Static Enforcement of Timing Policies Using Code Certification

C. Joseph Vanderwaart

August 7, 2006
CMU-CS-06-143

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Karl Crary (chair)
Robert Haper
Peter Lee
Stephanie Weirich (University of Pennsylvania)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was partially supported by the National Science Foundation (NSF) grants CCR-0121633 and CCR-9984812, and by the Defense Advanced Research Projects Agency (DARPA) grant F19628-95-C-0050.

Keywords: Type systems, type theory, typed assembly language, certified code, certifying compilation, type safety, timing, responsiveness, liveness, resource management

Abstract

Explicit or implicit, enforced or not, safety policies are ubiquitous in software systems. In the many settings where third-party software is executed in the context of a larger client program, the supervisor usually enforces a safety policy that prevents the foreign code from behaving in ways that would disrupt the client, corrupt data or destabilize the system. Certified code provides a static means for controlling the behavior of untrusted programs or components by bringing the power of type systems and formal logic to bear on the problem. Code certification systems that prevent bad memory accesses and enforce the abstractions provided by libraries and runtime system interfaces have been well studied.

This thesis presents a system for certifying conformance to timing requirements. The approach is simple, comprising an incremental change to an existing type system for assembly language, but flexible in the set of policies it can enforce. Moreover, in principle, it can be extended to support arbitrarily complex coding idioms. Focusing on a particular timing policy of interest, I describe a compiler that produces certifiably compliant programs with no help from the programmer and only a small impact on runtime performance. Later, I discuss the applicability of both the type system and the compilation techniques to other timing and resource control problems.

Acknowledgements

I would like to thank my thesis advisor, Karl Crary, for his years of help and guidance, and thesis committee members Bob Harper, Peter Lee and Stephanie Weirich for their invaluable help with this thesis and the research that went into it. Frank Pfenning, though not a member of my thesis committee, contributed greatly to my graduate education and I thank him as well.

I also want to thank my fellow students at Carnegie Mellon, including Derek Dreyer, William Lovas, Tom Murphy VII, Leaf Petersen, Susmit Sarkar and Dan Spoonhower. I gratefully acknowledge the particular contribution of Aleksey Kliger, who wrote most of the front end of what I refer to in this thesis as "my compiler."

Portions of this dissertation were written while I was a visiting faculty member at Pomona College. I thank my sometime mentor and longtime friend Kim Bruce for his advice and support during that time, and Rett Bull, Yi Chen, Jim Marshall and Kathy Sheldon of the Department of Mathematics and Computer Science at Pomona for their hospitality and friendship. I also thank the computer science students of Pomona, Scripps and Harvey Mudd Colleges for the unforgettable teaching experience of the year I spent with them.

I would like to thank my parents-in-law, Timi and Bob Hallem, for more love and support than I could ever have expected, as well as for providing me with room and board during some of the writing of this dissertation.

I also thank my parents, Polly and Peter Vanderwaart, and the rest of my family for encouraging me to pursue my interest in computer science.

Finally, I express my deepest gratitude to Elissa Anyon Hallem, without whose unshakeable faith that this thesis was possible this thesis would not have been possible.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**
1.1 Certified Code . . . 4
1.1.1 Classic Proof-Carrying Code . . . 4
1.1.2 Typed Assembly Language . . . 5
1.1.3 Foundationalism . . . 5
1.1.4 Static Safety in Operating Systems . . . 6
1.2 Timing Properties for Safety . . . 7
1.3 Thesis Overview . . . 8

2 TALT Background **11**
2.1 Metalogical Foundational Certified Code . . . 11
2.1.1 LF, Elf and Twelf . . . 11
2.1.2 The Metalogical Skeleton . . . 12
2.1.3 Certified Binaries and Verification . . . 15
2.2 TALT . . . 16
2.2.1 TALT, XTALT and EXTALT . . . 16
2.3 MiniTALT . . . 19
2.3.1 Basic Syntax . . . 21
2.3.2 Type System . . . 22
2.3.3 Instruction Typing . . . 23
2.3.4 Operational Semantics . . . 25
2.4 Chapter Summary . . . 25

3 TALT-R: A Typed Assembly Language for Responsiveness **27**
3.1 A Responsiveness Policy . . . 28
3.2 MiniTALT-R . . . 28
3.2.1 New Instructions . . . 28
3.2.2 The MiniTALT-R Abstract Machine . . . 29
3.3 Static Semantics . . . 29
3.3.1 The Constraint Subsystem . . . 30
3.3.2 The Virtual Clock . . . 31
3.3.3 Guarded and Singleton Types . . . 32
3.3.4 Expanding Singleton Reasoning . . . 34

3.4 Certification and Verification . . . 35
3.4.1 XTALT-R . . . 36
3.4.2 EXTALT-R . . . 37
3.5 Chapter Summary . . . 38

4 The TALT-R Constraint Logic 39
4.1 The Logic . . . 39
4.1.1 Terms and Formulas . . . 39
4.1.2 Defining Truth . . . 40
4.2 Decidability . . . 43
4.2.1 Proof Overview . . . 43
4.2.2 Interpretation of Terms . . . 44
4.2.3 Interpretation of Formulas . . . 46
4.2.4 Semantic Proofs . . . 48
4.2.5 The Decidability Theorem . . . 52
4.2.6 Implementation . . . 53
4.3 Incompleteness . . . 53
4.4 Chapter Summary . . . 54

5 Lilt: A Low-Level Source Language 55
5.1 Syntax . . . 55
5.2 Static Semantics . . . 57
5.3 Lilt Examples . . . 62

6 Yield Placement and Polling Techniques 65
6.1 Local Placement . . . 66
6.2 Global Placement with Call-Return Yielding . . . 68
6.3 Global Placement with Feeley Yielding . . . 69
6.4 Exceptional Placement . . . 72
6.5 Clocks and Polling . . . 72
6.5.1 Clocks . . . 73
6.5.2 Minor Yields . . . 74
6.5.3 The Minor Clock . . . 74
6.5.4 Tricks With Polling . . . 77
6.6 Chapter Summary . . . 78

7 Compilation of Lilt 79
7.1 Type-Directedness . . . 79
7.2 Conventions and Notations . . . 80
7.2.1 Variable Naming . . . 80
7.2.2 Minor Clock Notation . . . 81
7.3 Types and Data Representation . . . 81
7.4 Clock Specifiers . . . 83
7.5 Stacks, Register Files and Labels . . . 84
7.6 Translating Operands . . . 87
7.7 Compiling Expressions . . . 88
7.8 Complete Translation Rules . . . 90

8 Diverse Safety Policies **97**
8.1 Adaptive Responsiveness . . . 97
8.2 The Engine Abstraction . . . 98
8.3 Running Time . . . 99
8.4 Virtual Versus Real Clocks . . . 100
8.4.1 Unpredictability . . . 101
8.4.2 Better Static Approximations . . . 102
8.5 Bandwidth . . . 102
8.6 Stack . . . 103
8.7 Heap Allocation . . . 104
8.8 Chapter Summary . . . 106

9 Conclusions **107**
9.1 Performance Evaluation . . . 107
9.2 Discussion and Future Directions . . . 112
9.2.1 Improvements to Implemented System . . . 112
9.2.2 Applicability . . . 114
9.3 Conclusion . . . 115

A Complete MiniTALT Semantics **117**
A.1 Static Semantics . . . 117
A.1.1 $\boxed{\Delta \vdash c : K,\ \Delta \vdash \Gamma}$ Static Term Formation . . . 117
A.1.2 $\boxed{c_1 \equiv c_2,\ \Gamma \equiv \Gamma'}$ Static Term Equivalence . . . 118
A.1.3 $\boxed{\Delta \vdash \tau_1 \leq \tau_2,\ \Delta \vdash \Gamma \leq \Gamma'}$ Subtyping . . . 118
A.1.4 $\boxed{\Delta; \Psi; \Gamma \vdash o : \tau}$ Operand Typing . . . 120
A.1.5 $\boxed{\Delta; \Psi; \Gamma \vdash d : \tau \to \Gamma'}$ Destination Typing . . . 120
A.1.6 $\boxed{\Delta; \Psi; \Gamma \vdash I}$ Instruction Typing . . . 120
A.1.7 $\boxed{\Psi; \Delta \vdash I : \tau\ \mathsf{block},\ \Delta \vdash P}$ Block and Program Typing . . . 122
A.2 Operational Semantics . . . 122
A.2.1 $\boxed{H, V_s, R \vdash o \rightsquigarrow v}$ Operand Resolution . . . 122
A.2.2 $\boxed{H, V_s, R \vdash d(v) \rightsquigarrow H', V_s', R'}$ Destination Propagation . . . 122

B Complete MiniTALT-R Typing Rules **123**
B.1 $\boxed{\Delta \vdash c : K}$ Static Term Formation . . . 123
B.2 $\boxed{\Delta \vdash \varphi\ \mathsf{prop}}$ Constraint Formula Formation . . . 123
B.3 $\boxed{c_1 \equiv c_2,\ \varphi_1 \equiv \varphi_2}$ Static Term and Formula Equivalence . . . 123
B.4 $\boxed{\Delta \vdash \varphi\ \mathsf{true}}$ Constraint Truth . . . 124
B.4.1 Rule For Rational Extension . . . 124
B.5 $\boxed{\Delta \vdash \tau_1 \leq \tau_2,\ \Delta \vdash \Gamma \leq \Gamma'}$ Subtyping . . . 124
B.5.1 Unofficial Rules . . . 124
B.6 $\boxed{\Delta; \Psi; \Gamma \vdash o : \tau}$ Operand Typing . . . 125
B.7 $\boxed{\Delta; \Psi; \Gamma \vdash I}$ Instruction Typing . . . 125
B.7.1 Unofficial Rules . . . 126
B.8 $\boxed{\Delta; \Psi; \Gamma \vdash I\ \text{inits}\ r{:}\mathtt{mbox}(\tau)}$ Object Initialization . . . 126

B.9 $\boxed{\Psi; \Delta \vdash I : \tau \text{ block}}$ Block Typing 127

C Rational Semantic Proofs **129**
C.1 Linear Programming Duality 129
C.2 Characteristic Linear Programs 131
C.3 Rational Semantic Proofs 133
C.4 Augmented Syntactic Proof System 133

D Typing Rules for Lilt **135**

List of Figures

2.1 The skeleton of metalogical certified code. . . . 13
2.2 TBF File Layout . . . 15
2.3 Overview of the TALT safety structure. . . . 17
2.4 Machine-Specific Notation for IA-32 . . . 20
2.5 MiniTALT Program Syntax . . . 20
2.6 MiniTALT Type System Syntax . . . 22
2.7 Selected Instruction Typing Rules. . . . 24
2.8 MiniTALT Abstract Machine Configurations . . . 24

3.1 MiniTALT-R Type System Syntax . . . 30
3.2 Formation Rules for Constraints . . . 31
3.3 Elementary Rules for Singletons . . . 33

4.1 Truth of Formulas . . . 43

5.1 Lilt Syntax . . . 56
5.2 Lilt Example: Recursive Fibonacci . . . 61
5.3 Lilt Example: Iterative Fibonacci . . . 61
5.4 Lilt Example: List Reversal . . . 62

6.1 A Flow Graph With a Join . . . 66
6.2 Fibonacci using Feeley Yielding . . . 71
6.3 Yields Under a Polling Strategy . . . 73
6.4 Code for a Minor Yield . . . 74
6.5 Fibonacci using Feeley Polling . . . 76
6.6 A Minor Yield with F on the Clock . . . 77
6.7 Resetting the Clock from F to R . . . 77

7.1 Translation of kinds and types (except function types) . . . 81
7.2 Translation of function types . . . 82
7.3 Clock Specifiers . . . 83
7.4 A Lilt function's stack frame . . . 85
7.5 Determining the Stack Type . . . 86
7.6 Determining the Register File Type . . . 86
7.7 Label and Block Types . . . 87
7.8 Translation Contexts . . . 88

8.1 Code for a Clock Check . . . 102
8.2 Typing Rules for a Stack Usage Policy . . . 104

8.3 Typing Rules for a Heap Allocation Policy 105

9.1 Normalized execution time ($Y = 1$ billion, $L = 500$, $E = 100$, $H = 50$) 107
9.2 Effect of Y on Yielding Performance 109
9.3 Effect of L on Yielding Performance (Y=100M, E=100, H=50) 110
9.4 Effect of E on Yielding Performance (Y=100M, L=500, H=50) 111
9.5 Effect of L on Competition (Y=10M, E=100, H=50) 112

List of Tables

2.1 Variants of TALT 19
2.2 MiniTALT Typing Judgment Forms 23
2.3 MiniTALT Abstract Machine Evaluation 26

3.1 New Typing Judgments of MiniTALT-R 30

5.1 Lilt typing judgment forms 58

9.1 Yields under Feeley Polling 108
9.2 Yield Frequencies (Yields/sec) 108

Chapter 1

Introduction

> *[An]* operating system *. . . is a program that keeps track of other programs in a computer and gives each its due in space and time.* — Guy L. Steele Jr. [65]

Computers are useful precisely because they can be programmed. The success of programming depends on the ability of programmers to construct sequences of instructions whose behavior, when they are executed faithfully by hardware, is consistent with some design. This is harder than it sounds, partly because although the basic operation of computer hardware is deterministic, programs are executed in a complex environment of simultaneously running processes multiplexed onto a machine that usually must serve multiple purposes for multiple users. This arrangement is intended to improve performance (multiple processes can make more effective use of a computer's resources), usability (users like to have more than one application active at a time) and modularity (programs responsible for different functions can be designed independently), but these gains cannot be realized if the resulting environment is no longer predictable enough to be programmed. To keep the complexity under control, we use *operating systems*, which supervise application programs and prevent their uncontrolled interference with one another. The job of an operating system necessarily involves the setting and enforcement of *safety policies* regarding diverse aspects of program behavior, including memory access, privileged instruction execution, and resource usage.

This relationship that exists between an operating system and the application programs running under it, characterized by the setting and enforcement of rules that restrict the behavior of each process for the benefit of the whole system, is not unique to OS-application interaction. A similar relationship can be found in sophisticated modern software systems based on the dynamic linking and execution of third-party code, such as mobile agents, web applets or application plugins: the principal roles are the ***supervisor,*** which is an application or server process the computer's owner trusts implicitly, and one or more ***subprocesses***, which may be untrusted. The supervisor sets up and enforces specific behavioral rules that the subprocess must obey; such a set of rules is called a ***safety policy.*** (For the purposes of this thesis, I call it a "safety" policy even if it is not a safety property in the sense of Lamport [40] or of Alpern and Schneider [2]. This is consistent with the usage of the term in the certified code literature.) It is worth noting that safety policies are often complex and application-specific, particularly when the supervisor-subprocess relationship exists between two user-level processes, or between an application and an untrusted module that runs as part of the same process. The safety policy of an operating system is generally as permissive as the operating system implementor can allow.

Indeed, the lower the level of abstraction at which one observes the operation of a computer,

the more permissive the safety policy appears to be. The native instruction set of a general-purpose microprocessor (such as Intel's IA-32, also known as x86) is a language with a very small number of types: IA-32 has three different integer types, three different floating-point types, and nothing else. Furthermore, all of the primitive operations of the language — address calculations, loads, stores and ALU and FPU operations — are syntactically restricted so that any well-formed application of any operation to any value has a well-defined outcome [38]. In fact, we can say that as far as the processor architect is concerned, machine language is type safe — this counter-intuitive assertion is accurate because the hardware designer is not concerned with any notion of safety other than that the machine always behaves according to its specification.[1]

When a hardware implementation of IA-32 is used for a nontrivial purpose, such as serving as the CPU of a personal computer, a different notion of safety is called for. Application programs are designed in relative isolation but are executed in an unforeseeable and continually changing context of other concurrent processes. To control the interactions between processes, operating systems impose certain requirements on program behavior: a process must not access memory outside of its designated address space, and it must not attempt to perform certain "privileged" operations. Any violation of these rules is detected at run time by the hardware, and the operating system abruptly and unapologetically terminates the offending process. The rules, and their enforcement by the operating system, greatly limit the range of environmental conditions and events an application programmer must anticipate. This makes it possible to write programs that respond to these conditions predictably, even though the behavior of the other processes on the system cannot be known in advance.

As is well known, most operating systems achieve this end by constant monitoring of every running process to detect violations of its safety policy and forcible correction of errant behavior. Indeed, for a long time conventional wisdom held that this was the only way to do it: after all, conformance to any nontrivial safety policy is an undecidable property of program behavior, so detecting potential safety violations *before* run time seems impossible. The advent of ***certified code***, however, has made it clear that this inference is not valid: in particular, it is possible to design *refinements* of the overly permissive type system of machine language that are strong enough to rule out many forms of unacceptable behavior. Furthermore, using these type systems, static rejection of potentially unsafe programs can be achieved by requiring programs to be accompanied by additional information that establishes their safety.

Enforcing safety policies by certification rather than run-time monitoring has most of the benefits of type-safe programming in general. Although hardware support for detecting violations lessens the cost of monitoring a running program (catching references to unmapped memory pages involves a considerably smaller overhead on most hardware architectures than, say, the tag-checking necessary to implement dynamically-typed languages like Scheme), some expense must be incurred when transferring control to a user process in order for the OS to be ready for anything that process might do. Furthermore, since the hardware does play such a critical role, the range of possible safety policies an OS can implement (and consequently the range of reliability guarantees it can make to application developers) is limited by the capabilities of the hardware; in contrast, the set of policies implementable by certification is relatively independent of the hardware. Finally, the actions taken by an OS in response to a misbehaving process are often drastic and disruptive, and the unexpected termination of one process due to a safety violation can often do harm to other processes that interact with it. By ruling out bad behavior before allowing a program to run, an operating system could reasonably promise users that the program will not be prematurely terminated.

[1] In fact, anecdotal evidence suggests that even this is too strong to accurately describe chip designers' goals.

To date, advocates of safety by certification have focused their rhetoric on so-called "mobile code", and even more specifically "*untrusted* mobile code". This apparent restriction of focus tends to undersell the technology, because after all, **all software is mobile — and most of it is untrusted.** It is very rare that a program resides and runs on the same computer from the moment it is created until it is discarded for good. Furthermore, almost all software must pass from one human owner to another at some point in its life cycle. Sometimes these transfers between individual or corporate owners are well-controlled commercial transactions, but in an increasing variety of scenarios, computers execute code from sources their owners do not know well. Often this is the result of an explicit decision by the owner: it is common practice for users to download programs from unfamiliar web sites, and install and run them even though they have no reason to trust the authors' intentions, skills, or choice of development tools. On occasion, untrustworthy code is executed at a user's apparent request, but as a result of confusion or accident: many e-mail worms spread this way.

Of course, a growing number of applications involve sending code from one machine to another for automatic execution, without any human participation in the process. This, as opposed to the above, is the phenomenon uncontroversially referred to as ***mobile code.*** Most computer users have observed mobile code in action: ever since the late 1990's, many World Wide Web pages have contained embedded "applets" that are executed by browsers with the aid of a Java Virtual Machine implementation [42], and many if not most Web pages today contain embedded code in Javascript or a similar language to be interpreted by the browser.

Some applications of mobile code use it in ways that are not directly visible to users. *Mobile agents*, programs that autonomously migrate from host to host to take advantage of localized resources, have achieved buzzword status over the past decade or so [41, 7]. More recently, ***grid computing*** has emerged as a powerful paradigm that relies heavily on mobile code. For the purposes of this thesis, "grid computing" simply means large-scale distributed computing on a heterogeneous collection of computers connected by the Internet. This includes scientific computing endeavors such as SETI@Home [62] and Folding@Home [24] as well as CMU's ConCert infrastructure [9] and a host of others being developed in academia and industry. Since the hosts participating in a grid computation are owned by numerous different people or organizations and located all over the world, the task of distributing application code to all of the participants is nontrivial. In the case of general grid computing infrastructures like ConCert (as opposed to single-purpose grids such as the SETI@Home network), the automatic transport and execution of mobile code — *untrusted* mobile code, in fact — is essential. Indeed, ConCert programs are designed to start on a single host and recruit additional hosts as they run, spawning new copies of themselves that migrate to new locations in much the same way mobile agents do.

The security risks associated with automatically executing untrusted code are hard to overstate. The proliferation of e-mail worms that infect a computer when the user opens an attached file shows that even well educated human users can be tricked into running harmful programs. Removing the user from the scenario and executing downloaded software automatically (as mobile code hosts do) without some kind of security measure would clearly be a disaster. The standard advice to users regarding e-mail worms — not to open attachments unless they both trust the apparent sender and were expecting the message — is hard to apply to machines that are supposed to play host to mobile agents or ConCert grid programs. After all, mobile agents arrive for execution unsolicited and without warning, and they often do not come directly from their place of origin. Furthermore, it is often necessary or desirable that hosts be willing to execute code whose authors they do not know.

1.1 Certified Code

Static safety and security verification of software has become increasingly commonplace over the past decade, beginning with the introduction of Java by Sun Microsystems in 1995–6 [66]. Designed expressly for network-based computing, Java technology is based on an object-oriented virtual machine (the Java Virtual Machine or JVM [42]) whose high-level bytecode language was intended for use as an interchange format for software. Security was a major selling point: the Java Virtual Machine could be configured to support a range of different security policies, and because the JVM language was supposed to be type safe, a Java program could not interact with the network or file system except through the carefully designed interfaces provided by the virtual machine. To ensure that no Java bytecode, no matter how malicious or incompetent its author, could circumvent the protection of the type system, programs would be ***verified*** prior to execution.

The central role of the Java type system in the security of Java-based software inspired many detailed and critical investigations into whether or not it was sound. The results, published in papers with titles like "Java is not type-safe" [61] and "Java is type-safe — probably" [20], showed mainly that the Java type system was large and complex, with many dark corners in which unsafety might be overlooked. The resistance of Java to formal analysis no doubt helped to fuel the trend toward foundationality in certified code, described later in this section. In the end, formal analysis of Java led to positive results for subsets of the type system [20, 64, 26] as well as uncovering a number of bugs (*e.g.,* [61, 25]).

1.1.1 Classic Proof-Carrying Code

The foundations for more formal code certification were laid in 1996-7 in the sequence of papers by Necula and Lee that introduced ***proof-carrying code*** (PCC) [52, 50, 51]. Among other things, these papers established the basic vocabulary of the field, including the terms *code producer, code consumer,* and *safety policy.* In this original approach to PCC (which I often call "*classic*" PCC to distinguish it from the variations that appeared subsequently), an operational semantics for a safe but undecidable subset of assembly language is used as an informal guide to the manual construction of a program called a *verification condition generator* (VCGen). The VCGen analyzes the code to be certified and computes a first-order formula that implies the safety of the code (the *verification condition* or *VC*); a theorem prover is then used to generate a formal proof of the VC which constitutes the safety evidence for the program. The code consumer validates the certified binary by running the VCGen to extract the code's verification condition and then using a proof checker to verify that the certificate is a valid proof of the VC.

Necula and Lee's early experiments applied the PCC technique to packet filters hand-coded in assembly language [52]. Necula's Ph.D. thesis developed certifying compilation, focusing on a high-level language called Safe-C [54]. Years later, a very similar certification infrastructure was the target of a certifying compiler for Java, called Special J, developed by Cedilla Systems [8]. A striking difference between the certification of hand-coded packet filters and that of full-fledged Java programs, other than the matter of scale, is that the formalized logical discourse (verification conditions and proofs) of the early experiments was conducted at a very low level of abstraction, determined almost entirely by a fairly realistic operational semantics for the machine. The certification in Special J, by contrast, made heavy use of a collection of language-specific predicates, whose meanings were defined only by a set of *ad hoc* axioms; these predicates and axioms connected the instruction-by-instruction actions of the machine code with the higher-level Java type system that guaranteed the safety of the source program. Other researchers would later question

the wisdom of removing the formalized logical activity so far from the hardware level of abstraction.

1.1.2 Typed Assembly Language

The use of type systems for code certification was made quite explicit with the introduction of ***typed assembly language*** (TAL) by Morrisett *et al.* in 1999 [47]. The original TAL paper concerned a language based on a generic RISC-like architecture and showed how various programming idioms, including procedure linkage conventions, could be described using typing constructs that amounted more or less to System F with sums, products and existential types. The abstract machine in the first TAL paper had no stack; this technical limitation was overcome in Stack-Based TAL (STAL) [46], enabling a concrete implementation on the Intel IA-32 called TALx86 [45].

The theory and the implementation of TAL had a good deal of influence over certified code research that followed, including the present thesis. The deep connections it forged between conventional type theory and low-level code have allowed ever more complex and powerful type systems to be brought to bear on code certification problems. In addition, the implementation of TALx86 has been used in a number of subsequent research endeavors and has even proven robust enough to serve as the target of a full-scale ML compiler [56].

1.1.3 Foundationalism

Classic PCC and TALx86 introduced a certain degree of formalism to code certification, but it was soon apparent that there was room for more. In 2001, Appel *et al.* kicked off the trend of *foundationalism* in certified code by observing that despite the intentions of Necula and Lee and of Morrisett *et al.* to base their systems on sound type theory and logic, the security of these systems depended on the correctness of large amounts of code and *ad hoc* theory that were verified only informally or not at all.[2] Specifically, Appel noted, it would have been "a daunting task" to meticulously check every aspect of the type safety proof for TALx86 (even though the simpler abstract TAL had been subjected to rigorous scrutiny), let alone to formally verify the *software* implenting TAL's type-checker or PCC's VCGen [3].

Appel *et al.'s* ***foundational proof-carrying code*** (FPCC) aims to place code certification on a sound formal footing by reducing as much as possible the ***trusted computing base***, that body of code and theory that one needed to trust in order to believe in its security guarantees. The "proof" attached to an FPCC program is nothing other than a machine-checkable proof (in higher-order logic) of the proposition that that program obeys the safety policy. The main problem with this enterprise was one of scale: how in the world could a proof of safety for a practical-sized program be automatically generated, let alone represented compactly enough to be transmitted with mobile code over a network and verified in a reasonable amount of time?

A type system for machine language played a central role in the solution developed by Appel *et al.* They defined a semantic model of the types in their system, which amounts to defining a logic predicate for each type characterizing the structure of values of that type [4]. The model, together with a formal specification of the operational semantics of the machine, allowed the soundness of each typing rule to be proven as a lemma. A proof of safety for a given program could then be constructed from a typing derivation for that program, replacing application of typing rules with applications of the corresponding lemmas.

[2] "*Ad hoc*" is not pejorative here. The basic principles of type theory were considered to rest on solid ground; it was the metatheoretic work concerning the specific properties of PCC and TAL that was potentially in question.

Constructing formal semantic models and proving them correct in machine-checkable higher-order logic is not easy, and was perceived as the weak link in FPCC. In 2002, Hamid *et al.* [28, 29] proposed an alternative: instead of formalizing proofs of type safety in what amounted to a denotational semantics for a low-level type system, they formalized a *syntactic* safety proof in the style of Wright and Felleisen [71]. This style of proof was already known to be much more tractable than model-theoretic soundness proofs. Once the soundness of the system as a whole was proven, the proof obligation for any specific program was reduced, as before, to showing that it was well-typed.

The proofs in the syntactic FPCC of Hamid *et al.* were encoded in the Calculus of Inductive Constructions [55]. The next year, Crary unveiled TALT, a "foundational typed assembly language" with a machine-checkable safety proof encoded in the Twelf meta-logic [14, 13]. As I will explain later on, TALT was not only the most advanced and expressive type system for machine language to date, but served a higher purpose as the underlying type system of the first implemented instance of a ***metalogical foundational certified code*** framework. As this framework provides the context for all the technical work in this thesis, my first order of business will be to describe its operation; this exposition makes up Chapter 2 of the thesis.

1.1.4 Static Safety in Operating Systems

The notion of enforcing safety policies by static rather than dynamic means seems to have received only sporadic interest from the operating systems community until fairly recently. This is presumably due to the historical focus on performance as the primary goal of operating systems research and a widespread belief that static policy enforcement hurts performance; in recent years, however, security has emerged as an issue of paramount concern [67].

The most-cited example of an operating system leveraging the static safety properties of a programming language is SPIN [6], an extensible operating system that allows applications to extend the kernel with specialized paging algorithms, network protocol implementations, and other performance-enhancing modules written in the type-safe Modula-3 language. As noted, the original PCC experiments conducted by Necula and Lee focused on the ability to link untrusted code into an operating system kernel safely.

Singularity is an operating system under development at Microsoft Research [37]. It differs from SPIN in that not only the microkernel and all device drivers and system services, but all user-level applications as well, must be written in a type-safe language and verified prior to execution. In defense of this somewhat radical shift, the Singularity team conducted some measurements of the costs of hardware-based dynamic versus software-based static isolation of processes, the results of which led them to declare that the benefit of eliminating the overhead of dynamic safety policy enforcement is significant [1].

Admittedly, the dynamic approach has a substantial head start, and may have influenced software and hardware design too much for static approaches to be adopted in commercial operating systems any time soon. On the other hand, type-based certification makes this otherwise impossible idea a potentially viable alternative and has not existed for very long compared to the amount of time the systems community has spent engineering systems based on the dynamic approach. Whether these recent developments are the leading edge of a major change in computer software architecture remains to be seen.

Even if the policies of mainstream operating systems continue to be dynamically enforced for the foreseeable future, many applications that rely on untrusted third-party code have policies of their own that they must enforce. Furthermore, these policies might be concerned with aspects

of program behavior over which the operating system does not provide direct control, or applications may require a finer degree of control than the operating system offers. In situations like these, code certification can provide a means to enforce the application-specific policies.

1.2 Timing Properties for Safety

In order for all of the processes on a computer system to function properly, it is important that each of them be given enough time in which to run. In addition, for applications requiring user interaction or real-time control of devices, it is important that the intervals during which a given process is not running be short enough to allow that process to react to events in a timely manner. To address these requirements, operating systems manage the scheduling of processes. Time is allocated to processes in quanta known as *time slices*: at the start of a time slice, the OS hands over almost total control of the CPU to a user process with the understanding that it will only keep that control for a certain amount of time. At the end of that interval, if the user process has not performed any action to return control to the operating system, it is *preempted*; the state of the process is saved and it becomes dormant until the system chooses to give it another time slice.

It is usual to view this multiplexing of processes onto the CPU as a resource allocation task performed by the operating system. However, it can also be viewed as a part of the problem of enforcing rules of behavior for processes. As I have said, the operating system must set up and enforce behavioral rules that programs running on the machine must obey. The requirement that a program not keep control of the CPU for too long at a time is not fundamentally different from the requirement that it access only its own memory pages or that it refrain from performing privileged instructions; in all of these cases, one program's failure to comply with the operating system's policy might affect other programs' behavior in ways the other programs' authors could not have anticipated. Therefore, I choose to see pre-emption as an *enforcement mechanism for timing policies*. The policy states that no program shall keep continuous control of the CPU for more than the length of one time slice, and the enforcement mechanism relies on detecting violations at run time and forcibly correcting an offending program's behavior.

It is natural to ask, then, whether it is possible to enforce timing policies such as this one statically. Is it possible to use code certification to guarantee *before running a program* that it does not violate the policy? If it is possible, then doing so could relieve operating systems of the responsibility of enforcing the rules by detecting violations at run time — if a program passes the necessary verification prior to running, there will be nothing to detect. Just as the static process isolation in Singularity eliminates the need for hardware-supported memory protection, static enforcement of timing policies could eliminate the need for preemption.

Simplifying operating system implementation is only the beginning. Fundamentally, an operating system must regard non-certified programs with unreserved suspicion. They must be treated as adversaries and firmly regulated in order to keep the system as a whole secure. Certified programs, on the other hand, can be expected to play fair, and can therefore be given some freedom to control their own execution. A certified program can be trusted to choose, within appropriate limits, exactly when it will yield. Consequently, it can be given the responsibility of saving its own state before a context switch – which it knows how to do better than the OS can – and can arrange to avoid yielding at inconvenient times, such as inside short-lived loops where the loss of data from the cache might be particularly disruptive.

Certification-based enforcement of yielding policies is consistent with the so-called "pay-as-you-go" principle, which encourages the design of systems in which one does not incur costs for unused features. The overhead of preemptive multitasking is generally understood to be a

significant but unavoidable cost of guaranteeing the stability and reliability of a computer system. Many programs, though, such as those that perform a lot of blocking I/O operations or spend a lot of time waiting for GUI events, tend not to hold on to the CPU for very long. The overhead of preemption would be unnecessary for such programs, if only the scheduler could identify them with certainty. In traditional systems with non-certified executables, this clearly cannot be done. However, if programs were known in advance to be cooperative, the overhead of arranging for preemption by a timer interrupt that will never occur could be avoided. Certification of timing policies would make this possible.

The idea of enforcing timing policies with code certification goes back all the way to Necula and Lee, who observed that PCC could be used to guarantee bounded running time of programs [51]. In Crary and Weirich's type theory LXres and assembly language TALres, the type of a function specifies its running time, often as a function of the structure of its arguments [18]. More recently, Naik [49] described a Typed Interrupt Calculus, a core language for interrupt programming whose type system guarantees that all interrupts will be handled before their associated deadlines. To date, none of these approaches has produced a workable solution for general-purpose programming on a realistic scale. Nonetheless, the basic structures needed for reasoning about time introduced by Necula and Lee and picked up by Crary and Weirich and by Naik form the basis for my work as well.

Some timing properties can be enforced using a type system based on linear logic. In particular, Hofmann has shown that any program in a certain linear λ-calculus denotes a polynomial-time function [35]. This is a weaker property than any realistic safety policy, since even a constant-time function can take longer to finish than a supervisor is willing to wait. Also, the linear type system is not particularly user-friendly. The real advantage of Hofmann's calculus is that it controls *space* usage of programs by forbidding them to allocate new storage; the linear typing discipline allows preallocated space to be reused in a type-safe way. Hofmann and Jost later showed how to allow a limited amount of allocation while controlling the total memory usage of programs [36]. The issue of space usage was also addressed by Aspinall *et al.* in a logic for reasoning about resource usage in a fragment of JVM bytecode [5].

1.3 Thesis Overview

> *My claim in this thesis is that static enforcement is possible for a wide range of timing policies. Indeed, I claim that the complexity of enforcing this policy is a small increment over that of enforcing memory safety. Finally, I claim that certifiable adherence to the policy presents no burden to most application programmers and requires only a modest contribution from the implementors of the development tools they use.*

The technical content of this thesis begins in Chapter 2 with an overview of the Crary-Sarkar metalogical code certification framework and the TALT type system. The next five chapters comprise a case study in certifying compilance with a specific timing policy. Chapter 3 describes this policy, called *responsiveness,* and the type system called TALT-R that I developed to certify programs that obey it. Chapter 4 explores the metatheory of a key subsystem of TALT-R.

The generation and certification of responsive programs is laid out in the next three chapters. To support my claim that certifiable responsiveness presents no burden to most programmers, I consider the problem of generating compliant binaries without the benefit of any timing-related input from the programmer, because programmers of traditional preemption-based systems are

not accustomed to providing any. Chapter 5 describes a timing-ignorant typed intermediate language that will serve as the source language in my discussion of compilation. In Chapter 6 I covers the basic techniques I have worked out for making programs certifiably responsive, and Chapter 7, which gives a formal translation from the language of Chapter 5 to TALT-R.

Chapter 8 discusses the application of the ideas in TALT-R to certification of other timing policies and to other resource control problems. In Chapter 9 I present the results of some empirical experiments with TALT-R, discuss possible directions for further work on this subject, and give my final conclusions.

Chapter 2

TALT Background

> *A language design can no longer be a thing. It must be a pattern — a pattern for growth — a pattern for growing the pattern for defining the patterns that programmers can use for their real work and their main goal.* — Guy Steele [65]

The code certification machinery I have designed to enforce timing policies is based on the existing body of work by Crary and Sarkar on a so-called *metalogical approach* to foundational certified code [15], including the type system TALT [14]. In this chapter I review the basic ideas necessary to understand what this means, and sketch the type system and semantics of TALT. Readers familiar with Crary and Sarkar's work may find that this chapter is mostly review; however, it establishes the meanings of several terms that I will use throughout the remainder of the thesis.

2.1 Metalogical Foundational Certified Code

The metalogical approach to foundational certified code was developed concurrently with the TALT type system, and the needs of each influenced the design of the other. It is a mistake, though, to think of the formalized metatheory developed by Crary and Sarkar simply as "the TALT safety proof," however common it may be to call it that. The intent was much more general: to formalize a single, foundational safety policy, in a logic capable of proving the safety of as wide a variety of programs as practical. The preferred way to structure safety proofs for programs was to base them on type systems; TALT is merely one point in the vast design space of type systems for machine language whose safety can be proven within this framework. It serves (among other purposes) to demonstrate how type safety proofs in the metalogical framework may be constructed and as a starting point for the design of more expressive or more specialized type systems.

2.1.1 LF, Elf and Twelf

The "metalogical" aspect of Crary and Sarkar's approach to foundational certified code lies in its use of the Twelf metalogic for the expression of the safety policy and the statements and proofs of all safety theorems. It appears to be common in colloquial speech among users of the Twelf system to say that these definitions and proofs are conducted "in LF," but my opinion is that this leads to confusion. For the purposes of this thesis, therefore, I make the following definitions.

LF, the Edinburgh *L*ogical *F*ramework, is either the type theory first given that name by Harper, Honsell and Plotkin [31] or its subsequent reformulation by Harper and Pfenning [32]. LF is

instantiated by specifying a set of uninterpreted constants, each with a type or kind as appropriate; this information comprises a *signature*. The LF type theory itself is too weak to be of practical interest without a nontrivial signature.

Elf is a logic programming system based on LF. An LF signature is treated as a logic program; Elf attempts to answer queries of the form "Is there a substitution of closed terms for the free variables in A such that the resulting type is inhabited?" by conducting a search based on unification and backtracking. The name "Elf" is rarely heard nowadays, since the software implementing it was expanded considerably and renamed "Twelf" circa 1999. Nevertheless, I will refer to LF signatures (or fragments thereof) that are intended to be executed by the logic programming interpreter as *Elf logic programs*.

Twelf is the current generation of the Elf software package [59]. It includes all of the logic programming functionality of Elf, plus the capability to check that a logic program is *total* (roughly, that all queries of a certain form have answers). In addition, it includes a theorem prover capable of proving facts about the existence of LF terms of particular types — however, this facility is not used at all in the Crary-Sarkar certified code methodology, so I will not discuss it further.

The Twelf meta-logic is the method of using the totality checker of Twelf as a proof assistant [30]. Under a slight variation of the well-known programs-as-proofs correspondence, a total logic program is essentially a constructive proof of a "theorem" in that it shows how, given closed LF terms of certain types, to find closed terms of other, related types. Since such theorems are *about* the existence of LF terms, and LF is usually instantiated to coincide with some logic of interest, the theorems are called *meta-theorems* and the programs that prove them *meta-proofs*. Note that, although proofs do take the form of programs, the fact that they are logic programs (which are sets of constants with declared types and kinds) rather than functional programs (which are simply λ-terms) means that the propositions they prove do not precisely correspond to anything in the LF type theory itself — certainly not to types as in the familiar Curry-Howard isomorphism.

2.1.2 The Metalogical Skeleton

The basic operation of the metalogical certification framework is as follows. The two principals involved in the use of a certification system are the *producer*, who generates the code, and the *consumer*, who wishes to run it on his or her computer. The consumer in general does not trust the producer, but does trust his or her own computer, including its operating system, the runtime environment in which the untrusted code will execute, and any part of the certification and verification machinery over which he or she has control.

The first step is to specify the *safety policy*. As with other approaches to foundational certified code, the safety policy takes the form of an operational semantics for the target machine, that is, a description of the possible states of the machine and a transition relation between those states. Any program whose behavior can be described by this semantics is considered safe; thus it must differ from the "real" semantics of the hardware in that it must *not* contain any transitions corresponding to behaviors the consumer wishes to rule out. In addition to this abstract machine model, the safety policy provides a relation between programs — represented at this foundational level as strings of bytes — and machine states that relates any given program to all of the possible initial states from which execution of that program might begin.

Figure 2.1 is an outline of the body of Twelf code needed for metalogical certification. It contains the names and kinds of all the important "top-level" types, and indicates which principal, the producer or the consumer, is responsible for filling in the definition of each. The three components of the safety policy just described are covered by the first few lines of the outline. Machine

```
% The safety policy
state : type.
...                                          % consumer provides definition
transition : state -> state -> type.
...                                          % consumer provides definition
initial_state : astring -> state -> type.
...                                          % consumer provides definition

% Certificates and Validity
certificate : type.
...                                          % producer provides definition
check : certificate -> astring -> type.
%mode check +CERT +AS.
...                                          % producer provides definition

% The safety theorem
reaches : astring -> state -> type.
reaches_z : reaches AS S <- initial_state AS S.
reaches_s : reaches AS S
              <- reaches AS S'
              <- transition S' S.
safety : check CERT AS -> reaches AS S -> transition S S' -> type.
%mode safety +DC +DR -DT.
...                                          % producer provides definition
%worlds () (safety _ _ _).
%total  (safety _ _ _).                      % consumer checks totality
```

Figure 2.1: The skeleton of metalogical certified code.

states are represented by objects of type `state`, and the relation `transition` defines the dynamic semantics of the machine. Strings of bytes have type `astring`; the relation `initial_state` connects programs, represented as strings of bytes, to machine states. It is the prerogative of the code consumer, who will be running the certified binaries on his or her machine, to define the safety policy, so it is the consumer who is responsible for providing the definitions of these LF types.

