \Delta' \\ S(\Gamma')(gd) = \langle G, int, 0\rangle \\ S(\Gamma')(pc_G) = \langle G, int, E'_r\rangle \\ S(\Gamma')(pc_B) = \langle B, int, E_r\rangle \\ \Delta \vdash \Gamma \leq\ S(\Gamma') \\ \Delta \vdash\ \overline{(E_d, E_s)} = S(\overline{(E'_d, E'_s)}) \\ \Delta \vdash\ E_m = S(E'_m) \\ \Delta \vdash\ \varsigma \leq S(\varsigma')\end{array}}{\begin{array}{c}\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\ bz_B\ r_z\ r_d\ \Rightarrow \\ (\Delta; \Gamma{+}{+}; \overline{(E_d, E_s)}; E_m; \varsigma)\end{array}}\ (bz_B\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(gd) = \langle G, (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m; \varsigma') \rightarrow void, E'_r\rangle \\ \Gamma(r_d) = \langle B, (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m; \varsigma') \rightarrow void, E_r\rangle \\ \Delta \vdash\ E_r = E'_r \\ \exists S. \Delta \vdash\ S : \Delta' \\ S(\Gamma')(gd) = \langle G, int, 0\rangle \\ S(\Gamma')(pc_G) = \langle G, int, E'_r\rangle \\ S(\Gamma')(pc_B) = \langle B, int, E_r\rangle \\ \Delta \vdash \Gamma \leq\ S(\Gamma') \\ \Delta \vdash\ \overline{(E_d, E_s)} = S(\overline{(E'_d, E'_s)}) \\ \Delta \vdash\ E_m = S(E'_m) \\ \Delta \vdash\ \varsigma \leq S(\varsigma')\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \vdash\ jmp_B\ r_d\ \Rightarrow void}\ (jmp_B\text{-}t)$$

C.3.7 Machine State Typing

$\boxed{\Psi \vdash^{Z} R : \Gamma}$

$$\frac{\begin{array}{c}\forall a \in Dom(\Gamma).\ \Psi; \cdot \vdash^{Z} R(a) : \Gamma(a) \\ \cdot \vdash \Gamma(pc_G) \leq \langle G, int, E_G \rangle \\ \cdot \vdash \Gamma(pc_B) \leq \langle B, int, E_B \rangle \\ \cdot \vdash E_G = E_B\end{array}}{\Psi \vdash^{Z} R : \Gamma}\ (\textit{reg-file-t})$$

$\boxed{\Psi \vdash C}$

$$\frac{\begin{array}{l}0 \notin Dom(C) \\ \ \forall n \in Dom(C).\quad \Psi(n) = \Theta \rightarrow void\ \wedge\ \Psi;\Theta \vdash C(n) \Rightarrow RT\ \wedge \\ \qquad\qquad\qquad\quad (RT = \Theta'\ implies\ \Psi(n+1) = \Theta' \rightarrow void)\end{array}}{\Psi \vdash C}\ (\textit{C-t})$$

$\boxed{M = M_s \overset{\mathcal{L}}{\#} M_m}$

$$\frac{\begin{array}{c}Dom(M) = Dom(M_1) \cup Dom(M_2) \\ Dom(M_1) \cap Dom(M_2) = \emptyset \\ \forall \ell_1 \in Dom(M_1).\ \forall \ell_{\mathcal{L}} \in \mathcal{L}.\ \forall \ell_2 \in Dom(M_2).\ \ell_1 < \ell_{\mathcal{L}} < \ell_2\end{array}}{M = M_1 \overset{\mathcal{L}}{\#} M_2}\ (\textit{\#-def})$$

$\boxed{\Psi; M; Q \vdash^{Z} \ell : b\ ref^{\varphi}}$

$$\frac{\Psi(\ell) = b\ ref^1 \qquad \Psi \vdash M(\ell) : b}{\Psi; M; Q \vdash^{Z} \ell : b\ ref^1}\ (\textit{init-t})$$

$$\frac{\Psi(\ell) = b\ ref^0}{\Psi; M; Q \vdash^{Z} \ell : b\ ref^0}\ (\textit{uninit-t})$$

$$\frac{\Psi(\ell) = b\ ref^{\frac{1}{2}} \qquad Z \neq G \implies \exists n.\ (\ell, n) \in Q}{\Psi; M; Q \vdash^{Z} \ell : b\ ref^{\frac{1}{2}}}\ (halfinit\text{-}t)$$

$$\boxed{\Psi \vdash^{Z}\ Q : \overline{(E_d, E_s)}}$$

$$\frac{}{\Psi \vdash^{Z}\ () : ()}\ (Q\text{-}emp\text{-}t) \qquad \frac{\Psi \vdash^{G}\ \overline{(n_1', n_2')} : \overline{(E_d', E_s')} \qquad \cdot \vdash\ E_d : \kappa_{int} \qquad \cdot \vdash\ E_s : \kappa_{int}}{\Psi \vdash^{G}\ (n_1, n_2), \overline{(n_1', n_2')} : (E_d, E_s), \overline{(E_d', E_s')}}\ (Q\text{-}zap\text{-}t)$$

$$\frac{Z \neq G \qquad \Psi \vdash^{Z}\ \overline{(n_1', n_2')} : \overline{(E_d', E_s')} \qquad \cdot \vdash E_d = n_1 \qquad \cdot \vdash E_s = n_2 \qquad \Psi \vdash n1 : b\ ref^{\varphi} \qquad \varphi \leq \frac{1}{2} \qquad \Psi \vdash n2 : b}{\Psi \vdash^{Z}\ (n_1, n_2), \overline{(n_1', n_2')} : (E_d, E_s), \overline{(E_d', E_s')}}\ (Q\text{-}t)$$

$$\boxed{\Psi \vdash^{Z}\ (M, Q) : (E_m, \overline{(E_d, E_s)})}$$

$$\frac{\forall \ell \in Dom(M).\ \exists \varphi.\ \Psi; M; Q \vdash^{Z} \ell : b\ ref^{\varphi} \qquad [\![E_m]\!] = M \qquad \Psi \vdash^{Z} Q : \overline{(E_d, E_s)}}{\Psi \vdash^{Z}\ (M, Q) : (E_m, \overline{(E_d', E_s')})}\ (heap\text{-}t)$$

$$\boxed{\Psi \vdash^{Z}\ M : \varsigma}$$

$$\frac{\cdot \vdash E = \ell \qquad Dom(M) = \{\ell\}}{\Psi \vdash^{Z}\ M : (E : sbase)}\ (\varsigma\text{-}t\text{-}base)$$

$$\frac{\cdot \vdash (E : t :: \varsigma')\ wf \qquad \cdot \vdash E = \ell \qquad M = \{\ell \to n\} \dot{\#} M' \qquad \Psi; \cdot \vdash^{Z} n : t \qquad \Psi \vdash^{Z} M' : \varsigma'}{\Psi \vdash^{Z} M : (E : t :: \varsigma')}\ (\varsigma\text{-}t\text{-}cons)$$

$\boxed{\vdash^Z (R, C, M, Q, ir)}$

$\boxed{\vdash^Z (R, C, M, Q, ir) : \Psi, \Gamma, \varsigma}$

$$\frac{\begin{array}{l} Dom(\Psi) = Dom(C) \cup Dom(M_m) \\ M = M_s \overset{Dom(C)}{\#} M_m \\ \Psi \vdash C \\ \forall c \neq Z.\ ir \neq \cdot \implies C(R(pc_c)) = ir \\ \forall c \neq Z.\ \Psi(R(pc_c)) = (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m; \varsigma) \rightarrow void \\ \exists S.\ \cdot \vdash S : \Delta \\ \Psi \vdash^Z M_s : S(\varsigma) \\ \Psi \vdash^Z (M_m, Q) : (S(E_m), S(\overline{(E_d, E_s)})) \\ \Psi \vdash^Z R : S(\Gamma) \\ K = extractK(R, \Gamma),\ extractK(M_h),\ extractK(M_s, \varsigma) \end{array}}{\vdash^Z (R, C, M, Q, ir) : K} \ (\Sigma\text{-}t)$$

C.4 Simulation of Machine States

C.4.1 Color Extraction

$$\begin{array}{llll} \textit{extended color} & k & ::= & c \mid GB \mid none \\ \textit{coloring} & K & ::= & \cdot \mid a \mapsto k \mid \ell \mapsto k \end{array}$$

$$\begin{array}{lcl} extractK_t(\langle c, b, E \rangle) & = & c \\ extractK_t(E_z = 0 \Rightarrow \langle c, b, E \rangle) & = & c \\ extractK_t(ns) & = & none \\ \\ extractK(R, \Gamma) & = & \forall a \in R.\ a \in Dom(\Gamma)\ ? \\ & & \qquad a \mapsto extractK_t(\Gamma(a)) \ : \ a \mapsto none \\ \\ extractK(M_s, \varsigma) & = & \forall \ell \in Dom(M_s).\ \ .; \varsigma \vdash \ell : t \ \Rightarrow\ \ell \mapsto extractK_t(t) \\ \\ extractK(M_h) & = & \forall \ell \in Dom(M_h).\ \ell \mapsto GB \end{array}$$

C.4.2 Simulation

$\boxed{k\ n_1\ sim^Z\ k\ n_2}$

$$\frac{}{k\ n\ sim^Z\ k\ n}\ (\mathit{sim\text{-}val}) \qquad \frac{}{c\ n\ sim^c\ c\ n'}\ (\mathit{sim\text{-}val\text{-}zap})$$

$$\frac{}{none\ n\ sim^Z\ none\ n'}\ (\mathit{sim\text{-}val\text{-}no\text{-}color})$$

$\boxed{K \vdash R\ sim^Z\ R'}$

$$\frac{\forall a.\ \ K(a)\ R(a)\ sim^Z\ K(a)\ R'(a)}{K \vdash R\ sim^Z\ R'}\ (\mathit{sim\text{-}R})$$

$\boxed{K \vdash M\ sim\ ^Z M'}$

$$\frac{\begin{array}{c} Dom(M) = Dom(M') \\ \forall \ell \in Dom(M).\quad K(\ell)\ M(\ell)\ \ sim^Z\ \ K(\ell)\ M'(\ell) \end{array}}{K \vdash M\ sim^Z\ M'}\ (\mathit{sim\text{-}M})$$

$\boxed{Q\ sim^Z\ Q'}$

$$\frac{}{\cdot\ sim^Z\ \cdot}\ (\mathit{sim\text{-}Q\text{-}empty})$$

$$\frac{G\ n_1\ sim^Z\ G\ n_1' \qquad G\ n_2\ sim^Z\ G\ n_2' \qquad Q\ sim^Z\ Q'}{((n_1,n_2),Q)\ \ sim^Z\ ((n_1',n_2'),Q')}\ (\mathit{sim\text{-}Q})$$

$\boxed{\Sigma_1\ sim^Z\ \Sigma_2}$

$$\frac{\begin{array}{c}\vdash (R, C, M, Q, ir) : K \\ \vdash^{Z} (R', C, M', Q', ir) : K \\ K \vdash R \; sim^{Z} \; R' \\ K \vdash M \; sim^{Z} \; M' \\ Q \; sim^{Z} \; Q'\end{array}}{(R, C, M, Q, ir) \; sim^{Z} \; (R', C, M', Q', ir)} \; (sim\text{-}\Sigma)$$

Appendix D

ETAL$_{\text{FT}}$ **Formal Results**

This appendix gives expanded details on the formal results for ETAL$_{\text{FT}}$ that are summarized in Section 3.3. All lemmas and theorems are included, along with proof sketches. The complete proofs in note form appear in the companion technical report [49].

D.1 Modified Lemmas

Many of the lemmas used in Appendix A to prove properties of TAL$_{\text{FT}}$ must be modified for use with ETAL$_{\text{FT}}$. A selection of the modified lemmas is provided below to illustrate the types of changes that are necessary.

Substituting an expression of kind κ for a free variable of type κ preserves typing. *(Modifications: Parts 5 and 7 contain the stack type and also to substitute in to the instruction as well (since* malloc *contains a base type). Parts 6 and 7 hold for incomplete substitutions as Δ is extended while type checking* malloc*.)*

Lemma 34 (Substitution Lemma)

1. *If* $\Delta, x:\kappa \vdash E' : \kappa'$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E'[E/x] : \kappa'$.

2. *If* $\Delta, x:\kappa \vdash E_1 = E_2$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E_1[E/x] = E_2[E/x]$.

3. *If* $\Delta, x:\kappa \vdash E_1 \neq E_2$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E_1[E/x] \neq E_2[E/x]$.

4. *If* $\Psi;\Delta, x:\kappa \vdash^Z v : t$ *and* $\Delta \vdash E : \kappa$ *then* $\Psi;\Delta \vdash^Z v : t[E/x]$.

5. *If* $\Psi;(\Delta, x:\kappa;\Gamma;\overline{(E_d,E_s)};E_m;\varsigma) \vdash^Z ir \Rightarrow RT$ *and* $\Delta \vdash E : \kappa$
then $\Psi;(\Delta;\Gamma[E/x];\overline{(E_d,E_s)}[E/x];E_m[E/x];\varsigma[E/x]) \vdash^Z ir[E/x] \Rightarrow RT[E/x]$.

6. *If* $\Delta' \vdash S : \Delta$ *and* $\Psi;\Delta \vdash^Z v : t$ *then* $\Psi;\Delta' \vdash^Z v : S(t)$.

7. *If* $\Delta' \vdash S : \Delta$ *and* $\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m;\varsigma) \vdash^Z ir \Rightarrow (\Delta;\Gamma';\overline{(E'_d,E'_s)};E'_m;\varsigma')$
then $\Psi;(\Delta';S(\Gamma);S(\overline{(E_d,E_s)});S(E_m)) \vdash^Z S(ir) \Rightarrow (\cdot;S(\Gamma');S(\overline{(E'_d,E'_s)});S(E'_m);S(\varsigma'))$.