The rest of the code to fill in the outline is to be provided by the code producer. The producer's first obligation is to define what the certificates in certified binaries will look like. This amounts to filling in the definition of the type `certificate` in the figure. This is where the flexibility of the metalogical approach begins to show: the framework does not require any particular form of certificate, so the code producer is free to define certificates to be anything from terse typing annotations (as in TALx86) to oracle strings (as in PCC [53]) to proof terms in higher-order logic (as in FPCC [3]) or the Calculus of Inductive Constructions (like Hamid *et al.* [28]) — provided, of course, that he or she can fulfill the remaining obligations using certificates of the form chosen.

Since the code producer gets to define the type `certificate`, he or she must also define what it means for a certified binary to be valid. The producer accomplishes this by filling in the definition of the relation `check` as shown in the figure. The certified binary consisting of program code AS : `astring` and certificate term $CERT$: `cert` is considered valid if the type `check` $CERT\ AS$ is inhabited.

The largest and most important contribution of the producer, of course, is the proof of the safety theorem. The safety theorem is stated in terms of the consumer's safety policy and the producer's method of certification, but modulo these definitions it is always the same. It says:

> If the program AS is well certified, then any state that can be encountered while executing AS has a successor in the `transition` relation.

To state this as a Twelf metatheorem, we first define the relation `reaches` between programs (again, represented as byte strings) and machine states that identifies those states that may be encountered while executing a given program. In particular, the type `reaches` $AS\ S$ is inhabited iff S is reachable via the `transition` relation from some initial state of AS. The kind, mode and totality of the type `safety` together comprise the theorem: Given (any program AS, any certificate $CERT$, any state S and) evidence that $CERT$ is a valid certificate for AS and that S is reachable from an initial state of AS, a state S' can be found such that there is a transition from S to S'. The code producer fills in a definition of `safety` that is well-moded and total, proving the theorem.

To see why this formulation of the safety theorem makes sense, note that it essentially means that the execution of a well-certified program will never get stuck *with respect to the operational semantics defined in the safety policy*. Presumably, the design of the concrete hardware that will run the program defines a successor for every possible state of that concrete machine, even those that the consumer considers unsafe and wishes to avoid. The operational semantics in the safety policy, on the other hand, is constructed on purpose to lack these states (or the transitions that lead to them). Assuming that the safety policy's `transition` relation mirrors the behavior of the hardware closely enough,[1] the import of the safety theorem is that when a well certified program is executed on a concrete machine, the machine will never visit any state (or perform any transition) not covered by the safety policy. In other words, well certified programs stay within the realm of allowable behavior.

The alert reader will also notice that this formulation of safety seems to imply that a well-certified program never terminates. This is of little importance. If the safety policy were required to define some notion of "terminal" state, then the theorem could be modified to say that any reachable state *either* has a successor *or* is terminal. Crary and Sarkar apparently considered this nonuniformity irksome, so instead the safety policy endows the machine with an imaginary "halted" state whose successor is itself. Instructions that terminate the certified program and return control to the runtime system cause transitions into this state. In fact, this decision is quite sensible if we consider the mapping between states of the concrete machine and states of the safety policy (which of course is not formally defined, because the states of the concrete machine are not formally defined) to identify concrete states that are indistinguishable by safe programs. The execution of the concrete machine can continue indefinitely, long after any given program has terminated — but of course no program can observe anything that happens after it has finished, so it is fitting that a "halting" execution of the abstract machine ends with an unbounded sequence of transitions between indistinguishable states.

[1] (and that issues of nondeterminism and underspecification are handled properly — this gets tricky and I won't discuss it)

2.1.3 Certified Binaries and Verification

Generic and Specific Obligations

As I have just explained, the producer must make several contributions to the process of metalogical certification: he or she must define the type of certificates, define what it means for a certificate to be valid, prove that programs that have valid certificates are safe, and provide a (valid) certificate for each program he or she asks a consumer to run. The first two of these constitute the definition of a *safety condition*; the producer claims that any program satisfying a certain property (namely, the existence of a valid certificate) is safe and promises to demonstrate that any program he or she submits for execution will have that property (by providing such a certificate). The next contribution, the proof that well certified programs are safe, is known as the producer's *generic obligation* because it addresses the safety, not of any particular program, but of all programs that satisfy the safety condition. The presentation of a certificate along with a program to be run is called the producer's *specific obligation*, since it establishes the particular program in question as a member of that generic class.

Certified Binaries and Verification

The file format for certified binaries in the Crary-Sarkar system is called TBF, for Trustless Binary Format. Figure 2.2 shows the layout of a TBF file. A certified binary contains a magic number that identifies the version of the format, the program code, a second magic number that identifies the safety condition the certificate is supposed to satisfy, and the certificate. The code is IA-32 machine code in raw binary form. The certificate consists of definitions in Twelf syntax; it may contain any number of definitions, but the last one in the file must define a term `thecert` of type `certificate`.

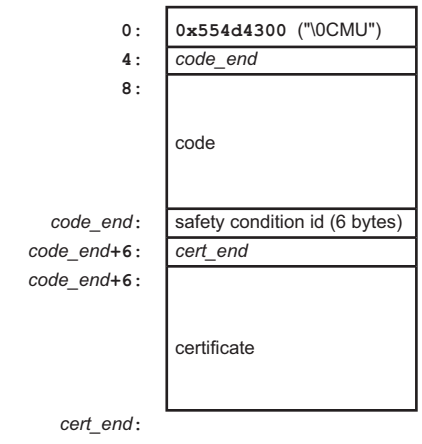


Figure 2.2: TBF File Layout

Upon receiving a TBF file for execution, the consumer consults the magic numbers in the file to determine the safety condition the program is supposed to satisfy. If the consumer is familiar with the particular safety condition claimed and believes it to be sound, only the program-specific proof obligation, the certificate itself, needs to be verified. If the safety condition is unknown to the consumer, however, the producer must first fulfill the generic proof obligation by supplying the complete definition of the safety condition and the generic safety proof.

Since safety proofs are large and are shared between programs, they are not included in TBF files. When the consumer encounters an unfamilar safety policy, it contacts the producer and asks for the Twelf code necessary to fill in the ellipses in the skeleton. Upon receiving it, the consumer feeds the entire "fleshed-out" skeleton into Twelf and asks the program to check the totality of the `safety` relation — remember that if `safety` is a total relation with the specified mode, then it constitutes a proof of the soundness of the safety condition. If Twelf fails to verify that `safety` is total, the consumer has no basis for believing the untrusted program is safe, and rejects it. If Twelf succeeds in verifying the totality of `safety`, then the consumer can not only go on to check the certificate of the particular program in question, but can also remember this generic result and skip the step of safety proof verification when it encounters programs certified to this safety

condition in the future.

Once the soundness of the safety condition is established, the consumer moves on to checking that the particular program under consideration is well certified. The basic idea is simply to translate the sequence of bytes in the code section of the TBF file into an LF term *theprog* of type `astring` and check that the type (`check thecert` *theprog*) is inhabited, but as of this writing there are two different mechanisms under development for accomplishing this. The first relies on the logic programming features of Twelf to check the certificate; the second requires the producer to supply a checker written in a different language.

The first approach, which has been under development longer and is closer to completion, is to assume that the safety condition `check` is defined in such a way that it can be interpreted as an Elf logic program. Under this approach, the consumer simply passes the query "`check thecert` *theprog*" to the Twelf interpreter. If the query succeeds, then the type named by the query is inhabited and the program is well certified; if the query fails (or fails to terminate after a reasonable period of time), then the program is rejected. The second approach is the subject of Susmit Sarkar's forthcoming Ph.D. thesis. Briefly, it requires the producer to submit, along with the definition of the safety condition and the proof of its soundness, a checker for certified programs. These checkers are to be written in a functional language with a highly specialized type system capable of guaranteeing that any well-typed checker is correct with respect to the LF definition of the safety condition.

Regardless of which approach is used to verify the correctness of the certificate, once this step is complete the consumer should believe that the program obeys the safety policy and be willing to run it. Running a program simply consists of loading the bytes from the code section of the TBF file into an executable area of memory and jumping to the first instruction in a manner consistent with the safety policy.

2.2 TALT

The most effective known way to fulfill a code producer's proof obligations in a certified code setting is to use a type system. Indeed, in early non-foundational certified code architectures (PCC and TAL), the safety policy itself was expressed as a type system. In all foundational proof-carrying code implementations that I am aware of, proofs of safety are constructed using a type system: a program is shown to be well-typed, and this fact is shown to imply its adherence to the safety policy. Metalogical certification is no exception to this pattern, as the only safety conditions and safety proofs that have been created for use within the framework so far have been based on type systems. The details of the implementation have evolved slightly since it was first conceived and presented by Crary [14], but there has always been a type system at the core of the safety argument. That type system is known as TALT, and it was the starting point for the type system design I will discuss in detail in upcoming chapters. In this section, I describe the most important features of TALT.

2.2.1 TALT, XTALT and EXTALT

In an ideal instantiation of the Crary-Sarkar framework, the safety argument might be built around a "type system for machine language," that is, a system of formal judgments that apply directly to sequences of bytes and characterize their structure as data and their behavior as code. A certificate for a particular string of bytes would simply be a typing derivation, possibly compressed so as to remove information that could be reconstructed by examining the byte string and certificate

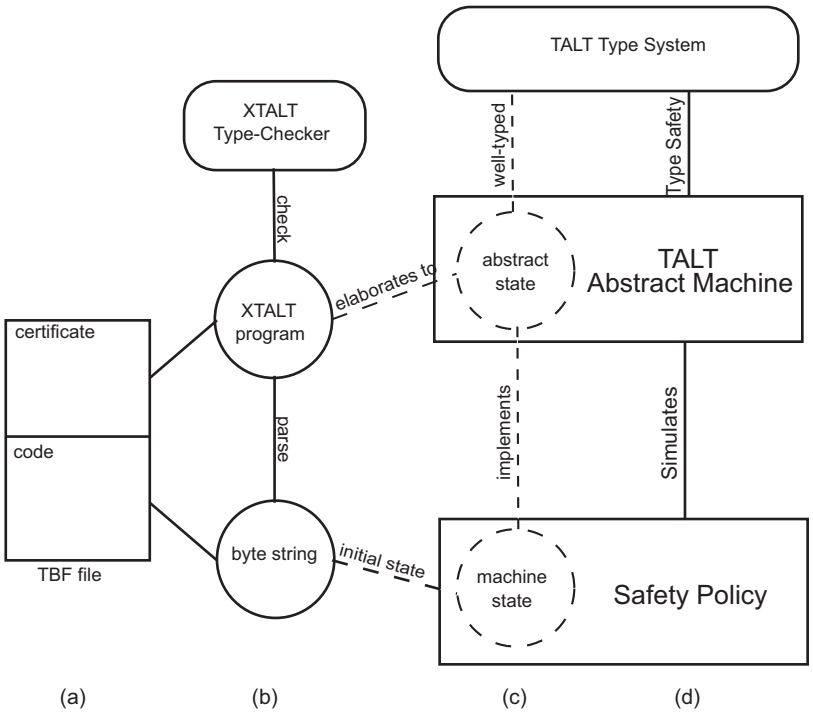


Figure 2.3: Overview of the TALT safety structure.

together. Validity of a certificate would be checked by attempting this reconstruction and, if successful, checking that the resulting derivation proves the appropriate judgment. The safety proof would be based on progress and type preservation lemmas for the type system with respect to the safety policy's operational semantics.

The notion of a "type system for machine language" is so vague that it could probably be argued that this is how TALT works. However, such an argument would stretch the limits of what most practitioners in the field mean by most of the terms involved. The intuitions behind the TALT system are therefore best understood by casting it in a somewhat different light, which reveals at least three closely related but distinct type systems and two different operational semantics. As I have already explained, the safety policy defines an abstract machine and its operational semantics; the "TALT abstract machine" is defined in the course of the safety proof and is the focus of much of the type safety argument. The TALT type system itself is a type assignment system for the TALT abstract machine; XTALT is an explicitly typed version of TALT used in certificates; and EXTALT is the more user-friendly input language of the certifying assembler.

The relationships between these components are sketched in Figure 2.3. The TALT abstract machine is the core of the system. This abstract machine is intended to closely resemble the one whose operational semantics constitute the safety policy (which in turn is an abstracted view of the IA-32) but to be abstract enough to insulate the type safety proof from machine details such as instruction encoding and the precise layout of a program's address space. Differences between the TALT abstract machine and the safety policy include:

- The TALT abstract machine distinguishes between instructions and the bytes that encode them, whereas in the safety policy the only values are bytes or sequences of bytes. In ad-

dition, the instruction set of the abstract machine has some instructions that do not exist in the safety policy or in the concrete IA-32 instruction set. Each of these "fake instructions" is encoded by a sequence of zero or more IA-32 instructions.

- The TALT abstract machine allows arbitrary combinations, and arbitrary-depth nestings, of operand and destination addressing modes. This is for the sake of uniformity; combinations that do not correspond to real IA-32 instructions will simply never come up in actual programs.
- The TALT abstract machine treats the stack as a special object, distinct from the heap, whereas the safety policy treats all of memory uniformly and treats the stack pointer register just like the other general-purpose registers.

As the figure shows (column (d)), the TALT type system is sound with respect to the TALT abstract machine, which in turn is related to the abstract machine of the safety policy by a simulation theorem. These two facts together imply that if the concrete machine starts in a state that "implements" a well-typed abstract machine state, the subsequent evaluation will not get stuck.

One important characteristic the safety policy and the TALT abstract machine have in common is that they are basically untyped. Values, instruction sequences, and states of the TALT abstract machine are all completely free of typing annotations.[2] The type system properly called TALT is a type assignment system, or a Curry-style type system, for the TALT abstract machine.

TALT is a fairly powerful type system, including (among other things) System F-style polymorphism; thus it is presumed that since typing is undecidable for Curry-style F [70] it is undecidable for TALT as well. As a result, a certificate for the binary representation of a TALT program must provide not only enough information to parse the machine code as a sequence of TALT instructions and values, but also enough typing information to reconstruct a typing derivation for the implicitly-typed TALT code. In the current implementation, these requirements are met by using an explicitly-typed program as the certificate. The language of explicitly-typed TALT programs is called XTALT; the types of XTALT are the same as those of TALT, but the term language is very different and so are the typing rules.

An XTALT program is a sequence of blocks, each with a label and an explicit type annotation. (As in conventional assembly language, labels are intended to denote the memory addresses at which their associated blocks reside.) A theory of *coercions* takes the place of TALT's rich subtyping relation, and the syntax of instructions and values is constructed so as to make syntax-directed type-checking of XTALT programs feasible. No operational semantics is directly defined for XTALT; instead, a relation called *elaboration* is defined between XTALT programs and TALT values; if a program X elaborates to V, then V is the TALT representation of the sequence of bytes encoding X. Thus XTALT is built for the convenience of the type-checker, while TALT is built for the convenience of the safety proof.

The assembler that produces certified binaries must therefore output an LF representation of the XTALT representation of the machine code it generates. The input to the assembler is in a third language, which for the purposes of this thesis I will call EXTALT (because it is the *ex*ternal language of the assembler). At present EXTALT does not differ significantly from XTALT. As I will describe in the next chapter, however, my resource-bounded versions EXTALT-R and XTALT-R do differ from each other in an important way, which will have an important implication for the assembler that must translate between them.

[2] Crary [11] has stated that this was simply more convenient when proving the safety theorem, but I think it can be justified on the aesthetic grounds that it avoids putting a lot more distance between the abstract and concrete operational semantics.

Figure 2.3 also shows how certificate verification in TALT works. When a TBF file is received for execution (column (a)), its two components are extracted. The code is a sequence of bytes (an LF term of type `astring`), and the certificate is an XTALT program (also represented as an LF term). The safety condition consists of the relationships depicted in column (b): the XTALT program must be well-typed, and the byte string must in fact be an encoding of that program. Verification of a certified binary amounts to checking that these two relationships hold. That is all that need be checked for any particular program: the generic safety proof for TALT argues that if the safety condition is met, then the relationships indicated by dashed lines in column (c) also hold. The XTALT program determines an initial state of the abstract machine, and the byte string determines an initial state of the safety policy's machine. The safety condition implies that the abstract state is well typed and the concrete state implements the abstract state; thus, by the type safety and simulation theorems, execution of the program will not get stuck.

2.3 MiniTALT

Unfortunately, none of the variations of TALT just described will do for the purposes of presentation in this thesis. The typing annotations of EXTALT and XTALT are bulky and cumbersome, although these languages do have the advantage of being geared toward the presentation and static analysis of programs rather than proofs of safety. TALT itself, being a type assignment system, is concise, but really applies to machine states and values rather than a convenient notion of "program". Furthermore, and most annoyingly for human readability, the TALT abstract machine deals very explicitly with the operands of jump instructions, which (when not indirect) are almost always pc-relative; it also is explicit about the sizes of instruction encodings, which on the IA-32 vary from one instruction to another. Thus finding the target of a direct jump instruction requires knowing the sizes of all the instructions in between the jump and the target — not something that readers of a thesis should be asked to do in order to understand simple examples.

Therefore, for this thesis, I adopt a compromise. For the purposes of all the code examples, translations and proofs herein I will use a block-structured, implicitly typed language called MiniTALT. Like TALT, MiniTALT is implicitly typed, so examples are unburdened by typing annotations (except as comments where they are helpful). Like XTALT, MiniTALT treats a program as an entity unto itself rather than as a value in the memory of a machine. A program is a sequence of blocks, each with a label; a label can occur as an operand, where it is intended to denote a pc-relative reference to the location where its associated block resides. Table 2.1 summarizes these relationships.

	Implicitly Typed	Explicitly Typed
Blocks	MiniTALT	XTALT
Flat	TALT	*none*

Table 2.1: Variants of TALT

In addition to its more readable syntax, the MiniTALT type system I will use in this thesis is a considerably simpler theory than the TALT actually implemented as an instance of the metalogical certification framework. Specifically, I have removed many types that do not appear in any of the code examples in the thesis, and many inference rules that do not play a role in any of the typings we will encounter. For the most part, these omissions merely serve to exclude distracting information much of which has already been covered by Crary and Sarkar [14, 15].

$W = 4$ (word size in bytes)
$B \in \mathit{Wordval} = \{0, \ldots, 2^{8W} - 1\}$

$r \in \mathit{Reg} = \{\texttt{eax}, \texttt{ebx}, \texttt{ecx}, \texttt{edx}, \texttt{esi}, \texttt{edi}, \texttt{ebp}\}$
$\bar{r} \in \mathit{Genreg} = \mathit{Reg} \cup \{\texttt{esp}\}$

Figure 2.4: Machine-Specific Notation for IA-32

Operands	o	::=	$B \mid \ell \mid \bar{r} \mid i\texttt{`}[o + j] \mid i\texttt{`}[o_1 + j + j' \cdot o_2]$
Destinations	d	::=	$\bar{r} \mid i\texttt{`}[o + j] \mid i\texttt{`}[o_1 + j + j' \cdot o_2]$
Conditions	κ	::=	`e` | `ne` | `b` | `be` | `a` | `ae` | `o` | `no`
Instruction Sequences	I	::=	ϵ
		|	`add` $d, o_1, o_2\ I$
		|	`addsptr` $d, o, n\ I$
		|	`call` $o\ \ell$
		|	`cmp` $o_1, o_2\ I$
		|	`cmpjcc` $o_1, o_2, \kappa, o_3\ I$
		|	`halt`
		|	`jcc` $\kappa, o\ I$
		|	`jmp` $o\ I$
		|	`malloc` $d, n\ I$
		|	`mallocarr` $d, n, o\ I$
		|	`mov` $d, o\ I$
		|	`pop` $n, d\ I$
		|	`push` $o\ I$
		|	`ret` I
		|	`salloc` $n\ I$
		|	`sfree` $n\ I$
		|	`sub` $d, o_1, o_2\ I$
Programs	P	::=	$\ell_1 = I_1, \ldots, \ell_n = I_n$

Figure 2.5: MiniTALT Program Syntax

2.3.1 Basic Syntax

The syntax of MiniTALT programs is given in Figures 2.4 and 2.5. Figure 2.4 defines some basic notation that is specific to the IA-32 architecture: the word size $W = 4$ and the names of the registers. Figure 2.5 has the syntax for operands, destinations, instruction sequences and programs.

Most of the MiniTALT syntax should be recognizable to those familiar with Intel-style assembler syntax [38]. Unlike informal accounts of conventional assembler syntax, MiniTALT distinguishes between operands, which produce values, and destinations, where values can be stored. Also, unlike the concrete IA-32 instruction set, MiniTALT does not restrict the combinations of operands and destinations that can appear together in an instruction.

An operand is either a literal word, a label, a register, or a memory operand. A destination may be any of these except for a literal or label.[3] Memory operands are annotated with the size (in bytes) of the value being fetched from memory; thus $1\text{`}[\texttt{eax} + 2]$ means a single byte, located at offset 2 from the pointer in `eax`. I will elide the size prefix for word-sized operands, and I will elide the offset (the $j$ in $[o + j]$) when it is zero. Thus the 4-byte word pointed to by `ebp` can be written simply $[\texttt{ebp}]$. For array element operands $i\text{`}[o_1 + j + j' \cdot o_2]$, the scaling factor j' may be elided if it is 1. Similar conventions apply to destinations.

Many of the instructions of MiniTALT (*e.g.* `add`, `jmp`) are familiar IA-32 instructions. The `call` instruction is slightly unusual in that each `call` specifies the label that should be used as the return address when it is executed. (This is for the convenience of the abstract operational semantics, so that every code pointer ever manipulated by the machine corresponds to a label that occurs in the program text.) The instruction $\texttt{call}\ o\ \ell$ will therefore usually mean "Call the function o, and when it returns, continue with the code at ℓ." This is more general than the `call` of concrete IA-32 implementations, in which the return address is always the address of the instruction immediately following the `call`. For readability, code examples will be written in a style that appeals to this intuition: *e.g.*, the two-block program fragment $\ell_1 = \texttt{call}\ o\ \ell_2,\ \ell_2 = \texttt{jmp}\ \ell_1$ will be written with a single block, as $\ell_1 = \texttt{call}\ o\ \texttt{jmp}\ \ell_1$.

Most of the other instructions of MiniTALT are implemented straightforwardly with IA-32 instructions or short sequences of instructions. The `addsptr` instruction, for example, adds a constant value to a pointer into the stack; this is implemented by an `add` instruction on an actual hardware (that is, the same bytes that encode `addsptr` also encode `add`), but since the abstract machine treats pointers differently from integers, and the stack as distinct from the heap, a different syntax and typing rule are needed at the type level. Similarly, `sfree` is "really" just an `add` instruction.

The remaining instructions of MiniTALT are implemented by sequences of two or more IA-32 instructions. For instance, a `cmpjcc` is implemented by a `cmp` followed by a `jcc`; the combined instruction at the TALT level has a special typing rule allowing typings that could not easily be achieved with two separate instructions.[4] Less obviously, `salloc`, which allocates space on the stack, is implemented by a sequence of two instructions: a `sub` to decrement the stack pointer, and a `mov` into the newly allocated space that triggers a page fault if the stack has overflowed. Finally, the `malloc` and `mallocarr` instructions are implemented as `call` instructions that invoke functions in the runtime library to allocate space in the heap.

[3] In conventional IA-32 assembly language a label may be used as a destination; this is disallowed in TALT to keep code position-independent.

[4] "Easily" is a weasel word here. Presumably one *could* come up with special rules for `cmp` and `jcc` that relied on some *ad hoc* device to make sure they were only ever used when the instructions appeared together. I do not even mean to suggest that that would be a bad idea, but it is not how TALT works.

Kinds	K	::=	$\mathsf{T} \mid \mathsf{T}i \mid \mathsf{TD} \mid \mathsf{N} \mid k_1 \to k_2$
Static Terms	c, τ, x	::=	α
		$\mid$	$\mathtt{ns}i \mid \mathtt{B0} \mid \mathtt{B}i \mid \tau_1 \times \tau_2 \mid \tau \uparrow x \mid \mathtt{box}(\tau) \mid \mathtt{mbox}(\tau) \mid \mathtt{sptr}(\tau)$
		$\mid$	$\Gamma \to 0 \mid \mathtt{set}_{=}(x) \mid \mathtt{set}_{<}(x) \mid \mathtt{set}_{>}(x) \mid \forall\alpha{:}K.\tau \mid \exists\alpha{:}K.\tau$
		$\mid$	$\tau_1 \wedge \tau_2 \mid \tau_1 \vee \tau_2 \mid \mathtt{void} \mid \mu\alpha.\tau \mid \overline{n} \mid \lambda\alpha{:}K.c \mid c_1\, c_2$
Static Contexts	Δ	::=	$\cdot \mid \Delta, \alpha{:}K \mid \Delta, \varphi\ \mathsf{true}$
Register File Types	Γ	::=	$\{\mathtt{eax}{:}\tau_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{bp}}, \mathtt{esp}{:}\tau_{\mathtt{sp}}\}$
Memory Types	Ψ	::=	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$

Figure 2.6: MiniTALT Type System Syntax

2.3.2 Type System

The syntax of the MiniTALT type system is given in Figure 2.6, and the judgment forms are summarized in Table 2.2. At the top level of the system are four *kinds*, which classify the terms at the second level, which we call *static terms*. The class of static terms is comprised of the *types* (of kinds $\mathsf{T}i$, TD and T) and the *number terms* (of kind N). By convention, I will use the metavariables τ and x in place of the general metavariable c to indicate that the static term referred to is a type or a number term, respectively. Furthermore, I will use the letter a instead of α for variables intended to be of kind N. The only terms of kind N other than variables are the *numerals*, written $\overline{n}$, where n is a nonnegative integer. (The set of static terms will be expanded significantly when I extend MiniTALT to MiniTALT-R in Chapter 3.) I have chosen to call the syntactic category containing the types "static terms" rather than the more usual "type constructors" (or simply "constructors") because although number terms may appear in types, they cannot really be said to *construct* anything. The name "static terms" also highlights my intention that these terms are part of the (static) type assignment system only; they do not appear in raw MiniTALT programs.

As usual, the role of types is to classify values. The need for three different kinds for types comes from an unusual feature of TALT, namely that values are not all the same size — in fact, it is not even the case that values of the same type have the same size (for example, consider the type $\mathtt{B0} \vee \mathtt{B4}$). For each natural number $n \geq 0$, $\mathsf{T}n$ is the kind of types whose values are exactly n bytes in size. TD is the kind of types τ such that all values of type τ are the same size. That is, if τ has kind TD and v_1 and v_2 have type τ, then v_1 and v_2 must be the same size; types of kind TD "*d*etermine" the size of the values they contain. Thus any type of kind $\mathsf{T}i$ also has kind TD. T is the kind of any type whatsoever, even those types that contain values of more than one size. Most of the values manipulated by MiniTALT-R code will have types of kind $\mathsf{T}W$, where W is the word size of the architecture (for IA-32, $W = 4$). The notable exception is the stack, which is permitted to vary greatly in size and hence usually has a type of kind TD.

For $i > 0$, the "nonsense" type $\mathtt{ns}i$ may be given to any value whatsoever of size i; any type of kind $\mathsf{T}i$ is therefore a subtype of $\mathtt{ns}i$. For $i \geq 0$, $\mathtt{B}i$ is the type of integer values i bytes in width. Values of the product type $\tau_1 \times \tau_2$ consist of a value of type τ_1 and one of type τ_2 appended together; hence if $\tau_1 : \mathsf{T}i$ and $\tau_2 : \mathsf{T}j$ then the product has kind $\mathsf{T}(i + j)$. There are subtyping rules that make the product constructor associative and B0 a unit. The array type $\tau \uparrow x$, where x is a number term, describes values that consist of x values of type τ appended together. Thus if τ has kind $\mathsf{T}i$, then $\tau \uparrow \overline{n}$ has kind $\mathsf{T}(ni)$.

Pointers to code (and in particular the labels associated with MiniTALT instruction blocks) have arrow types of the form $\Gamma \to 0$, where Γ is a *register file type*. It is safe to jump to a pointer of

Judgment	Meaning
$\Delta \vdash c : k$	c has kind k
$\Delta \vdash \Gamma$	Γ is well-formed
$\Delta \vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\Delta \vdash \Gamma_1 \leq \Gamma_2$	Γ_1 is a subtype of Γ_2
$\Delta; \Psi; \Gamma \vdash o : \tau$	Operand o has type τ
$\Delta; \Psi; \Gamma \vdash d : \tau \to \Gamma'$	Propagating a value of type τ to d yields Γ'
$\Delta; \Psi; \Gamma \vdash I$	I is well-typed
$\Delta; \Psi \vdash I : \tau$ block	I constitutes a block of type τ
$\vdash P$	P is well-typed

Table 2.2: MiniTALT Typing Judgment Forms

type $\Gamma \to 0$ if the current register state has type Γ. Pointers to data in the heap have type $\texttt{box}(\tau)$; pointers to *mutable* data in the heap have type $\texttt{mbox}(\tau)$. The type $\texttt{sptr}(\tau)$ describes a pointer into the stack. Universal quantification $\forall\alpha{:}K.\tau$ and existential quantification $\exists\alpha{:}K.\tau$ have their usual meanings, as do recursive ($\mu\alpha.\tau$), intersection ($\wedge$) and union types ($\vee$).

The type $\texttt{set}_{=}(x)$, where x is a word term, is a singleton type whose sole element is the word-sized binary representation of the number denoted by x. (If the number is not representable, then $\texttt{set}_{=}(x)$ is an empty type.) The subrange types $\texttt{set}_{<}(x)$ and $\texttt{set}_{>}(x)$ have as their elements the word-sized unsigned representations of numbers less than x and greater than x respectively. In TALT the singleton and subrange types are used mainly for array bounds checking and the implementation of disjoint union types; in TALT-R I will have another important use for the singleton type.

2.3.3 Instruction Typing

Since a MiniTALT program consists of a set of labeled instruction sequences, the central judgment in the type system is the one pertaining to instruction sequences. The judgment

$$\Delta; \Psi; \Gamma \vdash I$$

means that in the context consisting of the kinding assumptions Δ, the memory type Ψ and the register file type Γ, the instruction sequence I is well-typed. In effect, it states that the sequence I is safe to execute when the heap is of the form described by Ψ and the registers contain values of the types described by Γ.

Some of the rules defining this judgment are shown in Figure 2.7; these rules make heavy use of auxiliary typing judgments for operands and destinations, whose meanings are summarized in Table 2.2. One of the simplest instruction typing rules in the system is the one for the `mov` instruction, the first rule in the figure. This rule states that the instruction sequence consisting of $\texttt{mov}\ d, o$ followed by I is well typed if:

- the operand o is well-typed (with type τ), and
- the destination d is well-formed, and Γ' describes the state of the registers after propagating a value of type τ to d, and
- the continuation I is well-typed under Γ'.

$$\frac{\Delta;\Psi;\Gamma \vdash o : \tau \quad \Delta;\Psi;\Gamma \vdash d : \tau \to \Gamma' \quad \Delta;\Psi;\Gamma' \vdash I}{\Delta;\Psi;\Gamma \vdash \texttt{mov}\ d, o} \qquad \frac{\Delta;\Psi;\Gamma \vdash o_1 : \texttt{B4} \quad \Delta;\Psi;\Gamma \vdash o_2 : \texttt{B4} \quad \Delta;\Psi;\Gamma \vdash d : \texttt{B4} \to \Gamma' \quad \Delta;\Psi;\Gamma' \vdash I}{\Delta;\Psi;\Gamma \vdash \texttt{add}\ d, o_1, o_2\ I} \qquad \frac{\Delta;\Psi;\Gamma \vdash o : \Gamma \to 0}{\Delta;\Psi;\Gamma \vdash \texttt{jmp}\ o\ I}$$

$$\frac{(\Gamma(\texttt{esp}) = \tau_s) \quad \Delta;\Psi;\Gamma \vdash o : \tau \quad \Delta;\Psi;\Gamma\{\texttt{esp}{:}\tau \times \tau_s\} \vdash I}{\Delta;\Psi;\Gamma \vdash \texttt{push}\ o\ I} \qquad \frac{(\Gamma(\texttt{esp}) = \tau_s) \quad \Delta;\Psi;\Gamma \vdash \ell : \Gamma_r \to 0 \quad \Delta;\Psi;\Gamma \vdash o : \Gamma\{\texttt{esp} : (\Gamma_r \to 0) \times \tau_s\} \to 0}{\Delta;\Psi;\Gamma \vdash \texttt{call}\ o\ \ell}$$

Figure 2.7: Selected Instruction Typing Rules.

Values	v	::=	$B \mid \ell \mid \texttt{sptr}(n)$
Heap values	V	::=	$I \mid \langle v_1, \ldots, v_n \rangle$
Memories	H	::=	$\{\ell_1 \mapsto V_1, \ldots, \ell_n \mapsto V_n\}$
Flags	b	::=	$0 \mid 1$
Flag sets	ϕ	::=	$\{\texttt{cf} \mapsto b_c, \texttt{zf} \mapsto b_z, \texttt{sf} \mapsto b_s, \texttt{of} \mapsto b_o\}$
Register Files	R	::=	$\{\texttt{eax} \mapsto v_{\texttt{ax}}, \ldots, \texttt{ebp} \mapsto v_{\texttt{bp}}, \mathit{flags} \mapsto \phi\}$
Machine Configurations	M	::=	(H, V_s, R, I)

Figure 2.8: MiniTALT Abstract Machine Configurations

The `add` rule is similar to the `mov` rule, except that the two operands must both have type `B4` — that is, produce 32-bit integer values — and the value propagated to the destination has type `B4` as well.

The typing of control transfer instructions is illustrated by the typing rule for `jmp`. It states that the instruction `jmp` o is well-typed if the operand o has type $\Gamma \to 0$, where Γ is the current register file type. In other words, (the value of) o must be a pointer to code that is safe to execute under precisely those conditions that happen to hold at the time.

The rule for `push` (in the second row of the figure) illustrates the typing of stack manipulation instructions. If the stack has type τ_s, and the operand o has type τ, then the effect of `push` o is to append the value of o to the existing stack, producing a stack of type $\tau \times \tau_s$; the continuation I must be well-typed assuming the stack has this new type.

The IA-32 function call instruction takes a single operand, which is the address of a function. The `call` instruction pushes the specified return address label onto the stack and jumps to the start of the function. Thus `call` is essentially a combination of `push` and `jmp`. The typing rule captures this: in order for the one-instruction sequence `call` $o\ \ell$ to be well-typed, the return label ℓ must have some code type $\Gamma_r \to 0$, and the code pointed to by the value of o must be safe to execute in the state that results from pushing ℓ onto the stack.

The typing rules selected for discussion here cover most of the main ideas at work in MiniTALT type system. The complete set of typing rules, including rules for kinding, subtyping and auxiliary judgments, can be found in Appendix A.

2.3.4 Operational Semantics

The dynamic semantics of MiniTALT is defined in terms of the abstract machine whose configurations have the form shown in Figure 2.8. Because the purpose of MiniTALT is to support clear explanation of key concepts within the pages of this thesis, rather than to play a direct role in the certification of IA-32 programs, the semantics I present here is in the style of TAL [47] and STAL [46], rather than the more realistic but more complicated "official" semantics of TALT. A configuration of the MiniTALT machine consists of a memory (H), a stack (V_s), a register file (R), and an instruction sequence (I). The memory maps locations, which are abstract pointer values not confusable with integers, to ***heap values,*** each of which is either an instruction sequence or a sequence of word-sized values. The stack is also a heap value. The register file associates a word-sized value with each of the machine's general-purpose register names and a bit with each of the four status flags commonly used for conditional jumps; the stack pointer register implicitly points to the beginning of the stack value V_s. A pointer into the stack has the form $\texttt{sptr}(n)$; this is a word-sized value and represents the location n bytes away from the base of the stack.

The (single-step) transition relation $\mapsto$ on machine configurations is defined in Table 2.3. The conditions in the second column of the table are stated in terms of auxiliary judgments for resolving operands and propagating values to destinations; the rules for these judgments can be found in Appendix A.2. The addition and subtraction operations $\oplus$ and $\ominus$ correspond to the 32-bit binary arithmetic performed by an IA-32 processor, and specify both the resulting word and the new values of the status flags; the definitions of these operations are omitted, as are the formal definitions of the condition satisfaction relations $\phi \models \kappa$. Informally, $\phi \models \kappa$ if the status flag values ϕ indicate that the condition κ holds of the most recent arithmetic operation. For details of how these things are determined, see either the TALT paper [14] or the IA-32 manual [38].

2.4 Chapter Summary

This chapter consitutes the background on Crary-Sarkar metalogical code certification necessary to understand the balance of the thesis, including specifics of the type system TALT. TALT is a type system for an abstract machine closely related to a safe subset of the IA-32 architecture, and is the primary exemplar of code certification using the Twelf metalogic. I have described the mechanisms of certification and verification of TALT programs, including the roles of the related systems XTALT and EXTALT, and given a more detailed presentation of the variant called MiniTALT that stands in for TALT in the bulk of the thesis.

If $I = \ldots$	*and...*	$(H, V_s, R, I) \mapsto (H', V_s', R', I')$ *where...*
`add` $d, o_1, o_2\ I'$	$H, V_s, R \vdash o_1 \rightsquigarrow B_1$ $H, V_s, R \vdash o_2 \rightsquigarrow B_2$ $B_1 \oplus B_2 = (B_3, \phi)$ $H, V_s, R \vdash d(B_3) \rightsquigarrow H', V_s', R_1$	$R' = R_1\{\mathit{flags} \mapsto \phi\}$
`mov` $d, o\ I'$	$H, V_s, R \vdash o \rightsquigarrow v$ $H, V_s, R \vdash d(v) \rightsquigarrow H', V_s', R'$	
`sub` $d, o_1, o_2\ I'$	$H, V_s, R \vdash o_1 \rightsquigarrow B_1$ $H, V_s, R \vdash o_2 \rightsquigarrow B_2$ $B_1 \ominus B_2 = (B_3, \phi)$ $H, V_s, R \vdash d(B_3) \rightsquigarrow H', V_s', R_1$	$R' = R_1\{\mathit{flags} \mapsto \phi\}$
`cmp` $o_1, o_2\ I'$	$H, V_s, R \vdash o_1 \rightsquigarrow B_1$ $H, V_s, R \vdash o_2 \rightsquigarrow B_2$ $B_1 \ominus B_2 = (B_3, \phi)$	$R' = R\{\mathit{flags} \mapsto \phi\}$
`jmp` $o\ I_0$	$H, V_s, R \vdash o \rightsquigarrow \ell$ $H(\ell) = I'$	$H' = H, V_s' = V_s, R' = R$
`call` $o\ \ell_r$	$H, V_s, R \vdash o \rightsquigarrow \ell$ $H(\ell) = I'$	$H' = H, R' = R, V_s' = \ell_r @ V_s$
`jcc` $\kappa, o\ I_0$	$R(\mathit{flags}) \models \kappa$ $H, V_s, R \vdash o \rightsquigarrow \ell$ $H(\ell) = I'$	$H' = H, V_s' = V_s, R' = R$
`jcc` $\kappa, o\ I_0$	$R(\mathit{flags}) \not\models \kappa$	$H' = H, V_s' = V_s, R' = R, I' = I_0$
`cmpjcc` $o_1, o_2, \kappa, o_3\ I_0$	$(H, V_s, R, \texttt{cmp}\ o_1, o_2\ \texttt{jcc}\ \kappa, o_3\ I_0)$ $\mapsto^2 (H', V_s', R', I')$	
`pop` $d\ I'$	$V_s = v_1 @ V_{s0}$ $H, V_{s0}, R \vdash d(v_1) \rightsquigarrow H', V_s', R'$	
`push` $o\ I'$	$H, V_s, R \vdash o \rightsquigarrow v$	$H' = H, R' = R, V_s' = v @ V_s$
`ret` I_0	$V_s = \ell @ V_s'$ $H(\ell) = I'$	$H' = H, R' = R$
`salloc` $n\ I'$	$n = mW$	$H' = H, V_s' = \langle v_1, \ldots, v_m \rangle @ V_s$ $R' = R$, $v_1, \ldots, v_m$ *are arbitrary values*
`sfree` $n\ I'$	$V_s = V_1 @ V_s'$ $\lvert V_1 \rvert = n$	$H' = H, R' = R$
`malloc` $d, n\ I'$	$n = mW$ $\ell \notin \mathrm{dom}(H)$ $H_1 = H\{\ell \mapsto \langle v_1, \ldots, v_m \rangle\}$ $H_1, V_s, R \vdash d(\ell) \rightsquigarrow H', V_s', R'$	
`mallocarr` $d, n, o\ I'$	$\ell \notin \mathrm{dom}(H)$ $H, V_s, R \vdash o \rightsquigarrow v$ $H_1 = H\{\ell \mapsto \langle \underbrace{v, \ldots, v}_{n} \rangle\}$ $H_1, V_s, R \vdash d(\ell) \rightsquigarrow H', V_s', R'$	

Table 2.3: MiniTALT Abstract Machine Evaluation

Chapter 3

TALT-R: A Typed Assembly Language for Responsiveness

The central claim of this thesis is that static enforcement of *timing policies* using type systems is possible. In this chapter I begin to offer support for that claim by describing a TALT-like type system that allows a range of timing policies to be certified within the metalogical framework described in Chapter 2. Karl Crary has suggested [11] that my work be considered the next version of TALT and essentially replace it; however, until that happens it will be useful to distinguish Crary's TALT from my own. Therefore, for the time being I call my assembly language TALT-R (for "Responsiveness").