Proof Parts 1, 4, and 5 – by induction on the respective typing derivation. Parts 2 and 3 – by inspection of the equality judgment definition. Parts 6 and 7 – by induction on the size of Δ, using parts 4 and 5 respectively. ■

The type of a value gives us information about the shape of the value. *(Modifications: The value's color tag is removed. The new memory typing judgment including the queue is used instead of the old judgment. Case 4 handles the initialization flag. Cases 7 and 8 are new. Note that the existence of a stack pointer tells nothing about the contents of the location it refers to.)*

Lemma 35 (Canonical Forms)

If $\Psi;\cdot \vdash^Z n : t$ *and* $Dom(\Psi) = Dom(C) \cup Dom(M)$ *and* $\Psi \vdash^Z (M,Q) : (E_m, \overline{(E_d,E_s)})$ *and* $\Psi \vdash C$*, then*

1. *If $t = \langle c, b, E\rangle$ or $t = (E' = 0) \Rightarrow \langle c, b, E\rangle$ and $c = Z$ then no particular properties of n are known.*

2. *If $t = \langle c, int, E\rangle$ and $c \neq Z$ then $\cdot \vdash E = n$.*

3. *If $t = \langle c, \Theta \rightarrow void, E\rangle$ and $c \neq Z$ then $\Psi(n) = \Theta \rightarrow void$ and $n \in Dom(C)$ and $\cdot \vdash E = n$ and $n \neq 0$.*

4. *If $t = \langle c, b\ ref^{\varphi}, E\rangle$ and $c \neq Z$ then*

   (a) $\Psi(n) = b\ ref^{\varphi'}$ and $\varphi' \leq \varphi$, and

   (b) $n \in Dom(M)$, and

   (c) $\cdot \vdash E = n$, and

   (d) $\varphi' = ref^{\frac{1}{2}} \wedge Z \neq G \implies (\exists\ (n,v) \in Q)\ \wedge\ (\forall\ (n,v) \in Q.\ \Psi \vdash v : b)$

   (e) $\varphi' = ref^{1} \implies \Psi \vdash M(\ell) : b\ \wedge\ (Z \neq G \implies \forall\ (n,v) \in Q.\ \Psi \vdash v : b)$

5. *If $t = (E' = 0) \Rightarrow t'$ and $c \neq Z$ and $\cdot \vdash E' = 0$ then $n \neq 0$.*

6. *If $t = (E' = 0) \Rightarrow t'$ and $c \neq Z$ and $\cdot \vdash E' \neq 0$ then $n = 0$.*

7. *If $t = ns$ then no particular properties of n are known.*

8. *If $t = \langle c, sptr, E\rangle$ and $c \neq Z$ then $\cdot \vdash E = n$*

Proof By induction on the structure of $\Psi; \cdot \vdash^{Z} n : t$. ■

If a value has a type, and this type has a supertype, then the value also has the supertype. *(Modifications: Part 3 is new.)*

Lemma 36 (Subtyping)

1. *If $\Psi; \Delta \vdash^{Z} v : t$ and $\Delta \vdash t \leq t'$ then $\Psi; \Delta \vdash^{Z} v : t'$.*

2. *If* $\Psi \vdash^{Z} R : \Gamma$ *and* $\Delta \vdash \Gamma \leq \Gamma'$ *then* $\Psi \vdash^{Z} R : \Gamma'$.

3. *If* $\Psi \vdash^{Z} M : \varsigma$ *and* $\Delta \vdash \varsigma \leq \varsigma'$ *then* $\Psi \vdash^{Z} M : \varsigma'$.

Proof Part 1 is by induction on the derivation of $\Psi; \Delta \vdash^{Z} v : t$. Parts 2 and 3 are by inversion and reconstruction of the appropriate typing rules and using Part 1. ■

If a judgment holds under the empty zap tag, then that judgment holds for a colored zap tag as well. *(Modifications: Part 3 is new.)*

Lemma 37 (Color Weakening)

1. *If* $\Psi; \cdot \vdash v : t$ *then* $\forall\, c.\ \Psi; \cdot \vdash^{c} v : t$.

2. *If* $\Psi \vdash R : \Gamma$ *then* $\forall\, c.\ \Psi \vdash^{c} R : \Gamma$.

3. *If* $\Psi \vdash M : \varsigma$ *then* $\forall\, c.\ \Psi \vdash^{c} M : \varsigma$.

Proof Part 1 by induction on the value typing judgment. The remaining parts are by inversion and reconstruction of the judgment using Part 1 as necessary. ■

D.2 New Lemmas For Stacks

If an expression in a stack type is referred to explicitly and memory M has that stack type, then there is some location ℓ equal to that expression that exists in the domain of M.

Lemma 38 (Valid Stack Location)

1. *If* $\Psi \vdash^{Z} M : \varsigma$ *and* $\cdot; \varsigma \vdash E' : t$ *then* $\exists \ell.\ \cdot \vdash E' = \ell$ *and* $\ell \in Dom(M)$.

2. *If* $\Psi \vdash^{Z} M : \varsigma$ *and* $\cdot \vdash \varsigma[E' \mapsto t] = \varsigma'$ *then* $\exists \ell.\ \cdot \vdash E' = \ell$ *and* $\ell \in Dom(M)$.

3. *If* $\Psi \vdash^{Z} M : \varsigma$ *and* $\varsigma = E_1 : t_1 :: \ldots E_n : t_n :: \varsigma'$ *then* $\exists \ell.\ \cdot \vdash E_i = \ell_i$ *and* $\ell_i \in Dom(M)$.

Proof By repeated inversion of ς-*t-cons* and ς-*t-base*, all expressions E on the spine of ς are equal to a location ℓ in the domain of M. By ς-*update* and ς-*lookup*, E' is equal to an expression on the spine of ς. ■

In addition, if an expression is looked up on the stack and is equal to a location, then the contents of that location have the type given by the stack type.

Lemma 39 (Stack Lookup)

If $\Psi \vdash^Z M : \varsigma$ *and* $\cdot;\varsigma \vdash E : t$ *and* $\cdot \vdash E = \ell$ *then* $\Psi;\cdot \vdash^Z M(\ell) : t$*.*

Proof By induction on the structure of $\cdot;\varsigma \vdash E : t$. ■

If a memory has a stack type and we update that stack type, then the memory updated with a value of the same type has the new stack type.

Lemma 40 (Stack Update)

If $\Psi \vdash^Z M : \varsigma$ *and* $\cdot \vdash \varsigma[E' \mapsto t] = \varsigma'$ *and* $\cdot \vdash E = \ell$ *and* $\Psi;\cdot \vdash^Z n : t$ *then* $\Psi \vdash^Z M[\ell \mapsto n] : \varsigma'$*.*

Proof By induction on the structure of $\cdot \vdash \varsigma[E' \mapsto t] = \varsigma'$. ■

D.3 New Lemmas For Dynamic Memory Allocation

Judgments that hold with Ψ continue to hold after adding extra information to Ψ.

Lemma 41 (Heap Extension Lemma)

1. *If* $\Psi;\cdot \vdash^Z v : t$ *and* $n \notin Dom(\Psi)$ *then* $\Psi, n \mapsto b;\cdot \vdash^Z v : t$

2. *If* $\Psi \vdash^Z M : \varsigma$ *and* $n \notin Dom(\Psi)$ *then* $\Psi, n \mapsto b \vdash^Z M : \varsigma$

3. *If* $\Psi; M; Q \vdash^Z \ell : b\ ref^{\varphi}$ *and* $n \notin Dom(\Psi)$ *then* $\Psi, n \mapsto b; M; Q \vdash^Z \ell : b\ ref^{\varphi}$

4. *If* $\Psi;\Theta \vdash ir \Rightarrow RT$ *and* $n \notin Dom(\Psi)$ *then* $\Psi, n \mapsto b;\Theta \vdash ir \Rightarrow RT$

Proof By induction on the appropriate derivations. ■

Judgments that hold with Ψ continue to hold after updating an entry in Ψ to contain a subtype of the original value.

Lemma 42 (Ψ Subtyping Lemma)

1. *If* $\Psi;\cdot \vdash^{Z} v : t$ *and* $b' \leq \Psi(n)$ *then* $\Psi[n \mapsto b'];\cdot \vdash^{Z} v : t$

2. *If* $\Psi;M;Q \vdash^{Z} \ell : b\ ref^{\varphi}$ *and* $b' \leq \Psi(n)$ *then* $\Psi[n \mapsto b'];M;Q \vdash^{Z} \ell : b\ ref^{\varphi}$

3. *If* $\Psi;\Theta \vdash ir \Rightarrow RT$ *and* $b' \leq \Psi(n)$ *then* $\Psi[n \mapsto b'];\Theta \vdash ir \Rightarrow RT$

Proof By case analysis of the appropriate derivations. Insert *addr-subtp-t* where the derivation uses $\Psi(n)$. ■

Judgments that hold with a substitution S continue to hold after adding extra information to S.

Lemma 43 (Substitution Extension Lemma)

1. *If* $\Psi;\cdot \vdash^{Z} v : S(t)$ *and* $\cdot \vdash S : \Delta$ *and* $x \notin \Delta$ *then* $\Psi;\cdot \vdash^{Z} v : (S, E/x)(t)$

2. *If* $\Psi \vdash^{Z} M : S(\varsigma)$ *and* $\cdot \vdash S : \Delta$ *and* $x \notin \Delta$ *then* $\Psi \vdash^{Z} M : (S, E/x)(\varsigma)$

3. *If* $\Psi \vdash^{Z} Q : S(\overline{(E_d, E_s)})$ *and* $\cdot \vdash S : \Delta$ *and* $x \notin \Delta$ *then* $\Psi \vdash^{Z} Q : (S, E/x)(\overline{(E_d, E_s)})$

4. *if* $\Psi \vdash^{Z} R : S(\Gamma)$ *and*$\cdot \vdash S : \Delta$ *and* $x \notin \Delta$ *then* $\Psi \vdash^{Z} R : (S, E/x)(\Gamma)$

5. *if* $M = [\![S(E_m)]\!]$ *and* $\cdot \vdash S : \Delta$ *and* $x \notin \Delta$ *then* $M = [\![(S, E/x)(E_m)]\!]$

Proof By induction on the appropriate derivations. ■

D.4 Type Safety

The statements of Progress and Preservation do not change, but the proofs are modified significantly.

Theorem 44 (Progress)

1. *If* $\vdash \Sigma$ *then* $\Sigma \longrightarrow^{s}_{0} \Sigma'$ *and* $\Sigma' \neq \mathit{fault}$.

2. *If* $\vdash^{c} \Sigma$ *then* $\Sigma \longrightarrow^{s}_{0} \Sigma$.

Proof By case analysis on the instruction *ir* in state Σ. *(Modifications: The existing cases require minor modifications to remove the color tags and use the modified typing judgments. Cases are added for the new instructions.)* ■

Theorem 45 (Preservation)

1. *If* $\vdash^{Z} \Sigma$ *and* $\Sigma \longrightarrow^{s}_{0} \Sigma'$ *and* $\Sigma' \neq \mathit{fault}$ *then* $\vdash^{Z} \Sigma'$.

2. *If* $\vdash \Sigma$ *and* $\Sigma \longrightarrow^{s}_{1} \Sigma'$ *then* $\exists\, c.\ \vdash^{c} \Sigma'$.

Proof By case analysis of the structure of the derivation $\Sigma \longrightarrow^{s}_{k} \Sigma'$. *(Part 1 Modifications: The new structure of the top level typing rules requires more significant changes to the existing cases. The new cases for stack load and store use Lemmas 38 (Valid Stack Location), 39 (Stack Update), and 40 (Stack Lookup). The rules for stack allocation and deallocation require proof that the affected locations are part of the stack and not the heap. The rule for memory allocation adds new information to Ψ, and heap stores modify entries in Ψ to contain subtypes. These rules use Lemmas 41 (Heap Extension), 42 (Ψ Subtyping), and 43 (Substitution Extension). Part 2 Modifications: The main change is handling faults to registers with no corresponding colors information. In this case, the resulting state can be typed using any zap tag.)* ■

D.5 Fault Tolerance

The main change affecting the fault tolerance results comes from the reorganization of the simulation relation explained in Section 3.3.2. This change affects the statements of the lemmas and theorems since the meaning of the sim^Z relation now includes typing information, but actually has little significant impact on the structure of the proofs.

The separation of the value of memory into a heap and a stack changes the definition of ”observable” memory. The new relationships $s' \overset{\ell}{\simeq} s$ and $s' \overset{\ell}{\preceq} s$ are used to relate two output sequences of address-value pairs. Because the faulty computation may store faulty values into the stack portion of memory, we can no longer use simple equality to compare address-value pairs. The judgment $\overset{\ell}{\simeq}$ say that the addresses in the two sequences are equal, and for all addresses greater than ℓ, the values are also equal. The judgment $s' \overset{\ell}{\preceq} s$ is similar, but only requires that the locations in the first sequence are a subsequence of those in the second sequence. In other words, when given $max(Dom(C))$ as ℓ, these judgments check that the stores committed to the heap are identical and the stores committed to the stack are to the same locations, though the values may differ.

The lemmas and theorems are restated with these modifications.

Lemma 46 (Singlestep Fault Detection)

If Σ sim^c Σ_f *and* $\Sigma \longrightarrow_0^s \Sigma'$ *then* $\Sigma_f \longrightarrow_0^{s_f} \Sigma'_f$ *and either*

1. Σ' sim^c Σ'_f *and* $s' \overset{max(Dom(\Sigma.C))}{\simeq} s$*, or*

2. $\Sigma'_f = fault$ *and* $s_f = ()$*.*

Proof By case analysis of $\Sigma \longrightarrow_0^s \Sigma'$. *(Modifications: Other than the slight restructurings to access the color information, the existing cases are essentially unmodified. New cases are added for each of the new singlestep rules.)* ■

Lemma 47 (Multistep Fault Detection)

If $\Sigma\ sim^c\ \Sigma_f$ *and* $\Sigma \xrightarrow{n}{}_0^s \Sigma'$ *then either*

1. $\Sigma_f \xrightarrow{n}{}_0^{s_f} \Sigma'_f$ *and* $\Sigma'\ sim^c\ \Sigma'_f$ *and* $s_f \overset{max(Dom(\Sigma.C)}{\simeq} s$*, or*
2. *Exists* $m \leq n$. $\Sigma_f \xrightarrow{m}{}_0^{s_f} fault$ *and* $s_f \overset{max(Dom(\Sigma.C)}{\preceq} s$.

Proof By induction on the structure of $\Sigma \xrightarrow{n}{}_0^s \Sigma'$. *(Modifications: Minor changes to support the new output comparisons.)* ■

Lemma 48 (Fault Similarity)

If $\Sigma \longrightarrow_1 \Sigma_f$*, then* $\exists\ c.\ \Sigma\ sim^c\ \Sigma'$.

Proof By case analysis on the definition of $\Sigma \longrightarrow_1 \Sigma'$. *(Modifications: Minor changes to support the changes to the simulation relation.)* ■

Theorem 49 (Fault Tolerance)

If $\vdash \Sigma$ *and* $\Sigma \xrightarrow{n}{}_0^s \Sigma'$ *then either* $\Sigma \xrightarrow{(n+1)}{}_1^{s'} \Sigma'_f$
or $\exists m \leq (n+1)\ .\ \Sigma \xrightarrow{m}{}_1^{s'} fault$*, and*

1. *For all derivations* $\Sigma \xrightarrow{(n+1)}{}_1^{s'} \Sigma'_f$ *where* $\Sigma'_f \neq fault$.
   $s' \overset{max(Dom(C))}{\simeq} s$ *and* $\exists\ c.\ \Sigma'\ sim^c\ \Sigma'_f$.

2. *For all derivations* $\Sigma \xrightarrow{m}{}_1^{s'} fault$ *where* $m \leq (n+1)$.
   $s' \overset{max(Dom(C))}{\preceq} s$.

Proof By case analysis on the definition of $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$. *(Modifications: Minor changes to support the new output comparisons.)* ■

Appendix E

MiniC to $\mathrm{ETAL}_{\mathrm{FT}}$ Translation

This appendix provides the complete rules for the translation from MiniC to $\mathrm{ETAL}_{\mathrm{FT}}$ and proof sketches of the formal results. A condensed version is given in Section 3.4, and the full proofs are available in [49].

E.1 Overview

In order to simplify the translation, it uses the designated registers pc_G, pc_B, sp_G, sp_B, and gd and then as many fresh temporary registers $t_1, t_2, \ldots$ as needed. Many of these temporary registers can be easily removed by coalescing move instructions. Section 3.6 discusses how to support register allocation if the number of temporary registers is greater than the number of actual registers.

The translation uses a simplistic calling convention. In order to support fault tolerance, all function arguments and return values need to be duplicated. Arguments are passed on the stack, with the last argument pushed on first. There will be two assembly-level arguments for each MiniC argument, and the green argument of each pair is always below the corresponding blue argument on the stack. Below the arguments are the blue and green copies of the return address. When a function returns, it pops the return address and all the arguments and pushes two copies of the return value.

On function entry, each function loads all the arguments into temporary registers. When making a function call, all temporary registers that correspond to local variables are spilled to the stack before the arguments and return address are pushed. After the call returns, the registers for the local variables are restored and then the return values are moved to their destinations.

At a high level, the translation works by passing around a code memory C and continually accumulating new instructions onto the end. The notation $C@i$ is used to append instruction i onto the end of code memory C. In addition, many judgments track the following additional information:

n	*the number of temporary registers required so far. freshReg*(n) *generates two fresh temporary registers and increments* n *appropriately.*
V	*a mapping from MiniC variables* x *to pairs of registers* (r_g, r_b). $V_g(x)$ *refers to the first component of the pair and* $V_b(x)$ *refers to the second.*
B	*a mapping from function variables to addresses in code memory.*

Because ETAL$_{\text{FT}}$ has no notion of termination, the program ”completes” by jumping to a designated label l_{halt} which contains four instructions that create an infinite loop.

E.2 Translating Typing Information

E.2.1 Type Translation

The type translation $[\![\tau]\!] = b$ in Figure E.1 translates a MiniC type τ into a ETAL$_{\text{FT}}$ base type b. The translation of integers and references is straightforward. The only thing to note is that the translation of a MiniC reference is a fully initialized ETAL$_{\text{FT}}$ reference.

The translation of the function types into code types is quite a bit more complicated. First, the two components of the function type are used to generate the type of the stack on entry to the function (defined in a moment) and the base type b_τ of the return value.

Next, we choose a number of fresh expression variables. At function entry, the memory is described by α_m, the program counters by α_p, and the return addresses by α_r. α'_m and α_τ will be used to describe the return memory and return values.

The contexts with a subscript r describe the return state of the function. When the function returns, the two program counters will contain the return address. The arguments and return address will have been removed from the stack and the return values pushed on in their place.

On function entry, the top of the stack contains two copies of the return address, followed by the stack type constructed using the function arguments. The stack pointers point to the top of the stack, the program counters are equal, and the destination register is zero.

E.2.2 Context Generation

The typing information can be used to generate a number of different ETAL$_{\text{FT}}$ contexts.

$\boxed{[\![\tau]\!] = b}$

$$\frac{}{[\![int]\!] = int}\ (\textit{trans-int}) \qquad\qquad \frac{}{[\![b\ ref]\!] = b\ ref^{1}}\ (\textit{trans-ref})$$

$$\frac{\begin{array}{lcl}
[\![A]\!] & = & \alpha_\ell{-}2{*}n : \langle G, b_1, \alpha_1\rangle \ :: \alpha_\ell{-}2{*}n{-}1 : \langle B, b_1, \alpha_1\rangle \\
 & & \quad :: \ldots \\
 & & \quad :: \alpha_\ell{-}2 : \langle G, b_n, \alpha_n\rangle \ :: \alpha_\ell{-}1 : \langle B, b_n, \alpha_n\rangle \\
 & & \quad :: \alpha_\ell : \alpha_\sigma \\
[\![\tau]\!] & = & b_\tau \\
 & & \\
\multicolumn{3}{l}{\alpha_m, \alpha_p, \alpha_r, \alpha_\tau, \alpha'_m \ fresh} \\
 & & \\
\Delta_r & = & \alpha'_m : \kappa_{mem}, \alpha_\tau : \kappa_{int} \\
\Gamma_r & = & pc_G \rightarrow \langle G, int, \alpha_r\rangle, sp_G \rightarrow \langle G, sptr, (\alpha_\ell{-}2)\rangle, \\
 & & \quad pc_B \rightarrow \langle B, int, \alpha_r\rangle, sp_B \rightarrow \langle B, sptr, (\alpha_\ell{-}2)\rangle, \\
 & & \quad gd \rightarrow \langle G, int, 0\rangle \\
\varsigma_r & = & (\alpha_\ell{-}2) : \langle G, b_\tau, \alpha_\tau\rangle \ :: (\alpha_\ell{-}1) : \langle B, b_\tau, \alpha_\tau\rangle \ :: \alpha_\ell : \alpha_\sigma \\
\Theta_r & = & (\Delta_r, \Gamma_r, (), \alpha'_m, \varsigma_r) \\
 & & \\
\Delta & = & \alpha_m : \kappa_{mem}, \alpha_p : \kappa_{int}, \alpha_r : \kappa_{int}, \alpha_\ell : \kappa_{int}, \alpha_\sigma : \kappa_\sigma, \\
 & & \quad \alpha_1 : \kappa_{int}, \ldots, \alpha_n : \kappa_{int} \\
\Gamma & = & pc_G \rightarrow \langle G, int, \alpha_p\rangle, sp_G \rightarrow \langle G, sptr, \alpha_\ell{-}2{*}n-2\rangle, \\
 & & \quad pc_B \rightarrow \langle B, int, \alpha_p\rangle, sp_B \rightarrow \langle B, sptr, \alpha_\ell{-}2{*}n-2\rangle, \\
 & & \quad gd \rightarrow \langle G, int, 0\rangle \\
\varsigma & = & (\alpha_\ell{-}2{*}n{-}2) : \langle G, \Theta_r \rightarrow void, \alpha_r\rangle \\
 & & \quad :: (\alpha_\ell{-}2{*}n{-}1) : \langle B, \Theta_r \rightarrow void, \alpha_r\rangle \ :: [\![A]\!]
\end{array}}{[\![A \rightarrow \tau]\!] = (\Gamma, \Delta, (), \alpha_m, \varsigma) \rightarrow void}\ (\textit{trans-}(X \rightarrow \tau))$$

Figure E.1: Type Translation.

The type of the stack on function entry is generated from the function arguments A. The arguments appear in reverse order. There are two copies of each argument, one green and one blue. The base type for each of these two copies is given by the type translation.

$\boxed{[\![A]\!] = \varsigma}$

$$\frac{\alpha_\ell, \alpha_\sigma \ \mathit{fresh}}{[\![\cdot]\!] = \alpha_\ell : \alpha_\sigma} \ (\mathit{gen}\text{-}\varsigma\text{-}\mathit{base})$$

$$\frac{[\![\tau]\!] = b_\tau \qquad [\![A]\!] = E_s : \sigma \qquad \alpha_\tau \ \mathit{fresh}}{[\![x : \tau,\ A]\!] = (E_s{-}2) : \langle G, b_\tau, \alpha_\tau\rangle :: (E_s{-}1) : \langle B, b_\tau, \alpha_\tau\rangle :: E_s : \sigma} \ (\mathit{gen}\text{-}\varsigma)$$

Using the function arguments A and the corresponding stack type ς, we can generate the register file type that results after loading each of these arguments into the register specified by V.

$\boxed{[\![A;\varsigma]\!]_V = \Gamma}$

$$\frac{}{[\![\ \cdot;\ \alpha_\ell : \alpha_\sigma\ ]\!]_V = \cdot} \ (\mathit{gen}\text{-}\Gamma\text{-}\mathit{args}\text{-}\mathit{emp})$$

$$\frac{\begin{array}{c}[\![A;\ E_s : \sigma]\!]_V = \Gamma \\ \Gamma' = \Gamma, V_g(x) \rightarrow t_G, V_b(x) \rightarrow t_B\end{array}}{[\![\ x : \tau, A;\ \ (E_s{-}2) : t_G :: (E_s{-}1) : t_B :: E_s : \sigma]\!] = \Gamma'} \ (\mathit{gen}\text{-}\Gamma\text{-}\mathit{args})$$

Given a variable context X and a mapping V from each variable in X to a pair of registers, $[\![X]\!]_V$ is the register file type that contains typing information for each of these registers. Each pair of registers is described by a different expression variable.

$\boxed{[\![X]\!]_V = \Gamma}$

$$\frac{}{[\![\cdot]\!]_V = \cdot}\ (\textit{gen-}\Gamma\textit{-X-}\cdot)$$

$$\frac{[\![X]\!]_V = \Gamma \qquad [\![\tau]\!] = b_\tau \qquad \alpha_\tau \textit{ fresh}}{[\![x : \tau, X]\!] = \Gamma, V_g(x) \rightarrow \langle G, b_\tau, \alpha_\tau\rangle, V_b(x) \rightarrow \langle B, b_\tau, \alpha_\tau\rangle}\ (\textit{gen-}\Gamma\textit{-X})$$

Finally, the judgment $[\![A \rightarrow \tau; L]\!]_V = \Theta$ can be used to generate the static context used within the body of a function. It includes the information about function entry generated by $[\![A \rightarrow \tau]\!]$, plus the additional registers corresponding to the arguments and local definitions in L.

$\boxed{[\![A \rightarrow \tau; L]\!]_V = \Theta}$

$$\frac{\begin{array}{c}[\![A \rightarrow \tau]\!] = (\Delta, \Gamma, (), E_m, \varsigma) \\ \Gamma' = \Gamma, [\![L]\!]_V, [\![A; \varsigma]\!]_V\end{array}}{[\![A \rightarrow \tau; L]\!]_V = (\Delta, \Gamma', (), E_m, \varsigma)}\ (\textit{gen-}\Theta)$$

E.3 Extending Code Memory

As the translation progresses, it continually adds new code to the code memory C. We define a series of judgments below that can be used to state that the in-progress code memory has the properties we desire.

As a reminder, the original code typing judgment is shown below. It requires that for all addresses in the code memory, if code type $\Theta \rightarrow \textit{void}$ is assigned to that address, then Θ can be used to type check the instruction in the address. In addition, if type checking the instruction gives a modified static context Θ', then the type assigned to the next location is $\Theta' \to void$. In other words, the only time two adjacent instructions do not need to agree on the intermediate type, is when the first instruction is a blue jump.

$\boxed{\Psi \vdash C}$

$$\frac{\begin{array}{l} 0 \notin Dom(C) \\ \ \forall n \in Dom(C). \quad \Psi(n) = \Theta \to void \ \wedge \ \Psi;\Theta \vdash C(n) \Rightarrow RT \ \wedge \\ \qquad\qquad\qquad\quad (RT = \Theta' \ implies \ \Psi(n+1) = \Theta' \to void) \end{array}}{\Psi \vdash C} \ (C\text{-}t)$$

E.3.1 Invariants Between Functions

The judgment $B;F \vdash C : P$ describes a code memory that contains the translations of the functions in F. The code memory should be well-typed using C-t. In addition, the mapping B should map each function in F to an address that has the ETAL$_{\text{FT}}$ type corresponding to the function type. The only items in the type Ψ are those for the actual addresses in C. And finally, the designated label l_{halt} that contains the code for an infinite loop, should have the type provided.

$\boxed{B;F \vdash C : P}$

$$\frac{\begin{array}{l} \Psi \vdash C \\ \forall f \in F.\ \Psi(B(f)) = [\![F(f)]\!] \\ \mathrm{dom}(\Psi) = \mathrm{dom}(C) \\ \Psi(l_{halt}) = \ (\ \begin{array}{l} (\ ap : \kappa_{int},\ \alpha_m : \kappa_{mem},\ \alpha_\ell : \kappa_{int},\ \alpha_\sigma : \kappa_\sigma \ ) \qquad\quad , \\ (\ pc_G \to < G, int, l_{halt} >, \quad pc_B \to < B, int, l_{halt} > \\ \ \ sp_G \to < G, sptr, \alpha_\ell > \quad\ \ sp_B \to < B, sptr, \alpha_\ell > \\ \ \ gd \to < G, int, 0 > \ ), \\ (),\ \alpha_m,\ \alpha_\ell : \alpha_\sigma \ \ ) \end{array} \end{array}}{B;F \vdash C : \Psi} \ wf\text{-}F$$

E.3.2 Partial Code Memories

While we are in the process of generating code, we are often in the situation where the last instruction added is in the middle of a code block. Since the remaining instructions have not yet been added, the code memory is not well-typed according to *C-t*.

Instead, we define the judgment below. $\Psi \vdash C : \Theta$ states that all the addresses in code memory C are well-typed in the usual manner, but that the last instruction results in static context Θ. Ψ already contains the typing information for the next instruction that will be added.

$$\boxed{\Psi \vdash C : \Theta}$$

$$\frac{\begin{array}{l} 0 \notin \mathrm{dom}(C) \\ l_m = \max(\mathrm{dom}(C)) \\ \mathrm{dom}(\Psi) = \mathrm{dom}(C) \cup (l_m + 1) \\ \forall l \in \mathrm{dom}(C). \\ \quad \Psi(l) = \Theta \to \mathrm{void} \wedge \\ \quad \Psi; \Theta \vdash C(l) \Rightarrow RT \wedge \\ \quad RT = \Theta' \Longrightarrow \Psi(l+1) = \Theta' \to \mathrm{void} \\ \Psi(l_m + 1) = \Theta \to \mathrm{void} \end{array}}{\Psi \vdash C : \Theta} \; \textit{C-partial-}\Theta$$

E.3.3 Invariants within a Function

In addition, there are some extra invariants which should hold within the body of a function. Before defining these, we need to define two sub-judgments.

A static context Θ is a subtype of another static context Θ' when all the corresponding elements have a subtyping relationship.

$\boxed{\Theta \leq \Theta'}$

$$\frac{\begin{array}{l}\exists S.\ \Delta \vdash\ S : \Delta' \\ \Delta \vdash \Gamma \leq S(\Gamma') \\ \Delta \vdash \overline{(E_d, E_s)} = S(\overline{(E'_d, E'_s)}) \\ \Delta \vdash\ E_m = S(E'_m) \\ \Delta \vdash\ \varsigma \leq S(\varsigma')\end{array}}{(\Delta, \Gamma, \overline{(E_d, E_s)}, E_m, \varsigma) \leq (\Delta', \Gamma', \overline{(E'_d, E'_s)}, E'_m, \varsigma')}\ \Theta\text{-}\mathit{subtp}$$

The judgment $X;V \vdash\ \Theta$ wf states that the static context Θ is consistent with program variables in X and the mapping V. In other words, for every variable in $X : \tau$, the two registers in $V(x)$ have base type $[\![\tau]\!]$, the first is green and the second is blue, and they are described by equivalent expressions.

$\boxed{X;V \vdash\ \Theta\ \text{wf}}$

$$\frac{\begin{array}{l}\Theta = (\Delta, \Gamma, seq, E_m, \varsigma) \\ \forall x : \tau \in X. \\ \quad V(x) = (r, r') \wedge \\ \quad \Gamma(r) =< G, [\![\tau]\!], E > \\ \quad \Gamma(r') =< B, [\![\tau]\!], E' > \\ \quad \Delta \vdash\ E = E'\end{array}}{X;V \vdash\ \Theta\ \text{wf}}\ \Theta\text{-}V\text{-}\mathit{wf}$$

The judgment $f;V;B;F;A;L \vdash\ C : \Psi$ should hold for any code memory C used in the process of translating function f. There are five components to this judgment

- The code memory is well formed and ends with some context Θ.
- Θ is consistent with the arguments and local variables.