Like TALT, TALT-R is actually a number of different, but closely related, languages that play different roles in the certified code process. The three most prominent are:

- TALT-R itself, which is a Curry-style type system in which type-checking is presumed undecidable. TALT-R is analogous to Crary's TALT, the language for which Crary and Sarkar directly proved a safety metatheorem. I have not undertaken a formal safety proof for TALT-R, but I am confident in the conjecture that such a proof would be a mostly straightforward extension of the proof for TALT.

- XTALT-R (analogous to XTALT), which is an explicitly-typed version of TALT-R for which type-checking is tractable. The certificate for a TALT-R program is an XTALT-R program.

- EXTALT-R (analogous to EXTALT) which is the external language of the TALT-R assembler (and therefore also the direct target of high-level language compilers using TALT-R for certification).

However, as I explained for their TALT analogues in Chapter 2, none of these three is a particularly good language to use when formally describing a compiler as I must do in this thesis. Therefore, this chapter introduces the core language MiniTALT-R, which extends the MiniTALT of Chapter 2 in just the same way that TALT-R extends TALT.

To make the discussion of concrete, I start by describing a specific timing policy that will serve as the main motivating example for the design of TALT-R (this chapter and Chapter 4) and my compiler implementation (Chapters 5, 6 and 7). In Chapter 8 I will finally leave this particular example behind and explore the range of policies that can be certified using TALT-R.

3.1 A Responsiveness Policy

As discussed in Section 1.2, *periodic yielding* is an important timing requirement that must be enforced by operating systems and other kinds of supervisors in multithreaded applications. A process that fails to return control to the scheduler promptly can disrupt the behavior of other processes or bring the entire system to a halt. It seems, therefore, that any proposal for static enforcement of timing policies as an alternative to dynamic enforcement by pre-emptive scheduling must address the issue of cooperation among user processes.

Because of the effects non-conformant programs can have on other processes, I choose to call this "cooperativeness" requirement ***responsiveness.*** I state it semi-formally as follows:

> *I assume there is some system-specific set of operations, the **yielding operations,** that certified programs are expected to perform with at least a certain frequency. In particular, I assume that, for some large integer Y chosen in advance, a certified program must never execute more than Y non-yielding instructions in a row.*

The specific set of yielding operations will vary between systems. In the archetypical example of an operating system, any system call that gives the kernel an opportunity to deschedule the user process will count as a yield. In other scenarios, such as user-level thread schedulers, application plugin frameworks, or mobile code host environments, the interface presented to untrusted code will be application-defined and so will the designation of some of the available procedures as "yielding."

For the purposes of the exposition in this thesis, I will assume there is exactly one yielding operation, which I simply call "yield". This procedure is called from a TALT-R program by a new `yield` instruction, implemented as a function call. The type system of TALT-R will therefore be designed to enforce the simple policy that no more than Y instructions are ever executed between two successive `yield`'s.

3.2 MiniTALT-R

The remainder of this chapter describes the core language MiniTALT-R, which extends the MiniTALT of Chapter 2. Because the two languages, abstract machines and type systems are so closely related, my presentation in this chapter will cover only the differences between them, *i.e.* the extensions and refinements that distinguish MiniTALT-R from MiniTALT. A complete formal definition of MiniTALT-R is given in Appendix B.

Since the design of this new language is motivated by the desire to certify programs with respect to a particular safety policy, I begin with the components of the design that implement that policy: the extension of the instruction set to include a yielding operation, and the refinement of the operational semantics that makes it unsafe to run too long without yielding. After laying this groundwork I will explain the extensions and refinements to the type assignment system that are needed to guarantee programs are safe.

3.2.1 New Instructions

The MiniTALT-R programming language extends that of MiniTALT with just two new instructions. Formally, the grammar for instruction sequences gets two new productions:

$$\textit{Instruction Sequences} \qquad I ::= \cdots \mid \texttt{subjae}\ r_d, o_1, o_2, o_3\ I \mid \texttt{yield}\ I$$

The `yield` instruction is the key addition: it is this instruction that must be executed with at least a certain frequency under the new safety policy. The other new instruction is `subjae` ("subtract and jump if above or equal"), a compound instruction comprised of a comparison and a conditional jump. The instruction sequence `subjae` $\texttt{r}_d, o_1, o_2, o_3\ I$ has the same operational behavior as ($\texttt{sub}\ \texttt{r}_d, o_1, o_2; \texttt{jcc}\ \texttt{ae}, o_3; I$): it subtracts o_2 from o_1, stores the result in $\texttt{r}_d$, and jumps to o_3 if o_1 is greater than or equal to o_2 (interpreting these values as unsigned integers). A special typing rule reflects the result of the conditional jump into the type system. In this sense, `subjae` is related to the `cmpjcc` instruction inherited from TALT. The addition of `subjae` to the language may seem arbitrary at this point, but its usefulness in producing safe-but-efficient programs will become clear later on.

3.2.2 The MiniTALT-R Abstract Machine

As discussed earlier (Section 2.1), the operational semantics with respect to which the type safety theorem is proved comprises the safety policy in a foundational certification system. Thus, to certify that programs yield at least once every Y instructions, I must provide an operational semantics in which any valid execution necessarily obeys this policy and a type system that is sound with respect to that semantics.

The dynamic semantics of MiniTALT-R are defined in terms of the MiniTALT-R abstract machine, which is a refinement of the MiniTALT abstract machine. The key change is the addition of a ***virtual clock register*** to the register file:

$$\textit{Register files} \quad R ::= \{\ldots, \texttt{ck} = n\} \qquad (n \geq 0)$$

The value of the virtual clock is a nonnegative integer. Any non-yielding instruction executed by the MiniTALT-R abstract machine decrements the virtual clock, while the yielding instruction sets the virtual clock to Y.

These properties of the virtual clock capture the essence of the responsiveness policy. Since the virtual clock can never be negative, any machine state in which the next instruction is non-yielding but the virtual clock is zero is stuck (and thus forbidden). Therefore, any *safe* execution starting from a state where $\texttt{ck} = n$ must perform at least one `yield` in its first $n + 1$ steps. Since the `yield` instruction sets the clock to Y, it follows that successive yields must occur no more than Y instructions apart.

This method of instruction counting is not new: Necula and Lee proposed the use of a virtual clock for proof-carrying code [51], and Crary and Weirich used one in their languages LXres and TALres [18]. Unlike these others, however, I am not attempting to bound total running time; I am only interested in bounding the time until the next yield.

3.3 Static Semantics

The type system of MiniTALT-R, like its instruction set and abstract machine, is arrived at by means of a few changes to its MiniTALT counterpart. The modifications to the type system are shown in Figure 3.1.

The first and most important syntactic change is the inclusion of a *clock term* in every register file type. The clock term assignment $\texttt{ck} : t$, where t is a static term of type `N`, asserts that the value of the virtual clock is *at least* the number denoted by t. The kind `N` is extended to include formal sums of the form $t_1 + t_2$; the members of this expanded kind `N` will be called ***constraint terms***.

Static Terms	c, t, τ, x	::=	$\cdots$	
		$\mid$	$\varphi \Rightarrow \tau \mid \mathcal{S}(t)$	
		$\mid$	$\overline{n}$	*(replaces B)*
		$\mid$	$t_1 + t_2$	
Constraint Formulas	φ	::=	$t_1 \leq t_2 \mid t_1 = t_2$	
Static Contexts	Δ	::=	$\cdot \mid \Delta, \alpha{:}K \mid \Delta, \varphi\ \mathsf{true}$	
Register File Types	Γ	::=	$\{\mathtt{eax}{:}\tau_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{bp}}, \mathtt{esp}{:}\tau, \mathtt{ck}{:}t\}$	

Figure 3.1: MiniTALT-R Type System Syntax

Another important addition is the class of ***constraint formulas***, which are assertions of equality or non-strict inequality between constraint terms.[1] The constraint formulas and their role in typing will be discussed below (Section 3.3.1) and the proof theory of the logic they comprise will be explored in depth in Chapter 4.

MiniTALT-R adds only one form of type to MiniTALT: the ***guarded type*** $\varphi \Rightarrow \tau$ describes values which may be given type τ if the formula φ is satisfied. As a syntactic convenience, I also introduce the notation $\mathcal{S}(t)$ as a synonym for $\mathtt{set}_{=}(t)$ — by convention, I write $\mathtt{set}_{=}(t)$ when using this type in the implementation of disjoint union types or array bounds checks (*i.e.,* the purposes it serves in plain TALT), and $\mathcal{S}(t)$ when it occurs in the typing of time-keeping idioms.

3.3.1 The Constraint Subsystem

The purpose of the constraint terms and formulas is to allow the type system to reason about the time remaining before the next yield instruction must be performed. This constraint logic is largely separable from the rest of the type system; in fact, there is a certain degree of flexibility in its design. The version I will describe in this proposal is engineered mostly for clarity of presentation.

As mentioned above, the constraint terms include the natural numbers (written $\overline{n}$, where $n \geq 0$) and are closed under addition; the language of formulas contains equality ($t_1 = t_2$) and ordering ($t_1 \leq t_2$) on constraint terms. It would be a simple matter to add propositional connectives ($\wedge$, $\vee$, $\supset$, $\bot$) to the constraint logic; however, there is surprisingly little need for them to enforce the simple responsiveness policy of TALT-R. I therefore leave them out of this presentation for simplicity.

Judgment	Meaning
$\Delta \vdash \varphi\ \mathsf{prop}$	φ is a well-formed constraint formula.
$\Delta \vdash \varphi\ \mathsf{true}$	The constraint φ is true.

Table 3.1: New Typing Judgments of MiniTALT-R

The type system of MiniTALT-R defines two judgment forms not present in MiniTALT. Both of these have to do with constraint formulas; their meanings are summarized in Table 3.1. The judgment $\Delta \vdash \varphi\ \mathsf{prop}$ means that in context Δ, the formula φ is well-formed. The rules for this judgment (along with two relevant kinding rules) are given in Figure 3.2. Note that a formula need not be "true" in order to be well-formed.

The notion of "truth" for constraint formulas is captured by the other new judgment form: the judgment $\Delta \vdash \varphi\ \mathsf{true}$ means that the truth of the formula φ follows from the assumptions in Δ. Note that according to Figure 3.1, Δ may contain both kinding assumptions of the form $\alpha{:}K$ and hypotheses of the form $\varphi\ \mathsf{true}$. The inference rules defining the truth judgment are given in Chapter 4 along with extensive discussion of the rationale for their design and of their proof-

[1] In my thesis proposal [69], the constraint formulas were themselves static terms of a particular kind. Although aesthetically tempting, this formulation proved not to scale soundly to the full implemented TALT.

$$\frac{(n \geq 0)}{\Delta \vdash \overline{n} : \mathsf{N}} \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 + t_2 : \mathsf{N}} \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 \leq t_2 \ \mathsf{prop}} \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 = t_2 : \ \mathsf{prop}}$$

Figure 3.2: Formation Rules for Constraints

theoretic consequences. They capture a useful, if naïve, theory of addition of natural numbers that allows all of the idioms discussed in the remainder of this thesis to be certified. (Impatient readers can find the rules in Figure 4.1.)

3.3.2 The Virtual Clock

Accounting for the virtual clock in the type system is a fairly straightforward matter. TALT-R's treatment of the clock is more or less analogous to that of TALres. Register file types, in addition to giving types for the machine's general-purpose registers and the stack, give a constraint term that conservatively approximates the value of the virtual clock. That is to say, if $\Delta; \Psi; \Gamma \vdash I$, then the instruction sequence I may safely be executed if the value of the virtual clock is at least (the number denoted by) $\Gamma(\mathtt{ck})$. Typing rules for mundane instructions involving no control flow reflect this with some simple bookkeeping. For example, the typing rule for the `add` instruction is:

$$\frac{\begin{array}{c}\Delta; \Psi; \Gamma \vdash o_1 : \mathsf{B4} \quad \Delta; \Psi; \Gamma \vdash o_2 : \mathsf{B4} \\ \Delta; \Psi; \Gamma \vdash d : \mathsf{B4} \rightarrow \Gamma' \quad \Delta; \Psi; \Gamma'\{\mathtt{ck}{:}t\} \vdash I\end{array}}{\Delta; \Psi; \Gamma \vdash \mathtt{add}\ d, o_1, o_2; I} \ (\Gamma(\mathtt{ck}) = \overline{1} + t)$$

Note the two differences from the analogous rule in TALT: First, the side condition requires that the clock term $\Gamma(\mathtt{ck})$ have the form $\overline{1} + t$ for some term t, since it is a type error to perform an `add` when the clock is zero. Second, the final premise requires that the continuation I be well-typed assuming only t on the virtual clock, since the `add` will have used one time unit.

Correspondingly, a code pointer of type $\Gamma' \rightarrow 0$ is safe to jump to only if the virtual clock is at least $\Gamma'(\mathtt{ck})$ *after the jump*; that is, the clock must read at least one more than $\Gamma'(\mathtt{ck})$ in order for the jump instruction itself to be safe:

$$\frac{\Delta; \Psi; \Gamma \vdash o : (\Gamma\{\mathtt{ck}{:}t\}) \rightarrow 0}{\Delta; \Psi; \Gamma \vdash \mathtt{jmp}\ o; I} \ (\Gamma(\mathtt{ck}) = \overline{1} + t)$$

The `yield` instruction may be performed at any time, and resets the virtual clock to Y:

$$\frac{\Delta; \Psi; \Gamma\{\mathtt{ck}{:}\overline{Y}\} \vdash I}{\Delta; \Psi; \Gamma \vdash \mathtt{yield}; I}$$

The three rules just presented preserve or improve the accuracy of the constraint term $\Gamma(\mathtt{ck})$ with respect to the actual value of `ck`. In general, though, $\Gamma(\mathtt{ck})$ is an inexact approximation of the virtual clock. The imprecision is due to TALT-R's rule for register file subtyping, which allows the constraint term assigned to `ck` to vary:

$$\frac{\Delta \vdash t' \leq t \ \mathsf{true} \quad \Delta \vdash \tau \leq \tau' \quad \Delta \vdash \tau_i \leq \tau_i' \text{ for } 1 \leq i \leq N}{\Delta \vdash \{\mathtt{r1}{:}\tau_1, \ldots, \mathtt{rN}{:}\tau_N, \mathtt{sp}{:}\tau, \mathtt{ck}{:}t\} \leq \{\mathtt{r1}{:}\tau_1', \ldots, \mathtt{rN}{:}\tau_N', \mathtt{sp}{:}\tau', \mathtt{ck}{:}t'\}}$$

According to this rule, a register file type where the virtual clock reads t can be a subtype of one where it reads t' if the formula $t' \leq t$ can be proved in the constraint logic. Intuitively, the register file type on the left specifies that the value of the virtual clock is *at least* t; if $t' \leq t$, then anything that is at least t will also be at least t'. The register file type specifying $\mathtt{ck}{:}t$ is a stronger requirement on the state of the machine, consistent with the usual meaning of subtyping.

Because the register file subtyping rule involves reasoning about the virtual clock, the subtyping rule for arrow types and the subsumption rule for instruction sequences take on additional meaning in TALT-R as well. To be specific, the subsumption rule (inherited unchanged from TALT):

$$\frac{\Delta; \Psi; \Gamma' \vdash I \quad \Delta \vdash \Gamma \leq \Gamma'}{\Delta; \Psi; \Gamma \vdash I}$$

now allows an instruction sequence to "forget" about some of the remaining ticks on the virtual clock. The subtyping rule for code pointer types $\Gamma \to 0$ is contravariant in Γ as always:

$$\frac{\Delta \vdash \Gamma' \leq \Gamma}{\Delta \vdash \Gamma \to 0 \leq \Gamma' \to 0}$$

Coupled with the register file subtyping rule, this means that a pointer to an instruction sequence expecting t on the clock may be used in place of one expecting t' if $t \leq t'$. Intuitively speaking, this is because any subsequent jump to that pointer will have to provide a clock of at least t', which will be at least enough since the instruction sequence requires only t.

I pause here to note that if the premise $\Delta \vdash t' \leq t$ true in the register file subtyping rule were replaced by $\Delta \vdash t \leq t'$ true, then the sense of the approximation of $R(\mathtt{ck})$ by $\Gamma(\mathtt{ck})$ would be reversed. That is, a register file type Γ would describe machine states in which the value of the virtual clock was *at most* $\Gamma(\mathtt{ck})$. If the premise were replaced by $\Delta \vdash t' = t$ true, then the static term in the register file type would always correspond exactly to the value of the clock. I will discuss some applications of these variants of the system in Chapter 8.

3.3.3 Guarded and Singleton Types

There are two forms of type in TALT-R that need to be discussed here: the *singleton types* ($\mathcal{S}(t)$), which are really the same as the singletons written $\mathtt{set}_{=}(t)$ in TALT but have been endowed with some new capabilities, and the *guarded types* ($\varphi \Rightarrow \tau$), which are new. The intuitive meanings of these types are simple, but their usefulness may not be obvious until I discuss yield-placement strategies later on in the thesis. Basically, I will use them to construct more precise types for functions than would otherwise be possible, so that the constraint reasoning built into the type system can recognize more efficient code as safe. They are not strictly necessary in the sense that it is possible to write a compiler whose output is well-typed without them, but they deliver significant performance benefits for a reasonably small metatheoretic investment.

A guarded type $\varphi \Rightarrow \tau$ describes values that may be used at type τ only if the formula φ is true. This is captured by a subtyping rule:

$$\frac{\Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash \varphi \text{ true}}{\Delta \vdash \varphi \Rightarrow \tau \leq \tau}$$

Using this rule, an operand o of type $\varphi \Rightarrow \tau$ may be promoted to type τ if φ is provable in the constraint logic. If the truth of φ cannot be derived, then no interesting use can be made of o.

$$\frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \mathcal{S}(t) : \mathsf{T}W} \qquad \frac{(0 \leq n \leq 2^{8W} - 1)}{\Delta \vdash n : \mathcal{S}(\overline{n})} \qquad \frac{\Delta \vdash t_1 = t_2\ \mathsf{true}}{\Delta \vdash \mathcal{S}(t_1) \leq \mathcal{S}(t_2)} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \mathcal{S}(t) \leq \mathsf{B}W}$$

Figure 3.3: Elementary Rules for Singletons

The introduction mechanism for guarded types differs slightly between TALT-R and MiniTALT-R. In both systems, there is a guarded type introduction rule for values:

$$\frac{(\Delta, \varphi\ \mathsf{true}); \Psi \vdash v : \tau}{\Delta; \Psi \vdash v : \varphi \Rightarrow \tau}$$

According to this rule, to conclude that v has type $\varphi \Rightarrow \tau$ it suffices to show that v has type τ, under the assumption that φ is true. Importantly, the derivation of $v : \tau$ may depend on the hypothesis φ true; v need not be well-typed at all without it. It is worth noticing that guarded types bear a certain similarity to $\forall$-types: both are introduced by typing a value under some new assumption, and both are eliminated by subtyping rules that "validate" the assumption.

It is very important that one be able to give guarded types to code pointers—more important, in fact, than for any other kind of value. In TALT-R, blocks of code are simply values, and so the above rule is sufficient. In MiniTALT-R, instruction sequences are treated specially, so an additional guarded type introduction rule for blocks is required:

$$\frac{\Psi; (\Delta, \varphi\ \mathsf{true}) \vdash I : \tau\ \mathsf{block}}{\Psi; \Delta \vdash I : \varphi \Rightarrow \tau\ \mathsf{block}}$$

This rule is analogous to the rule for values, and states that one may give a guarded type to (the address of) a block of instructions that is well-typed under the assumption that the guard is true.

Singleton types in TALT and TALT-R play a role similar to that of singletons in DTAL [72] and LTT [16]. In DTAL one writes a singleton type as $\mathrm{int}(x)$, where x is an "index expression"; in LTT one writes $S_{Int}(M)$, where M is the proof-language representation of an integer. The TALT-R type $\mathcal{S}(t)$ is well-formed when t is a well-formed constraint term (*i.e.*, it has kind N), and contains at most one value: the word-sized unsigned binary representation of the natural number denoted by t. (If the meaning of t is outside the representable range, then $\mathcal{S}(t)$ is an empty type.) The most elementary rules for singleton types are shown in Figure 3.3.

In DTAL and LTT, programs may perform arithmetic on values of singleton type, and the type system tracks this manipulation symbolically by giving an appropriate singleton type to the result. As it happens, the particular use I have in mind for singleton types is to describe a counter which is repeatedly decremented until it reaches zero. Consequently, the only form of arithmetic I will need for singletons is a combined subtract-and-conditional-jump operation; it is for this reason that the `subjae` instruction is included in TALT-R. As I have already mentioned, the instruction sequence ($\texttt{subjae}\ \mathsf{r}_d, o_1, o_2, o_3\ I$) subtracts the value of o_2 from o_1 and stores the result in r_d; if this result is greater than or equal to zero, control jumps to the address in o_3; otherwise, execution continues with I. The `subjae` instruction has a special singleton-aware typing rule:

$$\frac{\begin{array}{c}\Delta; \Psi; \Gamma\{\mathsf{r}_d{:}\mathsf{B}W, \mathsf{ck}{:}t\} \vdash I \\ \Delta; \Psi; \Gamma \vdash o_3 : \forall a{:}\mathsf{N}.(u = v + a) \Rightarrow \Gamma\{\mathsf{r}_d{:}\mathcal{S}(a), \mathsf{ck}{:}t\} \rightarrow 0 \\ \Delta; \Psi; \Gamma \vdash o_1 : \mathcal{S}(u) \quad \Delta; \Psi; \Gamma \vdash o_2 : \mathcal{S}(v) \quad (\Gamma(\mathsf{ck}) = 2 + t)\end{array}}{\Delta; \Psi; \Gamma \vdash \texttt{subjae}\ \mathsf{r}_d, o_1, o_2, o_3\ I}$$

This rule shows how to type a `subjae` instruction when the two operands to be subtracted have singleton types $\mathcal{S}(u)$ and $\mathcal{S}(v)$ respectively. Notice the different typing conditions associated with the two possible outcomes of the conditional jump. If the branch is taken, then the result is non-negative and hence the subtraction falls within the domain of natural number arithmetic; the target of the jump is therefore allowed to assume that the result is some natural number a such that the larger operand is equal to the sum of a and the smaller operand. If the branch is not taken, however, the result of the subtraction is negative and cannot be reasoned about in my theory of natural numbers; hence the instruction sequence I must be well-formed assuming only that the destination register contains an integer. Finally, note that the virtual clock is decremented by two instead of by one; this is because `subjae` is implemented by a sequence of two instructions on a concrete IA-32 machine.

3.3.4 Expanding Singleton Reasoning

Although `subjae` is the only singleton instruction required for the compilation strategies I describe in this thesis, and the only one supported by my implementation of TALT-R, there are a few other singleton-related instructions and typing rules that are sound in principle and for which support could easily be added.

Checked Addition It may be desirable to include a singleton-aware `add` instruction. The main difficulty here is that the TALT-R constraint logic is concerned with (arbitrary) natural numbers whereas arithmetic in assembly language is performed modulo 2^{8W}. Expressing the results of modular arithmetic in the constraint logic presents two difficulties: first, it requires adding multiplication to the logic; second, it does not allow one to reason about inequalities as easily. A more attractive solution is for the singleton addition operation to be a "double" instruction like `subjae`, so that it automatically detects when its result is inconsistent with natural number arithmetic. Just as a subtraction can be reflected in the logic as long as the result is not negative, an addition can be accounted for as long as it does not overflow. The appropriate compound instruction for singleton addition is therefore `addjnc`, or "add and jump if no carry." The syntax and typing rule are analogous to `subjae` (but the typing premise for the jump target is simpler):

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma\{\mathsf{r}_d{:}\mathsf{B}W, \mathsf{ck}{:}t\} \vdash I \\ \Delta;\Psi;\Gamma \vdash o_3 : \Gamma\{\mathsf{r}_d{:}\mathcal{S}(u+v), \mathsf{ck}{:}t\} \to 0 \\ \Delta;\Psi;\Gamma \vdash o_1 : \mathcal{S}(u) \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathcal{S}(v) \quad (\Gamma(\mathsf{ck}) = 2 + t)\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{addjnc}\ \mathsf{r}_d, o_1, o_2, o_3\ I}$$

Inverted Checked Arithmetic In the `subjae` and `addjnc` instructions, the conditional branch is taken if the result of the arithmetic operation can be given a useful singleton type and falls through if it cannot. This is convenient for the particular idiom I have in mind for `subjae` (to be covered in Chapter 6), but the system would be more symmetrical if there were alternative checked singleton arithmetic instructions with the branch conditions reversed. There is no difficulty in principle with adding `subjb` ("subtract and jump if below") and `addjc` ("add and jump if carry") to TALT-R. The following typing rules soundly describe their semantics:

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o_3 : \Gamma\{\mathsf{r}_d{:}\mathsf{B}W, \mathsf{ck}{:}t\} \to 0 \\ (\Delta, a{:}\mathsf{N}, (u = v + a)\ \mathsf{true});\Psi;\Gamma\{\mathsf{r}_d{:}\mathcal{S}(a), \mathsf{ck}{:}t\} \vdash I \\ \Delta;\Psi;\Gamma \vdash o_1 : \mathcal{S}(u) \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathcal{S}(v) \quad (\Gamma(\mathsf{ck}) = 2 + t)\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{subjb}\ \mathsf{r}_d, o_1, o_2, o_3\ I}$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma\{\mathtt{r}_d{:}\mathcal{S}(u+v),\mathtt{ck}{:}t\}\vdash I\\ \Delta;\Psi;\Gamma\vdash o_3:\Gamma\{\mathtt{r}_d{:}\mathsf{B}W,\mathtt{ck}{:}t\}\to 0\\ \Delta;\Psi;\Gamma\vdash o_1:\mathcal{S}(u)\quad \Delta;\Psi;\Gamma\vdash o_2:\mathcal{S}(v)\quad (\Gamma(\mathtt{ck})=2+t)\end{array}}{\Delta;\Psi;\Gamma\vdash \mathtt{addjc}\ \mathtt{r}_d,o_1,o_2,o_3\ I}$$

In fact, it is important for performance reasons to allow the programmer or compiler to choose the sense of conditional jumps. Most IA-32 processors use a static branch prediction heuristic which assumes (until more information is available) that backward conditional jumps are taken and forward conditional jumps are not taken. In order to produce fast code for superscalar architectures with deep pipelines, programmers are encouraged to arrange their code so that these assumptions are likely to be accurate [39].

Unchecked Arithmetic When I discuss the application of TALT-R to different safety policies in Chapter 6, I will encounter situations where a subtraction of two singleton values is required *and* it is statically known which of the two quantities is larger. In this case the `jae` part of the `subjae` instruction is unnecessary (and its presence obnoxious) because that branch will never be taken. The following rule for unchecked singleton subtraction allows the operation to proceed under those conditions:

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma\vdash o_1:\mathcal{S}(u+v)\quad \Delta;\Psi;\Gamma\vdash o_2:\mathcal{S}(u)\\ \Delta;\Psi;\Gamma'\{\mathtt{ck}{:}t\}\vdash I\quad \Delta;\Psi;\Gamma\vdash d:\mathcal{S}(v)\to\Gamma'\quad (\Gamma(\mathtt{ck})=1+t)\end{array}}{\Delta;\Psi;\Gamma\vdash \mathtt{sub}\ d,o_1,o_2\ I}$$

A rule for unchecked singleton addition, analogous to this one, is presumably sound but seems less likely to be helpful in practice.

Ordering and Addition Another reasonable typing rule that could be added to TALT-R to enrich the capabilities of its singleton types, but for which I have not found an immediate need, is the following subtyping rule:

$$\frac{\Delta\vdash t\le u\ \mathsf{true}}{\Delta\vdash \mathcal{S}(u)\le \exists a{:}\mathsf{N}.\mathcal{S}(t+a)}$$

It states that if $t \le u$, then the number u can be thought of as the sum of t and an unknown natural number.

3.4 Certification and Verification

The process of producing and verifying certified binaries using TALT-R is analogous to the process for TALT described in Chapter 2. Specifically, a compiler wishing to target TALT-R outputs programs in EXTALT-R, a user-friendly explicitly-typed assembly language. The TALT-R assembler transforms an EXTALT-R program into a TBF file (see Section 2.1.3, Figure 2.2) by translating the assembly instructions into binary machine code and generating a certificate. The certificate is (the LF representation of) an XTALT-R program. The consumer-side certificate verifier is analogous to the one described for TALT.

Because of the strength of the analogies between the TALT and TALT-R families of languages, I will sometimes refer to XTALT and XTALT-R as "the 'X' languages" and to EXTALT and EXTALT-R as "the 'EX' languages."

3.4.1 XTALT-R

The XTALT-R language, like XTALT, is designed to be as easy as possible to type-check. This means removing all the ambiguity and implicitness that make type-checking for TALT-R itself (presumably) impossible. Like the "Mini" languages detailed in this thesis, XTALT and XTALT-R divide the contents of a "program", which in TALT and TALT-R is just a single large value, into labeled blocks; unlike the "Mini" languages, the "X" languages require the programmer to specify a type for each label. This essentially identifies the memory typing (Ψ) under which the whole program is supposed to be well-typed, eliminating an ambiguity that would be difficult to resolve by inference.

An even greater source of difficulty in type-checking is the richness of the theory of subtyping in TALT and TALT-R. (Most of the difficulties here are reflected in the "Mini" languages.) To avoid the need to decide the (presumably) undecidable subtyping relation, the "X" languages replace the subtyping judgment $\Delta \vdash \tau_1 \leq \tau_2$ by a calculus of *coercions*. Essentially, a coercion is a reified subtyping derivation: the coercion typing assertion $\Delta \vdash q : \tau_1 \leq \tau_2$ means that the coercion q represents a proof that τ_1 is a subtype of τ_2. The subsumption rule for operands in TALT or TALT-R,

$$\frac{\Delta; \Psi; \Gamma \vdash o : \tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta; \Psi; \Gamma \vdash o : \tau}$$

is replaced in the respective "X" languages by *coercion application*, written $@q\,o$ and having the typing rule:

$$\frac{\Delta; \Psi; \Gamma \vdash o : \tau' \quad \Delta \vdash q : \tau' \leq \tau}{\Delta; \Psi; \Gamma \vdash @q\,o : \tau}$$

The coercions themselves, in turn, correspond almost exactly to subtyping derivations. There is a form of coercion for each subtyping rule in the underlying theory, so that $\Delta \vdash \tau \leq \tau'$ is derivable if and only if there is a coercion q such that $\Delta \vdash q : \tau \leq \tau'$. For instance, the rule stating that any type is a subtype of nonsense (of the appropriate size) corresponds to a coercion called `forget`. That is:

$$\frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau \leq \mathsf{ns}i} \qquad \textit{corresponds to} \qquad \frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \texttt{forget} : \tau \leq \mathsf{ns}i}$$

XTALT-R must extend XTALT with coercion forms for all of the new subtyping rules TALT-R adds to TALT. The ones that require the most novelty are the rules with constraint-truth premises, like the guard satisfaction rule:

$$\frac{\Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash \varphi\ \mathsf{true}}{\Delta \vdash (\varphi \Rightarrow \tau) \leq \tau}$$

In order for a typechecker to accept a coercion witnessing this relation as well-formed, it must be evident that φ is true. The easiest way to achieve this is to require the coercion itself to provide the evidence; in other words, it must contain a *proof* of φ. Thus, the coercion has the form $\texttt{satisfy}\,\pi$, where π is a proof term:

$$\frac{\Delta \vdash \pi : \varphi}{\Delta \vdash \texttt{satisfy}\,\pi : (\varphi \Rightarrow \tau) \leq \tau}$$

Proof terms reify truth derivations in exactly the same way that coercions reify subtyping derivations. (The truth derivations themselves are discussed in Chapter 4.) They appear in the coercion forms associated with all TALT-R subtyping rules that have constraint-truth premises.

3.4.2 EXTALT-R

The "EX" languages, the explicitly-typed languages generated by compilers and processed by the certifying assemblers, are the most concrete, public and visible incarnations of TALT and TALT-R. They are therefore the variants for which human-readability and -writability are the most important. This creates tension between the desire for ease of use on the one hand, and the fact that removing almost any of the annotations in the "X" languages leads to undecidability on the other.

EXTALT-R follows EXTALT in retaining the calculus of coercions used in the "X" languages to circumvent the presumed undecidability of subtyping. The one source of nonuniformity is the need for proof terms in XTALT-R to reify constraint truth derivations. EXTALT-R does not use proof terms, for two reasons: First, since proof terms are a new syntactic class, distinct from type constructors and coercions, adding them to EXTALT-R would represent a significant increase in syntactic complexity compared to EXTALT. Second, the very concept of proof terms being unfamiliar to most programmers other than the few who are type theory experts, requiring them to appear in programs would greatly increase the steepness of the learning curve for EXTALT-R programming or, more importantly, certifying compiler development.[2] Third, truth deriviations are ubiquitous in the typings of even the most elementary TALT-R programs, because the `ck` terms in register file types must very often be rewritten (using an equality formula and the register file subsumption rule) in order to match the form required by the omnipresent side conditions in instruction typing rules.

The purpose of these side conditions is to capture the idea that the virtual clock is decremented for every instruction. The EXTALT-R assembler performs this symbolic decrementation automatically, using a very simple heuristic. When the type-checker encounters an instruction requiring k "ticks" of the virtual clock and the current register file type is Γ, it updates $\Gamma(\texttt{ck})$ to $\texttt{dec}(\Gamma(\texttt{ck}), k)$ before moving to the next instruction as long as the latter is well-defined according these rules:

$$\texttt{dec}(\overline{n}, k) = \overline{n-k} \quad \textit{if } n \geq k$$
$$\texttt{dec}(t + t', k) = \texttt{dec}(t, k) + t'$$

If application of these rules fails, the assembler gives up and reports a type error.

The other constructs that demand proof terms are subtyping judgments, most commonly the register-file subtyping that must be checked for jump operands and the GUARD-ELIM rule reified as the `satisfy` coercion, discussed earlier. In each of these cases, the assembler does not require the program text to contain a proof term, but instead tries to construct one using a semi-decision procedure for the TALT-R constraint logic called *depth-limited semantic proof search* (DLP), discussed in the next chapter. In fact, the logic is decidable; however, DLP is sufficient for the purposes of my responsiveness-certifying compiler, and since I have not found a decision procedure as simple and efficient I have not attempted to implement anything more advanced. Future work on TALT-R may require replacing the existing constraint logic with something more heavyweight, in which case the certification process would presumably need to incorporate a serious theorem prover.

[2] Coercions, of the kind found in the "X" and "EX" languages, are also unfamiliar. Their presence in the language can be rationalized to the extent that they act more or less like functions that change the type of a value, like C-style type casts which everyone understands. On the other hand, the verbosity of the coercions that appear in most EXTALT programs indicates that the concise and user-friendly representation of the information they carry is a language design issue worthy of attention.

3.5 Chapter Summary

In this chapter, I stated a specific timing policy that applies to many real-world situations. Motivated by this policy, I presented most of the design of a type system to certify compliance with this policy (the remainder of the static semantics will be revealed in the next chapter).

The type system, TALT-R, is an extension of TALT with clock reasoning in the style of TALres and dependent types in the style of DTAL. I have described the salient features of its static semantics, including potential variations which, as I will show later on, open the door to certification of other interesting timing and resource control policies.

Chapter 4

The TALT-R Constraint Logic

One of the most important features of TALT-R that makes it possible to generate certifiable programs without inserting excessively many yields is its *constraint logic*. Through the mechanism of guarded types, discussed in Chapter 3, the typing of a program can depend on the "truth" of some *constraint formulas*; this feature of the type system allows portions of programs to be given types that describe their clock behavior more precisely than would otherwise be possible, which in turn amounts to the ability to type programs whose clock behavior requires more subtle justification than would otherwise be allowed.

In order to define a type system that has this built-in constraint logic, and to prove theorems about that system formally, it is necessary to give a formal definition for the logic itself. This involves not only specifying the "language" of formulas upon which typing can depend, but also defining precisely what it means for a formula to be true. In the design of TALT-R it was critical to find a balance between simplicity and power: the simpler the logic, the less effort a formal safety proof would require, but a certain amount of proving power was necessary in order to type interesting programs.[1] My goal, therefore, was to find the simplest possible logic that could accomodate my compilation strategy. The results of that exercise make up this chapter. First I set down the definition of the TALT-R constraint logic, after first discussing some general considerations that influenced its design. The remainder of the chapter investigates the metatheory of the logic more deeply, to better understand its algorithmic properties and characterize its power.

4.1 The Logic

4.1.1 Terms and Formulas

The TALT-R constraint logic can be seen as a language of first-order predicates that lacks any logical connectives or quantifiers. The propositional connectives pose no theoretical difficulty, and I conjecture that some limited use of quantifiers would not either.

Formally, the terms of the logic are the static terms of kind N and the formulas are the constraint formulas as defined in Chapter 3. In this, chapter, though, I want to consider the logic in isolation from the rest of the type system. I therefore restrict my attention to "simple" terms and formulas, which are characterized by a very simple structure.

[1] As mentioned at the beginning of Chapter 3, I did not actually do a formal safety proof for TALT-R; however, it was still an important design criterion to make the execution of such a proof as straightforward as possible.

Definition 4.1 *The **simple terms** are the static terms generated by the grammar:*

$$t ::= a \mid \overline{n} \mid t_1 + t_2.$$

*A **simple formula** is a constraint formula containing only simple terms.*
*A **pre-simple context** is one that contains no kinding assumptions $a{:}K$ with $K \neq \mathsf{N}$.*
*A **simple context** is a pre-simple context in which all the constraint terms that appear are simple.*

(Alternatively, the simple terms are the β-normal terms t for which there exists a pre-simple context Δ such that $\Delta \vdash t : \mathsf{N}$.) Beginning with Section 4.1.2, I shall tacitly assume that all contexts, terms and formulas encountered in the remainder of this chapter are simple. Fortunately, the "simple" system and the unrestricted system are of comparable power: in particular, it can be shown that for any well-formed context Δ and well-formed formula φ there exist a simple context Δ' and simple formula φ' such that $\Delta \vdash \varphi$ true if and only if $\Delta' \vdash \varphi'$ true. Moreover, the simple versions can be computed from the originals, so decidability of the simple system implies decidability for well-formed judgments in the unrestricted system.[2]

4.1.2 Defining Truth

While discussing of TALT-R's static semantics in Chapter 3 I referred to, but did not define, the *constraint truth* judgment form $\Delta \vdash \varphi$ true, which appears as a premise in a few key rules of the static semantics. It is the business of this section to give a definition for this judgment; first, though, I will review some of the requirements for this definition.

That the truth judgment should be *sound* seems too obvious a necessity to need any discussion. After all, how can the type system be sound for programs as a whole if one of its judgments does not have the intended meaning? What makes this requirement interesting for TALT-R is that it is not enough for the truth judgment merely to *be* sound; I need to be able to *prove* its soundness *in Twelf*. As I shall argue, this rules out simple "denotational" definitions of truth such as the one proposed for integer constraints in DTAL [72]; instead, I will define truth "syntactically", using a carefully chosen set of axioms and inference rules.[3]

So, what does soundness of the constraint logic mean, and where does it come up in the Twelf proof of type safety? To answer these questions, consider the rule for register file subtyping, which has a truth premise (the one in the box):

$$\frac{\boxed{\Delta \vdash t' \leq t \text{ true}} \quad \Delta \vdash \tau \leq \tau' \quad \Delta \vdash \tau_i \leq \tau_i' \text{ for } 1 \leq i \leq N}{\Delta \vdash \{\mathsf{eax}{:}\tau_{\mathsf{ax}}, \ldots, \mathsf{ebp}{:}\tau_{\mathsf{bp}}, \mathsf{esp}{:}\tau_{\mathsf{sp}}, \mathsf{ck}{:}t\} \leq \{\mathsf{eax}{:}\tau'_{\mathsf{ax}}, \ldots, \mathsf{ebp}{:}\tau'_{\mathsf{bp}}, \mathsf{esp}{:}\tau'_{\mathsf{sp}}, \mathsf{ck}{:}t'\}}$$

Let Γ be the register file type on the left and Γ' be the one on the right. Then this rule permits Γ to be a subtype of Γ' only if the judgment $\Delta \vdash \Gamma'(\mathsf{ck}) \leq \Gamma(\mathsf{ck})$ true holds. The key lemma in which the effect of this premise is felt is the *register file subsumption lemma*:

Lemma 4.1 *If $\cdot \vdash \Gamma \leq \Gamma'$ and $\Psi \vdash R : \Gamma$, then $\Psi \vdash R : \Gamma'$.*

Here R is a register file. The judgment form $\Psi \vdash R : \Gamma$ (which we have not encountered before because register files appear only in the dynamic semantics) means that the register file R has type

[2] This is perhaps an unfair simplification of the TALT type theory as implemented; in particular, it depends on the fact that the static term language of MiniTALT-R is strongly β-normalizing. In fact, the static term language of TALT as formalized in LF is not normalizing, and the truth judgment in that system is undecidable.