- Θ is a subtype of the type obtained by translating the function type and the local declarations. (In other words, all the information from the function entry still holds, though there may be additional information about temporary registers.)
- All functions in the function context have correctly typed start addresses.
- The halt label l_{halt} has the appropriate type.

$\boxed{f;V;B;F;A;L \vdash C:\Psi}$

$$\frac{\begin{array}{l}\Psi \vdash C:\Theta \\ (A \cup L);V \vdash \Theta \ \text{wf} \\ \Theta \leq [\![F(f);L]\!]_V \\ \forall f' \in F.\ \Psi(B(f)) = [\![F(f')]\!] \\ \Psi(l_{halt}) = \ (\quad ap:\kappa_{int},\ \alpha_m:\kappa_{mem},\ \alpha_\ell:\kappa_{int},\ \alpha_\sigma:\kappa_\sigma\ ) \qquad , \\ \qquad\qquad (\ pc_G \rightarrow < G, int, l_{halt} >, \quad pc_B \rightarrow < B, int, l_{halt} > \\ \qquad\qquad\ sp_G \rightarrow < G, sptr, \alpha_\ell > \quad sp_B \rightarrow < B, sptr, \alpha_\ell > \\ \qquad\qquad\ gd \rightarrow < G, int, 0 >), \\ \qquad\qquad (),\ \alpha_m,\ \alpha_\ell:\alpha_\sigma\ )\end{array}}{f;V;B;F;A;L \vdash C:\Psi}\ \textit{wf-fun-body}$$

E.3.4 Adding Instructions to Code Memory

The Code Addition Lemma states that if we have a code memory that is well-typed ending in Θ and a sequence of instructions, the first of which is well-typed given Θ and who agree on the interleaving static contexts, then the sequence of instructions can be appended to the code memory, and the result will be the static context generated by the last instruction.

Lemma 50 (Code Addition)

If $\Psi \vdash C : \Theta_1$ *and* $\Psi;\Theta_1 \vdash i_1 : \Theta_2$ *and* $\Psi;\Theta_2 \vdash i_2 : \Theta_3$ *and* $\ldots$ *and* $\Psi;\Theta_n \vdash i_n : \Theta'$ *then* $\Psi' \vdash C@i_1@\cdots@i_n : \Theta'$ *where* $l_m = \max(\mathrm{dom}(\Psi))$ *and* $\Psi' = \Psi', l_m+1 \to (\Theta_2 \to \mathrm{void}), l_m+3 \to (\Theta_3 \to \mathrm{void}), \ldots, l_m+n \to (\Theta' \to \mathrm{void})$

Proof By inversion and reconstruction of *C-partial-t*. ∎

The Block Extension Lemma is similar except the added sequence of instructions begins a new block instead of appending on to a block in progress.

Lemma 51 (Block Extension)

If $\Psi \vdash C$ *and* $\Psi;\Theta_1 \vdash i_1 : \Theta_2$ *and* $\Psi;\Theta_2 \vdash i_2 : \Theta_3$ *and* $\ldots$ *and* $\Psi;\Theta_n \vdash i_n : \Theta'$ *then* $\Psi' \vdash C@i_1@\cdots@i_n : \Theta'$ *where* $l_m = \max(\mathrm{dom}(\Psi))$ *and* $\Psi' = \Psi', l_m+1 \to (\Theta_1 \to \mathrm{void}), \ldots, l_m+n \to (\Theta_n \to \mathrm{void}), l_m+n+1 \to (\Theta' \to \mathrm{void})$

Proof Similar to Lemma 50; deconstruct *C-t* to build first *C-partial-t*. ∎

E.4 Value Translation

The value translation appends code to the existing code memory that moves the value into the two registers r and r'. For a constant value n, n is moved into two fresh registers. For variables, it simply returns the existing register pair, and does not add any new instructions. References are created by allocating a new memory location, initializing that location, and then returning the two registers contain the new address.

$$\boxed{[\![X \vdash v : \tau]\!]\; C\; n\; V = C'\; n'\; r\; r'}$$

$$\frac{\begin{array}{l}(n',r,r') = \mathit{freshReg}(n)\\ C' = C \mathbin{@} \mathit{mov}\ r\ n \mathbin{@} \mathit{mov}\ r'\ n\end{array}}{[\![X \vdash\ n : \mathit{int}]\!]\ C\ n\ V\ =\ C'\ n'\ r\ r'}\ (\mathit{trans\text{-}v\text{-}n})$$

$$\frac{}{[\![X \vdash\ x : X(x)]\!]\ C\ n\ V\ =\ C\ n\ V_g(x)\ V_b(x)}\ (\mathit{trans\text{-}v\text{-}x})$$

$$\frac{\begin{array}{l}[\![X \vdash v : \tau]\!]\ C\ n\ V\ =\ C_v\ n_v\ r_v\ r'_v\\ [\![\tau]\!] = b_\tau\\ (n',r,r') = \mathit{freshReg}(n_v)\\ C' = C \mathbin{@} \mathit{malloc}[b_\tau]\ r\ r' \mathbin{@} \mathit{st}_G\ r\ r_v \mathbin{@} \mathit{st}_B\ r'\ r'_v\end{array}}{[\![X \vdash\ \mathit{ref}\ v : b\ \mathit{ref}]\!]\ C\ n\ V\ =\ C'\ n'\ r\ r'}\ (\mathit{trans\text{-}v\text{-}ref})$$

The Value Translation Lemma states that after adding the instructions to translate the value, the types of the two destination registers are based on the type of the value. The resulting register file type is a subtype of the starting register file type because additional temporaries may be added in rule *trans-v-ref*.

Lemma 52 (Value Translation)

If $[\![X \vdash\ v : \tau]\!]\ C\ n\ V\ =\ C'n'\ r\ r'$ *and* $\Psi \vdash\ C : (\Delta,\Gamma,\mathit{seq},E_m,\varsigma)$ *and* $X;V \vdash\ \Theta$ wf *then* $\exists E'_m,\Psi'.\ \ \Psi' \vdash\ C' : (\Delta,\Gamma',\mathit{seq},E_m,\varsigma)$ *and* $\Gamma' \leq \Gamma$ *and* $\Gamma'(r) = \langle G,[\![\tau]\!],E\rangle$ *and* $\Gamma'(r') = \langle B,[\![\tau]\!],E'\rangle$ *and* $\Delta \vdash\ E = E'$

Proof By case analysis on $[\![X \vdash\ v : \tau]\!]\ C\ n\ V\ =\ C'\ n'\ r\ r'$. ■

E.5 Value List Translation

The value list translation $[\![X \vdash vs : X]\!]\ C\ n\ V\ n_p = C'\ n'$ is used to push the arguments to a call onto the stack. It assumes that the stack space has already been allocated.

$$\boxed{[\![X \vdash vs : X]\!]\ C\ n\ V\ n_p = C'\ n'}$$

$$\frac{}{[\![X \vdash\ \cdot : \cdot]\!]\ C\ n\ V\ n_p\ =\ C\ n}\ (\textit{trans-vs-}\cdot)$$

$$\frac{\begin{array}{c}[\![X \vdash v : \tau]\!]\ C\ n\ V\ =\ C_v\ n_v\ r_v\ r_v' \\ C_v' = C_v\ @\ sst\ (2 * n_p)\ r_v\ @\ sst\ (2 * n_p + 1)\ r_v' \\ [\![X \vdash\ v, vs\ :\ x : \tau, X]\!]\ C_v'\ n_v\ V\ (n_p{+}1)\ =\ C'\ n'\end{array}}{[\![X \vdash\ v, vs\ :\ x : \tau, X]\!]\ C\ n\ V\ n_p\ =\ C'\ n'}\ (\textit{trans-vs})$$

The Value List Translation Lemma states that after the translation, the resulting stack type contains the types of the arguments in the appropriate slots. The translation may generate additional temporary registers while getting the values to be stored, and so the resulting register file type may contain extra information.

Lemma 53 (Value List Translation)

If $[\![X \vdash\ vs : ps]\!]\ C\ n\ V\ P_{num}\ =\ C'\ n'$ *and* $vs = x_1 : \tau_1, \cdots, x_n : \tau_n$ *and* $\Psi \vdash\ C : (\Delta, \Gamma, seq, E_m, \varsigma)$ *and* $\varsigma = E_l : t :: \cdots :: E_l + 2P_{num} : t_1 :: \cdots :: E_l + 2(P_{num} + n - 1) : t_n :: \varsigma''$ *then* $\Psi \vdash\ C' : (\Delta, \Gamma', seq, E_m', \varsigma')$ *where* $\varsigma' = E_l : t :: \cdots :: E_l + 2P_{num} :< G, [\![\tau_1]\!], E_1 >:: \cdots :: E_l + 2(P_{num} + n - 1) :< B, [\![\tau_n]\!], E_n >:: \varsigma''$ *and* $\Gamma' \leq \Gamma$.

Proof By induction on the structure of $[\![X \vdash\ vs : ps]\!]\ C\ n\ V\ P_{num}\ =\ C'\ n'$. The recursive case uses Lemma 52 and then calls the Induction Hypothesis.

■

E.6 Local Declaration Translation

The local declaration translation is a recursive judgment. For each declaration, the value is obtained and then moved into two fresh registers. The value mapping V is extended to map the variables to these registers.

$$\boxed{[\![F;A;L \vdash\ lds : L']\!]\ C\ n\ V = C'\ n'\ V'}$$

$$\frac{}{[\![F;A;L \vdash\ \cdot : L]\!]\ C\ n\ V\ =\ C\ n\ V}\ (\textit{trans-lds-empty})$$

$$\frac{\begin{array}{l}[\![(A \cup L) \vdash\ v : \tau]\!]\ C\ n\ V\ =\ C_v\ n_v\ r_v\ r'_v \\ (n_{ld}, r, r') = \textit{freshReg}(n_v) \\ V_{ld} = V[x \mapsto (r, r')] \\ C_{ld} = C_v @\textit{mov}\ r\ r_v\ @\textit{mov}\ r'\ r'_v \\ [\![F;A;L[x:\tau] \vdash\ lds : L']\!]\ C_{ld}\ n_{ld}\ V_{ld}\ =\ C'\ n'\ V'\end{array}}{[\![F;A;L \vdash\ \tau\ x\ =\ v;\ lds : L']\!]\ C\ n\ V\ =\ C'\ n'\ V'}\ (\textit{trans-lds})$$

The Local Declaration Translation Lemma states that the code for the translation of local variables lds consistently adds the types of those declarations.

Lemma 54 (Local Decl. Translation)

$$\begin{array}{ll} \textit{If} & [\![F;A;L \vdash\ lds : L']\!]\ C\ n\ V\ =\ C'\ n'\ V' \\ & f;V;B;F;A;L \vdash\ C : \Psi \\ \textit{then} & \exists \Psi'.\ f;V';B;F;A;L' \vdash\ C' : \Psi' \end{array}$$

Proof By induction on the structure of $[\![F;A;L \vdash\ lds : L']\!]\ C\ n\ V\ =\ C'\ n'\ V$. The recursive case uses Lemma 52 and then calls the Induction Hypothesis. ■

$$\boxed{[\![F;A;L \vdash s \text{ wf}]\!]_f\ C\ n\ V\ B = C'\ n'}$$

$$\frac{\begin{array}{l}[\![F;A;L \vdash\ s_1\ \text{ wf}]\!]_f\ C\ n\ V\ B\ =\ C_1\ n_1 \\ [\![F;A;L \vdash\ s_2\ \text{ wf}]\!]_f\ C_1\ n_1\ V\ B\ =\ C_2\ n_2\end{array}}{[\![F;A;L \vdash\ \ s_1;s_2\ \text{ wf}]\!]_f\ C\ n\ V\ B\ =\ C_2\ n_2}\ (\textit{trans-s-seq})$$

$$\frac{\begin{array}{l}[\![\,(A \cup L) \vdash v : \tau]\!]\ C\ n\ V\ =\ C_v\ n_v\ r_v\ r'_v \\ C' = C_v\ \ @\ mov\ V_G(x)\ r_v\ \ \ @\ mov\ V_B(x)\ r'_v\end{array}}{[\![F;A;L \vdash\ \ x = v\ \text{ wf}]\!]_f\ C\ n\ V\ B\ =\ C'\ n_v}\ (\textit{trans-s-assign})$$

$$\frac{\begin{array}{l}[\![\,(A \cup L) \vdash v_1 : int]\!]\ C\ n\ V\ =\ C_1\ n_1\ r_1\ r'_1 \\ [\![\,(A \cup L) \vdash v_2 : int]\!]\ C_1\ n_1\ V\ =\ C_2\ n_2\ r_2\ r'_2 \\ C' = C_2\ \ @\ op\ V_G(x)\ r_1\ r_2\ \ \ @\ op\ V_B(x)\ r'_1\ r'_2\end{array}}{[\![F;A;L \vdash\ \ x = v_1\ op\ v_2\ \text{ wf}]\!]_f\ C\ n\ V\ B\ =\ C'\ n2}\ (\textit{trans-s-op})$$

$$\frac{\begin{array}{l}[\![\,(A \cup L) \vdash v : \tau\ ref\,]\!]\ C\ n\ V\ =\ C_v\ n_v\ r\ r' \\ C' = C_v\ \ @\ ld_G\ V_G(x)\ r\ \ \ @\ ld_B\ V_B(x)\ r'\end{array}}{[\![F;A;L \vdash\ \ x =\ !v\ \text{ wf}]\!]_f\ C\ n\ V\ B\ =\ C'\ n_v}\ (\textit{trans-s-deref})$$

$$\frac{\begin{array}{l}[\![\,(A \cup L) \vdash v_1 : \tau\ ref\,]\!]\ C\ n\ V\ =\ C_1\ n_1\ r_1\ r'_1 \\ [\![\,(A \cup L) \vdash v_2 : \tau]\!]\ C_1\ n_1\ V\ =\ C_2\ n_2\ r_2\ r'_2 \\ C' = C_v\ \ @\ st_G\ r_1\ r_2\ \ \ @\ st_B\ r'_1\ r'_2\end{array}}{[\![F;A;L \vdash\ \ v_1 := v_2\ \text{ wf}]\!]_f\ C\ n\ V\ B\ =\ C'\ n2}\ (\textit{trans-s-update})$$

Figure E.2: Statement Translation - Basic Statements.

$$\frac{\begin{array}{l} l_{start} = \max(\text{dom}(C)) + 1 \\ [\![(A \cup L) \vdash\ v : int]\!]\ C\ n\ V\ =\ C_v\ n_v\ r_v\ r'_v \\ (n_c, re, re', rs, rs') = freshReg(n_v) \\ l_{fixend} = \max(\text{dom}(C_v)) + 1 \\ C_c = C_v\ \ @\ mov\ re\ l_{halt}\ \ \ @\ mov\ re'\ l_{halt} \\ \qquad\quad @\ bz_G\ r_v\ re\ \ \ \ @\ bz_B\ r'_v\ re' \\ [\![F; A; L \vdash\ s\ \ \text{wf}]\!]_f\ C_c\ n_c\ V\ = C_s\ n_s \\ C'_s = C_s\ \ @\ mov\ rs\ l_{start}\ \ \ @\ mov\ rs' @ l_{start} \\ \qquad\quad @\ jmp_G\ rs\ \ \ \ @\ jmp_B\ rs' \\ l_{end} = \max(\text{dom}(C'_s)) + 1 \\ C' = C'_s\ \ [l_{fixend} \ \ \mapsto\ \ mov\ re\ l_{end}] \\ \qquad\quad [l_{fixend} + 1\ \ \mapsto\ \ mov\ re'\ l_{end}] \\ n' = n_s \end{array}}{[\![F; A; L \vdash\ \texttt{while}\ v\ \texttt{do}\ s\ \ \text{wf}]\!]\ C\ n\ V\ B\ =\ C'\ n'}\ (\textit{trans-s-while})$$

Figure E.3: While Statement Translation.

$$\frac{\begin{array}{l} l_c = \max(\text{dom}(C)) + 1 \\ [\![(A \cup L) \vdash\ v : int]\!]\ C\ n\ v =\ C_v\ n_v\ r_v\ r'_v \\ (n_c, rt, rf, rjt, rjf, rt', rf', rjt', rjf', re, re') = freshReg(n_v) \\ l_{fixtrue} = \max(\text{dom}(C_v)) + 1 \\ C_c = C_v\ \ @\ mov\ rt\ l_{halt}\ \ \ @\ mov\ rt'\ l_{halt} \\ \qquad\quad @\ bz_G\ r_v\ rt\ \ \ \ @\ bz_B\ r'_v\ rt' \\ [\![F; A; L \vdash\ s_2\ \ \text{wf}]\!]_f\ C_c\ n_c\ V\ =\ C_f\ n_f \\ l_{fixjoin} = \max(\text{dom}(C_f)) + 1 \\ C'_f = C_f\ \ @\ mov\ rjf\ l_{halt}\ \ \ @\ mov\ rjf'\ l_{halt} \\ \qquad\quad @ jmp_G\ rjf\ \ \ \ @\ jmp_B\ rjf' \\ l_{true} = \max(\text{dom}(C'_f)) + 1 \\ [\![F; A; L \vdash\ s_1\ \ \text{wf}]\!]_f\ C'_f\ n_f\ V\ =\ C_t\ n_t \\ l_{join} = \max(\text{dom}(C_t)) + 1 \\ C'_t = C_t\ \ @\ mov\ re\ l_{join}\ \ \ @\ mov\ re'\ l_{join} \\ \qquad\quad @\ jmp_G\ re\ \ \ \ @\ jmp_B\ re' \\ C' = C'_t\ \ [l_{fixjoin} \ \ \mapsto\ \ mov\ rjf\ l_{join}] \\ \qquad\quad [l_{fixjoin} + 1\ \ \mapsto\ \ mov\ rjf'\ l_{join}] \\ \qquad\quad [l_{fixtrue} \ \ \mapsto\ \ mov\ rt\ l_{true}] \\ \qquad\quad [l_{fixtrue} + 1\ \ \mapsto\ \ mov\ rt'\ l_{true}] \\ n' = n_t \end{array}}{[\![F; A; L \vdash\ \texttt{if}\ v\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \ \text{wf}]\!]_f\ C\ n\ V\ B\ =\ C'\ n'}\ (\textit{trans-s-if})$$

Figure E.4: If Statement Translation.

$$\frac{\begin{array}{l}
argspace = (\mathrm{size}(vs) * 2) + 2 \\
vspace = \mathrm{size}(\mathrm{dom}(V)) * 2 \\
spillspace = vspace + argspace \\
C_{temps} = \begin{array}[t]{l} @\ salloc\ argspace \\ @\ sst\ (argspace + 0)\ V_g(x_1) \\ @\ sst\ (argspace + 1)\ V_g b(x_1) \\ @\ \cdots @\ sst\ (argspace + vspace - 2)\ V_g(x_n) \\ \\ @\ sst\ (argspace + vspace - 1)\ V_b(x_n) \end{array} \\
[\![(A \cup L) \vdash\ vs : ps]\!]\ C_{temps}\ n\ V\ 1 =\ C_{vs}\ n_{vs} \\
(n_{call}, rf, rf', ra, ra') = freshReg(n_{vs}) \\
retaddr = \max(\mathrm{dom}(C_{vs})) + 9 \\
C_{call} = C_{vs} \begin{array}[t]{ll} @\ mov\ ra\ retaddr & @sst\ 0\ ra \\ @\ mov\ ra'\ retaddr & @sst\ 1\ ra' \\ @\ mov\ rf\ B(g) & @mov\ rf'\ B(f) \\ @\ jmp_G\ rf & @jmp_B\ rf' \\ @\ sld_G\ 2\ V_g(x_1) & \\ @\ sld_G\ 3)\ V_b(x_1) & \\ @\ \cdots & \\ @\ sld_B\ (vspace) V_g(x_n) & \\ @\ sld_B\ (vspace + 1) V_b(x_n) & \\ @\ sld_G\ V_g(x)\ 0 & @\ sld_B\ V_b(x)\ 1 \\ @\ sfree(vspace + 2) & \end{array}
\end{array}}{[\![F; A; L \vdash\ \ x = g(vs)\ \ \mathrm{wf}]\!]_f\ C\ n\ V\ B\ =\ C_{call}\ n_{call}}\ (\textit{trans-s-call})$$

Figure E.5: Function Call Translation.

E.7 Statement Translation

The statement translations are shown in Figures E.2, E.3, E.4, and E.5. The rules that do not involve control flow transfers are very straightforward.

The rules for `while` and `if` are quite a bit more involved. The main complication comes from the backpatching required to fill in the correct addresses for jump targets. Because the code is generated sequentially, those instructions that jump forward do not know the correct address to use at the point they are generated. Instead, they temporarily jump to l_{halt} and are later patched to jump to the correct address. The call instruction generates a lot of code, but this code just implements the calling convention discussed earlier.

The Statement Translation Lemma says that translating a statement s has no effect on the main typing information. Again, there may be changes to additional temporary registers and the description of memory, but the final type is still consistent with the function arguments and local declarations.

Lemma 55 (Statement Translation)

If $[\![F;A;L \vdash\ s\ \text{ wf}]\!]_f\ C\ n\ V\ B\ =\ C'\ n'$ *and* $f;V;B;F;A;L \vdash C : \Psi$
then $\exists\Psi'.\ f;V;B;F;A;L \vdash\ C' : \Psi'$.

Proof By induction on the structure of $[\![F;A;L \vdash\ s\ \text{ wf}]\!]_f\ C\ n\ V\ B\ =\ C'\ n$ and Lemma 52. ■

E.8 Function Translation

The function translation depends on two auxiliary judgments to generate the function entry and exit sequences.

E.8.1 Prologue and Epilogue

The prologue generates code that loads the arguments from the stack into fresh registers and updates the variable mapping appropriately.

$$\boxed{\mathit{genPrologue}(C,n,V,n_p,A) = (C',n',V')}$$

$$\frac{}{\mathit{genPrologue}(C,n,V,n_p,\cdot) = (C',n',V')}\ (\mathit{gen\text{-}prologue\text{-}emp})$$

$$\frac{\begin{array}{l}(n_x,r,r') = \mathit{freshReg}(n)\\ C_x = C\ \ @\ \ \mathit{sld}_G\ (2*n_p+2)\ r\ \ @\ \ \mathit{sld}_B\ (2*n_p+3)\ r'\\ \mathit{genPrologue}(C_x,\ n_x,\ V[x\mapsto(r,r')],\ (n_p+1),\ A) = (C',\ n',\ V')\end{array}}{\mathit{genPrologue}(C,\ n,\ V,\ n_p,\ (x:\tau,A)\ ) = (C',\ n',\ V')}\ (\mathit{gen\text{-}prologue})$$

Before the prologue code, C is well-typed with regard to the set of functions defined. Afterwards, C' is well-typed with regard to the body of function f (though there are no local declarations yet).

Lemma 56 (Prologue Generation)

If $\mathit{genPrologue}(ps)\ C\ n\ V\ P_{num}\ =\ C'\ n'\ V'$ *and* $F;B\vdash\ C$

then $\exists\Psi'.\ f;V';B;F[f:ps\rightarrow\tau];ps;\cdot\vdash\ C':\Psi'$.

Proof By induction on the structure of $\mathit{genPrologue}(ps)\ C\ n\ V\ P_{num}\ =\ C'\ n'\ V'$. ■

The epilogue generates code to free the function arguments and return addresses and push the return values.

$$\boxed{genEpilogue(X \vdash v : \tau,\ C, n,\ V,\ A) = (C', n')}$$

$$\frac{\begin{array}{lll} [\![X \vdash v : \tau]\!]\ C\ n\ V\ =\ C_v\ n_v\ r_v\ r'_v & & \\ (n', r, r') = freshReg(n_v) & & \\ C' = C & @\ sld_G\ 0\ r & @\ sld_B\ 1\ r' \\ & @\ sfree\ 2 * size(A) + 2 & @\ salloc\ 2 \\ & @\ sst\ 0\ r_v & @\ sst\ 1\ r'_v \\ & @\ jmp_G\ r & @\ jmp_B\ r' \end{array}}{genPrologue(X \vdash v : \tau,\ C, n,\ V,\ A) = (C', n')}\ (gen\text{-}epilogue)$$

Before the epilogue, C is well-typed with regard to the body of function f. Afterwards, C' is well-typed with regard to the set of functions.

Lemma 57 (Epilogue Generation)

If $genEpilogue((A \cup L) \vdash\ v : \tau, C, n, A) = (C', n')$ *and* $f; V; B; F; A; L \vdash\ C : \Psi$ *then* $\exists \Psi'.\ B; F \vdash\ C : \Psi'$.

Proof By deconstruction of (wf-fun-body) and the appropriate instruction typing rules. ■

E.8.2 Function Translation Judgment

The function translation judgment generates the function prologue, translates the local declaration, translates the function body, and generates the epilogue for each function. The maximum number of registers required by the set of functions is the maximum required by any single function.

$$\boxed{[\![F \vdash\ fds : F']\!]\ C\ B\ =\ C'\ B'\ n}$$

$$\frac{}{[\![F \vdash\ \cdot : F]\!]\ C\ B\ =\ C\ B}\ (\textit{trans-fds-empty})$$

$$\frac{\begin{array}{l} f_{addr} = \max(\mathrm{dom}(C)) + 1 \\ \mathit{genPrologue}(C, 0, V, 0, A) = C_p\ n_p\ V_p \\ [\![F[f : A \to \tau]; A; \cdot \vdash\ lds : L]\!]\ C_p\ n_p\ V_p\ =\ C_{lds}\ n_{lds}\ V_{lds} \\ [\![F[f : A \to \tau]; A; L \vdash\ s\ \ \mathrm{wf}]\!]_f\ C_{lds}\ n_{lds}\ B[f \mapsto f_{addr}]\ =\ C_s\ n_s \\ \mathit{genEpilogue}((A \cup L) \vdash v : \tau,\ C_s, n_s,\ V_{lds},\ A) = (C', n_r) \\ [\![F[f : A \to \tau] \vdash\ fds : F']\!]\ C\ B[f \mapsto f_{addr}]\ = C'\ B'\ n_{fds} \\ n = \max(n_r, n_{fds}) \end{array}}{[\![F \vdash\ \tau\ f(A)\ \{lds; s; \mathtt{return}\ v\}\ fds : F']\!]\ C\ B\ = C'\ B'\ n'}\ (\textit{trans-fds})$$

The Function Translation Lemma states that the code memory before and after translating a set of functions is well-typed with regards to those functions which are defined.

Lemma 58 (Function Translation)

If $[\![F \vdash\ fds : F']\!]\ C\ B\ =\ C'\ B'\ n$ *and* $B; F \vdash C : \Psi$

then $\exists \Psi'.\ B'; F' \vdash C' : \Psi'$

Proof By induction on the structure of $[\![F \vdash\ lds : F']\!]\ C\ B\ =\ C'\ B'\ n$. The recursive case uses Lemmas 56 (Prologue Generation), 54 (Local Declaration Translation), 55 (Statement Translation), and 57 (Epilogue Generation) and then calls the induction hypothesis.

∎

E.9 Program Translation

The final judgment $[\![\vdash\ p\ \text{ wf}]\!] =\ C\ n\ l$ takes a MiniC program p and generates a code memory C, a maximum number of registers n, and a start address l. It initializes the code memory to contain four instructions that create an infinite loop

$$\boxed{[\![\vdash\ p\ \text{ wf}]\!] = C\ n\ l}$$

$$\frac{\begin{array}{l} C_{start} = l_{start} \mapsto mov\ t_0\ l_{halt},\ \ l_{start}+1 \mapsto mov\ t_1\ l_{halt} \\ \qquad\qquad l_{start}+2 \mapsto jmp_G\ t_0,\ \ l_{start}+3 \mapsto jmp_B\ t_1 \\ [\![\cdot \vdash\ fds : F\,]\!]\ C_{start}\ 0\ \cdot\ =\ C_{fds}\ B_{fds}\ n_{fds} \\ l_{main} = \max(\text{dom}(C_{fds}))+1 \\ [\![F;\cdot;\cdot \vdash\ lds : L]\!]\ C_{fds}\ 0\ \cdot\ =\ C_{lds}\ n_{lds}\ V_{lds} \\ [\![F;\cdot;L \vdash\ s\ \text{ wf}]\!]_f\ C_{lds}\ n_l ds\ V_{lds}\ B_{fds}\ =\ C_s\ n_s \\ genEpilogue(L \vdash\ v:\tau,\ C_s, n_s,\ V_{lds},\ ())\ =\ C'\ n_r \\ n' = \max(n_r, n_{fds}) \end{array}}{[\![\vdash\ fds; lds; s; \texttt{return}\, v\ \text{ wf}]\!] =\ C'\ n'\ l_{main}}\ (\textit{trans-p})$$

Finally, the Translation Theorem states that the result of translating a well-formed program p can be used to create a well-typed ETAL$_{\text{FT}}$ machine state. The code memory is just the code memory returned by the translation. The contents of the heap is empty, and the stack contains two pointers to the designated label containing the termination code. The register file is built by calling the function $buildR(n)$ to generate a blank register with n temporary registers. The two program counters are set to the start address generated by the translation, and the stack pointers are set to the top of the stack.

Theorem 59 (Translation)

If $[\![\vdash fds; lds; \varsigma\ \text{wf}]\!] = C\ n\ l_{start}$

then $\vdash\ (R, C, M, (), \cdot)$

where $st = \min(\text{dom}(C)) - 3$

$R = \text{buildR}(n),\ pc_G \mapsto l_{start},\ pc_B \mapsto l_{start},\ sp_G \mapsto st,\ sp_B \mapsto st,\ gd \mapsto 0$

$M = st + 2 \mapsto 0, st + 1 \mapsto l_{halt}, st \mapsto l_{halt}$

Proof By Lemmas 58 (Function Translation), 54 (Local Declaration Translation), 55 (Statement Translation), and 57 (Epilogue Generation). ■

Appendix F

TAL$_{\text{CF}}$ **Formal Results**

This appendix expands on the formal results for TAL$_{\text{CF}}$ presented in Section 4.4. We provide proof sketches for all lemmas and theorems. The complete proofs appear in the companion technical report [49].