[3] Readers familiar enough with Twelf to consider this decision a no-brainer may skip the next few paragraphs.

Γ (under heap assumptions Ψ); it requires, among other things, that the clock value in R is greater than or equal to the number denoted by $\Gamma(\mathsf{ck})$. Thus in order to prove this lemma we need to know, among other things, that if the number denoted by $\Gamma(\mathsf{ck})$ is less than or equal to m, and the truth judgment $\cdot \vdash \Gamma'(\mathsf{ck}) \leq \Gamma(\mathsf{ck})$ true holds, then the number denoted by $\Gamma'(\mathsf{ck})$ is also less than or equal to m. We get this from the *soundness lemma*:

Lemma 4.2 (Soundness of Truth) *If* $\cdot \vdash \overline{n'} \leq \overline{n}$ true, *then* $n' \leq n$.

This is the most important soundness result for the TALT-R constraint logic.

Observe that the soundness lemma, which we wish to be able to prove as a Twelf metatheorem, has an instance of the truth judgment on the left side of an implication. This means that if the definition of truth involves any universal quantification, the soundness lemma will not be a Π_2 sentence and hence will not be provable with Twelf. As a result we can forget about "semantic" definitions like the following:

> **Non-Definition.** $\Delta \vdash \varphi$ true iff the entailment it denotes holds over the natural numbers: that is, iff for any substitution of natural numbers for the constraint term variables declared in Δ such that the constraint hypotheses in Δ hold, φ holds.

The fact that this definition presupposes knowledge of the natural numbers, which are not built into Twelf, is annoying but it is not the issue. The real problem is the quantification "for any substitution. . . ," which cannot be encoded with an LF type.

The answer to this, of course, is to define the truth judgment the way anything else is defined in Twelf: inductively, as the least set of judgments closed under certain inference rules. This means that TALT-R's notion of "truth" is really more like "provability", with the rules for constructing proofs fixed in advance as part of the type system.

Formal theories of the natural numbers seem to come in two main varieties, neither of which is appropriate for TALT-R. The first variety comprises theories like LXres [18], whose power comes from a rich term language (featuring addition, multiplication and primitive recursion over natural numbers, as well introduction and elimination forms for some other types) and the associated theory of equality (which understands basic properties of addition and multiplication as well as $\beta\eta$-conversion). Such a theory is simple to describe, but does not support hypothetical reasoning, which TALT-R must if guarded types are to make sense.

The second variety takes the form of a set of *axioms* expressed in a *logic*; the logic is generally first-, second- or higher-order classical or intuitionistic predicate logic, and the axioms usually resemble those of Peano or Presburger arithmetic. These theories generally do support hypothetical reasoning, and as a group they are very powerful: in principle, one could use the Zermelo-Fränkel axioms for sets as the basis for such a theory and formalize all the mathematics one needed. Unfortunately, this class of theories is also unacceptable for TALT-R. That most of them are undecidable (Presburger arithmetic being the notable exception) is the least of their drawbacks: after all, typing in TALT is already presumed undecidable. Of much greater concern is the fact that proofs of even the most routine facts in these theories are large, difficult to construct, and even harder (for humans) to read. This would be crippling for TALT-R, since the certificate for a program must include proofs of all the constraint judgments on which its typing depends. These proofs would have to be generated during certification and transmitted over a network, and a large investment of time would be required to produce formal proofs of all the "lemmas" required. The complexity of the proofs that would have to be provided to the certificate generation algorithm would reduce the human-readability and -writability of EXTALT-R, the explicitly typed input language of

the assembler. Finally, and most damningly, the consistency of the axiomatization would have to be proven in Twelf; perhaps this could be managed, but to the best of my knowledge there is no published work on applying Twelf to consistency proofs in theories this complex.

The goal for TALT-R, then, was to devise a simple, though necessarily incomplete, axiomatization of natural number arithmetic subject to the following three considerations. First, the soundness of the theory must be provable as a Twelf metatheorem. Second, although the theory need not be complete in any formal sense, it must be "complete enough" to derive all of the judgments necessary to type the output of my compiler. Finally, the theory must be decidable, and furthermore it must be possible to produce proofs for derivable judgments automatically.

Note on Decidability

Typing in in the implicitly-typed TALT language is undecidable. A certificate for a TALT program, therefore, must contain at least enough information to convince the verifier that a typing derivation exists. So that the TALT assembler can produce such a certificate, the EXTALT input to the assembler must be heavily annotated. In particular, wherever the typing of the program depends on subsumption, the EXTALT program must contain a *coercion*, which is really a representation of a derivation of the necessary subtyping relationship. It might seem reasonable, therefore, to require an EXTALT-R program to include proofs of constraint formulas on which its typing depends.

Forcing the arithmetic proofs to be present in the EXTALT-R representation of a program has a serious drawback: it requires the person or program that generates the EXTALT-R program to produce the proofs. This seems like an excessive burden. Proofs in any theory of arithmetic are likely to be dense and hard for humans to read, which means that EXTALT-R programs whose typing depends on them will be very difficult to write or debug by hand. As a result of these considerations, I adopted the view that while the TALT-R type theory itself is defined in terms of a particular axiomatization of the constraint logic, from the point of view of a programmer or compiler writer generating EXTALT-R code, the structure of proofs is an implementation detail that does not need to be understood; furthermore, theorem proving in the TALT-R constraint logic is a task common to all producers of TALT-R programs, so it is the responsibility of the TALT-R implementor (that is, me) to provide a tool that does it. The theorem prover is integrated into the assembler, so EXTALT-R programs never need to contain proofs.

In principle, the assembler's theorem prover does not need to be complete, even with respect to the TALT-R constraint theory (which itself need not be complete with respect actual natural number arithmetic). However, it is highly desirable that the input language of the assembler have a concise and accessible definition so that programmers and compiler writers have some basis on which to predict whether their EXTALT-R code will be accepted or not. Since the assembler now includes the constraint prover, the definition of EXTALT-R must include a description of the set of constraint judgments it will be able to derive. In other words, if the theorem prover I build into the assembler decides a proper subset of TALT-R's theory of arithmetic, I must be able to give a concise definition of that decidable subset.

In fact, I have done a combination of these things: the theory presented in this chapter is decidable as-is, as I prove in Section 4.2, but I have not implemented a complete decision procedure. Instead, I describe a convenient metric of proof complexity that can be used to turn a sound and complete but unbounded proof search into an incomplete but terminating procedure, which I have implemented in the TALT-R assembler. I claim, and will demonstrate in Chapter 7, that a proof search bounded in this way is "complete enough" for the compilation strategy described there.

$$\frac{((\varphi\ \mathsf{true}) \in \Delta)}{\Delta \vdash \varphi\ \mathsf{true}} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash t = t\ \mathsf{true}} \qquad \frac{\Delta \vdash t_2 = t_1\ \mathsf{true}}{\Delta \vdash t_1 = t_2\ \mathsf{true}} \qquad \frac{\Delta \vdash t_1 = t_3\ \mathsf{true} \quad \Delta \vdash t_3 = t_2\ \mathsf{true}}{\Delta \vdash t_1 = t_2\ \mathsf{true}}$$

$$\frac{\Delta \vdash t_1 = t_1'\ \mathsf{true} \quad \Delta \vdash t_2 = t_2'\ \mathsf{true}}{\Delta \vdash t_1 + t_2 = t_1' + t_2'\ \mathsf{true}} \qquad \frac{}{\Delta \vdash \overline{m} + \overline{n} = \overline{m+n}\ \mathsf{true}} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \overline{0} + t = t\ \mathsf{true}}$$

$$\frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 + t_2 = t_2 + t_1\ \mathsf{true}} \qquad \frac{\Delta \vdash t_i : \mathsf{N}\ (\text{for } i = 1, 2, 3)}{\Delta \vdash (t_1 + t_2) + t_3 = t_1 + (t_2 + t_3)\ \mathsf{true}} \qquad \frac{(m \leq n)}{\Delta \vdash \overline{m} \leq \overline{n}\ \mathsf{true}}$$

$$\frac{\Delta \vdash t_1 = t_2\ \mathsf{true}}{\Delta \vdash t_1 \leq t_2\ \mathsf{true}} \qquad \frac{\Delta \vdash t_1 \leq t_3\ \mathsf{true} \quad \Delta \vdash t_3 \leq t_2\ \mathsf{true}}{\Delta \vdash t_1 \leq t_2\ \mathsf{true}} \qquad \frac{\Delta \vdash t_1 \leq t_2\ \mathsf{true} \quad \Delta \vdash t_2 \leq t_1\ \mathsf{true}}{\Delta \vdash t_1 = t_2\ \mathsf{true}}$$

$$\frac{\Delta \vdash t_1 \leq t_1'\ \mathsf{true} \quad \Delta \vdash t_2 \leq t_2'\ \mathsf{true}}{\Delta \vdash t_1 + t_2 \leq t_1' + t_2'\ \mathsf{true}} \qquad \frac{\Delta \vdash t + t_1 \leq t + t_2\ \mathsf{true}}{\Delta \vdash t_1 \leq t_2\ \mathsf{true}} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \overline{0} \leq t\ \mathsf{true}}$$

Figure 4.1: Truth of Formulas

The Truth Judgment

The rules defining the truth judgment are given in Figure 4.1. As mentioned earlier, we will assume for the duration of this chapter that contexts Δ contain only constraint term kinding assumptions (a:N) and constraint hypotheses of the form φ true where φ is simple.

The following "substitution" or "cut" property will come in useful later on.

Proposition 4.1 (Cut) *If* $\Delta, \varphi \vdash \varphi'$ true *and* $\Delta \vdash \varphi$ true, *then* $\Delta \vdash \varphi'$ true.

Since truth is a hypothetical judgment rather than a sequent-style proof system, the proof of this property is very straightforward.

4.2 Decidability

The main result of this section is the decidability of the TALT-R constraint logic:

Given Δ *and* φ, *it is decidable whether or not* $\Delta \vdash \varphi$ true.

The fact that this logic is "complete enough" is part of the type preservation theorem for my compilation strategy, which I will cover later on. Moreover, its soundness can be proven in Twelf.[4]

4.2.1 Proof Overview

The proof of decidability is based on two main insights, which taken together reveal that the existence of a proof for a given formula (in a given context) is equivalent to the existence of a

[4]Essentially the same constraint logic was added to the TALT implementation by Karl Crary shortly after I proposed its inclusion in TALT-R; the Twelf safety proof for TALT therefore includes a soundness proof for this logic.

feasible solution to a certain integer linear program that is easy to extract from the formula and the context. (This may seem anticlimactic: after all, the use of integer programming to solve constraints of this kind is quite common. However, deciding the validity of a constraint over the integers is not the same thing as deciding its *derivability* in this logic, which is relatively weak; I therefore had no *a priori* expectation that any off-the-shelf algorithm would work.)

The first insight is that, modulo regrouping and reordering, a term t is just a finite sum of "atomic" terms, each of which is either a variable or a literal number. If we imagine "combining like terms" as in high-school algebra (which is really just counting the number of occurrences of each variable and adding together all the literals), the formula is essentially just a linear polynomial in several variables with natural number coefficients. Using this fact it is easy to imagine a notion of canonical form for terms. To reduce a term to canonical form, simply reassociate and reorder its atomic subterms until it is, say, a right-associated sum with all the occurrences of each variable appearing consecutively and a single literal at the end; this representation is canonical except for the ordering of the variables.

For the purposes of this proof, I have found that the notion of syntactic reduction to canonical form is not particularly convenient. Instead, I "factor" the extraction of a canonical form into an *interpretation* function $[\![\cdot]\!]$ mapping constraint terms to *linear polynomials* (a concept I will make precise shortly) and a *representation* function $\mathcal{R}$ in the other direction. The linear polynomials here are objects of the metatheory, not constraint terms. The composition $\mathcal{R}[\![\cdot]\!]$ can be viewed as the extraction of a canonical form in the sense that (under suitable well-formedness conditions), if Δ contains no hypotheses of the form φ true then $\Delta \vdash t = u$ true if and only if $\mathcal{R}[\![t]\!]$ and $\mathcal{R}[\![u]\!]$ are the same. (Defining the representation function requires assuming that the set Var of variables is well-ordered.) However, the linear polynomial $[\![t]\!]$ is a more convenient object to reason about than the term $\mathcal{R}[\![t]\!]$: the former lives in a metatheoretic structure with an addition operator that is commutative and associative, while the latter lives in a *syntactic* theory with a *formal* addition operator that is commutative and associative *up to provable equality*.

The second insight is that treating terms as polynomials has the effect of trivializing most of the axioms in the logic. I devise an interpretation for formulas that trivializes several other rules in the same way. The rules making provable equality an equivalence relation and $\leq$ a partial order become trivial, and importantly, so does the rule allowing cancellation of a subterm appearing on both sides of an inequality. Under this interpretation, the only rules that remain nontrivial are the hypothesis rule, the nonnegativity rule (giving $\overline{0} \leq t$ for any t) and the monotonicity rule (allowing inequality formulas to be "added" as in high school algebra) — that is, a proof simply adds together several hypotheses along with one instance of the nonnegativity axiom. The possibility of doing this for any particular goal formula and set of hypotheses is easily formulated as an integer program.

4.2.2 Interpretation of Terms

Syntactically, the terms of the constraint logic contain both the natural numbers and the constraint term variables and are closed under formal addition. Viewed modulo provable equality, formal addition is commutative and associative, agrees with integer addition on integer arguments, and has zero as an identity. My first step in showing that the provable equality and provable inequality relations are decidable is to give an interpretation of constraint terms into a structure with a computable addition operation that has these properties up to *equality* rather than only up to provable equality.

Definition 4.2 *A* **linear polynomial** *is a function $P : \mathrm{Var} \cup \{\overline{1}\} \to \mathbb{Z}$ that is zero at all but finitely many points. We call the set of all such functions* Poly*. If $P(x) \geq 0$ for all x, we say P is* nonnegative*;* Poly^+ *is set of all nonnegative linear polynomials. The letters F and G will be understood to range over* Poly^+*.*

As a matter of notation, if $P(\overline{1}) = m_0$ and $P(a_i) = m_i$ for $1 \leq i \leq n$ and $P(b) = 0$ for all $b \notin \{a_1, \ldots, a_n\}$, then we write P as

$$m_0 + m_1 a_1 + \cdots + m_n a_n$$

optionally omitting m_0 if it is zero. Note that the order of the terms in such a rendering is insignificant.

The sum $P + Q$ is defined as the pointwise sum of the functions P and Q; this gives the usual meaning of polynomial addition. Subtraction and scalar multiplication are defined analogously. I will also need the pointwise meet operation $\sqcap$ and the "bounded subtraction" operation defined as follows:

$$\begin{aligned}(P \sqcap Q)(x) &= \min(P(x), Q(x)) \\ (P \ominus Q)(x) &= P(x) - \min(P(x), Q(x)) \\ &= \begin{cases} P(x) - Q(x) & \text{if } P(x) \geq Q(x) \\ 0 & \text{if } Q(x) > P(x) \end{cases}\end{aligned}$$

It will be important that for any polynomials P and Q, $P = (P \ominus Q) + (P \sqcap Q)$.

If F is a nonnegative linear polynomial, we say that $\Delta \vdash F$ if for variables a, $F(a) \neq 0$ implies $a \in \Delta$. Clearly, if $\Delta \vdash F$ and $\Delta \vdash G$ then $\Delta \vdash F + G$; furthermore, if $\Delta \vdash F + G$ then $\Delta \vdash F$ and $\Delta \vdash G$.

Definition 4.3 *The interpretation $[\![\cdot]\!] : \mathrm{Term} \to \mathrm{Poly}^+$ of terms as nonnegative linear polynomials is defined as follows:*

$$[\![\overline{n}]\!] = n \qquad [\![a]\!] = a \qquad [\![t + u]\!] = [\![t]\!] + [\![u]\!]$$

Observe that Poly^+ contains convenient subsets corresponding to the variables and the natural numbers, and that it forms a commutative monoid whose unit is $[\![\overline{0}]\!]$.

Let $\preceq$ denote the pointwise partial ordering on polynomials:

$$P \preceq Q \text{ iff for all } x \in \mathrm{Var} \cup \{\overline{1}\},\ P(x) \leq Q(x)$$

We will say $P \prec Q$ when $P \preceq Q$ and $P \neq Q$. When restricted to nonnegative polynomials, $\prec$ is well-founded. Assume the set Var of variables is also well-ordered by some relation $\sqsubset$. The resulting induction principles ensure that the following representation function is well-defined.

Definition 4.4 *The* representation $\mathcal{R} : \mathrm{Poly}^+ \to \mathrm{Term}$ *of nonnegative linear polynomials as terms is defined as follows:*

- *For constants $m \in \mathbb{Z}$, $\mathcal{R}(m) = \overline{m}$.*
- *For non-constant polynomials F, $\mathcal{R}(F) = a + \mathcal{R}(F - a)$ where $a = \min\{x \in \mathrm{Var} \mid F(x) \neq 0\}$.*

The minimum is with respect to $\sqsubset$. Note that this is a valid inductive definition, since in the second case $F - a$ is nonnegative and $F - a \prec F$.

The representation of polynomials as terms is a right inverse of the interpretation of terms as polynomials, and a left inverse up to some syntactic manipulation.

Lemma 4.3 *For any nonnegative polynomial F, $[\![\mathcal{R}(F)]\!] = F$.*

Proof: By induction on F.

Lemma 4.4 *If $\Delta \vdash F$ and $\Delta \vdash G$, then $\Delta \vdash \mathcal{R}(F + G) = \mathcal{R}(F) + \mathcal{R}(G)$ true.*

Proof: By induction on F.

Lemma 4.5 *If $\Delta \vdash t : \mathsf{N}$, then $\Delta \vdash \mathcal{R}[\![t]\!] = t$ true. It follows that for well-formed terms t and u, if $[\![t]\!] = [\![u]\!]$ then $\Delta \vdash t = u$ true.*

Proof: By induction on (the kinding of) t, using Lemma 4.4.

As an aside, note that the ordering on Var is necessary only to make the choice of a in the second part of the definition of $\mathcal{R}$ unique; one could do without it entirely by defining $\mathcal{R}$ as a one-to-many relation rather than a function. This would make the proofs of these last three lemmas somewhat awkward, but would render the canonical form $\mathcal{R}[\![t]\!]$ insensitive to the names of the free variables in t. Since simple formulas contain no bound variables, I consider sacrificing this insensitivity a reasonable tradeoff.

4.2.3 Interpretation of Formulas

Next, I define an interpretation of formulas in the constraint logic as constraints on linear polynomials.

Definition 4.5 *A* **polynomial constraint** *is an assertion of the form $P = 0$ or $P \leq 0$, where P is a linear polynomial (not necessarily nonnegative).*

Definition 4.6 *The interpretation $[\![\cdot]\!] : \mathrm{Form} \to \mathrm{PConstr}$ of formulas as polynomial constraints is defined as follows:*

$$[\![t_1 \leq t_2]\!] = ([\![t_1]\!] - [\![t_2]\!] \leq 0) \qquad [\![t_1 = t_2]\!] = ([\![t_1]\!] - [\![t_2]\!] = 0)$$

Because of the reflexivity and antisymmetry rules for inequality in the constraint logic, the derivability of any equality formula $t_1 = t_2$ is equivalent to the derivability of both inequalities $t_1 \leq t_2$ and $t_2 \leq t_1$; furthermore, the exact same formulas can be derived in a context containing an equality hypothesis as in the context that contains inequality in both directions instead. Thus, as I will show, it suffices to restrict our attention to inequality formulas. First, though, I must define a representation function mapping polynomial constraints to constraint formulas, analogous to the one for terms. The first step in defining this mapping is to show how to decompose the polynomial on the left side of a polynomial constraint into the two nonnegative polynomials representing the terms on both sides of an inequality formula.

Definition 4.7 *If P is a linear polynomial, then its* left-hand part P_L *and* right-hand part P_R *are defined as follows:*

$$P_L(x) = \begin{cases} P(x) & \text{if } P(x) > 0 \\ 0 & \text{otherwise} \end{cases} \qquad P_R(x) = \begin{cases} -P(x) & \text{if } P(x) < 0 \\ 0 & \text{otherwise} \end{cases}$$

That is, P_L consists of the terms of P with positive coefficients, and P_R consists of the terms of P with negative coefficients.

The decomposition into left-hand and right-hand parts is unique, in a sense made precise by the following lemma.

Lemma 4.6 *Let P, P_L^* and P_R^* be linear polynomials. Then $P_L^* = P_L$ and $P_R^* = P_R$ iff all three of the following are true:*

1. *P_L^* and P_R^* are both nonnegative;*
2. *$P = P_L^* - P_R^*$; and*
3. *For any $x \in \mathrm{Var} \cup \{\overline{1}\}$, either $P_L^*(x) = 0$ or $P_R^*(x) = 0$.*

Proof:

($\Rightarrow$): That conditions (1) and (2) hold for P_L and P_R is obvious. For condition (3), suppose $x \in \mathrm{Var} \cup \{\overline{1}\}$. If $P(x) \leq 0$, then by definition $P_L(x) = 0$; if $P(x) > 0$, then by definition $P_R(x) = 0$.

($\Leftarrow$): Assume conditions (1)–(3) hold. I need to show that $P_L^* = P_L$ and $P_R^* = P_R$. So, suppose $x \in \mathrm{Var} \cup \{\overline{1}\}$. Condition (3) gives two cases:

> **Case:** $P_L^*(x) = 0$. By condition (2), $P(x) = -P_R^*(x)$. By condition (1), this means $P(x) \leq 0$. Hence $P_L(x) = 0 = P_L^*(x)$ and $P_R(x) = -P(x) = P_R^*(x)$.
>
> **Case:** $P_R^*(x) = 0$. By condition (2), $P(x) = P_L^*(x)$. By condition (2), then, $P(x) \geq 0$. Hence $P_L(x) = P(x) = P_L^*(x)$ and $P_R(x) = 0 = P_R^*(x)$.

End of Proof.

Henceforth I will gloss the third, somewhat awkward, condition by saying that P_L and P_R have "disjoint domains".

Definition 4.8 *The syntactic representation $\mathcal{R} : \mathrm{PConst} \to \mathrm{Form}$ of polynomial constraints as formulas is defined by:*

$$\mathcal{R}(P \leq 0) = (\mathcal{R}(P_L) \leq \mathcal{R}(P_R))$$

Lemma 4.7 *For any polynomial P, $[\![\mathcal{R}(P \leq 0)]\!] = (P \leq 0)$.*

Proof: Direct, using Lemma 4.3:

$$[\![\mathcal{R}(P \leq 0)]\!] = [\![\mathcal{R}(P_L) \leq \mathcal{R}(P_R)]\!] = ([\![\mathcal{R}(P_L)]\!] - [\![\mathcal{R}(P_R)]\!] \leq 0) = (P_L - P_R \leq 0) = (P \leq 0)$$

End of Proof.

Lemma 4.8 *If $\Delta \vdash t : \mathsf{N}$ and $\Delta \vdash u : \mathsf{N}$, then $t \leq u$ and $\mathcal{R}([\![t \leq u]\!])$ are interderivable in context Δ. That is,*

1. *$\Delta, (t \leq u) \vdash \mathcal{R}([\![t \leq u]\!])$ true and*
2. *$\Delta, \mathcal{R}([\![t \leq u]\!]) \vdash t \leq u$ true.*

Proof:

Let $F = [\![t]\!]$ and $G = [\![u]\!]$. Then $\mathcal{R}([\![t \leq u]\!]) = \mathcal{R}(F - G \leq 0) = ((F - G)_L \leq (F - G)_R$. Observe that

$$F - G = ((F \sqcap G) + (F \ominus G)) - ((F \sqcap G) + (G \ominus F)) = (F \ominus G) - (G \ominus F)$$

But $F \ominus G$ and $G \ominus F$ are both nonnegative and have disjoint domains, so it follows that $(F-G)_L = F \ominus G$ and $(F-G)_R = G \ominus F$. Hence $\mathcal{R}([\![t \leq u]\!]) = (\mathcal{R}(F \ominus G) \leq \mathcal{R}(G \ominus F))$.

So, to prove part (1), let $\Delta' = (\Delta, t \leq u)$.
By the hypothesis rule, $\Delta' \vdash t \leq u$ true.
Observe that $[\![t]\!] = F = (F \sqcap G) + (F \ominus G)$ and similarly $[\![u]\!] = (F \sqcap G) + (G \ominus F)$.
By Lemma 4.4, $\Delta' \vdash \mathcal{R}(F \sqcap G) + \mathcal{R}(F \ominus G) = \mathcal{R}[\![t]\!]$ true and $\Delta' \vdash \mathcal{R}[\![u]\!] = \mathcal{R}(F \sqcap G) + \mathcal{R}(F \ominus G)$ true.
Applying Lemma 4.3, $\Delta' \vdash \mathcal{R}(F \sqcap G) + \mathcal{R}(F \ominus G) = t$ true and $\Delta' \vdash u = \mathcal{R}(F \sqcap G) + \mathcal{R}(F \ominus G)$ true.
By reflexivity and transitivity, $\Delta' \vdash \mathcal{R}(F \sqcap G) + \mathcal{R}(F \ominus G) \leq \mathcal{R}(F \sqcap G) + \mathcal{R}(G \ominus F)$ true.
By the cancellation rule, $\Delta' \vdash \mathcal{R}(F \ominus G) \leq \mathcal{R}(F \sqcap G)$ true.
That is, $\Delta' \vdash \mathcal{R}([\![t \leq u]\!]$ true.

To prove part (2), let $\Delta'' = (\Delta, \mathcal{R}([\![t \leq u]\!]))$.
By the hypothesis rule, $\Delta'' \vdash \mathcal{R}([\![t \leq u]\!])$ true.
That is, $\Delta'' \vdash \mathcal{R}(F \ominus G) \leq \mathcal{R}(G \ominus F)$ true.
Clearly $\Delta'' \vdash \mathcal{R}(F \sqcap G) : \mathsf{N}$; therefore by the reflexivity rules $\Delta'' \vdash \mathcal{R}(F \sqcap G) \leq \mathcal{R}(F \sqcap G)$ true.
By the monotonicity rule, $\Delta'' \vdash \mathcal{R}(F \sqcap G) + \mathcal{R}(F \ominus G) \leq \mathcal{R}(F \sqcap G) + \mathcal{R}(G \ominus F)$ true.
Reasoning as in part (1), $\Delta'' \vdash t = \mathcal{R}(F \sqcap G) + \mathcal{R}(F \ominus G)$ true and $\Delta'' \vdash \mathcal{R}(F \sqcap G) + \mathcal{R}(G \ominus F) = u$ true.
By reflexivity and transitivity, $\Delta'' \vdash t \leq u$ true.
End of Proof.

Lemma 4.9 (Addition of Provable Constraints) *If* $\Delta \vdash \mathcal{R}(P \leq 0)$ true *and* $\Delta \vdash \mathcal{R}(Q \leq 0)$ true *then* $\Delta \vdash \mathcal{R}(P + Q \leq 0)$ true.

Proof:
By assumption, $\Delta \vdash \mathcal{R}(P_L) \leq \mathcal{R}(P_R)$ true and $\Delta \vdash \mathcal{R}(Q_L) \leq \mathcal{R}(Q_R)$ true.
By the monotonicity rule, $\Delta \vdash \mathcal{R}(P_L) + \mathcal{R}(Q_L) \leq \mathcal{R}(P_R) + \mathcal{R}(Q_R)$ true.
Using Lemma 4.4 and the reflexivity and transitivity rules, $\Delta \vdash \mathcal{R}(P_L + Q_L) \leq \mathcal{R}(P_R + Q_R)$ true.
By Lemma 4.8 and Proposition 4.1, $\Delta \vdash \mathcal{R}[\![\mathcal{R}(P_L + Q_L) \leq \mathcal{R}(P_R + Q_R)]\!]$ true.
By definition, $[\![\mathcal{R}(P_L + Q_L) \leq \mathcal{R}(P_R + Q_R)]\!] = ([\![\mathcal{R}(P_L + Q_L)]\!] - [\![\mathcal{R}(P_R + Q_R)]\!] \leq 0)$.
Applying Lemma 4.3, $([\![\mathcal{R}(P_L + Q_L)]\!] - [\![\mathcal{R}(P_R + Q_R)]\!] \leq 0) = ((P_L + Q_L) - (P_R + Q_R) \leq 0) = ((P_L - P_R) + (Q_L - Q_R) \leq 0) = (P + Q \leq 0) = (S \leq 0)$.
Thus I have $\Delta \vdash \mathcal{R}(S \leq 0)$ true, as desired.
End of Proof.

4.2.4 Semantic Proofs

To finish setting up the proof of decidability, I now define a notion of "proof" for polynomial constraints. After proving that the syntactic provable inequality relation of TALT-R and this new semantic provable inequality relation correspond, I will show that the existence of a semantic proof for a given constraint is decidable.

Definition 4.9 *The interpretation of contexts as sets of polynomial constraints is defined as follows:*

$$\begin{aligned}[\![\Delta]\!] = \quad & \{[\![t_1 \leq t_2]\!] \mid (t_1 \leq t_2) \in \Delta\} \cup \\ & \{[\![t_1 \leq t_2]\!] \mid (t_1 = t_2) \in \Delta\} \cup \\ & \{[\![t_2 \leq t_1]\!] \mid (t_1 = t_2) \in \Delta\}\end{aligned}$$

Definition 4.10 *A* **semantic proof** $M = (A, F)$ *consists of a finite multiset* A *of linear polynomials and a nonnegative linear polynomial* F*. The* **yield** *of* M *(written* $\sum M$*) is defined as*

$$\sum M = (\sum_{R \in A} R) - F$$

We say that M *is a semantic proof of* $P \leq 0$ *in context* Δ *(written* $\Delta \models M : P \leq 0$*) if* $\sum M = P$ *and, for every* $R \in A$*,* $(R \leq 0) \in [\![\Delta]\!]$*.*

It is easy to show that every semantic proof corresponds to a syntactic proof. First, I prove a lemma showing that each constraint in the interpretation of a context is syntactically provable, then I prove the main soundness lemma.

Lemma 4.10 *If* $(P \leq 0) \in [\![\Delta]\!]$*, then* $\Delta \vdash \mathcal{R}(P \leq 0)$ true*.*

Proof:
There are three cases.

> **Case 1:** $(P \leq 0) = [\![t \leq u]\!]$ and $(t \leq u) \in \Delta$. By the hypothesis rule, $\Delta \vdash t \leq u$ true. By Lemma 4.8 and Proposition 4.1, $\Delta \vdash \mathcal{R}[\![t \leq u]\!]$ true, that is, $\Delta \vdash \mathcal{R}(P \leq 0)$ true.
>
> **Case 2:** $(P \leq 0) = [\![t \leq u]\!]$ and $(t = u) \in \Delta$. By the hypothesis rule, $\Delta \vdash t = u$ true. By the reflexivity rule, $\Delta \vdash t \leq u$ true. The result follows as in part (1).
>
> **Case 3:** $(P \leq 0) = [\![u \leq t]\!]$ and $(t = u) \in \Delta$. By the hypothesis rule, $\Delta \vdash t = u$ true. By the symmetry rule for equality, $\Delta \vdash u = t$ true. By the reflexivity rule for $\leq$, $\Delta \vdash u \leq t$ true. The result follows as in part (1).

End of Proof.

Lemma 4.11 (Soundness)

1. *If* $\Delta \models M : [\![t \leq u]\!]$*, then* $\Delta \vdash t \leq u$ true*.*
2. *If* $\Delta \models M : [\![t \leq u]\!]$ *and* $\Delta \models M' : [\![u \leq t]\!]$ *then* $\Delta \vdash t = u$ true*.*

Proof: The proof of part (2) follows from part (1) by antisymmetry.

For part (1), let $M = (A, F)$ and $P = \sum M = \sum A - F$. I need to show $\Delta \vdash t \leq u$ true; by Lemma 4.8 and Proposition 4.1 it suffices to show that $\Delta \vdash \mathcal{R}[\![t \leq u]\!]$ true, that is, that $\Delta \vdash \mathcal{R}(P \leq 0)$ true.

To begin, notice that since F is nonnegative, $(-F)_L = 0$ and $(-F)_R = F$. By a proof rule, $\Delta \vdash \overline{0} \leq \mathcal{R}(F)$ true, *i.e.*, $\Delta \vdash \mathcal{R}((-F) \leq 0)$ true.

Also note that for every $R \in A$, $(R \leq 0) \in [\![\Delta]\!]$ and thus $\Delta \vdash \mathcal{R}(R \leq 0)$ true by Lemma 4.10. Since A is finite, repeated application of Lemma 4.9 gives $\Delta \vdash \mathcal{R}(\sum A \leq 0)$ true. Using Lemma 4.9 once more, $\Delta \vdash \mathcal{R}(\sum A - F \leq 0)$ true, *i.e.*, $\Delta \vdash \mathcal{R}(P \leq 0)$ true.
End of Proof.

Showing that anything syntactically provable is semantically provable is a bit more interesting.

Lemma 4.12 *If* $\Delta \models M : P \leq 0$ *and* $\Delta \models M' : Q \leq 0$ *then there exists an* N *giving* $\Delta \models N : (P+Q) \leq 0$*.*

Proof:
Suppose $M = (A, F)$ and $M' = (A', F')$. Then let $N = (A \uplus A', F + F')$, and observe that

$$\sum N = \sum(A \uplus A') - (F - F') = \sum A + \sum A' - (F - F') = (\sum A - F) + (\sum A' - F') = P + Q.$$

Thus by definition $\Delta \models M : (P + Q) \leq 0$.
End of Proof.

Lemma 4.13 (Completeness)

- *If* $\Delta \vdash t \leq u$ true *then there is an* M *such that* $\Delta \models M : [\![t \leq u]\!]$.
- *If* $\Delta \vdash t = u$ true *then there is an* M *such that* $\Delta \models M : [\![t \leq u]\!]$ *and an* M' *such that* $\Delta \models M' : [\![u \leq t]\!]$.

Proof:
I will prove both parts simultaneously by induction on derivations.

Case:

$$\frac{((\varphi \text{ true}) \in \Delta)}{\Delta \vdash \varphi \text{ true}}$$

Sub-case: $\varphi = (t \leq u)$. Then $[\![t \leq u]\!] \in [\![\Delta]\!]$ and so $\Delta \models (\{[\![t \leq u]\!]\}, 0) : [\![t \leq u]\!]$.

Sub-case: $\varphi = (t = u)$. Then $[\![t \leq u]\!]$ and $[\![u \leq t]\!]$ are both in $[\![\Delta]\!]$. Thus $\Delta \models (\{[\![t \leq u]\!]\}, 0) : [\![t \leq u]\!]$ and $\Delta \models (\{[\![u \leq t]\!]\}, 0) : [\![u \leq t]\!]$.

Case:

$$\frac{\Delta \vdash t_2 = t_1 \text{ true}}{\Delta \vdash t_1 = t_2 \text{ true}} \quad \frac{\Delta \vdash t_1 = t_2 \text{ true}}{\Delta \vdash t_1 \leq t_2 \text{ true}} \quad \frac{\Delta \vdash t_1 \leq t_2 \text{ true} \quad \Delta \vdash t_2 \leq t_1 \text{ true}}{\Delta \vdash t_1 = t_2 \text{ true}}$$

Each of these cases follows immediately from the induction hypothesis.

Case:

$$\frac{}{\Delta \vdash \overline{m} + \overline{n} = \overline{m+n} \text{ true}} \quad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \overline{0} + t = t \text{ true}} \quad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 + t_2 = t_2 + t_1 \text{ true}}$$

$$\frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash t = t \text{ true}} \quad \frac{\Delta \vdash t_i : \mathsf{N} \text{ (for } i = 1, 2, 3)}{\Delta \vdash (t_1 + t_2) + t_3 = t_1 + (t_2 + t_3) \text{ true}}$$

In each of these rules the formula being proved has the form $t = u$ where $[\![t]\!] = [\![u]\!]$.
Thus in each case, $[\![t \leq u]\!] = [\![u \leq t]\!] = (0 \leq 0)$, and so $(\emptyset, 0)$ is a canonical proof of either direction.

Case:

$$\frac{\Delta \vdash t_1 = t_1' \text{ true} \quad \Delta \vdash t_2 = t_2' \text{ true}}{\Delta \vdash t_1 + t_2 = t_1' + t_2' \text{ true}}$$

By the induction hypothesis, $[\![t_1 \leq t_1']\!]$ and $[\![t_2 \leq t_2']\!]$ have semantic proofs in Δ.
That is, there exist semantic proofs of $([\![t_1]\!] - [\![t_1']\!] \leq 0)$ and $([\![t_2]\!] - [\![t_2']\!] \leq 0)$.
By Lemma 4.12, $([\![t_1]\!] - [\![t_1']\!] + [\![t_2]\!] - [\![t_2']\!] \leq 0)$ is semantically provable.

But this constraint is the same as $([\![t_1 + t_2]\!] - [\![t'_1 + t'_2]\!] \leq 0)$, that is, $[\![t_1 + t_2 \leq t'_1 + t'_2]\!]$.
The other direction is the same.

Case:

$$\frac{\Delta \vdash t_1 = t_3 \ \mathsf{true} \quad \Delta \vdash t_3 = t_2 \ \mathsf{true}}{\Delta \vdash t_1 = t_2 \ \mathsf{true}}$$

I will show that $[\![t_1 \leq t_2]\!]$ is semantically provable; the other direction is similar.
By the induction hypothesis, there exist semantic proofs of $[\![t_1 \leq t_3]\!]$ and $[\![t_3 \leq t_2]\!]$, that is, $([\![t_1]\!] - [\![t_3]\!] \leq 0)$ and $([\![t_3]\!] - [\![t_2]\!] \leq 0)$.
By Lemma 4.12, there is a semantic proof of $([\![t_1]\!] - [\![t_3]\!] + [\![t_3]\!] - [\![t_2]\!] \leq 0)$, that is, of $([\![t_1]\!] - [\![t_2]\!] \leq 0)$, which in turn is the same as $[\![t_1 \leq t_2]\!]$.

Case:

$$\frac{\Delta \vdash t_1 \leq t_3 \ \mathsf{true} \quad \Delta \vdash t_3 \leq t_2 \ \mathsf{true}}{\Delta \vdash t_1 \leq t_2 \ \mathsf{true}}$$

By the induction hypothesis, there exist semantic proofs of $[\![t_1 \leq t_3]\!]$ and $[\![t_3 \leq t_2]\!]$, that is, of $([\![t_1]\!] - [\![t_3]\!] \leq 0)$ and $([\![t_3]\!] - [\![t_2]\!] \leq 0)$.
By Lemma 4.12, $([\![t_1]\!] - [\![t_3]\!] + [\![t_3]\!] - [\![t_2]\!] \leq 0)$ is semantically provable.
But $[\![t_1]\!] - [\![t_3]\!] + [\![t_3]\!] - [\![t_2]\!] = [\![t_1]\!] - [\![t_2]\!]$, so I have found a semantic proof of $[\![t_1 \leq t_2]\!]$.

Case:

$$\frac{(m \leq n)}{\Delta \vdash \overline{m} \leq \overline{n} \ \mathsf{true}}$$

Note that $[\![\overline{m} \leq \overline{n}]\!] = (m - n \leq 0)$. Since $m \leq n$, $(n - m)$ is a nonnegative linear polynomial; hence $\Delta \models (\emptyset, n - m) : [\![\overline{m} \leq \overline{n}]\!]$.

Case:

$$\frac{\Delta \vdash t_1 \leq t'_1 \ \mathsf{true} \quad \Delta \vdash t_2 \leq t'_2 \ \mathsf{true}}{\Delta \vdash t_1 + t_2 \leq t'_1 + t'_2 \ \mathsf{true}}$$

By the induction hypothesis, $[\![t_1 \leq t'_1]\!]$ and $[\![t_2 \leq t'_2]\!]$ are semantically provable.
That is, there exist semantic proofs of $([\![t_1]\!] - [\![t'_1]\!] \leq 0)$ and $([\![t_2]\!] - [\![t'_2]\!] \leq 0)$.
By Lemma 4.12, $([\![t_1]\!] - [\![t'_1]\!] + [\![t_2]\!] - [\![t'_2]\!] \leq 0)$ is semantically provable.
But this constraint is the same as $([\![t_1 + t_2]\!] - [\![t'_1 + t'_2]\!] \leq 0)$, that is, $[\![t_1 + t_2 \leq t'_1 + t'_2]\!]$.