F.1 Type Safety

F.1.1 Typing Lemmas

First, we briefly explain the main lemmas used to prove type safety.

Expression equality is transitive. Conditional expression $E_z?E_f : E_t$ is equal to either E_t or E_f depending on the value of E_z.

Lemma 60 (Expression Equality)

1. *If* $\Delta \vdash E_1 = E_2$ *and* $\Delta \vdash E_2 = E_3$ *then* $\Delta \vdash E_1 = E_3$

2. *If* $\Delta \vdash \sigma_1 = \sigma_2$ *and* $\Delta \vdash \sigma_2 = \sigma_3$ *then* $\Delta \vdash \sigma_1 = \sigma_3$

3. *If* $\Delta \vdash E_z = 0$ *then* $\Delta \vdash E_z?E_f : E_t = E_t$.

4. *If* $\Delta \vdash E_z \neq 0$ *then* $\Delta \vdash E_z?E_f : E_t = E_f$.

Proof: *By the definition of* $\Delta \vdash E = E$ *and* $\Delta \vdash \sigma = \sigma$ *and the definition of* $[\![E]\!]$.

Substituting an expression of kind κ for a free variable of kind κ preserves typing. Applying a substitution S that provides substitutions for a number of free variables also preserves typing.

Lemma 61 (Substitution)

1. *If* $\Delta, x : \kappa \vdash E' : \kappa'$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E'[E/x] : \kappa'$

2. *If* $\Delta, x : \kappa \vdash E_1 = E_2$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E_1[E/x] = E_2[E/x]$

3. *If* $(\Delta, x : \kappa); \Psi \vdash^Z v : t$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta; \Psi \vdash^Z v : t[E/x]$

4. *If* $(\Delta, x : \kappa); \Psi; \Gamma; \sigma; E_i; \tau\ opt \vdash b$ *and* $\Delta \vdash E : \kappa$
   then $\Delta; \Psi; \Gamma[E/x]; \sigma[E/x]; E_i[E/x]; \tau\ opt[E/x] \vdash b$

5. *If* $(\Delta_1, \Delta_2) \vdash E' : \kappa'$ *and* $\Delta_1 \vdash S : \Delta_2$ *then* $\Delta_1 \vdash S(E') : \kappa'$

6. *If* $(\Delta_1, \Delta_2) \vdash E_1 = E_2$ *and* $\Delta_1 \vdash S : \Delta_2$ *then* $\Delta_1 \vdash S(E_1) = S(E_2)$

7. *If* $(\Delta_1, \Delta_2); \Psi \vdash^Z v : t$ *and* $\Delta_1 \vdash S : \Delta_2$ *then* $\Delta_1; \Psi \vdash^Z v : S(t)$

8. *If* $(\Delta_1, \Delta_2); \Psi; \Gamma; \sigma; E_i; \tau\ opt \vdash b$ *and* $\Delta_1 \vdash S : \Delta_2$
   then $\Delta_1; \Psi; S(\Gamma); S(\sigma); S(E_i); S(\tau\ opt) \vdash b$

Proof:

1. *By induction on the structure of* $\Delta, x : \kappa \vdash E' : \kappa'$

2. *By case analysis on the structure of* $\Delta, x : \kappa \vdash E_1 = E_2$ *using Part 1.*

3. *By case analysis on the structure of* $(\Delta, x:\kappa);\Psi \vdash^{Z} v : t$ *using Parts 1 and 2.*

4. *By induction on the structure of* $(\Delta, x:\kappa);\Psi;\Gamma;\sigma;E_i;\tau\ opt \vdash b$ *using Parts 1-3. The case for rule (recovernz-t) divides into two subcases depending on if* $E_z = 0$ *and uses rule (recovernz-eq-t) or rule (recovernz-neq-t) as appropriate.*

5-8. *By induction on the size of* Δ_2*, using Parts 1-4 respectively.*

If a value has a type t and this type is a subtype of t', then the value can also be given type t'.

Lemma 62 (Subtyping)

If $\Delta \vdash t \leq t'$ *and* $\Delta;\Psi \vdash^{Z} v : t$ *then* $\Delta;\Psi \vdash^{Z} v : t'$

Proof: *By induction on the derivation of* $\Delta;\Psi \vdash^{Z} v : t$*. Each case uses inversion of the subtyping rules and Lemma 60 (Expression Equality).*

If a value $c\ n$ has type $\langle c', \tau', E' \rangle$ under zap tag Z, then our knowledge about n depends both on the base type τ' and also the relationship between Z and c'. If $Z = CF$ and the color in the type is G or B, then the judgment may be been derived using rule (*val-zap-CF-t*), so we know nothing about n. However, we do know that the expression has kind κ_{int}. If Z is c', then the judgment may have been derived by rule (*val-zap-c-t*), so again we only know that the expression E' has kind κ_{int}. The remaining case applies when Z is not equal to the color in the type c' and when either $Z \neq CF$ or the color in the type is neither G nor B. In this case the judgment must have been derived using rule (*val-t*), so we know the color tag on the value is equal to the color tag in the type and the expression

E' is equal to the value n. In addition, if τ' is a code type, we also know that n is a valid code address.

Lemma 63 (Canonical Forms)

If $\cdot\ ;\Psi \vdash^{Z}\ c\ n : \langle c', \tau', E'\rangle$ *and* $\vdash C : \Psi$ *then*

1. *If* $Z = CF$ *and* ($c' = B$ *or* $c' = G$)*, then* $\cdot \vdash E' : \kappa_{int}$.

2. *If* $Z = c'$ *then* $c = c'$ *and* $\cdot \vdash E' : \kappa_{int}$.

3. *If* $Z \neq c'$ *and* ($Z \neq CF$ *or* ($c' \neq B$ *and* $c' \neq G$)) *then*

    - $c = c'$
    - $\cdot \vdash E' = n$
    - $\tau' = \forall[\Delta](\Gamma, \sigma) \implies n \in Dom(C)$

Proof: *By inspection of* $\cdot;\Psi \vdash^{Z} c\ n : \langle c', \tau', E'\rangle$.

If a value has a type t under zap tag Z, then that value also has type t under any zap tag Z' that is a supertype of Z.

Lemma 64 (Color Weakening)

If $\Delta;\Psi \vdash^{Z} v : t$ *and* $Z \leq Z'$ *then* $\Delta;\Psi \vdash^{Z'} v : t$

Proof: *By case analysis of* $\Delta;\Psi \vdash^{Z} v : t$ *and the definition of* $Z \leq Z'$.

F.1.2 Type Safety

We have proven that the TAL$_{\text{CF}}$ type system is sound using the standard notion of Progress and Preservation. Progress asserts that machine states well-typed under the empty zap tag can take a step to another ordinary machine state. States that are well-typed under any zap can also take a step, but this step may reach any state, including `recover`(h) or `hwerror`(h).

Theorem 65 (Progress)

1. *If* $\vdash \Sigma$ *then* $\Sigma \longrightarrow_0 \Sigma'$.

2. *If* $\vdash^Z \Sigma$ *then* $\Sigma \longrightarrow_0 \mathcal{F}$.

Proof: *The proof for each part is by case analysis on the current block b of Σ using Lemma 60 (Expression Equality) and Lemma 63 (Canonical Forms).*

Preservation states that execution preserves typing. States well-typed under the empty zap tag continue to be so after taking a non-faulty step. States typed under any zap also remain well-typed after a non-faulty step, but the zap tag may escalate to a supertype. This elevation might occur at control flow transfers. A zap tag of B or G becomes CF whenever the corruption has spread to the operands being used in the transfer. This way the block that results from the transfer can be well-typed under CF even when control has transferred to a totally unexpected block. The intention register is always the only orange value that is live across control flow transfers, and we have already seen that it is well-typed even when a control fault has occurred. Finally, a state is well-typed under the empty zap tag and takes a faulty step, then the resulting state is well-typed under some color c.

Theorem 66 (Preservation)

1. *If* $\vdash \Sigma$ *and* $\Sigma \longrightarrow_0 \Sigma'$ *then* $\vdash \Sigma'$

2. *If* $\vdash^Z \Sigma$ *and* $\Sigma \longrightarrow_0 \Sigma'$ *then* $\exists Z' . \vdash^{Z'} \Sigma'$ *and* $Z \leq Z'$.

3. *If* $\vdash \Sigma$ *and* $\Sigma \longrightarrow_1 \Sigma'$ *then* $\exists c . \vdash^c \Sigma'$

Proof: *The proof for each part is by case analysis on the corresponding single step judgment using Lemma 60 (Expression Equality), Lemma 63 (Canonical Forms), and Lemma 64 (Color Weakening). Cases for the jump and branch rules also use Lemma 61 (Substitution) and Lemma 62 (Subtyping).*

F.2 Fault Tolerance Results

We first present a handful of definitions and lemmas relating machine states to other states, and then use these to formally state and prove the Fault Tolerance Theorem.

F.2.1 Machine State Simulation

We say that a faulty value simulates a fault-free value under color c if the values are equal when they are not colored by c.

$$\frac{}{c'\ n\ sim^c\ c'\ n}\ (sim\text{-}val) \qquad \frac{}{c\ n\ sim^c\ c\ n'}\ (sim\text{-}val\text{-}zap)$$

A faulty machine state Σ_f simulates a fault-free state Σ under color c if Σ_f is well-typed under c, Σ is well-typed under the empty zap tag, and the two states are identical modulo the values in registers colored c.

$$\frac{\vdash (C,h,R,b) \qquad \vdash^c (C,h,R_f,b) \qquad \forall r.R_f(r)\ sim^c\ R(r)}{(C,h,R_f,b)\ sim^c\ (C,h,R,b)}\ (sim\text{-}\Sigma)$$

F.2.2 Block Execution

The Block Step Lemma states that given a non-faulty computation and a corresponding faulty version Σ_f, if the non-faulty computation can take a non-faulty step to some other state *in the same block*, then the faulty computation will either also take a step within the current block or will take a single step to the recover state.

Lemma 67 (Block Step)

If $\Sigma_f\ sim^c\ (C,h,R,b)$ *and* $(C,h,R,b) \longrightarrow_0 (C,h,R',b')$ *then either*

1. $\Sigma_f \longrightarrow_0 \Sigma'_f$ *and* $\Sigma'_f\ sim^c\ (C,h,R',b')$*, or*

2. $\Sigma_f \longrightarrow_0 \texttt{recover}(h)$

Proof: *By case analysis of* $(C,h,R,b) \longrightarrow_0 (C,h,R',b')$ *and Theorem 66 (Preservation).*

In order to reason about block execution, we extend the single step relation $\Sigma \longrightarrow_k \Sigma'$ from Section 4.2 to create the judgment $\Sigma \rightsquigarrow_k \mathcal{F}$ which states that $\mathcal{F}$ is the result of executing the *current block* of Σ while incurring k faulty transitions. Execution proceeds up to the control-flow transfer statement at the end of the current block or the recover state if the block terminates prematurely by transitioning to recovery code. For example, if $\Sigma = (C,h,R,i_1;\ldots;i_n;\texttt{jmp}\ r_t)$, then either $\mathcal{F} = (C,h,R',\texttt{recover}(h))$ or $\mathcal{F} = (C,h,R',\texttt{jmp}\ r_t)$.

$\boxed{\Sigma \rightsquigarrow_k \mathcal{F}}$

$$\frac{(C,h,R,b) \longrightarrow_0 \texttt{recover}(h)}{(C,h,R,b) \rightsquigarrow_0 \texttt{recover}(h)}\ (\textit{blk-eval-recover})$$

$$\frac{}{(C,h,R,\texttt{jmp}\ r_t) \rightsquigarrow_0 (C,h,R,\texttt{jmp}\ r_t)}\ (\textit{blk-eval-jmp})$$

$$\frac{}{(C,h,R,\texttt{brz}\ r_z\ r_t) \rightsquigarrow_0 (C,h,R,\texttt{brz}\ r_z\ r_t)}\ (\textit{blk-eval-brz})$$

$$\frac{(C,h,R,b) \longrightarrow_{k_1} (C,h,R',b') \qquad (C,h,R',b') \rightsquigarrow_{k_2} \mathcal{F}}{(C,h,R,b) \rightsquigarrow_{(k_1+k_2)} \mathcal{F}}\ (\textit{blk-eval-sequence})$$

The Block Execution Lemma states that given a faulty computation Σ_f that simulates a non-faulty computation, the result of executing the faulty block will either simulate the result of executing the non-faulty block, or executing the faulty block will result in $\texttt{recover}(h)$.

Lemma 68 (Block Execution)

If $\Sigma_f \; sim^c \; (C,h,R,b)$ *and* $(C,h,R,b) \leadsto_0 \Sigma'$ *then either*

1. $\Sigma_f \leadsto_0 \Sigma'_f$ *and* $\Sigma'_f \; sim^c \; \Sigma'$*, or*

2. $\Sigma_f \leadsto_0 \texttt{recover}(h)$

Proof: *By induction on the structure of* $(C,h,R,b) \leadsto_0 \Sigma'$ *and Lemma 67 (Block Step).*

F.2.3 Fault Recovery

The *CF* Fault Step Lemma states that once a control flow fault has occurred, execution will either step within the same block or will step to recovery code.

Lemma 69 (*CF* Fault Step)

If $\vdash^{CF} (C,h,R,b)$ *then either*

1. $(C,h,R,b) \longrightarrow_0 (C,h,R',b')$ *and* $\vdash^{CF} (C,h,R',b')$

2. $(C,h,R,b) \longrightarrow_0 \texttt{recover}(h)$.

Proof: *By case analysis on the structure of* b *using Theorem 66 (Preservation).*

The Fault Recovery Lemma states that once a control flow fault has occurred, control will always reach recovery code before exiting the current block.

Lemma 70 (*CF* Fault Block Execution)

If $\vdash^{CF} (C,h,R,b)$ *then* $(C,h,R,b) \leadsto_0 \texttt{recover}(h)$.

Proof: *By induction on the length of* b *and Lemma 69 (CF Fault Step).*

F.2.4 Block Transitions

In order to reason about transitions *between* blocks, we define the judgment $\Sigma \Longrightarrow^{\ell} \Sigma'$ whenever $(C,h,R,b) \longrightarrow_0 (C,(h,\ell),R',b')$. In other words, control transfers from the end of one block to the beginning of another block ℓ in a single step.

$\boxed{\Sigma \Longrightarrow^{\ell} \Sigma'}$

$$\frac{(C,h,R,b) \longrightarrow_0 (C,(h,\ell),R',b')}{(C,h,R,b) \Longrightarrow^{\ell} (C,(h,\ell),R',b')}\ (\textit{trans-eval})$$

The Block Transition Lemma states that whenever a non-faulty computation transitions to a new block, the corresponding faulty computation will either (1) transition to the same block and continue to be indistinguishable from the non-faulty computation, (2) trigger a hardware error, or (3) transition to an incorrect block where the error will be detected before control leaves the incorrect block.

Lemma 71 (Block Transition)

If Σ_f *sim*^{c} (C,h,R,b) *and* $(C,h,R,b) \Longrightarrow^{\ell} \Sigma'$ *then either*

1. $\Sigma_f \Longrightarrow^{\ell} \Sigma'_f$ *and* Σ'_f *sim*^{c} Σ'
2. $\Sigma_f \longrightarrow_0 \texttt{hwerror}(h)$
3. $\Sigma_f \Longrightarrow^{\ell'} \Sigma'_f$ *and* $\Sigma'_f \leadsto_0 \texttt{recover}(h,\ell')$

Proof: *By case analysis of the structure of* $(C,h,R,b) \Longrightarrow^{\ell} \Sigma'$ *and Lemma 70 (CF Fault Block Execution).*

F.2.5 Program Execution

The judgment $\Sigma \Longrightarrow_k^h \mathcal{F}$ states that machine state Σ executes through a sequence of blocks h to reach state $\mathcal{F}$ while incurring k faulty transitions. In other words, if $\Sigma = (C,h_1,R,b)$, then $\mathcal{F}$ is either the regular state $(C,(h_1,h),R',\texttt{jmp}\ r_t)$, the regular state $(C,(h_1,h),R',\texttt{brz}\ r_z\ r_t)$, the hardware error state $\texttt{hwerror}(h_1,h)$, or the recovery state $\texttt{recover}(h_1,h)$.

$\boxed{\Sigma \Longrightarrow_k^h \Sigma'}$

$$\frac{\Sigma \rightsquigarrow_k \mathcal{F}}{\Sigma \Longrightarrow_k^{()} \mathcal{F}}\ (\textit{prog-exec-blk})$$

$$\frac{\Sigma \Longrightarrow_k^h \Sigma' \qquad \Sigma' \longrightarrow_0 \texttt{hwerror}(h',h)}{\Sigma \Longrightarrow_k^h \texttt{hwerror}(h',h)}\ (\textit{prog-exec-seq-hwerror})$$

$$\frac{\Sigma \Longrightarrow_{k_1}^h \Sigma' \qquad \Sigma' \Longrightarrow^{\ell} \Sigma'' \qquad \Sigma'' \rightsquigarrow_{k_2} \mathcal{F}}{\Sigma \Longrightarrow_{(k_1+k_2)}^{(h,\ell)} \mathcal{F}}\ (\textit{prog-exec-seq-trans-blk})$$

The Faulty Execution Lemma states that if a faulty execution Σ_f simulates a non-faulty execution Σ under some color c, then Σ_f behaves in one of four possible ways with regards to Σ. (1) Executing Σ_f results in the same sequence of blocks h as executing Σ and the resulting faulty state simulates the corresponding non-faulty state under the same color c. (2) Executing Σ_f results in an attempt to transfer control to an invalid address outside the domain of code memory and triggers a hardware fault. Prior to the occurrence of the hardware fault, the execution of Σ_f visits the same blocks as the execution of Σ.

(3) While executing Σ_f, a fault is detected and control is transferred to recovery code even though no incorrect blocks have been visited. This situation can be caused by a fault affecting the intention register or the checking code. (4) While executing Σ_f, control veers off course to a block that is not visited in the execution of Σ. In this case, the checking code in the invalid block catches the error and transfers control to the recovery code.

Lemma 72 (Faulty Execution)

If Σ_f sim^c Σ and $\Sigma \Longrightarrow_0^h \Sigma'$ then either:

1. $\Sigma_f \Longrightarrow_0^h \Sigma'_f$ *and* Σ'_f *sim*^{c} Σ'
2. $\Sigma_f \Longrightarrow_0^{h_f} \texttt{hwerror}(h', h_f)$ *and* h_f *is a prefix of* h
3. $\Sigma_f \Longrightarrow_0^{h_f} \texttt{recover}(h', h_f)$ *and* h_f *is a prefix of* h
4. $\Sigma_f \Longrightarrow_0^{h_f} \texttt{recover}(h', h_f)$ *and* $h_f = (h_1, l')$ *and* $h = (h_1, l, h_2)$

Proof: *By induction on the structure of* $\Sigma \Longrightarrow_0^h \Sigma'$*, Lemma 68 (Block Execution) and Lemma 71 (Block Transition).*

F.2.6 The Fault Tolerance Theorem

A program is fault-tolerant if any execution of the program with a single fault behaves in one of four possible ways with regards to the original, non-faulty computation. (1) The faulty computation visits the same sequence of blocks as the original and the resulting faulty state simulates the corresponding original state under some color c. (2) The faulty computation attempts to transfer control to an invalid address outside the domain of code memory and triggers a hardware fault. Prior to the occurrence of the hardware fault, the faulty computation visits the same blocks as the original computation. (3) The faulty computation detects a fault in software and jumps to recovery code even though no incorrect blocks have been visited. This situation can be caused by a fault affecting the intention register or the checking code. (4) The faulty computation veers off course to a block that does not match the corresponding block in the original computation. In this case, the checking code in the invalid block catches the error and transfers control to the recovery code.

Theorem 73 (Fault Tolerance)

If $\vdash \Sigma$ *and* $\Sigma \Longrightarrow_0^h \Sigma'$ *then at least one of the following cases applies and all derivations* $\Sigma \Longrightarrow_1^{h_f} \mathcal{F}$ *where* $length(h_f) \leq length(h)$ *fit one of these cases:*

1. $\Sigma \Longrightarrow_1^h \Sigma'_f$ *and* $\exists c\ .\ \Sigma'_f\ sim^c\ \Sigma'$

2. $\Sigma \Longrightarrow_1^{h_f} \texttt{hwerror}(h', h_f)$ *and* h_f *is a prefix of* h

3. $\Sigma \Longrightarrow_1^{h_f} \texttt{recover}(h', h_f)$ *and* h_f *is a prefix of* h

4. $\Sigma \Longrightarrow_1^{h_f} \texttt{recover}(h', h_f)$ *and* $h_f = (h_1, l')$ *and* $h = (h_1, l, h_2)$

Proof: *By case analysis on the structure of* $\Sigma \Longrightarrow_0^h \Sigma'$. *In essence, arbitrarily divide the computation* $\Sigma \Longrightarrow_0^h \Sigma'$ *into two pieces with some* Σ'' *as the intermediate state. Use one of the fault rules to step* Σ'' *to* Σ''_f. *If* c *is the color of the value that faults, then* $\Sigma''_f\ sim^c\ \Sigma''$. *Then use Lemma 72 (Faulty Execution) with* $\Sigma''_f\ sim^c\ \Sigma''$ *and the remainder of the computation from* Σ'' *to* Σ' *to determine what happens after the fault. Use these* *results and the first half of the computation to show that one of the four cases applies to the entire computation containing a single fault.*

Appendix G

TAL$_{\text{CF}}$ Translation

This appendix gives additional details for the translation from while loops to TAL$_{\text{CF}}$ presented in Section 4.5. We provide proof sketches for all lemmas and theorems. The complete proofs appear in the companion technical report [49].

G.1 A Simple While Loop Language

The while loop language statements consist of simple assignment, subtraction, if statements, while loops, and sequences of statements. As all the variables in this language contain integers, the well-formedness judgment $V \vdash s$ simply enforces that all variables v in s exist in the variable context V.

$$
\begin{array}{rcl}
s & ::= & v := n \mid v_d := v_a - v_b \\
 & & \mid \texttt{if0}\ v_z\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ v_z \neq 0\ \texttt{do}\ s \\
 & & \mid s_1 ; s_2
\end{array}
$$

G.2 Checking Code and Exit Code Macros

The translation rules and lemmas make use of the following macros that implement the protocol from Section 4.1. These macros make use of two temporary registers: t_g and t_b.

Macro “check ℓ” generates the checking code at the entry of block ℓ to check that control has correctly transferred to this block. Macro “intendjmp ℓ_t” sets the intention and then executes the jump to target block ℓ_t. Finally, macro “intendzbrz r_z ℓ_t ℓ_f” uses register r'_z to conditionally set the intention to fall through to block ℓ_f or branch to block ℓ_t, and then uses register r_z to execute the conditional branch.

$$\begin{array}{rcl}
\text{check } \ell & \equiv & \texttt{movi}\ t_g\ O\ \ell; \texttt{sub}\ t_g\ t_g\ r_i; \\
 & & \texttt{recovernz}\ t_g \\
 & & \\
\text{intendjmp } \ell_t & \equiv & \texttt{movi}\ t_b\ B\ \ell_t; \texttt{intend}\ t_b; \\
 & & \texttt{movi}\ t_g\ G\ \ell_t; \texttt{jmp}\ t_g \\
 & & \\
\text{intendzbrz } r_z\ \ell_t\ \ell_f & \equiv & \texttt{movi}\ t_b\ B\ \ell_f; \texttt{intend}\ t_b; \\
 & & \texttt{movi}\ t_b\ B\ \ell_t; \texttt{intendz}\ r'_z\ t_b; \\
 & & \texttt{movi}\ t_g\ G\ \ell_t; \texttt{brz}\ r_z\ t_g
\end{array}$$

G.3 Translating Variable Context V

Since all variables are considered live at all program points, every assembly-level instruction block will have essentially the same signature. If the context V contains variable $v_1,\ldots,v_n$, then each block in the translation requires $2n+3$ registers: a green copy r_k and a blue copy r'_k for each variable v_k, the intention register r_i, and two temporary registers t_g and t_b. (We have made no effort to optimize this translation, it merely serves to demonstrate the theoretical expressiveness of the target language.)

The function $[\![V]\!]_\ell$ generates the code type of the code at label ℓ. The generated Δ contains all the expression variables that are need in Γ and σ. Each label will have a slightly different history typing σ, since the sequence ends with the current label. Register file typing Γ gives types to each of the $2n+3$ registers. Register r_k is green and r'_k is blue. Both registers have basic type *int* and are described by the same expression variable x_k, which enforces that they are equal on entry to the block. Register r_i is orange, has basic type *check*, and is described by expression variable x_i. Again, since we have not optimized the translation, we will assume that during block transitions t_g always contains a green value and t_b always contains a green value. They may hold values of other colors during the body of a block.

$\boxed{[\![V]\!]_\ell = \forall[\Delta](\Gamma, \sigma)}$

$$\frac{\begin{array}{l}\texttt{choose fresh variables } x_1,\ ...,\ x_n,\ x_h,\ x_{r_i},\ x_g,\ x_b,\ x_o \\ \Delta = x_1 : \kappa_{int},\ \ldots,\ x_n : \kappa_{int},\ x_h : \kappa_{hist},\ x_{r_i} : \kappa_{int},\ x_g : \kappa_{int},\ x_b : \kappa_{int} \\ \sigma = x_h \circ \ell \\ \Gamma = \{\ r_1 : \langle G, int, x_1\rangle,\ r'_1 : \langle B, int, x_1\rangle, \ldots,\ r_n : \langle G, int, x_n\rangle,\ r'_n : \langle B, int, x_n\rangle, \\ \qquad r_i : \langle O, check, x_{r_i}\rangle, \\ \qquad t_g : \langle G, int, x_g\rangle,\ t_b : \langle B, int, x_b\rangle\ \ \} \end{array}}{[\![v_1, \ldots, v_n]\!]_\ell = \forall[\Delta](\Gamma, \sigma)}\ (\textit{trans-V})$$

The function $\texttt{Gen}\Psi(V, L)$ computes the heap typing Ψ that maps each label in L to its corresponding type

$\boxed{\texttt{Gen}\Psi(V, L)}$

$$\begin{array}{l}\texttt{Gen}\Psi(V, \cdot) = \cdot \\ \texttt{Gen}\Psi(V, (L, \ell)) = \texttt{Gen}\Psi(V, L),\ \ell \mapsto [\![V]\!]_\ell\end{array}$$

G.4 Partial Translations

A 4-tuple of objects $(L, C, \vec{i}, \ell)$ is used to track the code generated during the translation. C is the code memory that contains all blocks generated so far. L contains labels that may be referred to by blocks in C but whose corresponding blocks have not yet been generated. ℓ is the label that will be assigned to the block that is currently being generated. $\vec{i}$ contains the list of instructions for this block that have been generated so far. The instructions for checking the checking code and exit code are not included and will be added when the block is added to C.

The judgment $V; \Psi_1 \vdash C : \Psi_2$ is used to type code memory C as it is being generated. There are two disjoint heaping typings: Ψ_1 contains labels that may be referenced by C but whose corresponding code blocks may not have been generated yet, and Ψ_2 contains the types for the blocks that have already been generated. Both Ψ_1 and Ψ_2 map each label ℓ to $[\![V]\!]_\ell$. In addition, each label in Ψ_2 has a type that can be used to type check corresponding block.

$\boxed{V; \Psi_1 \vdash C : \Psi_2}$

$$\frac{\begin{array}{l} Dom(\Psi_1) \cap Dom(\Psi_2) = \emptyset \\ \forall \ell \in Dom(\Psi_1) \,.\, \Psi_1(\ell) = [\![V]\!]_\ell \\ Dom(C) = Dom(\Psi_2) \\ \forall \ell \in Dom(\Psi_2) \,. \\ \quad \Psi_2(\ell) = [\![V]\!]_\ell = \forall[\Delta](\ (\Gamma, r_i \mapsto \langle O, check, x_i\rangle),\ x_h \circ \ell\ ) \\ \quad \wedge\ \ \Delta;\ (\Psi_1 \cup \Psi_2);\ (\Gamma, r_i \mapsto \langle O, check, x_i\rangle);\ (x_h \circ \ell);\ x_i;\ (\Psi_1 \cup \Psi_2)(\ell + 1) \\ \qquad \vdash\ \ C(\ell) \end{array}}{V; \Psi_1 \vdash C : \Psi_2} \ (\textit{C-wf})$$

Judgment $V \vdash \vec{i}$ wf states that $\vec{i}$ is a sequence of pairs of instructions that perform duplicate moves and subtractions. For example, the following is a well-formed list of instructions.

$$\texttt{movi}\ r_3\ G\ 3;\ \texttt{movi}\ r'_3\ B\ 3;\ \texttt{sub}\ r_4\ r_5\ r_6;\ \texttt{sub}\ r'_4\ r'_5\ r'_6; \ldots$$

Using these definitions, we say a partial translation $(L, C, \vec{i}, \ell)$ is well-formed when the code memory C is well-formed using the heap typings calculated from the label ℓ and the labels in L and the labels already in the domain of C. In addition, the instruction list $\vec{i}$ is well-formed.

$\boxed{V \vdash (L, C, \vec{i}, \ell)\ \text{wf}}$

$$\frac{\begin{array}{l}\Psi_1 = \texttt{Gen}\Psi(V, (L, \ell)) \\ \Psi_2 = \texttt{Gen}\Psi(V, Dom(C)) \\ V; \Psi_1 \vdash C : \Psi_2 \\ V \vdash \vec{i}\ \text{wf}\end{array}}{V \vdash (L, C, \vec{i}, \ell)\ \text{wf}}\ (\textit{partial-trans-}\ \text{wf})$$

The Block Construction Lemma says that the instruction list $\vec{i}$ from a well-formed partial translation can be used to construct a block by adding checking code to the beginning and exit code to the end. The exit code can refer to any existing label ℓ' as the jump target. The exit code can be a conditional branch only if the fall-through block $\ell + 1$ exists. The new code memory formed by adding this new block is also well-formed.