Case:

$$\frac{\Delta \vdash t + t_1 \leq t + t_2 \ \mathsf{true}}{\Delta \vdash t_1 \leq t_2 \ \mathsf{true}}$$

By the induction hypothesis, $[\![t + t_1 \leq t + t_2]\!]$ is semantically provable.
By the definitions of the interpretation functions $[\![\cdot]\!]$ for terms and formulas,

$$[\![t + t_1 \leq t + t_2]\!] = ([\![t]\!] + [\![t_1]\!] - [\![t]\!] - [\![t_2]\!] \leq 0) = ([\![t_1]\!] - [\![t_2]\!] \leq 0) = [\![t_1 \leq t_2]\!])$$

Thus $[\![t_1 \leq t_2]\!]$ is semantically provable.

Case:

$$\frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \overline{0} \leq t \ \mathsf{true}}$$

Observe that $[\![\overline{0} \leq t]\!] = (-[\![t]\!] \leq 0)$; thus $\Delta \models (\emptyset, [\![t]\!]) : [\![\overline{0} \leq t]\!]$.

End of Proof.

4.2.5 The Decidability Theorem

The interpretation of terms as polynomials effectively gives a representation of terms that identifies those that differ only by rearrangement of terms and computation with numerals. The interpretation of formulas as polynomial constraints identifies those formulas that differ only by rearrangement of terms, computation with numerals, and cancellation. The only proof rules not covered are the hypothesis rule, the rules for adding equations or inequalities together, and the one stating that any term is greater than or equal to zero; thus the semantic content of a proof essentially consists of a collection of hypotheses to be added together, along with an axiomatically true formula of the form $\overline{0} \leq t$. Intuitively speaking, therefore, I have reduced the question of whether a formula is provable or not to a question of whether a certain polynomial is a linear combination of certain others or not. This latter question is clearly decidable.

Theorem 4.1 (Decidability) *The (derivability of the) judgment* $\Delta \vdash \varphi$ true *is decidable.*

Proof:

If $\varphi = (t = u)$, then it suffices to decide whether both $\Delta \vdash t \leq u$ true and $\Delta \vdash u \leq t$ true. If both of these hold, then so does $\Delta \vdash \varphi$ true; if either does not hold, then neither does the judgment of interest.

If $\varphi = (t \leq u)$, then suppose $[\![t \leq u]\!] = (P \leq 0)$; by Lemmas 4.11 and 4.13 it suffices to decide whether there exists a semantic proof M such that $\Delta \models M : P \leq 0$. Suppose $[\![\Delta]\!] = \{H_1 \leq 0, \ldots, H_n \leq 0\}$; then I claim that $(P \leq 0)$ is semantically provable iff there are natural numbers $x_1, \ldots, x_n$ satisfying the system of constraints:

$$\begin{array}{rl} H_1(\overline{1})x_1 + \cdots + H_n(\overline{1})x_n & \geq P(\overline{1}) \\ H_1(a_1)x_1 + \cdots + H_n(a_1)x_n & \geq P(a_1) \\ & \vdots \\ H_1(a_m)x_1 + \cdots + H_n(a_m)x_n & \geq P(a_m) \end{array}$$

where

$$\{a_1, \ldots, a_m\} = \mathrm{dom}(P) \cup \bigcup_{i=1}^{n} \mathrm{dom}(H_i)$$

(*i.e.*, $a_1, \ldots, a_m$ are all the variables appearing in the judgment to be decided). Therefore, to decide whether $\Delta \vdash t \leq u$ true is provable it suffices to generate this system of constraints and solve it using an algorithm for Integer Programming. Generating the constraints poses no difficulty, since the operations on polynomials required to extract P from t and u, and the H_i's from Δ, are certainly computable.

Now I must prove both directions of my claim. First, suppose that $M = (A, F)$ is a semantic proof of $(P \leq 0)$. Then each polynomial R in the multiset A is equal to H_i for some i. So, for $1 \leq i \leq n$, let x_i be the multiplicity of H_i in A; then

$$P = \sum M = \sum A - F = x_1 H_1 + \cdots + x_n H_n - F$$

Since F is nonnegative, $x_1, \ldots, x_n$ satisfy the constraints above.

Now, suppose that $x_1, \ldots, x_n$ satisfy the constraints. To produce a semantic proof of P, let A be the submultiset of $\{H_1, \ldots, H_n\}$ containing each H_i with multiplicity x_i and define $F(x) = (\sum A)(x) - P(x)$ for every $x \in \mathrm{Var} \cup \{\overline{1}\}$. Then clearly $\sum A - F = P$, so it remains only to show

that F is nonnegative. Clearly $F(z) = 0$ for any $z \in \mathrm{Var} \setminus \{a_1, \ldots, a_m\}$. On the other hand, if $z \in \{\overline{1}, a_1, \ldots, a_m\}$ then one of the constraints gives $(\sum A)(z) \geq P(z)$ and so $F(z) \geq 0$. Thus $M = (A, F)$ is a semantic proof and $\Delta \vdash M : [\![t \leq u]\!]$.

End of Proof.

4.2.6 Implementation

I have not implemented this Integer Programming-based decision procedure in the TALT-R assembler. Instead, I implemented a much simpler algorithm that decides a proper subset of the derivable judgments; this subset is sufficiently large that any program generated by my compiler will be accepted by the algorithm.

The algorithm, which I call *depth-limited semantic proof search*, is based on a size measure for semantic proofs that I call *depth*. Roughly speaking, the depth of a semantic proof corresponds to the number of uses of the hypothesis rule in a syntactic derivation.

Definition 4.11 (Depth) *If $M = (A, F)$ is a semantic proof, then the* depth *of M is the cardinality of the multiset A. If the constraint $P \leq 0$ is semantically provable, then the* depth of $(P \leq 0)$ *(relative to a context Δ) is the minimum depth of any semantic proof of P (in Δ); otherwise we say $(P \leq 0)$ has depth ∞. The depth of an inequality formula $t \leq u$ is the depth of $[\![t \leq u]\!]$; the depth of an equality formula $t = u$ is the maximum of the depth of $t \leq u$ and the depth of $u \leq t$.*

Clearly, formulas of any depth are provable; thus for any d,

$$\mathrm{DLP}_d = \{(\Delta, \varphi) \mid \varphi \textit{ has depth at most } d \textit{ relative to } \Delta\} \subseteq \{(\Delta, \varphi) \mid \Delta \vdash \varphi\ \mathsf{true}\}.$$

The *depth-limited semantic proof search algorithm with limit d* decides the set DLP_d: Given a context Δ and a formula $\varphi = (t \leq u)$, it enumerates all possible submultisets of $[\![\Delta]\!]$ of cardinality d or smaller (there are finitely many). For each such multiset A, it computes the difference $F = \sum A - P$. If F is nonnegative, then $\Delta \models (A, F) : [\![\varphi]\!]$ and so the depth of φ is at most $|A|$, which is at most d; thus $(\Delta, \varphi) \in \mathrm{DLP}_d$ and the algorithm returns success. If no A makes $\sum A - P$ nonnegative, the algorithm returns failure.

When I discuss type preservation for my translation into TALT-R, I will claim that there is a d such that the typing of the translation's output never depends on the truth of any formula with depth greater than d. This implies that depth-limited semantic proof search with limit d is "complete enough" in the sense I described at the start of this chapter.

4.3 Incompleteness

Since the rules of the TALT-R constraint logic do not include a schema for induction over natural numbers, it comes as no suprise that they are not complete in the sense that not every entailment $\Delta \vdash \varphi\ \mathsf{true}$ that is valid over the natural numbers can be derived using them. In fact, all of the rules of the constraint theory remain sound if the variables are allowed to range over all nonnegative rationals; thus any entailment that is valid over $\mathbb{N}$ but not over $\mathbb{Q}^{\geq 0}$ cannot possibly be derivable. For instance, $(a + a \leq 3) \nvdash a \leq 1\ \mathsf{true}$ because this judgment is not valid if a is allowed to take on non-integral values.

The next natural question is whether the TALT-R constraint logic can derive *all* formulas that *are* valid over the nonnegative rationals. The answer is no: for a counterexample, observe that $(a + a \leq 4) \nvdash a \leq 2\ \mathsf{true}$. An extension of the TALT-R constraint logic capable of deriving this judgment and others like it is described in Appendix C.

4.4 Chapter Summary

The clock reasoning in TALT-R, as well as the static semantics of guarded and singleton types, depends on a constraint logic whose soundness must be provable in Twelf. I have isolated an impoverished but useful theory of linear inequalities whose soundness is very straightforward, and proven that it is decidable (although I have not found an efficient decision procedure) and contains a convenient subset for which a simple decision procedure is easily designed.

Chapter 5

Lilt: A Low-Level Source Language

> **lilt** \lilt\ (*n*) **1** : a spirited and usually cheerful song or tune **2** : a rhythmical swing, flow, or cadence **3** : a springy buoyant movement [44]

So that I can formalize the process of resource-bound certifying compilation, this chapter presents a low-level typed language that will serve as the source of a translation into MiniTALT-R. I call this language Lilt,[1] and it serves as the intermediate language in a certifying compiler for a subset of the high-level language Popcorn. (Popcorn is best known as the source language designed for compilation into TALx86 [45].) The back-end of my compiler generates EXTALT-R from Lilt using a translation based on the Lilt-to-MiniTALT-R translation in Chapter 7.

Lilt is designed to be completely ignorant of timing issues, but it does have a number of unusual characteristics motivated by its intended use in a compiler for Popcorn. Specifically, functions in a Popcorn program usually declare mutable local variables which they read from and assign to frequently. Furthermore, Popcorn functions often contain loops and sometimes contain exception handling constructs, and it is essential that the state of the local variables be threaded through all this control flow with a minimum of work. The best implementation strategy seems to be the one (presumably) used in the majority of compilers for C-like languages, and described in many if not most traditional compiler design texts (*e.g.,* [48]): Each dynamic instance of a function allocates (at most) one stack frame in which to store its local variables, and register allocation is performed on (at least) an entire function at a time to minimize the amount of "shuffling" that must be performed.[2] Unfortunately, the decision to adopt this compilation model complicates the intermediate language, since it introduces a distinction between local (*intra*procedural) and non-local (*inter*procedural) transfers of control, and requires an intermediate language that can deal with mutable local variables.

5.1 Syntax

The syntax of Lilt is given in Figure 5.1. Lilt has three different syntactic classes of identifiers at the term level: ***function names*** (ranged over by f), which have global scope and stand for functions;

[1] The name was chosen because it is a near-acronym for "Low-level Intermediate Language," rhymes with TILT, is related to music (like most ConCert project terminology) and has implications of rhythm and liveliness, which is sort of like liveness.

[2] The parenthetical interjections acknowledge the possibilities of eliding the stack frame on an architecture with enough registers, and of performing interprocedural register allocation, respectively. However, our target architecture (IA-32) has few registers and we do not plan to implement any interprocedural optimizations, so we will not discuss these matters any further.

Operands	v	::=	$s \mid n \mid \texttt{tt} \mid \texttt{ff} \mid \star \mid f \mid q@v$
Coercions	q	::=	$\texttt{id} \mid [c_1, \ldots, c_n] \mid \texttt{roll}_\tau \mid \texttt{unroll} \mid \texttt{pack}[\tau, c_1, \ldots, c_n]$
Small Expressions	r	::=	$v \mid op(v_1, \ldots, v_n) \mid \pi_i v \mid \texttt{inj}_\tau(i, v) \mid \texttt{outj}(v)$
		$\mid$	$\langle v_1, \ldots, v_n \rangle \mid \{v_1, \ldots, v_n\}$
Conditions	$cond$	::=	$v_1 = v_2 \mid v_1 < v_2$
Expressions	e	::=	$\texttt{return}\ v \mid \texttt{raise}\ v \mid \texttt{goto}\ \ell[c_1, \ldots, c_n]$
		$\mid$	$\texttt{let}\ s = r\ \texttt{in}\ e$
		$\mid$	$\texttt{let}\ s = v(v_1, \ldots, v_m)\ \texttt{in}\ e$
		$\mid$	$\texttt{let}\ s = \texttt{sub}(v, v_1)\ \texttt{in}\ e \mid \texttt{let}\ \texttt{sub}(v_1, v_2) := v_3\ \texttt{in}\ e$
		$\mid$	$\texttt{let}\ \pi_i\, v := v_1\ \texttt{in}\ e$
		$\mid$	$\texttt{let}\ (\alpha_1, \ldots, \alpha_n, s) = \texttt{unpack}\ v\ \texttt{in}\ e$
		$\mid$	$\texttt{pushhandler}\ \ell[c_1, \ldots, c_n]\ \texttt{in}\ e \mid \texttt{pophandler}\ \texttt{in}\ e$
		$\mid$	$\texttt{if}\ cond\ \texttt{then}\ e_1\ \texttt{else}\ e_2$
		$\mid$	$\texttt{case}\ v\ \texttt{of}\ \texttt{inj}(i, s) \Rightarrow e_1\ \texttt{else}\ e_2$
Functions	F	::=	$\texttt{func}(\Delta; \Gamma; \tau).(\texttt{enter}(s_1, \ldots, s_n).e, \ell_1 = B_1, \ldots, \ell_m = B_m)$
Blocks	B	::=	$\texttt{block}(\Delta; \Xi; \Gamma).e \mid \texttt{hndl}(\Delta; \Xi; \Gamma; s).e$
Programs	P	::=	$f_1 = F_1, \ldots, f_n = F_n$
Kinds	k	::=	$T \mid k_1 \to k_2$
Type Constructors	c, τ	::=	$\alpha \mid \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \langle \tau_1, \ldots, \tau_k \rangle \mid [i_1{:}\tau_1, \ldots, i_n{:}\tau_n] \mid \texttt{ns}$
		$\mid$	$\tau\ \texttt{array} \mid (\tau_1, \ldots, \tau_m) \to \tau \mid \mu\alpha.\tau$
		$\mid$	$\forall\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau \mid \exists\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau \mid \lambda\alpha{:}k.c \mid c_1\, c_2$
Type Contexts	Δ	::=	$\cdot \mid \Delta, \alpha{:}k$
Block Types	γ	::=	$lbl(\Delta; \Xi; \Gamma) \mid hnd(\Delta; \Xi; \Gamma)$
Local Contexts	Γ	::=	$[s_1{:}\tau_1, \ldots, s_n{:}\tau_n]$
Exception Stack Types	Ξ	::=	$\cdot \mid \Xi, \Gamma$
Label Contexts	Λ	::=	$\ell_1{:}\gamma_1, \ldots, \ell_n{:}\gamma_n$
Function Contexts	Φ	::=	$f_1{:}\tau_1, \ldots, f_n{:}\tau_n$

Figure 5.1: Lilt Syntax

labels (ranged over by ℓ), which stand for code blocks within a function and are meaningful only inside that function, and ***local variable names*** (ranged over by s), which also have function scope. Local variables are used as the names of a function's arguments as well as the names of local storage locations allocated by a function.

A Lilt program is a sequence of mutually recursive ***function definitions***, and the body of each function consists of one or more ***blocks***. The first block in each function is a special ***entry block*** of the form $\texttt{enter}(s_1, \ldots, s_n).e$, which is made up of a declaration of the function's local variables and the expression that will be evaluated when the function is called. Each of the remaining zero or more blocks in the function body is either an ordinary block ($\texttt{block}(\Delta; \Xi; \Gamma).e$) or an exception handler ($\texttt{hndl}(\Delta; \Xi; \Gamma; s).e$). Corresponding to these different kinds of code blocks are four different control-transfer expression forms, namely function call, function return, unconditional jump and `raise`.

If v_f is a function value, the function call expression $\texttt{let}\ s = v_f(\vec{v})\ \texttt{in}\ e$ causes control to be transferred to v_f's entry block, binding the function's formal parameters to the values $\vec{v}$. If the function returns a value, that value is copied into the local variable s and the expression e is evaluated. The expression $\texttt{return}\ v$ immediately exits the current function and returns the value v to the calling function. The jump expression $\texttt{goto}\ \ell[\vec{c}]$ performs a one-way transfer of control to the block named ℓ, passing it the type arguments $\vec{c}$ and implicitly passing along the current values of the current function's arguments and local variables.

The expression $\texttt{raise}\ v$ is similar to $\texttt{return}\ v$ except that v must be an exception value, and it is passed not to the calling function but to the current exception handler, which may have been installed by any pending function including the current one. The handler has access to the current values of the arguments and local variables of the function that installed it, and designates one of these variables to receive the value v. The `pushhandler` and `pophandler` expression forms manipulate the stack of pending exception handlers, but cannot remove any handlers installed before the call to the current function. A `return` expression implicitly pops all exception handlers installed by the current function, restoring the handler that was current when the function was called.

The type system of Lilt is essentially that of the higher-order polymorphic λ-calculus F_ω [27] augmented with several useful types for programming. The language includes the base types `int`, `bool` and `unit` as well as the familiar n-ary product types ($\langle \tau_1, \ldots, \tau_n \rangle$), array types ($\tau\ \texttt{array}$) and function types ($(\tau_1, \ldots, \tau_n) \to \tau$). The variant type $[i_1{:}\tau_1, \ldots, i_n{:}\tau_n]$ is essentially similar to the more familiar n-ary sum type $(\tau_1 + \cdots + \tau_n)$ found in other calculi; the labels $i_1, \ldots, i_n$ are distinct integers, and serve to identify the summands. (They correspond directly to the "tag" words used by the implementation.) We have chosen to use labeled variant types rather than unlabeled sum types in Lilt because they admit a very straightforward translation into TALT. The Lilt type system also includes recursive types ($\mu\alpha.\tau$), and universal and existential quantification ($\forall\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau$, $\exists\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau$). Finally, higher-order type constructors may be formed by abstraction ($\lambda\alpha{:}k.c$) and applied in the usual way ($c_1\ c_2$).

5.2 Static Semantics

The judgment forms of the Lilt type system are listed in Table 5.1. The complete set of rules defining these judgments may be found in Appendix D; I will discuss only the more unusual aspects of the type system in this section.

The central typing judgment in Lilt is the one for expressions. The judgment $\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash e$ states that e is a well-formed expression, where:

Judgment	Meaning
$\Delta \vdash c : k$	c has kind k
$\Delta \vdash c_1 = c_2 : k$	c_1 and c_2 are equivalent at kind k
$\Delta \vdash \Gamma$	Γ is well-formed
$\Delta \vdash \Xi$	Ξ is well-formed
$\Delta \vdash q : \tau_1 \Rightarrow \tau_2$	q coerces from τ_1 to τ_2
$\Phi; \Delta; \Gamma \vdash r : \tau$	r has type τ
$\Phi; \Delta; \Gamma \vdash \mathit{cond}$ cond	cond is a well-formed condition
$\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash e$	e is well-formed
$\Phi; \Delta; \Lambda; \tau \vdash B : \gamma$	B is a block of type γ
$\Phi \vdash F : \tau$	F is a function of type τ
$\vdash P$	P is a well-formed program
$\Delta \vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\Delta \vdash \Gamma_1 \leq \Gamma_2$	Γ_1 is a subtype of Γ_2
$\Delta \vdash \Xi_1 \leq \Xi_2$	Ξ_1 is a subtype of Ξ_2
$\Delta \vdash \Xi$ handles Γ	(see discussion of `raise`)

Table 5.1: Lilt typing judgment forms

- Φ is a function context, which assigns types to the function symbols defined in the program.
- Δ is a type context, which assigns kinds to constructor variables. The contents of Δ will be the type parameters of the current function and those of the current block, plus any additional variables introduced by `unpack` expressions.
- Λ assigns types to the block labels in the current function.
- Ξ describes the pending exception handlers, if any, that have been installed by the current function.
- Γ is a local context, which assigns types to the local variable names that may appear in e.
- τ is the return type of the current function.

If this judgment holds, then the expression e performs zero or more primitive operations and then does one of three things: It may return a value of type τ from the current function, it may jump to one of the labels declared in Λ, or it may raise an exception. The typing rule for `return` expressions states that returning a value of the appropriate type is always permitted:

$$\frac{\Phi; \Delta; \Gamma \vdash v : \tau}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{return}\ v}$$

Jumping to a label is allowed provided the label identifies an ordinary block (as opposed to an exception handler) that can accept the current state of the local storage and exception stack. A block may require some type arguments in addition to those of the enclosing function; the `goto` expression must provide constructors of the appropriate kinds:

$$\frac{(\Lambda(\ell) = \mathit{lbl}(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \quad \Delta \vdash c_i : k_i \quad \Delta \vdash \Gamma \leq \Gamma'[\vec{c}/\vec{\alpha}] \quad \Delta \vdash \Xi \leq \Xi'[\vec{c}/\vec{\alpha}]}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{goto}\ \ell[c_1, \ldots, c_n]}$$

Installing an exception handler has similar typing requirements to jumping: the constructor arguments must be properly kinded and the current stack of exception handlers must be consistent with the new handler's expectations. However, it is not necessary that the local context match the one expected by the handler at the point the handler is installed; this requirement is deferred to the point at which an exception is raised. The rule for pushing an exception handler is as follows:

$$\frac{\begin{array}{c}(\Lambda(\ell) = hnd(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \\ \Delta \vdash c_i : k_i \quad \Delta \vdash \Xi \leq \Xi'[\vec{c}/\vec{\alpha}] \quad \Phi; \Delta; \Lambda; (\Xi, \Gamma'[\vec{c}/\vec{\alpha}]); \Gamma; \tau \vdash e\end{array}}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{pushhandler}\ \ell[c_1, \ldots, c_n]\ \texttt{in}\ e}$$

The typing rule for `raise` expressions requires that the local context match the one expected by the current handler. This is captured by the premise $\Delta \vdash \Xi\ \mathsf{handles}\ \Gamma$:

$$\frac{\Phi; \Delta; \Gamma \vdash v : \tau_{\text{exn}} \quad \Delta \vdash \Xi\ \mathsf{handles}\ \Gamma}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{raise}\ v}$$

The auxiliary judgment $\Delta \vdash \Xi\ \mathsf{handles}\ \Gamma$ (defined in Appendix D) holds if Ξ is empty, meaning that the current exception handler was not locally installed (in which case the contents of Γ are irrelevant because the current locals will be discarded), or if Ξ is nonempty and the local context Γ matches the expectations of the current locally installed handler as given by Ξ. Importantly, `raise` v is not the only form of expression that may raise an exception. Array subscript operations may do so (if the index is out of bounds), and so may function calls (if the callee raises an exception it does not handle itself); therefore the typing rules for these forms of expressions must also have premises of the form $\Delta \vdash \Xi\ \mathsf{handles}\ \Gamma$ to ensure that the state of the local variables is consistent with what the current handler requires.

Most of Lilt's operations are performed by a sort of let-binding expression: the expression `let` $s = r$ `in` e evaluates r, stores the result in location s, and continues with e. Its typing rule makes use of an auxiliary judgment to determine the type of r:

$$\frac{\Phi; \Delta; \Gamma \vdash r : \tau' \quad \Phi; \Delta; \Lambda; \Xi; \Gamma[s \mapsto \tau']; \tau \vdash e}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{let}\ s = r\ \texttt{in}\ e}$$

The terms ranged over by r (the so-called "small expressions") are generally single primitive operations performed on syntactic values; they involve no control flow, cannot raise exceptions, and have no side effects (except possibly allocation, which may fail and terminate the program). Of these operations, arithmetic, tuple allocation and projection are relatively standard and have the expected typing rules. Slightly unusual features of Lilt at this level are the treatment of labeled variant types (a generalization of disjoint union or sum types), and the use of coercions.

Variants A value of variant type is created as usual by the `inj` operation, which takes a tag integer j and a value v, and produces a value of any variant type containing a j variant whose type is that of v:

$$\frac{\begin{array}{c}\Delta \vdash \tau = [\ldots, j{:}\tau_j, \ldots] \\ \Phi; \Delta; \Gamma \vdash v : \tau_j\end{array}}{\Phi; \Delta; \Gamma \vdash \texttt{inj}_\tau(j, v) : \tau}$$

Given a value of variant type, accessing its contents is a two-stage process: the `case` expression form "narrows" the type until it has only one variant, and then the `outj` operation can extract the

carried value:

$$\frac{\Phi;\Delta;\Gamma \vdash v : [\overline{j{:}\tau}, i{:}\tau', \overline{j{:}\tau}'] \quad \Phi;\Delta;\Lambda;\Xi;\Gamma[s \mapsto [i{:}\tau']];\tau \vdash e_1 \quad \Phi;\Delta;\Lambda;\Xi;\Gamma[s \mapsto [\overline{j{:}\tau}, \overline{j{:}\tau}']];\tau \vdash e_2}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{case}\ v\ \texttt{of}\ \texttt{inj}(i,s) \Rightarrow e_1\ \texttt{else}\ e_2} \qquad \frac{\Phi;\Delta;\Gamma \vdash v : [i:\tau]}{\Phi;\Delta;\Gamma \vdash \texttt{outj}(v) : \tau}$$

The `case` expression typed in this rule examines the value v, which has a variant type, compares the tag of v to the number i and then continues with either e_1 or e_2, after placing a version of v with an appropriately refined type in the location s. (Here it is important that all the tags in the sum type are syntactically required to be distinct.) The typing of e_1 assumes that s has the unary variant type corresponding to the i branch of the type of v; the typing of e_2 assumes s has a variant type consisting of all the remaining branches of v's original type. The small expression $\texttt{outj}(v)$ assumes v has a unary variant type, and retrieves the value it carries.

Coercions The operations of $\forall$-elimination, $\exists$-introduction, and introduction and elimination of recursive types are intended to have the special property that, when applied to values, they require no run-time work to compute. It is reasonably common practice to simply include expression forms with this property among the syntactic values (or in Lilt, the operands) of the language. This is what I have done, except that I group these four different forms of values into one, namely the application of a *coercion* to a value (written $q@v$). From a typing point of view, coercions behave a bit like functions; in particular, the rule for coercion application is just like the usual function application rule:

$$\frac{\Phi;\Delta;\Gamma \vdash v : \tau_2 \quad \Delta \vdash q : \tau_2 \Rightarrow \tau}{\Phi;\Delta;\Gamma \vdash q@v : \tau}$$

The typing rules for the coercions themselves are derived from the standard typing rules for the constructs they replace. The $\forall$-elimination coercion, written $[c_1, \ldots, c_n]$, instantiates a value of a $\forall$-type:

$$\frac{\Delta \vdash c_i : k_i \text{ for } 1 \le i \le n}{\Delta \vdash [c_1, \ldots, c_n] : \forall \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau \Rightarrow \tau[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n]}$$

The $\exists$-introduction coercion, written $\texttt{pack}[\tau, c_1, \ldots, c_n]$, is similar:

$$\frac{\Delta \vdash \tau = \exists \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' : T \quad \Delta \vdash c_i : k_i \text{ for } 1 \le i \le n}{\Delta \vdash \texttt{pack}[\tau, c_1, \ldots, c_n] : \tau'[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n] \Rightarrow \tau}$$

The `roll` and `unroll` coercions mediate between a recursive type and its unrolling:

$$\frac{\Delta \vdash \tau = \mu\alpha.\tau' : T}{\Delta \vdash \texttt{roll}_\tau : \tau'[\tau/\alpha] \Rightarrow \tau} \qquad \frac{\Delta \vdash \mu\alpha.\tau : T}{\Delta \vdash \texttt{unroll} : \mu\alpha.\tau \Rightarrow \tau[\mu\alpha.\tau/\alpha]}$$

Roughly speaking, Lilt uses coercions for operations whose TALT equivalents are subtyping rules rather than value forms or instructions (and which are therefore represented as coercions in XTALT and EXTALT). This is not by accident, since the "operations" captured by subtyping rules in TALT (in which subtyping is resolutely inclusive rather than coercive) clearly amount to the identity.

```
int rfib( int n ) {
  if ( n < 2 ) return 1;
  return rfib(n-1) + rfib(n-2);
}
```

Popcorn

$$
\begin{array}{l}
\mathit{rfib} = \texttt{func}(\cdot;[n{:}\texttt{int}];\texttt{int}).(\\
\quad \texttt{enter}(\mathit{t1},\mathit{t2}). \\
\quad\quad \texttt{if } n < 2 \texttt{ then} \\
\quad\quad\quad \texttt{return } 1 \\
\quad\quad \texttt{else} \\
\quad\quad\quad \texttt{let } n = -(n,1) \texttt{ in} \\
\quad\quad\quad \texttt{let } \mathit{t1} = \mathit{rfib}(n) \texttt{ in} \\
\quad\quad\quad \texttt{let } n = -(n,1) \texttt{ in} \\
\quad\quad\quad \texttt{let } \mathit{t2} = \mathit{rfib}(n) \texttt{ in} \\
\quad\quad\quad \texttt{let } \mathit{t1} = +(\mathit{t1},\mathit{t2}) \texttt{ in} \\
\quad\quad\quad \texttt{return } \mathit{t1}\ )
\end{array}
$$

Lilt

Figure 5.2: Lilt Example: Recursive Fibonacci

```
int fib(int n) {
   int a,b,c;
   a = 1; b = 1;
   while (n != 0) {
      c = a + b;
      a = b;
      b = c;
      n--;
   }
   return a;
}
```

Popcorn

$$
\begin{array}{l}
\mathit{fib} = \texttt{func}(\cdot;[n{:}\texttt{int}];\texttt{int}).(\\
\quad \texttt{enter}(a,b,c). \\
\quad\quad \texttt{let } a = 1 \texttt{ in} \\
\quad\quad \texttt{let } b = 1 \texttt{ in} \\
\quad\quad \texttt{goto } \mathit{loop}\ , \\
\quad \mathit{loop} = \texttt{block}(\cdot;\cdot;[n{:}\texttt{int},a{:}\texttt{int},b{:}\texttt{int},c{:}\texttt{ns}]). \\
\quad\quad \texttt{if } n = 0 \texttt{ then} \\
\quad\quad\quad \texttt{return } a \\
\quad\quad \texttt{else} \\
\quad\quad\quad \texttt{let } c = +(a,b) \texttt{ in} \\
\quad\quad\quad \texttt{let } a = b \texttt{ in} \\
\quad\quad\quad \texttt{let } b = c \texttt{ in} \\
\quad\quad\quad \texttt{let } n = -(n,1) \texttt{ in} \\
\quad\quad\quad \texttt{goto } \mathit{loop})
\end{array}
$$

Lilt

Figure 5.3: Lilt Example: Iterative Fibonacci

```
union <a>list {
  void nil;
  *(a,<a>list) cons;
}

<a>list rev<a>(<a>list L) {
  <a>list M = ^.nil ;
  while (true) {
    switch (L) {
      case nil: return M;
      case cons*(h,t):
        M = ^.cons(^(h,M));
        L = t;
    }
  }
  // (Dead code)
  return M;
}
```

Popcorn

Define:

$listF = \lambda\alpha{:}T.\lambda\beta{:}T.\ [0{:}\texttt{unit}, 1{:}\langle\alpha,\beta\rangle]$
$list = \lambda\alpha{:}T.\ \mu\beta.listF\ \alpha\ \beta$
$listS = \lambda\alpha{:}T.\ listF\ \alpha\ (list\ \alpha)$

$rev = \texttt{func}(\alpha{:}T; [L{:}list\ \alpha]; list\ \alpha).($
  $\texttt{enter}(M, h).$
    $\texttt{let}\ M = \texttt{inj}_{listS\ \alpha}(0, \star)\ \texttt{in}$
    $\texttt{let}\ M = \texttt{roll}_{list\ \alpha}@M\ \texttt{in}$
    $\texttt{goto}\ loop\ ,$
  $loop = \texttt{block}(\cdot;\cdot;[L{:}list\ \alpha, M{:}list\ \alpha, h{:}\texttt{ns}]).$
    $\texttt{case}\ \texttt{unroll}@L\ \texttt{of}\ \texttt{inj}(0, L) \Rightarrow$
      $\texttt{return}\ M$
    $\texttt{else}$
      $\texttt{let}\ L = \texttt{outj}(L)\ \texttt{in}$
      $\texttt{let}\ h = \pi_0(L)\ \texttt{in}$
      $\texttt{let}\ h = \langle h, M\rangle\ \texttt{in}$
      $\texttt{let}\ M = \texttt{inj}_{listS\ \alpha}(1, h)\ \texttt{in}$
      $\texttt{let}\ M = \texttt{roll}_{list\ \alpha}@M\ \texttt{in}$
      $\texttt{let}\ L = \pi_1\ L\ \texttt{in}$
      $\texttt{goto}\ loop\ )$

Lilt

Figure 5.4: Lilt Example: List Reversal

5.3 Lilt Examples

A very simple Lilt function, illustrating the use of local variables, is shown in Figure 5.2. On the left side of the figure is a Popcorn (or C or Java) function that computes the n^{th} Fibonacci number using the obvious but inefficient recursive method; on the right is the approximate Lilt equivalent. Note that the entry block of the Lilt function declares the two local variables *t1* and *t2* but does not give types for them: at the start of the entry block, the local variables are uninitialized and so they have type `ns`. Also note that as in C-like languages, a function is allowed to assign into its arguments: the Lilt version of *rfib* destructively modifies its parameter n to compute the argument of each recursive call.

A somewhat more interesting function, involving some local control flow, is the function *fib* shown in Figure 5.3, which computes Fibonacci numbers using a linear-time loop instead of recursion. Again, note that the three local variables have type `ns` when they are first allocated. When the block called *loop* is invoked at the end of the entry block, a and b have been initialized, but c has not; therefore *loop*'s block header specifies the type `int` for a and for b (as well as for the argument n), but expects that c still has type `ns`. By the time *loop* invokes itself (in the last line of code), c has been assigned an integer; the jump is still well-typed because `int` is a subtype of `ns`.

A function with similar control-flow structure but more complex typing is the polymorphic list reversal function shown in Figure 5.4. This example uses the polymorphic type constructor

$list$, defined as follows:

$$list = \lambda\alpha{:}T.\ \mu\beta.\ [0 : \texttt{unit}, 1 : \langle\alpha, \beta\rangle]$$

(Note that the type $list\ \tau$ is recursive; this recursion is not marked by any special syntax in Popcorn, but must be written with a μ-type in Lilt.) For convenience, the constructor $listS$ is also defined in the figure; $listS\ \tau$ is simply the unrolling of the recursive type $list\ \tau$. At the beginning of the function rev, the variable M is initialized with an empty list; this is a two-stage process in Lilt, consisting of an injection (to produce a value of type $listS\ \alpha$) and an application of the coercion $\texttt{roll}_{list\ \alpha}$ to create the list itself. The block named $loop$ examines the list currently stored in the argument location L by unrolling it and performing a case analysis. In the case where the tag is 0—that is, L is the empty list—the current value of M is returned from the function. In the case where the tag is not 0—*i.e.*, the tag is 1 meaning L is a cons—the components of L are extracted by outjection and projection, the head of L is added to the front of M, the tail is stored back into L, and the loop is evaluated again.

Chapter 6

Yield Placement and Polling Techniques

The major novel element in compiling Lilt to TALT-R is, naturally, the placement of `yield` instructions so that the typing conditions regarding the virtual clock are satisfied. As mentioned briefly in Chapter 1, one of the key claims in this thesis is that a requirement that programs be certifiably responsive need not place any burden on the typical application programmer. To support this claim, I limit my discussion in this chapter and elsewhere to techniques for placing yields in a program with no timing-related input from the programmer at all. I shall comment further on this assumption in Chapter 9.

One possible strategy is to place a `yield` at the beginning of every basic block in the program and every Y instructions thereafter; this idea, while sound, is not very appealing because yielding is likely to be very expensive. (One can easily imagine a multiprocessing scenario where every yield allows an unbounded number of other processes to execute for up to Y instructions each.) I will describe a number of simple yield placement heuristics in this chapter, intended to increase the actual time between yields executed by programs as much as possible (while keeping it less than Y). These direct placement strategies, however, all fall short of optimal performance if Y is large. Later on, I will explain how the singleton and guarded types of TALT-R may be used to implement dynamic checks that avoid the limitations of direct yield placement strategies, greatly reducing the number of actual yields performed. However, even these checks are not free, so it is to one's advantage to minimize the number of them that are needed. Placement of checkpoints is essentially the same problem as placement of `yield` instructions, but the types involved are more complicated. Therefore, for the sake of clarity, I will structure the discussion as follows: first, I will explain some strategies for placing yield instructions with no dynamic checks; then, I will explain how dynamic checking is possible. The translation of Lilt to MiniTALT-R I give later will combine these ideas, using the placement strategies I discuss here to place dynamic checkpoints rather than actual `yield` instructions.

Yield placement in straight-line code is not interesting: one simply ensures that there are no more than Y non-yielding instructions in between any two consecutive yields. The challenge of yield placement is focused around instructions that perform transfers of control. If the virtual clock at the point of a jump is less than the value expected by the code being jumped to, a yield is necessary before the jump; on the other hand, if the virtual clock before a jump is greater than required, the next yield will happen sooner than necessary. There are essentially four different kinds of jumps in Lilt programs (function call, `return`, `goto` and `raise`), which subdivide yield placement in to three subproblems. *Local*, or *intra*procedural placement is the problem of ensuring that `goto` expressions obey the virtual clock rules; *global*, or *inter*procedural placement is concerned with function calls and returns; and finally *exceptional* placement deals with the timing properties

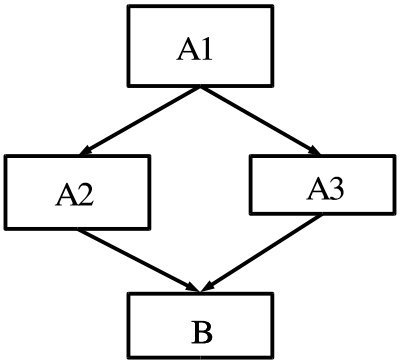


Figure 6.1: A Flow Graph With a Join

of exception handling. I will discuss each of these subproblems of yield placement in turn.

6.1 Local Placement

The problem of local, or *intra*procedural, yield placement is concerned with determining the initial virtual clock assumptions for all of the ordinary blocks in a Lilt function (that is, those that are not exception handlers and are not the entry block), and the placement of yield points consistent with these assumptions. This task is simplified by the fact that the targets of all local jumps (that is, `goto` expressions) are known, so an accurate flow graph for the ordinary blocks of the function can be built. Even so, *optimal* yield placement is tricky. I will describe three simple heuristics here, one of which I have implemented in my prototype compiler; after I discuss dynamic checks I will be able to formulate a fourth.

Yield-on-Jump The most naïve local yield placement strategy, but the simplest to implement, is to assume that every local jump will involve a yield. This can be accomplished either by assuming a virtual clock of zero at the start of every block, or by assuming a virtual clock of $Y - 1$ at the start of every block. In the former case, the first instruction in every block must be a yield; in the latter, the last instruction before every jump must be a yield.

Because these *yield-on-jump* strategies treat every block and every jump the same, making no use of one's static knowledge of each jump's target, it is easy to see that they place more yields than necessary. Figure 6.1, for example, shows a flow graph corresoponding to two Lilt blocks and containing one join point. (In Lilt, the extended basic block consisting of basic blocks A1, A2 and A3 is thought of as a single block.) If all of these basic blocks are short, and none of them contains any function calls (so that global yield placement does not affect the example), then it may be unnecessary to yield at the start of block B. In general, yield-on-jump appears to be badly behaved for acyclic Lilt functions that contain several blocks. The next two candidate strategies attempt to do better on acyclic functions by propagating approximate timing information between blocks.

Forward Propagation For the other two local yield placement heuristics I will consider, it is necessary to distinguish between *forward* and *backward* jumps. Specifically, I assume a total ordering

on the blocks in a function; a jump whose target is a *later* block than the one where the jump appears is called a forward jump, and one whose target is an earlier block, or the very one in which the jump occurs, is called a backward jump. Note that if the flow graph of a function is acyclic, then it is possible to arrange the ordering such that all jumps are forward; in a function containing loops, every loop necessarily contains at least one backward jump. Loops are a source of difficulty for local yield placement, since my system (probably) lacks the expressive power to avoid yielding at least once per iteration, so I expect that my heuristics will give the best results when the ordering on blocks minimizes the number of backward jumps. Rather than attempt to find such an ordering, however, I will simply use the order in which the blocks appear in the Lilt representation of the function.[1]

The first nontrivial local yield placement heuristic is based on the operation of propagating clock information forward through a block as code for the block is generated. The process is basically intuitive: starting with an initial assumption about the virtual clock at the start of the block, generate the instructions for the block, tracking the decrements to the virtual clock with each instruction. (The global yield placement strategy will determine the effect function calls have on the clock.) If the clock ever reaches zero (or becomes inconveniently small for any operation that must be compiled), insert a `yield` and reset it to Y. At each leaf of the extended basic block, one is faced with either a `return`, a `raise` or a `goto` and a certain predicted value on the clock. In each of these cases it may or may not be necessary to yield before the transfer of control. In the case of `return` and `raise`, the decision is made based on the global and exceptional placement strategies in use, respectively. It therefore remains only to show how to handle `goto`.

The *forward-propagation* method generates code for a function as follows. Compile the blocks in order, starting with the entry block of the function. The initial condition of the entry block is determined by the global placement strategy; the initial condition of a handler block is determined by the exceptional placement strategy. For ordinary blocks, note that by the time we compile a block (labeled by) ℓ, all forward jumps to ℓ have already been compiled. Therefore, these blocks may be handled using three rules:

- Do not yield before a forward jump, unless a yield is necessary to accomodate the `jmp` instruction itself.
- The initial condition for each ordinary block ℓ is the minimum virtual clock value seen at any forward jump to ℓ, adjusted to account for the jump instruction.
- For backward jumps, the target block has already been compiled; determine whether to yield before jumping based on the target block's initial condition.

This approach has the advantage that, although every loop needs at least one yield, there may not need to be a yield at every backward edge if the initial assumption at the top of the loop is small enough. It also may be more algorithmically convenient to use this heuristic, which processes blocks in a forward direction, than the next one in which blocks are scanned backwards.

Backward Propagation The forward propagation method started with an initial assumption about each block and determined what the block could guarantee at each leaf. It is also possible to place yields by starting with the *requirement* at each leaf of a block, and propagating *backward* to determine the requirement at the block's beginning. To do this, the instructions for each basic

[1] The problem of finding an optimal ordering is NP-hard (stated without proof by Manber [43], p. 429).

block must be generated in reverse order, incrementing the requirement (rather than decrementing an assumption) with each instruction until the value reaches Y. When this happens, a `yield` is inserted and the requirement is reset to zero. Conditional expressions within extended basic blocks require conservative approximation: the requirement before an `if` or a `case` instruction is computed based on the maximum of the requirements of the branches.

To generate code for a function using the backward propagation method, compile the blocks in reverse order. For each leaf of each block, determine the *final requirement*: for `return` and `raise` this comes from the global and exceptional placement policies, as before. For jumps, there are two cases.

- If the jump is forward, then its target has already been compiled. The final requirement of the current basic block is then the target block's initial requirement, plus the cost of the `jmp` instruction. If this is greater than Y, insert a yield before the jump.
- If the jump is backward, then insert a yield and assume a final requirement of zero.

Since the initial conditions of the exception handler blocks and of the function's entry block are not determined by local placement, it may be necessary to insert a `yield` at the beginnings of these blocks if the computed initial requirement exceeds this initial condition.

The backward propagation method has the advantage that it does not require tracking any additional information, whereas for forward propagation one has to remember the minimum clock value associated with each forward jump until the target block is compiled. However, the assumption that every `return` has the same requirement does not mesh well with the global placement strategy I developed for my compiler. I therefore used forward rather than backward propagation for local yield placement.

6.2 Global Placement with Call-Return Yielding

Global, or *inter*procedural, yield placement differs from local placement in that function pointers are first-class values in Lilt, and therefore for some call sites it may not be statically obvious which function is being called. Thus, finding a guaranteed optimal placement of yield points would seem to require interprocedural control flow analysis. Fortunately, I know of at least two global yield placement strategies that do not require this complexity: these methods treat all functions and all function call sites equally, avoiding the need to match up function calls with their targets. I will describe these two strategies, which I call *call-return yielding* and *Feeley yielding*, before moving on to discuss yield placement for the exception handling features of Lilt.

It is possible to devise a global placement heuristic that relies on only a small portion of the TALT-R type system. First, note that the inclusion of a term for `ck` in the register file type allows one to specify the time on the virtual clock at the start and end of a function, similarly to TALres [18]. For instance, the type

$$\forall\rho{:}\mathsf{TD}.\ \{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}(\{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}\rho, \mathtt{ck}{:}\overline{k_2}\} \to 0) \times \rho, \mathtt{ck}{:}\overline{k_1}\} \to 0$$

describes a function that takes an integer argument (in `eax`) and returns an integer (also in `eax`); further, this function may be called whenever there is at least $k_1 + 1$ on the virtual clock and is guaranteed to return with at least k_2 remaining. Unlike in TALres, however, this function may be called at any time (assuming that $0 \leq k_1 < Y$): if the value of the virtual clock at the desired call site is not known to be at least $k_1 + 1$, the caller simply yields before making the call, resetting

the virtual clock to Y. Similarly, if k_2 is not enough time for the caller to complete its own work, it has only to yield after the function returns. Furthermore, by similar arguments (and with the added assumption that $0 \leq k_2 < Y$), *any* function may be made to satisfy these timing properties by proper local yield placement (which, as discussed above, may include inserting `yield` instructions at the function's beginning and end).

As an interesting special case, consider setting $k_1 = k_2 = 0$ for every function in a program. This forces the first instruction of each function's body, and the instruction immediately following each `call` instruction, to be a `yield`, so I call this scheme ***call-return yielding***. (Choosing $k_1 = k_2 = Y-1$ would have a similar effect, except that the yields would need to occur just before, rather than just after, the jumps.) Call-return yielding is simple, but it is far from optimal if Y is large compared to the running time of most functions (a reasonable assumption). If some functions are very short compared to Y, it would be safe to perform several calls to these functions in succession with no yields at all, but the call-return strategy incurs the cost of the yield operation at least twice per call.

6.3 Global Placement with Feeley Yielding

It is possible to improve over call-return yielding by giving types to functions that more precisely capture their timing behavior. For example, by analogy with TALres, we might write the type

$$\forall a{:}\mathsf{N}.\forall \rho{:}\mathsf{TD}.\ \{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}(\{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}\rho, \mathtt{ck}{:}a\} \to 0) \times \rho, \mathtt{ck}{:}\overline{k} + a\} \to 0$$

to describe a function that takes time k. Quantifying over the amount of time remaining on return expresses the fact that this function returns with all but k of its initial virtual clock remaining, whatever that value happens to be. There is a problem, however: a function of this type cannot yield! To see why, note that the function must execute its return instruction with $a + 1$ remaining on the virtual clock; but as far as the function knows, a could be *any natural number*. In particular, a might be larger than Y—but Y is the largest clock value the function can ever ensure after it has performed a yield instruction.

In reality, of course, a will never be larger than Y; in fact, the initial clock value of $k + a$ can be at most $Y - 1$. Hence, if the function yields, the resulting clock value of Y is guaranteed to be greater than or equal to $a + 1$, allowing the function to return. As discussed in Section 3.3.3, code blocks in MiniTALT-R are permitted to depend on constraint assumptions; the addresses of such blocks are given guarded types so that they cannot be executed unless the constraints are satisfied. For example, if I decide the type of a function should be

$$\forall a{:}\mathsf{N}.\forall \rho{:}\mathsf{TD}.\ (\overline{k} + a \leq \overline{Y - 1}) \Rightarrow \{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}(\{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}\rho, \mathtt{ck}{:}a\} \to 0) \times \rho, \mathtt{ck}{:}\overline{k} + a\} \to 0$$

(the same type as the previous attempt at a function of cost k, except for the guard), then I add the hypothesis $(\overline{k} + a \leq \overline{Y - 1})$ true to the static context when typing the function's code. This hypothesis will then be available for use in proving formulas true within the function body. In particular, in order for the function to return after a yield, I need to show that $\overline{1} + a \leq \overline{Y}$. This is especially easy when $k \geq 1$, since (using the ordering axioms, monotonicity and transitivity) I can reason as follows:

$$\overline{1} + a \leq \overline{k} + a \leq \overline{Y - 1} \leq \overline{Y}$$

As a matter of fact, a function with the above type need not yield immediately before it returns, because a stronger fact holds:

Proposition 6.1 *If* $0 < k \leq Y$, *then* $(a{:}\mathsf{N}, (\overline{k} + a \leq \overline{Y - 1})\ \mathsf{true}) \vdash \overline{1} + a \leq \overline{Y - k}\ \mathsf{true}$.

Proof Sketch: Let Δ be the context in the judgment to be derived. Using commutativity, the addition axiom and reflexivity of ordering, $\Delta \vdash \overline{k-1} + (\overline{1} + a) \leq \overline{k} + a\ \mathsf{true}$. Using the addition axiom and reflexivity of ordering, $\Delta \vdash \overline{Y-1} \leq \overline{k-1} + \overline{Y-k}\ \mathsf{true}$. Invoking the hypothesis in Δ and using transitivity twice, we get $\Delta \vdash \overline{k-1} + (\overline{1} + a) \leq \overline{k-1} + \overline{Y-k}\ \mathsf{true}$. By the cancellation rule, $\Delta \vdash \overline{1} + a \leq \overline{Y-k}$ as required.

Alternatively, observe that the formulas on the left and right of the turnstile in this judgment have the same interpretation as polynomial constraints in the sense of Chapter 4, namely $(a + \overline{-Y + k + 1} \leq 0)$. The soundness results of Chapter 4 imply that the judgment is derivable (and is in DLP_1).
End of Sketch.

A consequence of this proposition is that a function with the type given above may execute up to k instructions between its last `yield` and its final `ret`. If j instructions have been executed since the last `yield` and $j \leq k$, then the virtual clock will read $Y - j$. It follows that $Y - j \geq Y - k \geq 1 + a$, making a return instruction well-typed.

As was the case in our discussion of call-return yielding, the function type just examined does not bound the number of instructions executed by a function. It merely guarantees that any function of that type that takes more than k instructions will yield after executing at most k instructions, and that if such a function does yield, the last time it does so is at most k instructions before it returns. By placing yields appropriately, any function can be made to obey these criteria.

Once again, an interesting special case arises if the value of k is fixed for all functions in the program: in this case, the result is essentially the yield-placement strategy described by Feeley [23]. Feeley, whose motivation was placing checkpoints in a program to detect interrupts, named his strategy *balanced polling*. (Feeley also inspired my use of the term *call-return yielding*.) I choose to refer to the yielding scheme I have just described as ***Feeley yielding***, and I follow Feeley in using the letter E to denote the fixed value we have chosen for k. The major advantage of Feeley yielding is that functions that contain no loops or function calls and are shorter than E instructions need not yield at all (whereas in call-return yielding *every* function must yield). Further, from the caller's point of view, any function appears to cost exactly E instructions. Thus if E is small enough compared to Y, several function calls may occur in succession without the caller having to yield in between.

A sample MiniTALT-R program fragment using the Feeley yielding strategy is shown in Figure 6.2. The function in the figure is a recursive function to compute Fibonacci numbers; it was hand-coded in MiniTALT-R and is displayed in approximately Intel assembler syntax. Note that the function has a "short path" corresponding to the case where the argument is less than or equal to 1, and a "long path" that performs two recursive calls if it is not. Notice that the short path does not need to yield (of course, this depends on E being chosen large enough). The long path must yield before the first recursive call, and between the last call and the final return instruction. This is typical of Feeley yielding, since any function might start out with as little as E on the clock, but any callee requires at least E; similarly, no callee can be assumed to return with more than $Y - E - 1$ on the clock, but the caller cannot return without at least $Y - E$. Notice, however, that no yield is needed in between the two recursive calls (again assuming appropriate values for Y and E).

Note: this example assumes that $E \geq 4$ and that $Y \geq 2E + 8$.

```
fib:
     // ck : E + a, (E + a  ≤  Y − 1)   true
     cmp eax,1
     ja L1                          // n  ≤  1?
     mov eax,1
     // ck : E − 3 + a
     ret                            //   Return 1
L1:
     // ck : E − 2 + a
     push eax
     sub eax,1
     // ck : E − 4 + a
     yield
     // ck : Y
     call fib                       // Compute fib(n-1)
     // ck : Y − E − 1
     pop ecx
     push eax
     mov eax,ecx
     sub eax,2
     // ck : Y − E − 5
     call fib                       // Compute fib(n-2)
     // ck : Y − 2E − 6
     pop ecx
     add eax,ecx                    // eax := fib(n-1)+fib(n-2)
     // ck : Y − 2E − 8
     yield
     // ck : Y
     ret                            // Return
```

Figure 6.2: Fibonacci using Feeley Yielding

6.4 Exceptional Placement

A simple heuristic suffices for exceptional yield placement. In particular, since it is often unknown at the site of a `raise` expression which handler is being invoked, the best solution is probably to use a fixed initial assumption for all handler blocks and treat `raise` expressions accordingly. If the initial condition of all exception handlers is taken to be H, then the requirement to generate a `raise` is clearly H plus the cost of raising the exception (a few instructions).

There is room for clever improvement of this method: if a `raise` occurs in a context where the current handler can be statically predicted, then it may be possible to avoid yielding before raising the exception if the handler block is short; however, if a handler might be invoked in a context where its identity is unknown, its initial requirement had better be at most H. It does not seem likely that any serious advantage can be gained from this flexibility, so I have not investigated it.

6.5 Clocks and Polling

The yield placement strategies I have discussed are straightforward and easy to implement, but they fall well short of the ideal goal of yielding exactly once for every Y other instructions executed. The reason is that, while the changes in the virtual clock can be precisely tracked over straight-line code or tree-structured code, this precision cannot be carried across extended basic block boundaries. Once the yield period Y is larger than the length of the longest extended basic block in the program, one cannot expect that increasing it any more will continue to lower the actual frequency with which the program will yield under these strategies.

One possible direction for further refinement is to enrich the static reasoning capabilities of the TALT-R type system, so that it can capture more and more complex coding idioms, including loops and recursion. This is the approach taken in LXres, where the equivalent of the static term language includes sum, product and inductive kinds (inherited from LX [17]) and primitive recursion in addition to basic arithmetic. Unfortunately, the potential benefits of this kind of system are difficult to realize without significant contributions from the programmer. Fundamentally, any improvement along the static reasoning axis involves two tightly coupled areas of simultaneous development: that of more and more sophisticated *program analyses* to detect opportunities for avoiding yields, and that of more and more expressive *type systems* to certify that the resulting optimized programs are still safe.

An alternative to improving the static reasoning capabilities of the language is to rely to some extent on *dynamic* mechanisms. That is, rather than implementing static analyses and compiler passes that safely hoist yield instructions out of loops, one can generate programs that keep track of time *as they run* and yield only when needed. Of course, some static reasoning is needed to certify the correctness of the instruction counting, but it turns out that this is not difficult. In fact, it is substantially easier than beefing up the type system's logical power to the point of being able to handle real programs.

Here is the idea: Let the program use one of the machine's general-purpose registers to maintain a dynamic approximation of the number of instructions remaining until the next yielding operation is due to occur. This approximation is maintained by periodically subtracting from the register until it becomes zero (or inconveniently close to zero); when that happens, the program must assume it has run out of time. It yields, resets the register and continues. I call this behavior ***polling.***

6.5.1 Clocks

To implement polling, I reserve one general-purpose register for timing purposes. I will use the name `rck` for this register and refer to it as the *clock register* (to distinguish it from the pseudoregister `ck`, the *virtual clock*). Note that although I give a descriptive name to the clock register for the sake of presentation, there is nothing special about this register as far as the type system is concerned. In fact, it is not strictly necessary to store the value of the clock register in a register at all: it would also be reasonable to stack-allocate it and save the register for other uses. It is perhaps most helpful to think of the name '`rck`' as referring to the *role* played by a certain register, rather than to the register itself.

The purpose of the clock register is to store an approximation of the number of clock cycles left before the next yield must happen. In particular, programs will maintain the invariant that the value of `rck` is always less than or equal to the value of the virtual clock. A special significance is attached to the *difference* between these two quantities (or the best available static approximation thereof): this is the maximum number of nonyielding instructions the program can execute before some action must be taken to maintain the invariant, either by yielding or by decreasing the value of `rck`.

For simplicity, let us assume that updates to the clock register occur in a highly stereotyped pattern: At points in the program where the virtual clock value cannot be proven to exceed the value of `rck`, a certain fixed quantity, call it L, is subtracted from the register. If the new value is negative, then the program assumes it has run out of time, performs a `yield`, and sets the register to $Y - L$; if it is nonnegative, then the virtual clock now exceeds the register's value by at least L and execution can proceed. The technique will be most effective if L is close to the length of the longest extended basic block in the program, since it is at extended basic block boundaries that precision tends to be lost.

The effect of this technique on yield timing is depicted in Figure 6.3. The graph on top shows value of the virtual clock as a function of time during the imagined execution of some program compiled using direct yield placement as described earlier. The downward sloping portions of the graph show the steady ticking of the virtual clock as nonyielding instructions are executed; the virtual clock value jumps back to Y each time the program yields. Clearly, this program makes rather ineffective use of the time it is given between yields. The graph on the bottom shows the same program, modified to use polling: each program point that performed a yield in the upper graph now decrements the instruction counter instead, yielding only when that value gets close to zero. Each decrement subtracts L from the counter; since in this picture Y is approximately $4L$, only every fourth clock check results in a yield. In practice, the ratio Y/L is much larger than four, giving an even more dramatic decrease in yield frequency.

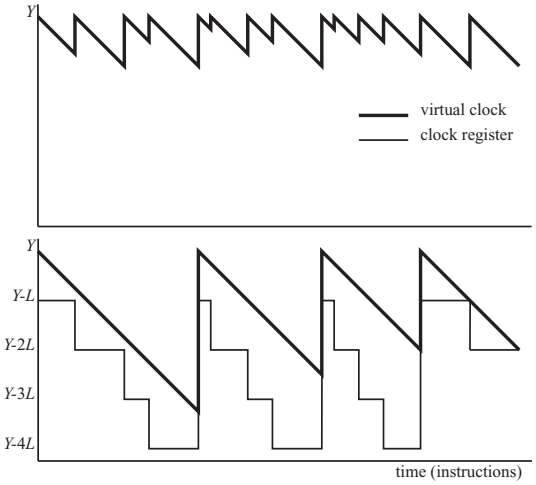


Figure 6.3: Yields Under a Polling Strategy

In general, if $Y = M \cdot L$, then each yield period (of Y instructions) can be thought of as M ***minor yield periods*** of L instructions each. The act of decrementing the clock register and yielding

if necessary is performed at least once per *minor* yield period, and every M'th time incurs an ordinary yield. To highlight this relationship, I call the sequence of instructions that updates the clock register a ***minor yield;*** the one of every M minor yields at run time that must peform a `yield` instruction is called a ***major yield.*** The task of yield placement is now reduced to the placement of minor yields; they must occur at least once every L instructions, and we will see that reasoning about the amount of time remaining before the next minor yield is not much harder than reasoning about the virtual clock itself. Since L is much closer to the lengths of actual basic blocks than Y, the loss of precision associated with each join point in a program will be smaller. Moreover, the cost of a minor yield is so much less than that of a major yield that the overhead of the instruction counting is insignificant.

```
YIELD =
     // a:N, rck:S(a), ck:2̄ + a
     subjae rck,rck,(L + 2),end
     // rck:int, ck:a
     yield
     // ck:Ȳ
     mov rck,(Y-L-3)
     // a′ ↦ Y − L − 3; rck:  S(Y − L − 3̄),
     // ck:Y − 1̄ = L̄ + (2̄ + a′)
end:
     // a′:N, rck:S(a′), ck:L̄ + (2̄ + a′)
```

Figure 6.4: Code for a Minor Yield

6.5.2 Minor Yields

A MiniTALT-R implementation of a minor yield is shown in Figure 6.4. Ignoring the type annotations for a moment, the effect of this code is clear. The `subjae` instruction decrements the clock register by $L + 2$. If the result is nonnegative, then execution continues at the label `end`; if the result of the subtraction is negative, a true yield is performed before `end` is reached. The typing annotations show that if, for some static term a, the clock register initially holds the value a and the virtual clock shows $\overline{2} + a$ remaining, then the code after the `end` label may assume that the clock register contains some value a' such that the virtual clock reads $\overline{L} + (\overline{2} + a)$. I will use the name `YIELD` to refer to this code sequence.

6.5.3 The Minor Clock

The informal description of the relationship between the clock register and the virtual clock must now be made precise. In order to ensure that a minor yield is always possible, programs maintain the invariant that the clock register `rck` always has some singleton type $\mathcal{S}(t)$ and the static approximation to the virtual clock is always $t' + (\overline{2} + t)$ for some other term t'. When this is the case I will say t' is the value of the ***minor clock.*** Intuitively, the minor clock captures the number of instructions that may be executed before the next minor yield. Notice that in straight-line code, the minor clock behaves just like the virtual clock in the sense that it decrements with every instruction (provided it is initially positive). More formally, the following rule for the `add` instruction is

derivable:

$$\frac{\begin{array}{c}(\Gamma(\mathtt{rck}) = \mathcal{S}(t))\ (\Gamma(\mathtt{ck}) = (\overline{1} + t') + (\overline{2} + t)) \\ \Delta; \Psi; \Gamma \vdash o_1 : \mathtt{int} \quad \Delta; \Psi; \Gamma \vdash o_2 : \mathtt{int} \\ \Delta; \Psi; \Gamma \vdash d : \mathtt{int} \to \Gamma' \quad \Delta; \Psi; \Gamma'\{\mathtt{ck}{:}\overline{t'} + (\overline{2} + t)\} \vdash I\end{array}}{\Delta; \Psi; \Gamma \vdash \mathtt{add}\ d, o_1, o_2; I}$$

This rule shows how to type an `add` instruction when the assumption that the minor clock is $\overline{1} + t'$; note that as long as the destination d is not `rck`, the continuation I will be typed under the assumption that `rck` still has type $\mathcal{S}(t)$, meaning that the new minor clock is just t'. Similar "minor clock rules" can be derived for all the instructions of TALT-R except for `yield`. Furthermore, the typing annotations in Figure 6.4 suggest that (if one ignores the syntactic inconvenience that it involves multiple blocks in MiniTALT-R), `YIELD` essentially acts like an instruction with a typing rule like the following:

$$\frac{(\Gamma(\mathtt{ck}) = \overline{2} + t) \quad \Delta; \Psi; \Gamma \vdash \mathtt{rck} : \mathcal{S}(t) \quad (\Delta, a{:}\mathsf{N}); \Psi; \Gamma\{\mathtt{rck}{:}\mathcal{S}(a), \mathtt{ck}{:}\overline{L} + \overline{2} + a\} \vdash I}{\Delta; \Psi; \Gamma \vdash \mathtt{YIELD}; I}$$

This rule states that `YIELD` has the effect of turning a state with *any* minor clock value into one where the minor clock is L—but it may change the value of the clock register.

As I have mentioned, the fact that `YIELD` behaves so much like `yield` means that the local, global and exceptional placement strategies I previously discussed for `yield` should also work for `YIELD`, tracking the minor clock instead of the virtual clock and placing yield points every L instructions instead of every Y. When a yielding strategy is adapted to placing minor yields, I call it a ***polling strategy***. For example, recalling the type of a function under Feeley yielding,

$$\forall a{:}\mathsf{N}.\forall \rho{:}\mathsf{TD}.\ (\overline{E} + a \leq \overline{Y - 1}) \Rightarrow \{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}(\{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}\rho, \mathtt{ck}{:}a\} \to 0) \times \rho, \mathtt{ck}{:}\overline{E} + a\} \to 0$$

and modifying it so that it specifies the function's behavior with respect to the minor clock instead of the virtual clock, one gets the type of a function under ***Feeley polling***:

$$\begin{array}{l}\forall a{:}\mathsf{N}.\forall b{:}\mathsf{N}.\forall \rho{:}\mathsf{TD}.\ (\overline{E} + a \leq \overline{L - 1}) \Rightarrow \\ \quad \{\mathtt{eax}{:}\mathtt{B4}, \mathtt{rck}{:}\mathcal{S}(b), \mathtt{esp}{:}(\forall b'{:}\mathsf{N}.\{\mathtt{eax}{:}\mathtt{B4}, \mathtt{rck}{:}\mathcal{S}(b'), \mathtt{esp}{:}\rho, \mathtt{ck}{:}a + (\overline{2} + b')\} \to 0) \times \rho, \\ \quad \mathtt{ck}{:}(\overline{E} + a) + (\overline{2} + b)\} \to 0\end{array}$$

Notice that, while under Feeley yielding a function called with $E + a$ on the virtual clock returns with a on the virtual clock, under Feeley polling a function called with $E + a$ on the *minor* clock returns with a on the minor clock. Notice also that the function may change the value of the clock register; the code at the return address must be well-typed for any possible value on the clock register, assuming only the relationship between the register and the virtual clock that defines the minor clock.

Figure 6.5 shows the Fibonacci function from Figure 6.2 implemented with Feeley polling. This new function has the type given above, and its code is exactly the same except that `yield` instructions have been replaced by the `YIELD` macro. Notice that every `YIELD`, and every recursive call, may change the value of the clock register.

Note: this example assumes that $E \geq 4$ and that $L \geq 2E + 8$.

```
fib:
     // a, b0 : N, rck:S(b0),
     // ck : (E + a) + (2 + b0), (E + a  ≤  L − 1)   true
     cmp eax,1
     ja L1                          // n  ≤  1?
     mov eax,1
     // ck : (E − 3 + a) + (2 + b0)
     ret                            //   Return 1
L1:
     // ck : (E − 2 + a) + (2 + b0)
     push eax
     sub eax,1
     // ck : (E − 4 + a) + (2 + b0)
     YIELD
     // b1 : N, rck:S(b1)
     // ck : L + (2 + b1)
     call fib                       // Compute fib(n-1)
     // b2 : N, rck:S(b2)
     // ck : L − E − 1 + (2 + b2)
     pop ecx
     push eax
     mov eax,ecx
     sub eax,2
     // ck : L − E − 5 + (2 + b2)
     call fib                       // Compute fib(n-2)
     // b3 : N, rck:S(b3)
     // ck : L − 2E − 6 + (2 + b3)
     pop ecx
     add eax,ecx                    // eax := fib(n-1)+fib(n-2)
     // ck : L − 2E − 8 + (2 + b3)
     YIELD
     // b4 : N, rck:S(b4)
     // ck : L + (2 + b4)
     ret                            // Return
```

Figure 6.5: Fibonacci using Feeley Polling

```
YIELD(F) =
     // a:N, rck:S(a), ck:F̄ + (2̄ + a)
     subjae rck,rck,(L − F + 2),end
     // if taken: rck:S(a′), a = a′ + L − F + 2̄ true,
     //           ck:(F̄ + a) = F̄ + L − F + 2̄ + a′ = L̄ + (2̄ + a′)
     // otherwise: rck:int, ck:F̄ + a
     yield
     // ck:Ȳ
     mov rck,(Y-L-3)
     // a′ ↦ Y − L − 3̄; rck: S(Y − L − 3̄), ck:Y − 1̄ = L̄ + (2̄ + a′)
end:
     // a′:N, rck:S(a′), ck:L̄ + (2̄ + a′)
```

Figure 6.6: A Minor Yield with F on the Clock

```
YIELD(F, R) =
     // a:N, rck:S(a), ck:F̄ + (2̄ + a)
     subjae rck,rck,(R − F + 2),end
     // if taken: rck:S(a′), a = a′ + R − F + 2̄ true,
     //           ck:(F̄ + a) = F̄ + R − F + 2̄ + a′ = R̄ + (2̄ + a′)
     // otherwise: rck:int, ck:F̄ + a
     yield
     // ck:Ȳ
     mov rck,(Y-R-3)
     // a′ ↦ Y − R − 3̄; rck: S(Y − R − 3̄), ck:Y − 1̄ = R̄ + (2̄ + a′)
end:
     // a′:N, rck:S(a′), ck:R̄ + (2̄ + a′)
```

Figure 6.7: Resetting the Clock from F to R

6.5.4 Tricks With Polling

In addition to reducing the difference between the "yield" period and basic block size, polling allows more precision than ordinary yielding because one has control over how much the clock register is decremented with every minor yield. For example, it seems to occur frequently that a yield must be placed at a location where there is known to be some time left on the clock. In an explicit polling scheme, one can take advantage of this by decrementing the clock register by a smaller amount—in effect, saving the unused cycles so that they can be used later. The code in Figure 6.6 illustrates this.

Of course, it is also possible to decrement the clock register by *more* than $L + 2$. In fact, there is no reason at all that the minor clock must be reset to L at every minor yield; if one finds oneself at the beginning of a basic block that is of length R (where $R \leq Y-3$), then one can subtract $R+2$ from `rck` and set the minor clock to exactly what the current block requires. This is accomplished by the code sequence `YIELD`(F, R) defined in Figure 6.7. Note that the first two forms of minor yield are really special cases of this last one: `YIELD`(F) is simply `YIELD`(F, L), and the `YIELD` from

Figure 6.4 is `YIELD`$(0, L)$. The formal translation in the next section will use the two-argument notation exclusively.

Using this precise minor yield in conjunction with yield-on-jump and call-return yield placement strategies results in a polling strategy that I call *precise yield-on-jump*. Under this strategy, every basic block in the program begins with a minor yield that "reserves" exactly the right number of minor clock cycles for that block. While this does introduce more minor yields than would be needed under, say, forward propagation and Feeley polling, it eliminates all of the error associated with join points. The only "lost cycles" now occur at major yields. A major yield happens when the cycles remaining on the virtual clock (there will nearly always be some left) are insufficient for the current basic block; these left-over cycles cannot be used, but the waste is bounded by the length of the longest basic block in the program.

6.6 Chapter Summary

If programs written by programmers who are ignorant of timing requirements are to satisfy a timing policy like that of TALT-R, yielding operations must be inserted into those programs by the certifying compiler. This process must balance the absolute and inviolable requirements of the type system, which serves as the proxy for the safety policy, with the desire to yield as infrequently as possible for the sake of performance.

I have described a number of techniques and approaches to yield placement, ranging from the very simple to the fairly complex. The more complicated techniques are based not on advanced static analyses but on dynamic instruction counting, an easily understood mechanism that offers low yield frequencies with a fairly small investment in type system complexity and low performance overhead.

Chapter 7

Compilation of Lilt

In this chapter, I will finally give a formal translation from Lilt to MiniTALT-R. The purpose of this formal translation is twofold. First, since it relates any well-typed Lilt program to an equivalent assembly language program, it resolves any ambiguitiy there may have been in my prose description of the semantics of Lilt language constructs. (Of course, giving an operational semantics for Lilt directly would have served the same need.) Second, and more importantly, it allows me to argue that the type system I propose for MiniTALT-R is sufficiently general to support all the constructs and idioms of a typical high-level programming language. In particular, it demonstrates that the polling technique I described in Section 6.5 is flexible enough that resource bound certification need not get in the programmer's way.

The translation I give here uses Feeley polling for *inter*procedural yield placement, but is nondeterministic with respect to local yield placement. In other words, there are many different ways to translate any Lilt function, differing in the number and location of minor yields in the MiniTALT-R code. An actual implementation of this translation must resolve the nondeterminism using a heuristic such as the ones I described earlier in this proposal. (The prototype compiler I have implemented uses forward propagation.)

Although the implications of polling are the main point of this proposal, the formal translation I give in this chapter addresses *all* aspects of type-directed compilation of Lilt. In particular, I give a complete translation from Lilt types to TALT-R types, and I show how to compile all the primitive operations of Lilt. This makes the translation as a whole rather technical. Before giving the translation rules themselves, therefore, I must take some time to introduce some conventions and notation.

7.1 Type-Directedness

Formal translations between languages generally come in two flavors: syntax-directed and type-directed.[1] Syntax-directed translations are the more naïve variety: they are defined recursively (that is, by induction) over the syntax of the source language, generally using little or no context information. A syntax-directed translation usually applies to any term, well-typed or not; the static correctness theorem for the translation states that if a source term is well-typed, then its

[1]There is a third type, called an *elaboration*, that differs from both of these in that it is used to define the static semantics of the source language in terms of the target. The archetypical elaboration is the Harper-Stone interpretation of Standard ML [33]; the translation of EXTALT-R to XTALT-R performed by the certifying assembler (but not discussed in this thesis) is an elaboration.

translation is well-typed. On the other hand, type-directed translations are (roughly speaking) defined by inference rules that are constructed to closely mirror the typing rules of the source language; they are often thought of as being defined by induction over typing derivations, rather than over terms. Because of this, it is usually very easy to prove that a term may be translated if and only if it is well-typed, and not very difficult in principle to prove that its translation is well-typed in the target language.

Although a syntax-directed translation is often simpler to define and implement, there are many cases where it simply does not make sense to use one. For instance, if the way a term is translated ever depends on the type of one of its subterms, then it is usually advisable to define the translation by induction on typing rather than syntax. Type-directed translations are also called for when the target language is explicitly typed, particularly if the target requires typing annotations in places where the source language does not. This latter case clearly arises when translating a typed language like Lilt into explicitly-typed assembly language: the assembly code for, say, a conditional statement will contain at least one label, which must be annotated with a type even though the relevant typing information is not explicitly present in the source program.

It may be a little surprising, then, that Lilt may (I conjecture) be translated to MiniTALT-R by a syntax-directed translation. This is so because MiniTALT-R (as opposed to EXTALT-R) is implicitly typed, so the translation does not have to generate any typing annotations. Furthermore, it happens to be the case that the (concrete) machine instructions implementing any Lilt expression can be computed independently of the types of any of its subterms. However, the translation I give in this chapter is supposed to be an abstract stand-in for the one implemented by my compiler, and that implementation targets EXTALT-R, not MiniTALT-R; because of the explicit typing annotations (and coercions) needed in EXTALT-R, my actual Lilt compiler is type-directed. Therefore, I give a type-directed translation in this chapter even though doing so renders the presentation a good deal less concise. I will use the context and typing information available in the setting of a type-directed translation to annotate the MiniTALT-R output with typing information for labels, even though such annotations are not officially part of MiniTALT-R. This will hopefully help make the intended meaning of the generated code more clear.

7.2 Conventions and Notations

7.2.1 Variable Naming

For the purposes of my translation from Lilt to TALT, I will make some assumptions about local variable names. First, I assume that local variable names have the following syntax:

$$s ::= \texttt{arg}(i) \mid \texttt{loc}(i)$$

Second, I assume that the context specifying a function's formal parameters has the form $\Gamma_a = [\texttt{arg}(1){:}\tau_1, \ldots, \texttt{arg}(m){:}\tau_m]$ and that the list of local variables declared by the function's entry block is always $\texttt{loc}(1), \ldots, \texttt{loc}(n)$. Note that I make these assumptions without any loss of generality, since any Lilt function may be α-varied into this form. With these conventions in place, the name of a local storage location s identifies it as either a function argument or a local variable, and I will show shortly how the TALToperand or destination corresponding to a location may be determined based on its name. Furthermore, it is no longer necessary to write the names of the arguments and local variables where they are declared at the start of the function, so to save space I will write

$$\texttt{func}(\Delta; [\tau_1, \ldots, \tau_A]; \tau).(\texttt{enter}(L).e, \ell_1 = B_1, \ldots, \ell_m = B_m)$$

$$
\begin{array}{rcl}
|T| & = & \mathtt{T4} \\
|k_1 \to k_2| & = & |k_1| \to |k_2| \\
 & & \\
|\alpha| & = & \alpha \\
|\mathtt{int}| & = & \mathtt{B4} \\
|\mathtt{bool}| & = & \mathtt{B4} \\
|\mathtt{unit}| & = & \mathtt{B4} \\
|\langle \tau_1, \ldots, \tau_n \rangle| & = & \mathtt{mbox}(|\tau_1| \times \cdots \times |\tau_n|) \\
|[i_1{:}\tau_1, \ldots, i_n{:}\tau_n]| & = & \mathtt{box}(\mathtt{set}_{=}(i_1) \times |\tau_1|) \vee \cdots \vee \mathtt{box}(\mathtt{set}_{=}(i_n) \times |\tau_n|) \\
|\tau\ \mathtt{array}| & = & \exists \alpha{:}\mathsf{N}.\mathtt{box}(\mathtt{set}_{=}(\alpha) \times \mathtt{mbox}(|\tau| \uparrow \alpha)) \\
|\forall \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau| & = & \forall \alpha_1{:}|k_1|. \ldots \forall \alpha_n{:}|k_n|.|\tau| \\
|\exists \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau| & = & \exists \alpha_1{:}|k_1|. \ldots \exists \alpha_n{:}|k_n|.|\tau| \\
|\lambda \alpha{:}k.c| & = & \lambda \alpha{:}|k|.|c| \\
|c_1\ c_2| & = & |c_1|\ |c_2|
\end{array}
$$

Figure 7.1: Translation of kinds and types (except function types)

instead of

$$\mathtt{func}(\Delta; [\mathtt{arg}(1){:}\tau_1, \ldots, \mathtt{arg}(A){:}\tau_A]; \tau).(\mathtt{enter}(\mathtt{loc}(1), \ldots, \mathtt{loc}(L)).e, \ell_1 = B_1, \ldots, \ell_m = B_m)$$

when I define the translation.

7.2.2 Minor Clock Notation

The translation uses a polling strategy for yielding, so all the code blocks in the MiniTALT-R output must make assumptions about the minor clock that are reflected in their types. To write these types, I will use some notation based on the fiction that there is a single register called `mck`, analogous to `ck`, that holds the value of the minor clock. In particular, for MiniTALT-R register file types Γ, define:

$$\Gamma[\mathtt{mck}_u \mapsto t] = \Gamma[\mathtt{esi} \mapsto \mathcal{S}(u), \mathtt{ck} \mapsto t + (\overline{2} + u)]$$

Here u (which will nearly always be a variable) is the constraint term representation of the clock register value. The register `esi` serves as the clock register. $\Gamma[\mathtt{mck}_u \mapsto t]$ is the register file type that specifies u on the clock register and t on the minor clock, and agrees with Γ on everything else. I will take the liberty of writing register files that specify a static term for `mck` in a similar way: $\{\mathtt{r}_1{:}\tau_1, \ldots, \mathtt{r}_n{:}\tau_n, \mathtt{mck}_u{:}t\}$ will denote the register file type $\{\mathtt{r}_1{:}\tau_1, \ldots, \mathtt{r}_n{:}\tau_n\}[\mathtt{mck}_u \mapsto t]$ as defined above.

7.3 Types and Data Representation

The translation of Lilt kinds and type constructors is defined in Figures 7.1 and 7.2. The translation of kinds is nearly trivial; the only point of interest is that the Lilt kind T is translated as $\mathtt{T4}$, which

$$|(\tau_1, \ldots, \tau_m) \to \tau| \quad = \quad \forall\rho_1{:}\mathsf{TD}.\forall\rho_2{:}\mathsf{TD}.\forall\alpha_f{:}\mathsf{T4}.\forall\alpha_h{:}\mathsf{T4}.\forall a{:}\mathsf{N}.\forall b{:}\mathsf{N}.\ (\overline{E} + a \leq \overline{L-1}) \Rightarrow \{\mathtt{edi}{:}\tau_e, \mathtt{ebp}{:}\alpha_f, \mathtt{esp}{:}\tau_r \times \sigma_0, \mathtt{mck}_b{:}\overline{E} + a)\} \to 0$$

where:
$$\sigma_0 = |\tau_1| \times \cdots \times |\tau_m| \times \rho_1 \times \tau_h \times \rho_2$$
$$\tau_h = \alpha_h \wedge \forall b'{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau_{\mathrm{exn}}|, \mathtt{esp}{:}\rho_2, \mathtt{mck}_{b'}{:}\overline{H}\} \to 0$$
$$\tau_e = \mathtt{sptr}(\tau_h \times \rho_2)$$
$$\tau_r = \forall b''{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau|, \mathtt{edi}{:}\tau_e, \mathtt{ebp}{:}\alpha_f, \mathtt{esp}{:}\sigma_0, \mathtt{mck}_{b''}{:}a\} \to 0$$

Figure 7.2: Translation of function types

means that any Lilt value (since it has a type of kind T) will be represented by something that is 32 bits wide. In particular, my translation will not require any run-time type constructor analysis (as in [19, 17, 57]) to compute the sizes of values.

The translations of base types, products and quantified types are not surprising. Sum types are translated using TALT's singleton and union types: for instance, a value of type $[i_1{:}\tau_1, i_2{:}\tau_2]$ is *either* a pointer to a pair consisting of the number i_1 and a value of type τ_1 *or* a pointer to a pair consisting of the number i_2 and a value of type τ_2. The translation of array types also makes use of singletons: a value of array type is a pair whose first element is the length of the array and whose second element is a pointer to the array data itself.

Unsurprisingly, the treatment of function types is the most complicated part of the type translation, because the type of a function must completely capture not only the interprocedural yielding or polling strategy used by the compiler, but also the procedure calling and linkage conventions, which in the case of Lilt includes not only the passing of parameters and the return address and the saving of registers, but also the (interprocedural) exception handling mechanism. As the translation in Figure 7.2 indicates, a Lilt function expects to receive its arguments and return address on the stack, and returns its result in `eax`. The frame pointer register, `ebp`, is managed using a callee-saves discipline: its initial value, of the unknown type α_f, is restored upon exit from the function.