Lemma 74 (Block Construction)

If $V \vdash (L, C, \vec{i}, \ell)$ wf *then* $\forall \ell' \in ((L, \ell) \cup Dom(C))$.

1. $\texttt{Gen}\Psi(V, L) \ \vdash\ C[\,\ell \mapsto \text{check}\ \ell;\ \vec{i};\ \text{intendjmp}\ \ell'\,] : \texttt{Gen}\Psi(V, (Dom(C), \ell))$

2. *If* $l+1 \in ((L,\ell) \cup Dom(C))$

   then $\texttt{Gen}\Psi(V,L) \vdash C[\ell \mapsto \text{check } \ell;\ \vec{i};\ \text{intendzbrz } r_z\ \ell'\ \ell+1] : \texttt{Gen}\Psi(V,(Dom(C),\ell))$

Proof: *Using the macro definitions, the definition of* $V \vdash (L,C,\vec{i},\ell)$ wf*, and instruction typing rules from Section 4.3.2.*

G.5 Translating Statements

The main translation judgment $[\![V \vdash s]\!](L,C,\vec{i},\ell) = (L',C',\vec{i}',\ell')$ extends the existing partial translation $(L,C,\vec{i},\ell)$ with the translation of statement s.

The statement translation rules are shown in Figure G.1. Translating simple assignment and subtraction statements simply adds pairs of assembly instructions to the end of the current instruction sequence. Sequencing two statements uses the partial translation from the first statement to translate the second.

Translating `if0` statements requires the addition of new blocks: ℓ_f contains the fall-through branch, ℓ_t contains the true branch, and ℓ_m is where the two branches merge. The function $NumBlock(s)$ calculates the number of blocks generated by the translation of s. The current block b_ℓ contains checking code, the code $\vec{i}$ generated for the block so far, and ends with a conditional branch to ℓ_t (and an automatic fall-through to ℓ_f). The new label ℓ_t is the starting point for the code generated for the true branch s_1. The ending label of this code ℓ_t' finishes by merging back to the common block at ℓ_m. The translation of the false branch is similar. The final code memory contains all blocks generated by either branch as well as the blocks ending each branch by jumping to the merge block ℓ_m. The label in the resulting partial translation is ℓ_m.

Translating `while` statements also requires the addition of new blocks. The current block at ℓ is terminated with an unconditional jump to a beginning block at ℓ_b that tests

$$\boxed{[\![V \vdash s]\!](L,C,\vec{i},\ell) = (L',C',\vec{i}',\ell')}$$

$$\frac{\vec{i}' = \vec{i};\ \texttt{movi}\ r_k\ G\ n;\ \texttt{movi}\ r'_k\ B\ n}{[\![V \vdash v := n]\!](L,C,\vec{i},\ell) = (L,C,\vec{i}',\ell)}\ (\textit{t-assign})$$

$$\frac{\vec{i}' = \vec{i};\ \texttt{sub}\ r_k\ r_a\ r_b;\ \texttt{sub}\ r'_k\ r'_a\ r'_b}{[\![V \vdash v_d := v_a - v_b]\!](L,C,\vec{i},\ell) = (L,C,\vec{i}',\ell)}\ (\textit{t-sub})$$

$$\frac{\begin{array}{c}[\![V \vdash s_1]\!](L,C,\vec{i},\ell) = (L_1,C_1,\vec{i}_1,\ell_1)\\ [\![V \vdash s_2]\!](L_1,C_1,\vec{i}_1,\ell_1) = (L_2,C_2,\vec{i}_2,\ell_2)\end{array}}{[\![V \vdash s_1;s_2]\!](L,C,\vec{i},\ell) = (L_2,C_2,\vec{i}_2,\ell_2)}\ (\textit{t-seq})$$

$$\frac{\begin{array}{l}\ell_f = \ell + 1\\ \ell_t = \ell_f + \mathit{NumBlocks}(s_1)\\ \ell_m = \ell_t + \mathit{NumBlocks}(s_2)\\ b_\ell = \text{check } \ell;\ \vec{i};\ \text{intendzbrz } r_z\ \ell_f\ \ell_t\\ [\![V \vdash s_1]\!]((L,\ell_f),C[\ell \mapsto b_\ell],\cdot,\ell_t) = (L'_t,C'_t,\vec{i}'_t,\ell'_t)\\ b'_t = \text{check } \ell'_t;\ \vec{i}'_t;\ \text{intendjmp } \ell_m\\ [\![V \vdash s_2]\!]((L,\ell_t),C[\ell \mapsto b_\ell],\cdot,\ell_f) = (L'_f,C'_f,\vec{i}'_f,\ell'_f)\\ b'_f = \text{check } \ell'_f;\ \vec{i}'_f;\ \text{intendjmp } \ell_m\\ C' = (C'_t \cup C'_f)[\ell'_t \mapsto b'_t][\ell'_f \mapsto b'_f]\end{array}}{[\![V \vdash \texttt{if0}\ v_z\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!](L,C,\vec{i},\ell) = (L,C',\cdot,\ell_m)}\ (\textit{t-if})$$

$$\frac{\begin{array}{l}\ell_b = \ell + 1\\ \ell_s = \ell_b + 1\\ \ell_e = \ell_s + \mathit{NumBlocks}(s)\\ C'' = \ C[\ell \mapsto \text{check } \ell;\ \vec{i};\ \text{intendjmp } \ell_b]\\ \qquad\quad [\ell_b \mapsto \text{check } \ell_b;\ \text{intendzbrz } r_z\ \ell_e\ \ell_s]\\ [\![V \vdash s]\!]((L,\ell_e),C'',\cdot,\ell_s) = (L'_s,C'_s,\vec{i}'_s,\ell'_s)\\ C' = C'_s[\ell'_s \mapsto \text{check } \ell'_s;\ \vec{i}'_s;\ \text{intendjmp } \ell_b]\end{array}}{[\![V \vdash \texttt{while}\ v_z \neq 0\ \texttt{do}\ s]\!](L,C,\vec{i},\ell) = (L'_s,C',\cdot,\ell_e)}\ (\textit{t-while})$$

Figure G.1: Translation of While Programs.

the condition and branches to an ending label ℓ_e if the condition fails. Otherwise it falls through to the block at ℓ_s which contains the translation of s and terminates with a jump back to the beginning block. The label in the resulting partial translation is ℓ_e.

The Statement Translation Lemma says that given a well-formed partial translation $(L, C, \vec{i}, \ell)$, translating a statement s results in another well-formed partial translation. In addition, the new set of undefined labels L' is equal to that in the original partial translation.

Lemma 75 (Statement Translation)
If $[\![V \vdash s]\!](L, C, \vec{i}, \ell) = (L', C', \vec{i}', \ell')$ *and* $V \vdash (L, C, \vec{i}, \ell)$ wf
then $V \vdash (L', C', \vec{i}', \ell')$ wf *and* $L = L'$

Proof: *Using the definition of* $V \vdash (L, C, \vec{i}, \ell)$ wf *and Lemma 74 (Block Construction).*

G.6 The Translation Theorem

To translate a statement s as a stand-alone program, it is translated as in the previous section with 1 as the starting label. Because there is no halt instruction in TAL$_{\text{CF}}$, code is added to the last block in the translation to create an infinite loop at label ℓ_{halt}. The function InitRegFile(V) creates an initial register file that maps each register used to translate V to 0.

The assembly language program corresponding to s is the TAL$_{\text{CF}}$ state consisting of the generated code memory, a history with only the first location, an initial register file, and code to jump to the first label in code memory. If the original statement is well-formed, then the translation is well-typed.

Theorem 76 (Translation)

If $[\![V \vdash s]\!](.,.,.,1) = (.,C,\vec{i},\ell)$ *then*

$\vdash (\ C',\ \ 0,\ \ \text{InitRegFile}(V),\ \ \text{intendjmp}\ 1\ )$

where $C' = [\ell \mapsto \text{check}\ \ell; \vec{i}; \text{intendjmp}\ \ell_{halt}][\ell_{halt} \mapsto \text{check}\ \ell_{halt}; \text{intendjmp}\ \ell_{halt}]$

Proof: *Using Lemma 75 (Statement Translation), Lemma 74 (Block Construction), the block typing rules from Section 4.3.3, and the machine state typing rules from Section 4.3.4.*

Bibliography

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*, Nov. 2005.

[2] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. A theory of secure control flow. In *International Conference on Formal Engineering Methods*, Nov. 2005.

[3] A. W. Appel. Foundational proof-carrying code. In *Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE, 2001.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. In *Proceedings of the IEEE*, volume 94, February 2006.

[5] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.

[6] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1 – 121_01.14, April 2002.

[7] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. pages 513–525, 1997.

[8] J. Blömer, M. Otto, and J.-P. Seifert. A new crt-rsa algorithm secure against bellcore attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 311–320, New York, NY, USA, 2003. ACM.

[9] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[10] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[11] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. In *IEEE Micro*, volume 25, pages 10–16, December 2005.

[12] C.-L. Chen and M. Y. B. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[13] J. Chen. A typed intermediate language for compiling multiple inheritance. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–30, New York, NY, USA, 2007. ACM.

[14] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers.

In *ACM Conference on Programming Language Design and Implementation*, June 2008.

[15] J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–49, New York, NY, USA, 2005. ACM.

[16] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 208–219, New York, NY, USA, 2003. ACM.

[17] K. Crary. Toward a foundational typed assembly language. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 198–212, New York, NY, USA, 2003. ACM.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[19] P. Dusart, G. Letourneux, and O. Vivolo. Differential fault analysis on A.E.S, 2003.

[20] M. Elsman. Fault-tolerant voting in a simply-typed lambda calculus. Technical Report ITU-TR-2007-99, IT University of Copenhagen, June 2007.

[21] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 254–267, New York, NY, USA, 2005. ACM.

[22] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 170–182, New York, NY, USA, 2008. ACM.

[23] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 98–109. ACM Press, 2003.

[24] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 Symposium on Security and Privacy*, pages 153–165, May 2003.

[25] R. Hamming. Error-detecting and error-correcting codes. In *Bell System Technical Journal*, volume 29(2), pages 147–160, 1950.

[26] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. In *Proceedings of the Symposium on VLSI Technology*, pages 73–74, 2001.

[27] C. Hawblitzel, H. Huang, L. Wittie, and J. Chen. A garbage-collecting typed assembly language. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 41–52, New York, NY, USA, 2007. ACM.

[28] P. Hazucha, T. Karnik, J. Maiz, S. Walstra, B. Bloechel, J. Tschanz, G. Dermer, S. Hareland, P. Armstrong, and S. Borkar. Neutron soft error rate measurements in a 90-nm cmos process and scaling trends in sram from 0.25-/spl mu/m to 90-nm

generation. In *Proceedings of the 2003 Electron Devices Meeting*, pages 21.5.1–21.5.4, Dec 2003.

[29] M. Y. B. Hsiao, W. C. Carter, J. W. Thomas, and W. R. Stringfellow. Reliability, availability, and serviceability of IBM computer systems: A quarter century of progress. *IBM Journal of Research and Development*, 25(5):453–465, 1981.

[30] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. In *IEEE Symposium on Logic in Computer Science*, pages 407–416, June 2005.

[31] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar. Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18. In *Proceedings of the Symposium on VLSI Technology*, pages 61–62, 2001.

[32] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 41–76, London, UK, 1994. Springer-Verlag.

[33] C. Lin, A. McCreight, Z. Shao, Y. Chen, and Y. Guo. Foundational typed assembly language with certified garbage collection. In *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 326–338, Washington, DC, USA, 2007. IEEE Computer Society.

[34] J.-L. Lin, M. H. Dunham, and M. A. Nascimento. A survey of distributed database checkpointing. *Distributed and Parallel Databases*, 5(3):289–319, 1997.

[35] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[36] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *MICRO ’07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[37] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in Los Alamos National Labratory’s ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.

[38] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based Typed Assembly Language. *Journal of Functional Programming*, 12(1):43–88, Jan. 2002.

[39] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, Jan. 1998.

[40] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[41] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110. IEEE Computer Society, 2002.

[42] G. Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[43] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, Oct. 1996.

[44] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Langauges*, pages 106–119, Paris, Jan. 1997.

[45] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[46] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, pages 41–49, January 1996.

[47] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. In *IEEE Transactions on Reliability*, volume 51, pages 111–122, March 2002.

[48] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.

[49] F. Perry. Reasoning about software in the presence of transient faults – complete proofs. Technical Report TR-831-08, Princeton University, 2008.

[50] F. Perry, C. Hawblitzel, and J. Chen. Simple and flexible stack types. In *International Workshop on Aliasing, Confinement, and Ownership*, July 2007.

[51] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. In *International Symposium on Programming Language Design and Implementation (PLDI)*, June 2007.

[52] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. Technical Report TR-776-07, Princeton University, 2007.

[53] F. Perry and D. Walker. Reasoning about control flow in the presence of transient faults. Technical Report TR-799-07, Princeton University, 2007.

[54] F. Perry and D. Walker. Reasoning about control flow in the presence of transient faults. In *International Static Analysis Symposium*, July 2008.

[55] R. Phelan. Addressing soft errors in ARM core-based SoC. ARM White Paper, December 2003.

[56] G. Piret and J.-J. Quisquater. A differential fault attack technique against spn structures, with application to the AES and KHAZAD. In *CHES*, pages 77–88, 2003.

[57] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36. ACM Press, 2000.

[58] G. A. Reis. *Software Modulated Fault Tolerance*. PhD thesis, Department of Electrical Engineering, Princeton University, Princeton, NJ, 2008.

[59] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery methods. In *IEEE Micro Top Picks*, volume 27, January 2007.

[60] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[61] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 148–159, June 2005.

[62] Z. Shao. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM.

[63] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–399, June 2002.

[64] D. P. Siewiorek. Fault tolerance in commercial computers. *Computer*, 23(7):26–37, 1990.

[65] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 Microprocessor design. In *IEEE Micro*, volume 19, pages 12–23, March 1999.

[66] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[67] M. Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems, 2005.

[68] M. Tremblay and Y. Tamir. Support for fault tolerance in VLSI processors. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 1, pages 388–392, May 1989.

[69] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 61–71, June 2006.

[70] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[71] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98. IEEE Computer Society, 2002.

[72] B. D. Vito and R. Butler. Provable transient recovery for frame-based, fault-tolerant computing systems, 1992.

[73] D. Walker, L. Mackey, J. Ligatti, G. A. Reis, and D. I. August. Static typing for a faulty lambda calculus. In *ACM International Conference on Functional Programming*, Portland, Oregon, Sept. 2006.

[74] A. Wood. Data integrity concepts, features, and technology. White Paper, Tandem Division, Compaq Computer Corporation, 1999.

[75] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.

[76] J. F. Ziegler and H. Puchner. *SER - History, Trends, and Challenges: A Guide for Designing with Memory ICs*. 2004.