Our treatment of exception handling is very similar to that of the TALx86 Popcorn compiler [45], which in turn appears to be based on the canonical translation into STAL [46]. A Lilt function expects to be passed the current *exception pointer* in register `edi`. The exception pointer points to the current exception handler, which is stored in an unknown location on the stack. The type of the stack expected by the function, therefore, consists of the return address (of type τ_r), the m arguments, a portion of unknown type ρ_1, the exception handler (of type τ_h), and finally a tail of unknown type ρ_2. The handler itself is a pointer to code that can accept a stack of type ρ_2; therefore, to raise an exception one may simply move the exception value to be raised into `eax`, move the exception pointer from `edi` into `esp`, and execute a `ret` instruction.

The typing of the exception handler itself is a bit complicated: on the one hand, the function must be able to jump to the handler when raising an exception, but on the other hand, the function is responsible for returning the handler to its caller when it exits. The exception handler thus behaves both like an argument or return address (the function requires it to have a certain type) and like a callee-save register (the act of calling a function must not result in the loss of any information about the current handler). This kind of pattern usually calls for bounded quantification; rather than add this feature to TALT-R, I use a known trick for simulating it using ordinary universal quantification and intersection types [60, 12]. Intuitively, the parameter α_h is the "real" type

$$\kappa ::= \text{just}\, n \mid \text{retplus}\, n \qquad |\text{just}\, n|_a = \overline{n} \qquad |\text{retplus}\, n|_a = \overline{n} + a$$

$$\begin{aligned} (\text{just}\, n) - m &= \text{just}(n - m), \text{ if } n \geq m \\ (\text{retplus}\, n) - m &= \text{retplus}(n - m), \text{ if } n \geq m \end{aligned}$$

$$\begin{array}{ll} \text{just}\, n \geq \text{just}\, m & \text{iff } n \geq m \\ \text{just}\, n \geq \text{retplus}\, m & \text{iff } n - (L - E - 1) \geq m \\ \text{retplus}\, n \geq \text{just}\, m & \text{iff } n \geq m \\ \text{retplus}\, n \geq \text{retplus}\, m & \text{iff } n \geq m \end{array}$$

Figure 7.3: Clock Specifiers

of the exception handler; since the value pointed to by `edi` is of the intersection type τ_h, it has the unknown type α_h but is additionally bounded above by the right conjunct, which is the code pointer type the function requires the handler to have.

Finally, observe that the translation of function types specifies a dynamic polling discipline for yielding, as described in Section 6.5. The minor clock pre- and postconditions of a function are expressed using the minor clock notation just defined in Section 7.2.2. The value of the clock register when the function is called is the static term parameter b. The translated function type also specifies a Feeley-style placement strategy for minor yields: the minor clock upon entry to the function is assumed to be $\overline{E} + a$ (where a is a static term parameter), and it will be a when the function returns. The exception handler pointed to by `edi` is expected to require a minor clock of H. The numbers L, E and H are parameters of the translation and have the same meanings as in Chapter 6. Note that just like in my earlier discussion of polling, the return address and exception handler must not care about the exact value of the clock register.

7.4 Clock Specifiers

In MiniTALT-R code produced by the translation, the minor clock at any point within a function will have one of two forms: either it will be a constant, or it will be $\overline{n}+a$, where a is the amount that must be present when the function returns. So that the translation rules do not have to mention the variable a, Figure 7.3 introduces *clock specifiers*, which are a more abstract way of describing the minor clock. The clock specifier $\text{just}\, n$ corresponds to n on the minor clock; $\text{retplus}\, n$ means that the value of the minor clock is n plus whatever is required for the function to return. Given the variable a, $|\kappa|_a$ is the static term representation of the minor clock denoted by κ if the function must return with a on the clock.

The figure also defines the operation of decrementing a clock specifier by an integer constant ($\kappa - m$); note that this operation is not always defined. Finally, the partial order $\geq$ specifies the constraints on clock specifiers that can be soundly inferred. Subtraction and ordering of clock specifiers will be used in the translation rules to determine when minor yields are needed.

The partial ordering and decrement operation on clock specifiers express constraints that can be proven in the TALT-R constraint logic in the context of a translated function. In particular, the type of any code block in a function will associate with the variable a a constraint hypothesis

$(\overline{E} + a \leq \overline{L-1})$, which is enough to prove any TALT-R constraint derived from the clock specifier notation.

Lemma 7.1 *Let* $\Delta = (a{:}\mathsf{N},\ (\overline{E} + a \leq \overline{L-1})\ \mathsf{true})$. *Then:*

1. *If* $\kappa - m = \kappa'$ *then* $\Delta \vdash |\kappa|_a = \overline{m} + |\kappa|_a$, *and this constraint is in* DLP_0.
2. *If* $\kappa_1 \geq \kappa_2$ *then* $\Delta \vdash |\kappa_2|_a \leq |\kappa_1|_a\ \mathsf{true}$, *and this constraint is in* DLP_1.

Proof: Part (1) is trivial and is left to the reader to check.
For part (2), there are four cases:

Case 1: $\kappa_1 = \mathrm{just}\, n$, $\kappa_2 = \mathrm{just}\, m$ and $n \geq m$. Then $[\![|\kappa_2|_a \leq |\kappa_1|_a]\!] = (m - n \leq 0)$, which is in DLP_0 and hence in DLP_1.

Case 2: $\kappa_1 = \mathrm{retplus}\, n$, $\kappa_1 = \mathrm{retplus}\, m$ and $n \geq m$. Similar to the previous case.

Case 3: $\kappa_1 = \mathrm{retplus}\, n$, $\kappa_2 = \mathrm{just}\, m$ and $n \geq m$. Then

$$[\![|\kappa_2|_a \leq |\kappa_1|_a]\!] = (m - (n + a) \leq 0) = (-a + (m - n) \leq 0)$$

which is in DLP_0.

Case 4: $\kappa_1 = \mathrm{just}\, n$, $\kappa_2 = \mathrm{retplus}\, m$ and $n - (L - E - 1) \geq m$. In this case we must finally use the constraint hypothesis in Δ. Note that

$$[\![\Delta]\!] = \{[\![\overline{E} + a \leq \overline{L-1}]\!]\} = \{(E - L + 1) + a \leq 0\}.$$

Now, $|\kappa_1|_a = \overline{n}$ and $|\kappa_2|_a = \overline{m} + a$. Thus

$$[\![|\kappa_2|_a \leq |\kappa_1|_a]\!] = (a + (m - n) \leq 0).$$

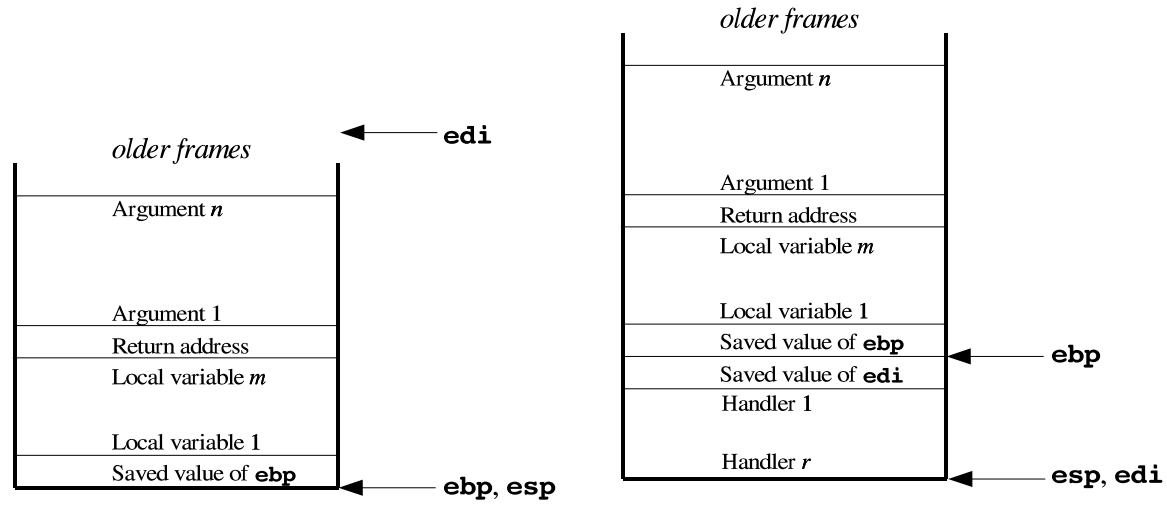
Subtracting the constraint in $[\![\Delta]\!]$ gives $(m-n+L-E-1 \leq 0)$. By assumption, the number on the left-hand side is nonpositive, so we have found a semantic proof of the desired judgment at depth 1 as desired.

End of Proof.

7.5 Stacks, Register Files and Labels

In order to give typing annotations for the labels in the output of my translation, I must be able to specify the types of all the registers, including the stack pointer, at every one of these program points. More generally, in order to argue that my translation is type-preserving, I must be able to specify the types I intend for the register file and stack at any point in the MiniTALT-R program I produce. This is more technically involved than might be expected, mostly because of the exception-handling constructs of Lilt.

The stack frame layout used by a Lilt function is shown in Figure 7.4. Note that the stack "grows downward" in the diagram just as it does in memory. All function arguments are passed and stored on the stack (above the return address) and all of the function's local variables are stack-allocated. The figure also illustrates the usage of two important registers (`ebp` and `edi`) that point into the stack. Register `ebp` plays its usual role as the frame pointer, except that it is



(before pushing local exception handlers) (after pushing r local handlers)

Figure 7.4: A Lilt function's stack frame

set up to point to the bottom of the stack frame instead of into the middle as is more customary. This is because I wish to address both arguments and local variables using displacements from `ebp`, and in TALT these displacements are not allowed to be negative. Each function stores its caller's frame pointer at the very bottom of its initial stack frame and reloads this value into `ebp` before returning. Register `edi` is the *exception pointer*; as I have already mentioned, its value is the address of a location on the stack where the current exception handler is stored. Thus at the beginning of a function, `edi` points somewhere *above* the function's own stack frame.

The left-hand side of Figure 7.4 shows the initial state of a function's stack frame; in particular, this frame has no pending local exception handlers. The right-hand side shows a frame in which r handlers have been pushed by the function. Notice that before pushing the first local exception handler, the function saves the initial value of `edi` on the stack; this value must be reloaded into `edi` when the function returns, or any time the non-local exception handler becomes current again. As long as the current exception handler is local to the current function, `edi` will have the same value as `esp`.

The type of the stack at any point in a Lilt program can be determined using the function ST in Figure 7.5. Intuitively, $ST_{\rho_1,\rho_2,\alpha_f,\alpha_h,a}(\Xi,\Gamma,\tau)$ is the type of the stack type corresponding to a Lilt exception context of Ξ and local context of Γ, in a function that returns type τ. The subscripts $\rho_1, \rho_1, \alpha_f, \alpha_h, a$ specify some special variables that are allowed to occur free in these types: ρ_1 and ρ_2 are the two unknown portions of the stack, α_f is the type of the saved value of `ebp`, α_h is the precise type of the exception handler, and a is the value that must be on the minor clock when the function returns. (To reduce verbosity, these subscripts are elided for occurrences of ST on the right-hand side of each clause when they are the same as on the left-hand side, and are elided on the left-hand side when they do not appear at all on the right.)

Figure 7.6 shows how to find the types of the registers for any point in a compiled Lilt program First, $RF_{\rho_1,\rho_2,\alpha_f,\alpha_h,a,u}(\Xi,\Gamma,\tau,t)$ is the register file type associated with the exception context Ξ and local context Γ, assuming τ is the return type of the current function and t is the value of the minor

$$
\begin{array}{rcl}
ST_{\rho_1,\rho_2,\alpha_f,\alpha_h,a}(\cdot,\Gamma,\tau) & = & \alpha_f \times |\tau_{l1}| \times \cdots \times |\tau_{lm}| \times \tau_r \times \sigma_0 \\
\\
\textit{where:} & & \Gamma = [\mathtt{arg}(1){:}\tau_{a1},\ldots,\mathtt{arg}(n){:}\tau_{an},\mathtt{loc}(1){:}\tau_{l1},\ldots,\mathtt{loc}(m){:}\tau_{lm}] \\
& & \sigma_0 = |\tau_{a1}| \times \cdots \times |\tau_{an}| \times \rho_1 \times \tau_h \times \rho_2 \\
& & \tau_h = \alpha_h \wedge \forall b{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau_{\mathrm{exn}}|,\mathtt{esp}{:}\rho_2,\mathtt{mck}_b{:}\overline{H}\} \to 0 \\
& & \tau_r = \forall b'{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau|,\mathtt{ebp}{:}\alpha_f,\mathtt{edi}{:}\tau_e,\mathtt{esp}{:}\sigma_0,\mathtt{mck}_{b'}{:}a\} \to 0 \\
& & \tau_e = \mathtt{sptr}(\tau_h \times \rho_2) \\
\\
ST_{\rho_1,\rho_2,\alpha_f,\alpha_h,a}((\cdot,\Gamma'),\Gamma,\tau) & = & (\forall b{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau_{\mathrm{exn}}|,\mathtt{esp}{:}\tau_e \times ST(\cdot,\Gamma',\tau),\mathtt{mck}_b{:}\overline{H}\} \to 0) \\
& & \quad \times \tau_e \times ST(\cdot;\Gamma;\tau) \\
\textit{where:} & & \tau_h = \alpha_h \wedge \forall b{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau_{\mathrm{exn}}|,\mathtt{esp}{:}\rho_2,\mathtt{mck}_b{:}\overline{H}\} \to 0 \\
& & \tau_e = \mathtt{sptr}(\tau_h \times \rho_2) \\
\\
ST((\Xi,\Gamma'),\Gamma,\tau) & = & (\forall b{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau_{\mathrm{exn}}|,\mathtt{esp}{:}ST(\Xi,\Gamma',\tau),\mathtt{mck}_b{:}\overline{H}\} \to 0) \\
& & \quad \times ST(\Xi,\Gamma,\tau) \\
& & \textit{if } \Xi \neq \cdot
\end{array}
$$

Figure 7.5: Determining the Stack Type

$$
\begin{array}{rcl}
RF_{\rho_1,\rho_2,\alpha_f,\alpha_h,a,u}(\cdot,\Gamma,\tau,t) & = & \{\mathtt{edi}{:}\tau_e,\mathtt{ebp}{:}\mathtt{sptr}(\sigma_1),\mathtt{esp}{:}\sigma_1,\mathtt{mck}_u{:}t\} \\
\textit{where:} & & \tau_h = \alpha_h \wedge \forall b'{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau_{\mathrm{exn}}|,\mathtt{esp}{:}\rho_2,\mathtt{mck}_{b'}{:}\overline{H}\} \to 0 \\
& & \tau_e = \mathtt{sptr}(\tau_h \times \rho_2) \\
& & \sigma_1 = ST_{\rho_1,\rho_2,\alpha_f,\alpha_h,a}(\cdot,\Gamma,\tau) \\
\\
RF_{\rho_1,\rho_2,\alpha_f,\alpha_h,a,u}(\Xi,\Gamma,\tau,t) & = & \{\mathtt{edi}{:}\mathtt{sptr}(\sigma_2),\mathtt{ebp}{:}\mathtt{sptr}(\sigma_1),\mathtt{esp}{:}\sigma_2,\mathtt{mck}_u{:}t\} \\
\textit{where:} & & \sigma_1 = ST_{\rho_1,\rho_2,\alpha_f,\alpha_h,a}(\cdot,\Gamma,\tau) \\
& & \sigma_2 = ST_{\rho_1,\rho_2,\alpha_f,\alpha_h,a}(\Xi,\Gamma,\tau) \\
& & \Xi \neq \cdot
\end{array}
$$

Figure 7.6: Determining the Register File Type

$$
\begin{array}{l}
LL(\Delta, \Xi, \Gamma, \tau, \kappa, [\bar{r}_1 \mapsto \tau_1, \ldots, \bar{r}_n \mapsto \tau_n]) = \\
\qquad \forall \alpha_1{:}|k_1| \ldots \forall \alpha_m{:}|k_m| . \forall \rho_1{:}\mathsf{TD} . \forall \rho_2{:}\mathsf{TD} . \forall \alpha_f{:}\mathsf{T4} . \forall \alpha_h{:}\mathsf{T4} . \\
\qquad \forall a{:}\mathsf{N} . \forall b{:}\mathsf{N} . \ (\overline{E} + a \leq \overline{L - 1}) \Rightarrow \\
\qquad\qquad RF_{\rho_1, \rho_2, \alpha_f, \alpha_h, a, b}(\Xi, \Gamma, \tau, |\kappa|_a)[\bar{r}_1 \mapsto \tau_1, \ldots, \bar{r}_n \mapsto \tau_n] \to 0 \\
\qquad \text{where } \Delta = \alpha_1{:}k_1, \ldots, \alpha_m{:}k_m
\end{array}
$$

$$
\begin{array}{rcl}
|lbl(\Delta'; \Xi; \Gamma)|_{\Delta, \tau, \kappa} & = & LL((\Delta, \Delta'); \Xi; \Gamma, \tau, \kappa, [\,]) \\
\\
|hnd(\Delta'; \Xi; \Gamma)|_{\Delta, \tau, \kappa} & = & \forall \alpha_1{:}|k_1| \ldots \forall \alpha_m{:}|k_m| . \forall \rho_1{:}\mathsf{TD} . \forall \rho_2{:}\mathsf{TD} . \forall \alpha_f{:}\mathsf{T4} . \forall \alpha_h{:}\mathsf{T4} . \\
& & \forall a{:}\mathsf{N} . \forall b{:}\mathsf{N} . \ (\overline{E} + a \leq \overline{L - 1}) \Rightarrow \\
& & \quad \{\mathtt{eax}{:}|\tau_{\mathrm{exn}}|, \mathtt{esp}{:}ST_{\rho_1, \rho_2, \alpha_f, \alpha_h, a}(\Xi, \Gamma, \tau), \mathtt{mck}_b{:}\overline{H}\} \to 0 \\
\text{where} & & (\Delta, \Delta') = \alpha_1{:}k_1, \ldots, \alpha_m{:}k_m
\end{array}
$$

Figure 7.7: Label and Block Types

clock. The subscripts $\rho_1, \rho_2, \alpha_f, \alpha_h, a, u$ are as in the definition of ST, with the addition of u, the static term representation of the register clock.

Finally, Figure 7.7 shows how to compute types for labels occurring within a translated function body and how to translate Lilt block types. First, $LL(\Delta, \Xi, \Gamma, \tau, \kappa, [\bar{r}_1 \mapsto \tau_1, \ldots, \bar{r}_n \mapsto \tau_n])$ is the type of a local label with type parameters given by Δ (this includes both the type parameters of the enclosing function and any additional parameters of the current block) and expecting exception handlers described by Ξ, local storage described by Γ, and κ describing the minor clock, where τ again is the return type of the function in which the label appears and the additional type assignments $\bar{r}_i \mapsto \tau_i$ specify the types of values stored temporarily in registers. The translation of an ordinary block type is easily defined using LL; LL is also used to annotate labels that occur in the interior of a Lilt block. Exception handler blocks are a little different: an exception handler block expects an exception value in `eax` and H on the minor clock.

7.6 Translating Operands

Because of my assumptions about the names of local storage locations, if the total number M of local variables allocated by the current function is known then the operand corresponding to location s (denoted by $|s|_M$) can be determined from the name s as follows:

$$
\begin{array}{rcl}
|\mathtt{loc}(i)|_M & = & \mathtt{[ebp+}(4i)\mathtt{]} \\
|\mathtt{arg}(i)|_M & = & \mathtt{[ebp+}(4(1 + M + i))\mathtt{]}
\end{array}
$$

In the MiniTALT-R syntax used in this proposal, stack operands such as these are written exactly the same as the destinations denoting the same locations. To refer to the *destination* corresponding to the location s I will write $|s|^d_M$.

I assume there is an obvious embedding of Lilt function symbols into assembly-level labels,

and extend the mapping $|\cdot|_L$ to all Lilt operands as follows:

$$\begin{array}{cc} |n|_M = \mathtt{im}(\overline{n}) & |\star|_M = \mathtt{im}(\overline{0}) \\ |\mathtt{tt}|_M = \mathtt{im}(\overline{1}) & |f|_M = f \\ |\mathtt{ff}|_M = \mathtt{im}(\overline{0}) & |q@v|_M = |v|_M \end{array}$$

7.7 Compiling Expressions

In general, a Lilt block may translate to more than one MiniTALT-R block; a Lilt expression will translate to a MiniTALT-R instruction sequence plus zero or more additional blocks. The translation rules will use the letter S to range over sequences of MiniTALT-R blocks:

$$S ::= \epsilon \mid \ell{:}\tau = I\ S$$

To make MiniTALT-R code look more like ordinary assembly code, I will freely concatenate sequences of blocks in the obvious way.

$$\begin{array}{rl} \mathcal{C} & ::= (\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau) \\ \mathcal{T} & ::= \ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n \end{array}$$

Figure 7.8: Translation Contexts

Since the translation is type-directed, its structure follows the typing rules of Lilt rather closely; however, to reduce the clutter on the left side of the turnstile in translation judgments, I collect all the context information for a Lilt expression into one *translation context*, ranged over by $\mathcal{C}$ as shown in Figure 7.8. The figure also shows the syntax for local timing contexts $\mathcal{T}$; a local timing context maps each local label in a Lilt function to the minor clock value that block expects. To manipulate the context information collected in a translation context $\mathcal{C}$ as required by the translation rules, some notation is required. In particular, if $\mathcal{C} = (\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau)$, then define the following:

- $\mathrm{locs}(\mathcal{C}) = \mathrm{dom}(\Gamma)$
- $\mathrm{handlers}(\mathcal{C}) = \mathrm{length}(\Xi)$
- $\mathcal{C}(\ell) = \Lambda(\ell)$
- $\mathcal{C}[s \mapsto \tau'] = (\Phi; \Delta; \Lambda; \Xi; \Gamma[s \mapsto \tau']; \tau)$
- $\mathcal{C} \oplus \Delta' = (\Phi; (\Delta, \Delta'); \Lambda; \Xi; \Gamma; \tau)$
- $\mathcal{C} \oplus \Gamma' = (\Phi; \Delta; \Lambda; (\Xi, \Gamma'); \Gamma; \tau)$
- $\mathrm{poph}(\mathcal{C}) = (\Phi; \Delta; \Lambda; \Xi'; \Gamma; \tau)$, if $\Xi = (\Xi', \Gamma')$
- $|s|_{\mathcal{C}} = |s|_B$, where $\mathrm{dom}(\Gamma) = \{\mathtt{arg}(1), \ldots, \mathtt{arg}(A), \mathtt{loc}(1), \ldots, \mathtt{loc}(B)\}$ (and similarly for $|s|^d_{\mathcal{C}}$)
- $\mathcal{C} \vdash c : k$ iff $\Delta \vdash c : k$
- $\mathcal{C} \vdash c_1 = c_2 : k$ iff $\Delta \vdash c_1 = c_2 : k$
- $\mathcal{C} \vdash v : \tau'$ iff $\Phi; \Delta; \Gamma \vdash v : \tau'$
- $\mathcal{C} \models \Gamma'$ iff $\Delta \vdash \Gamma \leq \Gamma'$
- $\mathcal{C} \models \Xi'$ iff $\Delta \vdash \Xi \leq \Xi'$

- $\mathcal{C} \models$ canraise iff $\Delta \vdash \Xi$ handles Γ

The complete translation rules are in Section 7.8. The translation judgment, $\mathcal{C}, \mathcal{T}, \kappa \vdash e \rightsquigarrow I\ S$, means that the instruction sequence I, together with the additional blocks S, implements the expression e assuming κ describes the minor clock. The translation is highly nondeterministic: in particular, it makes no commitment to either forward or backward propagation, and does not specify how to determine the initial minor clock requirement for each block within a function. Two translation rules ensure that a minor yield may be inserted before any subexpression, whether it is needed or not:

$$\frac{\mathcal{C}; \mathcal{T}; (\text{just } m) \vdash e \rightsquigarrow I\ S}{\mathcal{C}; \mathcal{T}; (\text{just } n) \vdash e \rightsquigarrow \texttt{YIELD}(n, m)\ I\ S} \qquad \frac{\mathcal{C}; \mathcal{T}; (\text{just } m) \vdash e \rightsquigarrow I\ S}{\mathcal{C}; \mathcal{T}; (\text{retplus } n) \vdash e \rightsquigarrow \texttt{YIELD}(n, m)\ I\ S}$$

Note that this rule takes advantage of the clock register "tricks" discussed in Section 6.5.4, setting the minor clock to an arbitrary value m. The rules do not specify the value of m; in practice an implementation may either use $m = L$ everywhere in a program, as my prototype does, or it may perform some analysis to determine good values for m at each minor yield it generates.

In the rule for translating an *intra*procedural jump, the timing context $\mathcal{T}$ is consulted to ensure the target block's clock expectations are met:

$$\frac{\begin{array}{c}(\mathcal{C}(\ell) = \mathit{lbl}(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \\ \kappa - 1 \geq \mathcal{T}(\ell) \quad \mathcal{C} \vdash c_i : k_i \quad \mathcal{C} \models \Gamma'[\vec{c}/\vec{\alpha}] \quad \mathcal{C} \models \Xi'[\vec{c}/\vec{\alpha}]\end{array}}{\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{goto}\ \ell[c_1, \ldots, c_n] \rightsquigarrow \texttt{jmp}\ \ell}$$

Since the initial minor clock is κ, it will be $\kappa - 1$ after the `jmp` instruction. Thus in order for this rule to apply, it must be the case that $\kappa - 1$ is greater than or equal to the minor clock value expected by block ℓ. (The other premises of this rule correspond directly to the premises of the typing rule for `goto`.) If it is not the case that $\kappa - 1 \geq \mathcal{T}(\ell)$, then this rule will not apply, but one of the two yielding rules will; thus a well-typed `goto` expression can always be compiled, possibly by yielding first.

The rule for returning from a function takes account of the fact that a clock specifier of $\text{retplus}\, n$ means minor clock is sufficient to execute n instructions, the last of which may be a `ret`. It takes a few instructions, however, to get ready to return:

$$\frac{\begin{array}{c}(\text{locs}(\mathcal{C}) = [\texttt{arg}(1), \ldots, \texttt{arg}(A), \texttt{loc}(1), \ldots, \texttt{loc}(B)]) \\ \mathcal{C} \vdash v : \tau \quad \kappa - 4 \geq \text{retplus}(0) \quad (\text{handlers}(\mathcal{C}) = 0)\end{array}}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{return}\ v \rightsquigarrow \\ \qquad \texttt{mov eax}, |v|_{\mathcal{C}} \\ \qquad \texttt{pop ebp} \\ \qquad \texttt{sfree}\ (4B) \\ \qquad \texttt{ret}\end{array}}$$

The code generated by this rule moves the value to be returned into `eax`, moves the caller's frame pointer back into `ebp`, frees the stack space allocated by the function, and finally returns. This takes four instructions, so the rule requires that $\kappa - 4 \geq \text{retplus}(0)$. (This is equivalent to requiring $\kappa \geq \text{retplus}(4)$.) A side condition in this rule requires that $\text{handlers}(\mathcal{C}) = 0$; there is a slightly different rule for returning when there are local exception handlers that must be removed from the stack.

Most of the other instructions simply decrement the clock specifier κ by the appropriate amount before translating their subexpressions. For example, translation of primitive arithmetic is straightforward:

$$\frac{\mathcal{C} \vdash v_i : \texttt{int} \text{ for } 1 = 1, 2 \quad \mathcal{C}[s \mapsto \texttt{int}]; \mathcal{T}; (\kappa - 3) \vdash e \rightsquigarrow I\ S}{\begin{array}{c}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = +(v_1, v_2)\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{mov eax}, |v_1|_{\mathcal{C}} \\ \quad \texttt{add eax}, \texttt{eax}, |v_2|_{\mathcal{C}} \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \quad I \\ S\end{array}}$$

(Note, though, that a simple addition takes three MiniTALT-R instructions because all local storage is on the stack. This highlights the need for a better register allocation scheme.) If the translation encounters an addition expression like this one in a Lilt program and the minor clock is less than 3 (that is, if $\kappa - 3$ is undefined), then it must translate that expression using the appropriate yielding rule. Unfortunately, some Lilt operations can in principle require an arbitrary number of instructions: allocating a tuple of size n requires as many as $2n + 2$ instructions, and calling a function with n arguments costs $n + E + 3$. It is therefore impossible to require these operations to be compiled to yield-free instruction sequences. The translation given here ignores these issues, but there is no reason a real compiler cannot be designed to deal with wide tuples and high-arity functions.

7.8 Complete Translation Rules

$$\frac{\begin{array}{c}\vdash \Delta \quad \Delta \vdash \tau_i : T \text{ for each } i \quad \Delta \vdash \tau : T \quad \Delta \vdash \Lambda \quad (\mathrm{dom}(\mathcal{T}) = \mathrm{dom}(\Lambda)) \\ (\Phi; \Delta; \Lambda; \cdot; \Gamma; \tau); \mathcal{T}; (\mathrm{retplus}(E - 2)) \vdash e \rightsquigarrow I\ S_0 \\ \Phi; \Delta; \Lambda; \tau; \mathcal{T} \vdash B_i : (\Lambda(\ell_i), \mathcal{T}(\ell_i)) \rightsquigarrow I_i\ S_i \text{ for } 1 \le i \le m\end{array}}{\begin{array}{l}\Phi \vdash \texttt{func}(\Delta; \vec{\tau}; \tau).(\texttt{enter}(L).e, \ell_1 = B_1, \ldots, \ell_m = B_m) : \forall\Delta.(\vec{\tau}) \to \tau \rightsquigarrow \\ \quad f : |\forall\Delta.(\vec{\tau}) \to \tau| = \\ \\ \quad\quad \texttt{salloc}\ (4L) \\ \quad\quad \texttt{push ebp} \\ \quad\quad I \\ \quad S_0 \\ \quad \ell_1 : |\Lambda(\ell_1)|_{\Delta,\tau,\mathcal{T}(\ell_1)} = I_1 \\ \quad S_1 \\ \quad \vdots \\ \quad \ell_m : |\Lambda(\ell_m)|_{\Delta,\tau,\mathcal{T}(\ell_m)} = I_m \\ \quad S_m\end{array}}$$

where

$\Gamma = [\texttt{arg}(1){:}\tau_1, \ldots, \texttt{arg}(p){:}\tau_p, \texttt{loc}(1){:}\texttt{ns}, \ldots, \texttt{loc}(L){:}\texttt{ns}]$
each B_i is either $\texttt{block}(\Delta_i; \Xi_i; \Gamma_i).e$ or $\texttt{hndl}(\Delta_i; \Xi_i; \Gamma_i; s).e$, and
$\mathrm{dom}(\Gamma_i) = \mathrm{dom}(\Gamma)$ for each i

$$\frac{\begin{array}{c}\Delta, \Delta' \vdash \Xi \\ \Delta, \Delta' \vdash \Gamma \quad (\Phi; (\Delta, \Delta'); \Lambda; \Xi; \Gamma; \tau); \mathcal{T}; \kappa \vdash e \rightsquigarrow I\ S\end{array}}{\Phi; \Delta; \Lambda; \tau; \mathcal{T} \vdash \texttt{block}(\Delta'; \Xi; \Gamma).e : (lbl(\Delta'; \Xi; \Gamma), \kappa) \rightsquigarrow I\ S}$$

$$\frac{\begin{array}{c}\Delta, \Delta' \vdash \Gamma \\ (\Phi; (\Delta, \Delta'); \Lambda; \cdot; \Gamma[s \mapsto \tau_{\text{exn}}]; \tau); \mathcal{T}; (\text{just}(H-3)) \vdash e \mapsto I\ S\end{array}}{\begin{array}{l}\Phi; \Delta; \Lambda; \tau; \mathcal{T} \vdash \texttt{hndl}(\Delta'; \cdot; \Gamma; s).e : (hnd(\Delta'; \cdot; \Gamma), \kappa) \rightsquigarrow \\ \quad \texttt{pop edi} \\ \quad \texttt{mov ebp, esp} \\ \quad \texttt{mov } |s|_\Gamma\texttt{, eax} \\ \quad I \\ S\end{array}}$$

$$\frac{\begin{array}{c}(E = \text{length}(\Xi) \neq 0) \quad \Delta, \Delta' \vdash \Xi \quad \Delta, \Delta' \vdash \Gamma \\ (\Phi; (\Delta, \Delta'); \Lambda; \Xi; \Gamma[s \mapsto \tau_{\text{exn}}]; \tau); \mathcal{T}; (\text{just}(H-4)) \vdash e \mapsto I\ S\end{array}}{\begin{array}{l}\Phi; \Delta; \Lambda; \tau; \mathcal{T} \vdash \texttt{hndl}(\Delta'; \Xi; \Gamma; s).e : (hnd(\Delta'; \Xi; \Gamma), \kappa) \rightsquigarrow \\ \quad \texttt{mov edi, esp} \\ \quad \texttt{mov ebp, esp} \\ \quad \texttt{addsptr ebp, ebp, } 4(E+1) \\ \quad \texttt{mov } |s|_\Gamma\texttt{, eax} \\ \quad I \\ S\end{array}}$$

$$\frac{\begin{array}{c}(\text{locs}(\mathcal{C}) = [\texttt{arg}(1), \ldots, \texttt{arg}(A), \texttt{loc}(1), \ldots, \texttt{loc}(B)]) \\ \mathcal{C} \vdash v : \tau \quad \kappa - 4 \geq \text{retplus}(0) \quad (\text{handlers}(\mathcal{C}) = 0)\end{array}}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{return } v \rightsquigarrow \\ \quad \texttt{mov eax, } |v|_{\mathcal{C}} \\ \quad \texttt{pop ebp} \\ \quad \texttt{sfree } (4B) \\ \quad \texttt{ret}\end{array}}$$

$$\frac{\begin{array}{c}(\text{locs}(\mathcal{C}) = [\texttt{arg}(1), \ldots, \texttt{arg}(A), \texttt{loc}(1), \ldots, \texttt{loc}(B)]) \\ \mathcal{C} \vdash v : \tau \quad \kappa - 5 \geq \text{retplus}(0) \quad (X = \text{handlers}(\mathcal{C}) \neq 0)\end{array}}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{return } v \rightsquigarrow \\ \quad \texttt{mov eax, } |v|_{\mathcal{C}} \\ \quad \texttt{mov edi, } [\texttt{esp} + 4X] \\ \quad \texttt{mov ebp, } [\texttt{esp} + (4(X+1))] \\ \quad \texttt{sfree } (4(B+X+2)) \\ \quad \texttt{ret}\end{array}}$$

$$\frac{\begin{array}{c}(\mathcal{C}(\ell) = lbl(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \\ \kappa - 1 \geq \mathcal{T}(\ell) \quad \mathcal{C} \vdash c_i : k_i \quad \mathcal{C} \models \Gamma'[\vec{c}/\vec{\alpha}] \quad \mathcal{C} \models \Xi'[\vec{c}/\vec{\alpha}]\end{array}}{\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{goto } \ell[c_1, \ldots, c_n] \rightsquigarrow \texttt{jmp } \ell}$$

$$\frac{\begin{array}{c}\kappa - 3 \geq \text{just } H \\ \mathcal{C} \vdash v : \tau_{\text{exn}} \quad \mathcal{C} \models \textsf{canraise}\end{array}}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{raise } v \rightsquigarrow \\ \quad \texttt{mov eax, } |v|_{\mathcal{C}} \\ \quad \texttt{mov esp, edi} \\ \quad \texttt{ret}\end{array}}$$

$$\frac{\mathcal{C} \vdash v : (\tau'_1 \ldots, \tau'_n) \to \tau'' \quad \mathcal{C} \models \mathsf{canraise} \quad \mathcal{C} \vdash v_i : \tau'_i \text{ for } 1 \le i \le n \quad \mathcal{C}[s \mapsto \tau'']; \mathcal{T}; (\kappa - (n+3+E)) \vdash e \mapsto I\,S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = v(v_1, \ldots, v_n)\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{push}\ |v_n|_{\mathcal{C}} \\ \quad \vdots \\ \quad \texttt{push}\ |v_1|_{\mathcal{C}} \\ \quad \texttt{call}\ |v|_{\mathcal{C}} \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \quad \texttt{sfree}\ 4n \\ \quad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v : \tau'\ \texttt{array} \quad \mathcal{C} \vdash v' : \texttt{int} \quad \mathcal{C}[s \mapsto \tau']; \mathcal{T}; (\kappa - 7) \vdash e \rightsquigarrow I\,S \quad \mathcal{C}; \mathcal{T}; (\kappa - 4) \vdash \texttt{raise}\, v_{\text{arrayexn}} \rightsquigarrow I_e\, S_e}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = \texttt{sub}(v, v')\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{mov}\ \texttt{eax}, |v|_{\mathcal{C}} \\ \quad \texttt{mov}\ \texttt{ecx}, |v'|_{\mathcal{C}} \\ \quad \texttt{cmpja}\ [\texttt{eax}], \texttt{ecx}, \ell_{pass} \\ \quad I_e \\ S_e \\ \ell_{pass} : \forall \alpha_{sz}{:}\mathsf{N}. \\ \qquad LL(\mathcal{C}, [\texttt{eax} \mapsto \texttt{box}(\texttt{set}_{=}(\alpha_{sz}) \times \texttt{mbox}(|\tau'| \uparrow \alpha_{sz})), \texttt{ecx} \mapsto \texttt{set}_{<}(\alpha_{sz})]) = \\ \quad \texttt{mov}\ \texttt{eax}, [\texttt{eax} + 4] \\ \quad \texttt{mov}\ \texttt{eax}, [\texttt{eax} + 0 + 4 \cdot \texttt{ecx}] \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \quad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v_1 : \tau'\ \texttt{array} \quad \mathcal{C} \vdash v_2 : \texttt{int} \quad \mathcal{C}; \mathcal{T}; (\kappa - 4) \vdash \texttt{raise}\, v_{\text{arrayexn}} \rightsquigarrow I_e\, S_e \quad \mathcal{C} \vdash v_3 : \tau' \quad \mathcal{C}; \mathcal{T}; (\kappa - 7) \vdash e \rightsquigarrow I\,S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ \texttt{sub}(v_1, v_2) := v_3\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{mov}\ \texttt{eax}, |v_1|_{\mathcal{C}} \\ \quad \texttt{mov}\ \texttt{ecx}, |v_2|_{\mathcal{C}} \\ \quad \texttt{cmpja}\ [\texttt{eax}], \texttt{ecx}, \ell_{pass} \\ \quad I_e \\ S_e \\ \ell_{pass} : \forall \alpha_{sz}{:}\mathsf{N}. \\ \qquad LL(\mathcal{C}, [\texttt{eax} \mapsto \texttt{box}(\texttt{set}_{=}(\alpha_{sz}) \times \texttt{mbox}(|\tau'| \uparrow \alpha_{sz})), \texttt{ecx} \mapsto \texttt{set}_{<}(\alpha_{sz})]) = \\ \quad \texttt{mov}\ \texttt{eax}, [\texttt{eax} + 4] \\ \quad \texttt{mov}\ \texttt{edx}, |v_3|_{\mathcal{C}} \\ \quad \texttt{mov}\ [\texttt{eax} + 0 + 4 \cdot \texttt{ecx}], \texttt{edx} \\ \quad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v : [\overline{j{:}\tau}, i{:}\tau', \overline{j{:}\tau}'] \quad \mathcal{C}[s \mapsto [i{:}\tau']]; \mathcal{T}; (\kappa-4) \vdash e_1 \rightsquigarrow I_1\, S_1 \quad \mathcal{C}[s \mapsto [\overline{j{:}\tau}, \overline{j{:}\tau}']]; \mathcal{T}; (\kappa-4) \vdash e_2 \rightsquigarrow I_2\, S_2}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{case}\ v\ \texttt{of}\ \texttt{inj}(i, s) \Rightarrow e_1\ \texttt{else}\ e_2 \rightsquigarrow \\ \quad \texttt{mov eax}, |v|_{\mathcal{C}} \\ \quad \texttt{cmpje [eax]}, i, \ell_{match} \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \quad I_2 \\ S_2 \\ \ell_{match} : LL(\mathcal{C}, [\texttt{eax} \mapsto |[i{:}\tau']|]) = \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \quad I_1 \\ S_1\end{array}}$$

$$\frac{\mathcal{C} \vdash v_i : \texttt{int for } i = 1, 2 \quad \mathcal{C}; \mathcal{T}; (\kappa-3) \vdash e_1 \rightsquigarrow I_1\, S_1 \quad \mathcal{C}; \mathcal{T}; (\kappa-3) \vdash e_2 \rightsquigarrow I_2\, S_2}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{if}\ v_1 = v_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \\ \quad \texttt{mov eax}, |v_1|_{\mathcal{C}} \\ \quad \texttt{cmp eax}, |v_2|_{\mathcal{C}} \\ \quad \texttt{jne}\ \ell_{else} \\ \quad I_1 \\ S_1 \\ \ell_{else} : LL(\mathcal{C}, [\,]) = \\ \quad I_2 \\ S_2\end{array}}$$

$$\frac{\mathcal{C} \vdash v : \langle \tau_0, \ldots, \tau_m \rangle \quad \mathcal{C} \vdash v : \tau_i \quad \mathcal{C}; \mathcal{T}; \kappa - 3 \vdash e \rightsquigarrow I\, S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ \pi_i\, v := v'\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{mov eax}, |v|_{\mathcal{C}} \\ \quad \texttt{mov ecx}, |v'|_{\mathcal{C}} \\ \quad \texttt{mov}\ [\texttt{eax} + 4i], \texttt{ecx} \\ \quad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v_i : \texttt{int for } i = 1, 2 \quad \mathcal{C}; \mathcal{T}; \kappa - 3 \vdash e_1 \rightsquigarrow I_1\, S_1 \quad \mathcal{C}; \mathcal{T}; \kappa - 3 \vdash e_2 \rightsquigarrow I_2\, S_2}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{if}\ v_1 < v_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \\ \quad \texttt{mov eax}, |v_1|_{\mathcal{C}} \\ \quad \texttt{cmp eax}, |v_2|_{\mathcal{C}} \\ \quad \texttt{ja}\ \ell_{else} \\ \quad I_1 \\ S_1 \\ \ell_{else} : LL(\mathcal{C}, [\,]) = \\ \quad I_2 \\ S_2\end{array}}$$

$$\frac{\mathcal{C} \vdash v : \exists \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' \quad (\mathcal{C} \oplus (\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n))[s \mapsto \tau']; \mathcal{T}; (\kappa-2) \vdash e \rightsquigarrow I\, S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}(\alpha_1, \ldots, \alpha_n, s) = \texttt{unpack}\ v\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{mov eax}, |v|_{\mathcal{C}} \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \quad I \\ S\end{array}}$$

$$\frac{(\mathcal{C}(\ell) = hnd(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \quad (\text{handlers}(\mathcal{C}) = 0) \quad \mathcal{C} \vdash c_i : k_i \quad \mathcal{C} \models \Xi'[\vec{c}/\vec{\alpha}] \quad \mathcal{C} \oplus (\Gamma'[\vec{c}/\vec{\alpha}]); \mathcal{T}; (\kappa - 3) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{pushhandler}\ \ell[c_1, \ldots, c_n]\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{push edi} \\ \quad \texttt{push}\ \ell \\ \quad \texttt{mov edi, esp} \\ \quad I \\ S\end{array}}$$

$$\frac{(\mathcal{C}(\ell) = hnd(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \quad (\text{handlers}(\mathcal{C}) \neq 0) \quad \mathcal{C} \vdash c_i : k_i \quad \mathcal{C} \models \Xi'[\vec{c}/\vec{\alpha}] \quad \mathcal{C} \oplus (\Gamma'[\vec{c}/\vec{\alpha}]); \mathcal{T}; (\kappa - 2) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{pushhandler}\ \ell[c_1, \ldots, c_n]\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{push}\ \ell \\ \quad \texttt{mov edi, esp} \\ \quad I \\ S\end{array}}$$

$$\frac{(\text{handlers}(\mathcal{C}) = 1) \quad \text{poph}(\mathcal{C}); \mathcal{T}; (\kappa - 2) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{pophandler in}\ e \rightsquigarrow \\ \quad \texttt{mov edi, [esp} + 4\texttt{]} \\ \quad \texttt{sfree}\ 8 \\ \quad I \\ S\end{array}}$$

$$\frac{(\text{handlers}(\mathcal{C}) > 1) \quad \text{poph}(\mathcal{C}); \mathcal{T}; (\kappa - 2) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{pophandler in}\ e \rightsquigarrow \\ \quad \texttt{sfree}\ 4 \\ \quad \texttt{mov edi, esp} \\ \quad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v : \tau' \quad \mathcal{C}[s \mapsto \tau']; \mathcal{T}; (\kappa - 2) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C} \vdash \texttt{let}\ s = v\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{mov eax,}\ |v|_{\mathcal{C}} \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}\texttt{, eax} \\ \quad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v_i : \texttt{int for}\ 1 = 1, 2 \quad \mathcal{C}[s \mapsto \texttt{int}]; \mathcal{T}; (\kappa - 3) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = +(v_1, v_2)\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{mov eax,}\ |v_1|_{\mathcal{C}} \\ \quad \texttt{add eax, eax,}\ |v_2|_{\mathcal{C}} \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}\texttt{, eax} \\ \quad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v_i : \tau_i\ \text{for}\ 1 \leq i \leq n \quad \mathcal{C}[s \mapsto \langle \tau_1, \ldots, \tau_n \rangle]; \mathcal{T}; (\kappa - (2n + 2)) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = \langle v_1, \ldots, v_n \rangle\ \texttt{in}\ e \rightsquigarrow \\ \quad \texttt{push}\ |v_n|_{\mathcal{C}} \\ \quad \vdots \\ \quad \texttt{push}\ |v_1|_{\mathcal{C}} \\ \quad \texttt{malloc eax, ebx,}\ 4n \\ \quad \texttt{pop [eax} + 4 \cdot 0\texttt{]} \\ \quad \vdots \\ \quad \texttt{pop [eax} + 4 \cdot (n - 1)\texttt{]} \\ \quad \texttt{mov}\ |s|^d_{\mathcal{C}}\texttt{, eax} \\ \quad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v : \langle \tau_0, \ldots, \tau_n \rangle \quad \mathcal{C}[s \mapsto \tau_i]; \mathcal{T}; (\kappa - 3) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = \pi_i\ v\ \texttt{in}\ e \rightsquigarrow \\ \qquad \texttt{mov eax}, |v|_{\mathcal{C}} \\ \qquad \texttt{mov eax}, [\texttt{eax} + 4i] \\ \qquad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \qquad I \\ S\end{array}}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \tau' = [\ldots, j{:}\tau_j, \ldots] : T \\ \mathcal{C} \vdash v : \tau_j \quad \mathcal{C}[s \mapsto \tau']; \mathcal{T}; (\kappa - 5) \vdash e \rightsquigarrow I\ S\end{array}}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = \texttt{inj}'_\tau(j, v)\ \texttt{in}\ e \rightsquigarrow \\ \qquad \texttt{push}\ |v|_{\mathcal{C}} \\ \qquad \texttt{malloc eax}, \texttt{ebx}, 8 \\ \qquad \texttt{mov}\ [\texttt{eax}], j \\ \qquad \texttt{pop}\ [\texttt{eax} + 4] \\ \qquad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \qquad I \\ S\end{array}}$$

$$\frac{\mathcal{C} \vdash v : [i{:}\tau'] \quad \mathcal{C}[s \mapsto \tau']; \mathcal{T}; (\kappa - 3) \vdash e \rightsquigarrow I\ S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = \texttt{outj}(v)\ \texttt{in}\ e \rightsquigarrow \\ \qquad \texttt{mov eax}, |v|_{\mathcal{C}} \\ \qquad \texttt{mov eax}, [\texttt{eax} + 4] \\ \qquad \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ \qquad I \\ S\end{array}}$$

$$\frac{\mathcal{C}; \mathcal{T}; (\text{just}\ R) \vdash e \rightsquigarrow I\ S}{\mathcal{C}; \mathcal{T}; (\text{just}\ F) \vdash e \rightsquigarrow \texttt{YIELD}(F, R)\ I\ S}$$

$$\frac{\mathcal{C}; \mathcal{T}; (\text{just}\ R) \vdash e \rightsquigarrow I\ S}{\mathcal{C}; \mathcal{T}; (\text{retplus}\ F) \vdash e \rightsquigarrow \texttt{YIELD}(F, R)\ I\ S}$$

Chapter 8

Diverse Safety Policies

Most of the thesis up to this point has focused on certification of programs with respect to one specific safety policy. This is all very well, but it is important to realize that the virtual clock mechanism and the type theory of TALT-R can be adapted to work with other safety policies, extending the applicability of this work to many other situations. In this chapter I shall describe a number of possible modifications to the TALT-R language and their application to a wide range of safety policies.

8.1 Adaptive Responsiveness

The version of TALT-R presented in detail in earlier chapters takes the maximum yield period Y to be a fixed number, chosen in advance and "hard-wired" into the type system and into the certifying compilation and verification machinery. For practical purposes, this lack of flexibility is likely to be a serious problem. The correct value of Y for optimal performance will vary from one situation to the next, depending on the cost of the yield operation and the system-specific timing requirements, among other factors. What is more, the optimal Y may vary over time even for the same supervisor.

Here is an idea for a flexible solution: each time the program yields, let the supervisor specify the deadline for the next yield by placing that value in a register before returning control to the program. The program can then load this value into a clock register and continue with a minor yielding strategy more or less exactly as described in Section 6.5. Of course, it will not do to allow the supervisor to specify *any* number it chooses. In order to write programs that are safe under this new policy, it must be possible to place clock checks — minor yields — close enough together that no yield deadline will ever be missed no matter what the supervisor does. Unless there is a lower bound on the inter-yield times the supervisor can demand, it will be impossible for a nontrivial program to satisfy the policy. So let the safety policy specify a *minimum maximum yield period*, Y_0, that is large enough to admit reasonably spaced minor yields.

Syntactically, we change the yield instruction so that it requires a destination. The typing rule becomes:

$$\frac{(\Delta, a{:}\mathsf{N}); \Psi; \Gamma \vdash d : \mathcal{S}(a + \overline{Y_0}) \to \Gamma' \quad (\Delta, a{:}\mathsf{N}); \Psi; \Gamma'\{\mathtt{ck}{:}a + \overline{Y_0}\} \vdash I}{\Delta; \Psi; \Gamma \vdash \mathtt{yield}\, d\, I}$$

According to this rule, the value returned by the modified yield instruction is the representation of the value to which the virtual clock has been set. This number, denoted by the static term $a+\overline{Y_0}$, is statically unknown but is clearly at least Y_0. Furthermore, assuming $Y_0 \geq L + 3$, if $d = \mathtt{rck}$ then

we have

$$\Gamma'\{\texttt{ck}{:}a + \overline{Y_0}\} \leq \Gamma\{\texttt{rck}{:}\mathcal{S}(a + \overline{Y_0 - L - 3} + \overline{L + 3}), \texttt{ck}{:}\overline{1} + \overline{L} + (\overline{2} + a + \overline{Y_0 - L - 3})\}.$$

If we use the unchecked singleton subtraction described in Section 3.3.4 to subtract $L + 3$ from `rck`, which takes one instruction, we obtain the register typing

$$\Gamma\{\texttt{rck}{:}\mathcal{S}(a + \overline{Y_0 - L - 3}), \texttt{ck}{:}\overline{L} + (\overline{2} + (a + \overline{Y_0 - L - 3}))\} = \Gamma\{\texttt{mck}_{a+\overline{Y_0-L-3}} \mapsto \overline{L}\},$$

i.e., we have set the minor clock to L. Moreover, L can be any number of our choosing that is less than or equal to $Y_0 - 3$. In fact, choosing $L = Y_0 - 3$ may sometimes make sense: remember that Y_0 is the *lower limit* of a dynamically varying inter-yield allowance, and so it is probably much smaller than the Y of earlier chapters. Depending on the application, instances where the supervisor returns $10Y_0$ or even $1000Y_0$ as the actual deadline may be common; if such is the case there may be little benefit in checking the clock much more often than every Y_0 instructions.

8.2 The Engine Abstraction

The *engine* abstraction is an approach to multitasking and preemption popular in Scheme programming. An engine is a computation that can be executed subject to a time limit, which may or may not be enough for it to finish. This time limit is referred to as the amount of *fuel* given to the engine. If the engine finishes its computation before running out of fuel, it returns the computed value; if it does not, it returns a new engine which, if invoked, will resume the computation where the old one left off. Haynes and Friedman [34] showed how to implement user-level threads using engines; Dybvig and Hieb [22] have shown that engines may in turn be implemented using `call/cc` and a timer interrupt. Finally, and most interestingly, Dybvig's Scheme programming book [21] shows how to implement a form of engines *without* the help of a system-provided asynchronous timer interrupt. (Sitaram's online text [63] provides another good introduction to engines for novice Scheme programmers.)

Dybvig's interrupt-free engine implementation is less satisfying than the alternatives from a pragmatic standpoint, in that the code executed by an engine is responsible for decrementing a timer periodically to track its consumption of fuel. (Thus, although Dybvig advertises engines as an abstraction of "timed preemption," the implementation he provides is not preemptive at all and may fail to work properly if engine code is not written in a certain way.) However, there are easy parallels to draw between the code run by an engine under Dybvig's system and that of a TALT-R-certified program.

The code to be executed by an engine is given as a thunk, or a function of no arguments. If that function returns normally, its return value is the engine's result. To handle the case of running out of time, Dybvig's implementation defines a global variable called `do-expire` whose value is a function to be called by the engine's code upon discovering it has run out of fuel. In turn, `do-expire` uses `call/cc` to create a new engine which, if invoked, will cause `do-expire` to return the amount of additional fuel provided to continue the computation; this new engine is then passed to the continuation of the engine invocation, returning control to the client. In other words, `do-expire` suspends the execution of the engine indefinitely and (if it returns at all) returns the number of "ticks" until it must suspend again. **This is exactly the same as the behavior of the adaptive `yield` instruction in Section 8.1.** What is more, the function responsible for calling `do-expire` (called `decrement-timer`) is precisely analogous to the adaptive minor yield: it

attempts to decrement the amount of fuel remaining, and if this reaches zero, it calls `do-expire` and resets the fuel level to the value thus obtained.

One can therefore think of TALT-R as a type system for writing cooperative engines — if `yield` is implemented as a call to `do-expire`, then any well-typed TALT-R program (written using the adaptive `yield`) describes an engine that is guaranteed to behave well under a non-preemptive implementation of the engine mechanism.

8.3 Running Time

In contrast to most previous work on certification of time bounds, the bulk of this thesis has been devoted to a policy to which *any* program, appropriately compiled, can be made to conform. It is only a matter of inserting enough yields, and I have assumed that the `yield` instruction has no observable effect from the certified program's point of view. This assumption is more or less consistent with an applet-like or mobile agent-like model in which the untrusted code is executed by a supervisor or host on behalf of some other party. The relationship between an operating system kernel and most of the user processes running under it is similar in that the supervisor is not interested in the correctness, or even the performance, of the subordinate processes, and the safety policy exists to isolate processes from one another rather than to govern any kind of critical interaction.

In order to conduct a meaningful discussion of applications that require certified bounds on the total running time of a program or function, it is necessary to distinguish between two broad subclasses. The first is a time-sensitive client-server model in which the consumer is a host that executes untrusted code on behalf of other parties. In these systems, the consumer does not care about the *results* of the untrusted computation, even though it may care about the time it takes to compute them. The second class is a plugin-like model in which the supervisor (perhaps an OS kernel) *uses* some untrusted code to perform a useful function (perhaps a device driver or packet filter). It may well be important for reliability that the routines exported by the untrusted plugin produce meaningful results or effects within a certain amount of time, and it is therefore legitimate to include such requirements in the safety policy that plugins are expected to obey.

If we remove the `yield` instruction from TALT-R, then we can certainly devise types for functions that capture the timing policy in either of these two classes of application. For instance, the TALres-like code type

$$\forall a{:}\mathsf{N}.\forall \rho{:}\mathsf{TD}.\ \{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}(\{\mathtt{eax}{:}\mathtt{B4}, \mathtt{esp}{:}\rho, \mathtt{ck}{:}a\} \to 0) \times \rho, \mathtt{ck}{:}\overline{k} + a\} \to 0$$

(previously encountered in Chapter 6's discussion of Feeley yielding) describes a function that takes at most k instructions—and indeed, without the `yield` instruction, any function with this type will obey a very strict time bound.

The problem is that the somewhat impoverished logic available for clock reasoning in TALT-R cannot give this type to any function that contains any loops, recursion, or other nontrivial control flow. In fact, TALT-R as described seems much less suited to this kind of policy than TALres, which includes a good deal more abstract reasoning power for proving interesting time bounds. An obvious way to remedy this shortcoming of TALT-R is to endow it with a more powerful logic, perhaps similar to that of TALres, and in fact, the version of TALT implemented by Crary includes some LX-like features similar to those that give TALres its power, which might go a long way in this direction. It is unknown at this time, however, whether or not "LX-ified" TALT-R would provide a scalable solution.

Interestingly, major additions are not necessary in the case of the client-server model. Let TALT-R be altered such that the `halt` instruction may be executed at any time, regardless of context. (The official version of `halt` may only be performed when the stack is empty.) Then replacing the `yield` instruction in the minor yield with a `halt` produces a code fragment with the same typing properties as a minor yield. Now, if we compile a program using this modified minor yield (or "minor halt"), the resulting code is guaranteed to terminate after Y instructions: it either terminates normally, as the programmer intended, or it runs out of time and halts.

From the consumer's point of view, this is exactly what was required. Things are a little more complicated from the producer's point of view. The best results from this side are obtained by using as precise a method of "yield" placement as possible, perhaps the precise yield-on-jump strategy of Section 6.5. If we use such a precise strategy and ignore the overhead of instruction counting, it is safe to say that if the original program *would have finished* in under Y instructions without the minor halts, then the program compiled with minor halts will finish normally and with the same result. In other words, if we include the timing requirement among the criteria for correctness and ignore the complications of overhead, we can say that **the insertion of minor halts does not affect the semantics of a correct program.** Of course, the type system is no help when it comes to guaranteeing correctness — but certification is for the consumer's benefit, not the producer's. That the program must produce a useful result after its allotment of Y instructions is a self-imposed requirement of the *producer* and is therefore irrelevant to certification. If the producer can come up with a correct program, and satisfy herself of its correctness by any means whatsoever, then it can be certified such that the consumer will accept and run it.

Unfortunately, the success of this idea depends on the availability of an "escape route" through which a program can terminate without producing meaningful results. It may not always be possible to provide such a convenience: for instance, in plugin-like systems with hard real-time constraints, it can be safety-critical that the untrusted code not merely terminate within the allotted time but provide a useful result. In systems of this kind, the problem of certifying that a computation produces its results correctly and on time remains as difficult as ever.

8.4 Virtual Versus Real Clocks

All the safety policies discussed so far measure elapsed time by counting instructions. For this to make sense, it has been necessary to make the tacit assumption that a reasonable upper bound can be placed on the time any instruction takes to execute, so that multiplying this quantity by the number of instructions in a sequence may be assumed to provide a reasonable upper bound on the execution time of the sequence. Perhaps at one time in the history of digital computers that may have been true, but it represents a tremendous oversimplification of the behavior of present-day desktop and server microprocessors. The actual execution time of a memory read, for example, depends upon the state of the memory hierarchy and can vary over several orders of magnitude. As a result, it is far from simple to produce conservative static estimates of running time that are precise enough to be useful.

There are two issues that must be dealt with in order to develop a real-time version of TALT-R. First, certain instructions and certain combinations of operands take more cycles to compute than others; and second, the execution time of any given instruction is highly variable (which means it is usually much less than its maximum value). These issues are independent and can be addressed separately.

8.4.1 Unpredictability

Because it is so difficult to make usefully precise conservative static predictions of execution time, it simplifies matters to treat time as utterly unpredictable, except for very loose upper bounds known in advance. In order to make sense of this, it is necessary to distinguish between the TALT-R *abstract machine* and the *concrete* machine (something like an Intel Pentium) on which programs will actually be run. For the time being, I continue to assume the existence of a single upper bound on the execution time of any concrete machine instruction; this amount of time is the *virtual clock unit* (or *vcu*). It follows that any sequence of instructions during which the virtual clock of the abstract machine decreases from Y to zero takes the concrete machine *at most* Y vcu to execute — in fact, it will usually be much faster than that. If the desired safety policy is that the concrete machine yield at least once every Y vcu, then the expression of that safety policy as the requirement that the virtual clock remain nonnegative is overly conservative. Most of the time, when the virtual clock reaches zero, a concrete machine will still have time to perform many more instructions before yielding. The bad news is that, by assumption, we cannot statically make any better predictions than the very loose bound of one vcu per concrete instruction. We can, however, endow the abstract machine with the means to discover the availability of more cycles after its virtual clock has run out.

Fortunately, most modern computers and embedded microcontrollers possess a reliable way to measure time. For instance, the Intel x86 family of processors, starting with the Pentium, have a `rdtsc` ("read timestamp counter") instruction that produces the 64-bit number of clock cycles that have elapsed since the processor was last reset [38, Vol. 2, p. 3-604]. (Unlike the virtual clock of the TALT-R abstract machine, the cycles of this clock all take the same amount of real time.) Even in the absence of certification, a program that needs to observe a strict time limit can take advantage of features like this one to monitor its own progress. It is possible to extend the type system of TALT-R to certify the correctness of such techniques, guaranteeing that programs obey the policy even if conformance depends on the clock-checking behavior in a critical way.

Let TALT-R be extended with a new compound instruction `check` d that computes the real time remaining until the program's next deadline (whether for termination or yielding), stores the result (in virtual clock units) in destination d, and resets the virtual clock to this quantity. Since `check` is an abstract machine instruction implemented by a sequence of more than one concrete machine instruction, the amount by which it decrements the virtual clock will be greater than one; call this number c_{check}. (In other words, let c_{check} be the maximum number of vcus it can take to execute a `check` instruction on a concrete machine.) The following typing rule describes this new instruction:

$$\frac{(\Gamma(\mathsf{ck}) = c_{\text{check}} + t) \qquad (\Delta, a{:}\mathsf{N}); \Psi; \Gamma \vdash d : \mathcal{S}(a+t) \to \Gamma' \quad (\Delta, a{:}\mathsf{N}); \Psi; \Gamma'\{\mathsf{ck}{:}a+t\} \vdash I}{\Delta; \Psi; \Gamma \vdash \mathtt{check}\, d\, I}$$

This rule is very similar to the one for the adaptive `yield` in Section 8.1, in that it connects the integer value returned by the instruction to the value of the virtual clock. Indeed, from the point of view of the abstract machine, `check` has almost exactly the same semantics as the adaptive yield: it resets the virtual clock to an unpredictable quantity and stores that quantity in the given destination. In this case, however, note that there must be enough time on the virtual clock to perform the check (it is not a yield, after all, so its cost must be counted). Since the virtual clock is known to read at least $c_{\text{check}} + t$ before this instruction, and it costs at most c_{check}, it is safe to assume that it reads at least t afterward.

If the `check` operation is very cheap, it makes sense to do away with the use of a "clock reg-

ister" and rely solely on the built-in real clock to keep track of time. Figure 8.1 shows a code fragment that implements a *clock check*: assuming it is executed with at least $c_{\texttt{check}} + 2$ virtual clock cycles remaining, `CHECK` ensures that when control reaches the label `end` there is enough time on the clock to perform L instructions plus another `CHECK`. The implementation is analogous to the minor yield, except that the result of the `check` instruction is not stored in a "clock register." If that result is too small to satisfy the postcondition without yielding, a `yield` is performed.

```
CHECK  =
       // ck:\overline{c_check + 2}
       check eax
       // a:N, eax:S(a + 2-bar), ck:a + 2-bar
       subjae eax,eax,(L + c_check + 2),end
       ck:a
       yield
       // a' ↦ \overline{Y - L - c_check - 2}
       // ck:Y-bar = L-bar + (\overline{c_check + 2} + a')
end:
       // a' : N, ck:L-bar + (\overline{c_check + 2}) + a'
       // hence ck:L-bar + (\overline{c_check + 2})
```

Figure 8.1: Code for a Clock Check

8.4.2 Better Static Approximations

It is probably not necessary to assume that the virtual clock unit is the best statically available upper bound on the execution time of *any* instruction. Some instructions are faster than others: a register-to-register `mov` instruction, for instance, probably has a smaller range of possible running times than an arithmetic instruction whose destination is a memory location. It is possible to take advantage of this knowledge simply by discarding the assumption that each and every concrete instruction corresponds to exactly one tick of the abstract machine's virtual clock. In other words, we can redefine the virtual clock unit to be any amount of time we choose (perhaps most conveniently, we can set it equal to one cycle of the concrete hardware's clock) and assign different virtual costs to different abstract machine instructions, or even to different combinations of operands.

To be more precise, we can assign to each TALT-R instruction i a virtual cost c_i. Executing an i instruction decrements the virtual clock by c_i and takes at most c_i vcu on a concrete machine; the timing conditions in the instruction typing rules must be adjusted accordingly. All of the yield placement strategies described in Chapter 6 still work, keeping in mind that not all instructions have the same cost. It is still possible to use a clock register, but of course all the constants in the definitions of the minor clock and minor yield must be adjusted.

8.5 Bandwidth

In client-server or mobile agent systems, safety policies must often address the consumption of host resources other than time. In particular, if foreign code is permitted to access the network or file system, the host may be concerned about denial-of-service attacks based on excessive use of

these resources. The relevant policy in such situations may be one of bandwidth limiting, which can be accomplished by setting a *minimum* time that must elapse between calls to certain resource-related operations (such as disk reads or network sends). "Time" might mean real time, as in the previous section, or might be measured in instructions as in most of this thesis.

A bandwidth-limiting policy, expressed as a lower bound on the time between events, is essentially the dual of the responsiveness policy of TALT-R which is expressed as an upper bound. This suggests that a few minor changes to the type system might be sufficient to turn TALT-R into a theory for bandwidth certification. Where TALT-R had a designated "yielding" operation, let the bandwidth-certifying theory have a designated "consuming" operation; instead of a maximum yield period Y, specify a minimum "rest period" R that must elapse between consuming operations; and last but not least, in the register file subtyping rule, reverse the sense of the inequality constraint on the clock, thus:

$$\frac{\boxed{\Delta \vdash t \leq t' \text{ true}} \quad \Delta \vdash \tau_r \leq \tau'_r \text{ for each register } r}{\Delta \vdash \{\mathsf{eax}{:}\tau_{\mathsf{ax}}, \ldots, \mathsf{ebp}{:}\tau_{\mathsf{bp}}, \mathsf{esp}{:}\tau_{\mathsf{sp}}, \mathsf{ck}{:}t\} \leq \{\mathsf{eax}{:}\tau'_{\mathsf{ax}}, \ldots, \mathsf{ebp}{:}\tau'_{\mathsf{bp}}, \mathsf{esp}{:}\tau'_{\mathsf{sp}}, \mathsf{ck}{:}t'\}}$$

In this modified system, the virtual clock represents the amount of time that *must pass* before a consume operation is safe. Since it is safe to wait longer, this virtual clock decrements with every instruction until it reaches zero, then (rather than getting stuck) remains zero until it is reset to R by a consume operation. As in TALT-R, the ck term in a register file type is a conservative approximation of the virtual clock (but "conservative" here means the opposite of what it means in TALT-R): because of the constraint premise in the rule above, a register typing Γ describes a machine state in which the virtual clock is *at most* $\Gamma(\mathsf{ck})$.

Responsiveness involves an upper bound on elapsed time and thus requires conservatively overestimating the time on the clock; bandwidth involves a lower bound and requires conservatively underestimating the clock. A tempting question is whether it is possible to accomodate a policy that places both an upper and a lower bound on the time between successive events. In order to accomplish this we must no longer allow unrestricted over- or underestimation; the subtyping rule must not allow any variance in the clock term:

$$\frac{\boxed{\Delta \vdash t = t' \text{ true}} \quad \Delta \vdash \tau_r \leq \tau'_r \text{ for each register } r}{\Delta \vdash \{\mathsf{eax}{:}\tau_{\mathsf{ax}}, \ldots, \mathsf{ebp}{:}\tau_{\mathsf{bp}}, \mathsf{esp}{:}\tau_{\mathsf{sp}}, \mathsf{ck}{:}t\} \leq \{\mathsf{eax}{:}\tau'_{\mathsf{ax}}, \ldots, \mathsf{ebp}{:}\tau'_{\mathsf{bp}}, \mathsf{esp}{:}\tau'_{\mathsf{sp}}, \mathsf{ck}{:}t'\}}$$

(For convenience, rewriting of the clock term is still allowed, but the old and new terms must denote the same number.) All imprecision in static clock reasoning must now be accounted for explicitly using guarded types and singleton arithmetic. I have not investigated the implications of such policies for compilation of timing-ignorant programs.

8.6 Stack

Ordinary TALT does not provide any protection against stack overflow. The implementation relies on the operating system (more specifically, the virtual memory system) to detect excessive stack allocation and terminate the program, and the safety policy reflects this by specifying that any stack-growing instruction may fail and send the machine to the "halt" configuration. It would be nice to have a type system that rules out stack overflows, so that neither the safety policy nor the runtime system would need to account for the possibility of that error.

$$\frac{(\Gamma(\texttt{fss}) = \overline{n} + t) \quad \Delta; \Psi; \Gamma\{\texttt{esp} : \texttt{ns}n \times \Gamma(\texttt{esp}), \texttt{fss}{:}t\} \vdash I}{\Delta; \Psi; \Gamma \vdash \texttt{salloc}\, n\, I} \qquad \frac{\Delta \vdash \Gamma(\texttt{esp}) \leq \tau_1 \times \tau_2 \quad \Delta \vdash \tau_1 : \texttt{T}n \quad \Delta \vdash \tau_2 : \texttt{TD} \quad \Delta; \Psi; \Gamma\{\texttt{esp}{:}\tau_2, \texttt{fss}{:}\overline{n} + \Gamma(\texttt{fss})\} \vdash I}{\Delta; \Psi; \Gamma \vdash \texttt{sfree}\, n\, I}$$

Figure 8.2: Typing Rules for a Stack Usage Policy

We can adapt TALT-R to monitor stack space usage if we replace the virtual clock with a virtual counter representing the number of words of available (unused) stack segment memory. Concretely, we change the name of the `ck` pseudoregister to `fss` ("free stack space"), so that register file types look like

$$\Gamma ::= \{\texttt{eax}{:}\tau_{\texttt{ax}}, \ldots, \texttt{ebp}{:}\tau_{\texttt{bp}}, \texttt{esp}{:}\tau, \texttt{fss}{:}t\}$$

One key difference between space and time is that not every instruction consumes any stack space; in this case, it is only the stack-related instructions `call`, `push` and `salloc` that cause the stack to grow and hence should cause `fss` to decrease. Another important difference between space and time is that the stack space "consumed" by one of the instructions just mentioned can be recovered by a `ret`, `pop` or `sfree` instruction. The typing rules for non-stack-related instructions are exactly the same as in ordinary TALT; the rules for stack-related instructions are shown in Figure 8.2.

It is sound to consider a register file type Γ to be a subtype of Γ' if $\Gamma'(\texttt{fss}) \leq \Gamma(\texttt{fss})$. Not surprisingly, this is the same as the rule for time in a responsiveness policy: it is always safe to forget about some of an available resource.

Clearly, in order for this static tracking of stack availability to work, there must be some equivalent of a minor yield: it must be possible for a running program to determine how much stack space remains in order to know whether it has run out. There can be no precise equivalent of a `yield`, since the machine does not have an unlimited amount of virtual memory or an unbounded address space and hence one cannot always make room for more stack allocation. Something analogous to the `check` instruction from Section 8.4, however, does make sense: we can have a new instruction, call it `sscheck` ("stack segment check"), that returns the number of bytes remaining below the stack pointer. A sequence of instructions similar to the clock check in Figure 8.1 can compare this number to the amount the program wishes to allocate, and halt (or otherwise escape) if it is too small. The `sscheck` instruction itself is very simple to implement: it only needs to subtract the address of the beginning of the stack segment from the address stored in the stack pointer register.

8.7 Heap Allocation

Copying garbage collectors typically support allocation of memory by providing programs with an *allocation area* in which the mutator can write new objects. When this region of memory is full, the collector is notified and, after some work, returns a pointer to a new, empty allocation area. Since new objects are written into a contiguous block of memory, there is no need to consult or modify a "free list" with each allocation, making this interface very efficient for languages and programs that create new heap objects frequently.

In noncertified systems that use this protocol, the mutator keeps track of two pointer values to control allocation: the *allocation pointer* (`ap`), which points to the first unused word in the allocation

$$\frac{\Delta;\Psi;\Gamma \vdash o : \mathcal{S}(t) \quad \Delta;\Psi;\Gamma\{\texttt{faa}{:}t\} \vdash I}{\Delta;\Psi;\Gamma \vdash \texttt{gc}\, o\, I} \qquad \frac{(\Gamma(\texttt{faa}) = \overline{n} + t) \quad \Delta;\Psi;\Gamma\{r_d{:}\texttt{ns}W, \texttt{faa}{:}t\} \vdash I \text{ inits } r_d{:}\texttt{mbox}(\texttt{ns}n)}{\Delta;\Psi;\Gamma \vdash \texttt{malloc}\, r_d, n\, I}$$

Figure 8.3: Typing Rules for a Heap Allocation Policy

area, and the *limit pointer* (`lp`), the address of the end of the allocation area. At any given time, there are $\texttt{lp} - \texttt{ap}$ bytes available for allocation. To make space for an n-byte object, the mutator must first make sure that $\texttt{lp} - \texttt{ap} \leq n$. If this is not the case, then the garbage collector is called, and returns two new pointers defining a new allocation area that is guaranteed to be big enough. The mutator saves the value of `ap`, which will be the address of the new object, and updates `ap` to $\texttt{ap} + n$.

If a program must allocate many objects in rapid succession, it is wasteful to perform the comparison between `ap` and `lp` for every object; compilers therefore attempt to *coalesce* these operations, checking once to ensure the availability of space for several objects. This makes code shorter, saves time, and cuts down the number of code points from which the collector may be called, which can be helpful for tag-free collectors that must be able to parse the stack at every such point [68]. On the other hand, this practice requires a more complex safety policy than the `malloc` pseudoinstruction of TALT and TALT-R.

Creation of heap objects in a contiguous arena is in many ways analogous to stack allocation, and a type system similar to the one for stack segment management in the previous section can also capture the idiom of coalesced allocation pointer checks. Instead of `ck` or `fss`, let the register file type specify a term for `faa` ("free allocation area"), the number of unused bytes left in the allocation area. The two important instructions for manipulating the allocation area are `gc`, which causes the creation of a fresh allocation area of the requested size, and `malloc`. Typing rules for these instructions under such a policy are shown in Figure 8.3. The rule for `gc` is analogous to TALT-R's rule for `yield`: it has no observable effect but to reset the amount of free space to the quantity specified by the operand (terminating the program if this is not possible). The `malloc` rule is essentially the same as in TALT or TALT-R,[1] except that it consumes n bytes when allocating an object of size n.

Petersen *et al.*[58] have described a type theory for memory allocation based on ordered linear logic. In their calculus, called λ^{ord}, object creation is a three-step process: *reservation* creates a block of uninitialized memory, which then undergoes *initialization*, and *allocation* makes the new object available for use by the program. The object-creation area of the heap is divided into three parts: the *free space* that has not been touched in any way, the *frontier*, which is space that has been reserved but not allocated, and the portion that contains allocated objects. The frontier is treated specially by the type system, in that it allows updates that change the types of locations (from `ns` to useful types); the ordered linear typing discipline applied to the frontier prevents aliasing so these updates can be sound.

Petersen *et al.* treat the reservation step as a primitive that ensures that the frontier has the requested size, calling the garbage collector if necessary. To be more precise, it compares `ap` (which points to the boundary between used space and the frontier) to `lp` (which points to the end of the

[1] Although this rule is not discussed at all in this thesis – see Crary's TALT papers [13, 14].

free space) and notifies the collector if the difference is less than the requested size n. Having made sure there is enough free space to accomodate the request, it then "relabels" the first n bytes following `ap` as the new frontier; subsequent instructions can perform initialization in this region. A TALT-R-like system as described above could expose even more of the fine structure of allocation by separating the limit check (analogous to a minor yield or to `sscheck`), the call to the garbage collector (the `gc` instruction just described), and the creation of a frontier.

8.8 Chapter Summary

Although most of this thesis has been focused on TALT-R, a type system for certifying conformance to a specific responsiveness policy, small adjustments to this system suffice to enable certification of a wide variety of resource management policies, both timing-related and not. Some of these changes address practical shortcomings of TALT-R, such as the need to commit to a specific yield latency as part of the safety policy or the imprecision of instruction-counting as a measurement of time. Others point the way to certifying bounded running time in client-server applications, bandwidth limits, stack usage and proper interaction with a garbage collector.

Chapter 9

Conclusions

In this chapter, I present the results of some performance measurements of my compiler and discuss some of their implications. I go on to mention some avenues for future research in this area. Finally, I present my overall conclusions.

9.1 Performance Evaluation

As a preliminary performance experiment, I measured the effects of my yielding strategies on four different programs. The benchmarks range in complexity and qualitative behavior: `msort` applies a polymorphic merge-sort procedure to a pseudorandomly-generated linked list of integers; `qsort` applies quicksort to an array; `comb` computes a row of Pascal's triangle; and `tempo` is a port of the grid-based chess player developed by the ConCert project. Figure 9.1 shows the impact on execution time: for each benchmark the graph shows the execution time using Feeley yielding and Feeley polling, normalized with respect to the running time in ordinary TALT with no yielding requirements. The test programs were linked against a version of the runtime system in which the `yield` operation does nothing other than count the number of times it is called; thus the increases in running time are due only to function call overhead and/or clock register operations. All timing experiments were performed on a 730 MHz Pentium III desktop with 384 MB of RAM running Linux. As the chart shows, Feeley yielding slowed down programs by up to 67%, while Feeley polling never altered execution time by more than 6% in this experiment.

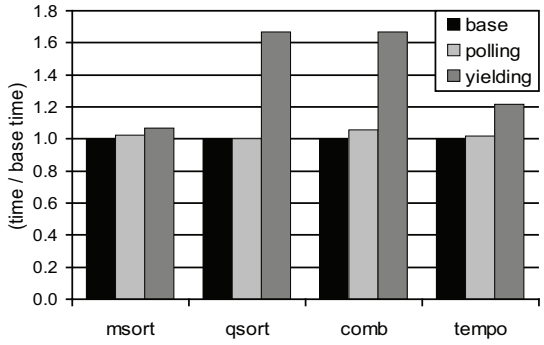


Figure 9.1: Normalized execution time ($Y = 1$ billion, $L = 500$, $E = 100$, $H = 50$)

These speed measurements clearly show that pure Feeley yielding without dynamic instruction counting is a losing strategy, and they seem to show that Feeley polling has a reasonably small effect on performance. This latter result, however, should be taken with a grain of salt, as there are many confounding factors. First of all, it is not fair to compare execution times between programs that use any particular yielding strategy and programs that perform no yields at all. The "base" version of each microbenchmark against which the others were compared is *not safe* with respect to the TALT-R yielding policy. Some of the difference between the base speed and the speed with Feeley polling is presumably "the price you pay for safety" and cannot be eliminated in *any* safe version of the program. On the other hand, all three versions of each program were produced by essentially the same compiler, and that compiler was a very naïve prototype that performed essentially no optimization and stored all temporary results on the stack. Since it did not do even the most basic kind of register allocation, it was completely insensitive to the increase in register pressure that dynamic instruction counting should have created. In other words, improving the code quality of the compiler may well have a greater impact on the non-yielding program than on the Feeley polling program, revealing my observed 6% differences as artificially small.

	Implicit Y/s	Explicit Y/s
`msort`	1 500 000	0.9
`qsort`	28	16
`comb`	0.77	13
`tempo`	210 000	0.03

Table 9.1: Yields under Feeley Polling

An important issue faced by the implementation but not apparent in my discussion of MiniTALT-R up to now is the timing behavior of TALT-R's `malloc` instruction, which allocates space in a garbage-collected heap. It is difficult to predict how long an invocation of `malloc` will take: those that trigger a garbage collection run much longer than those that do not. For my initial experiments I assumed that the runtime system would conservatively yield at every allocation. Table 9.1 shows that for `msort` and `tempo`, which do a lot of allocation, these "implicit yields" dominate the "explicit yields" introduced by the compiler: `msort` performs some 1.5 million yielding operations per second on average, only 0.9 of which are `yield` instructions. The `qsort` and `comb` benchmarks were carefully written to allocate as little as possible, and perform only a constant number of implicit yields per run.

These results clearly indicate a need for a better treatment of allocation. One possibility is to provide a version of `malloc` that has access to the program's clock register and yields only when needed. This "smart `malloc`" poses no problems in principle, but the implementation effort required is nontrivial and I have not attempted it. To estimate the performance improvement, I modified our implementation to assume a fixed cost for `malloc` (to simulate fast, non-collecting allocations) and instrumented the runtime system to count the number of garbage collections, which presumably would still have to yield. Table 9.2 shows the estimated yield rate for the smart `malloc` along with the rates we measured for Feeley yielding and polling. In all cases, the smart `malloc` reduces the total yield rate to less than one hundred yields per second. All of the remaining experiments discussed in this chapter use the simulated smart `malloc`.

	FY	FP	FPsm (est.)
`msort`	3.5×10^6	1.5×10^6	6.3×10^0
`qsort`	1.8×10^7	4.4×10^1	2.3×10^1
`comb`	1.7×10^7	1.4×10^1	1.3×10^1
`tempo`	6.6×10^6	2.1×10^5	6.4×10^1

Table 9.2: Yield Frequencies (Yields/sec)

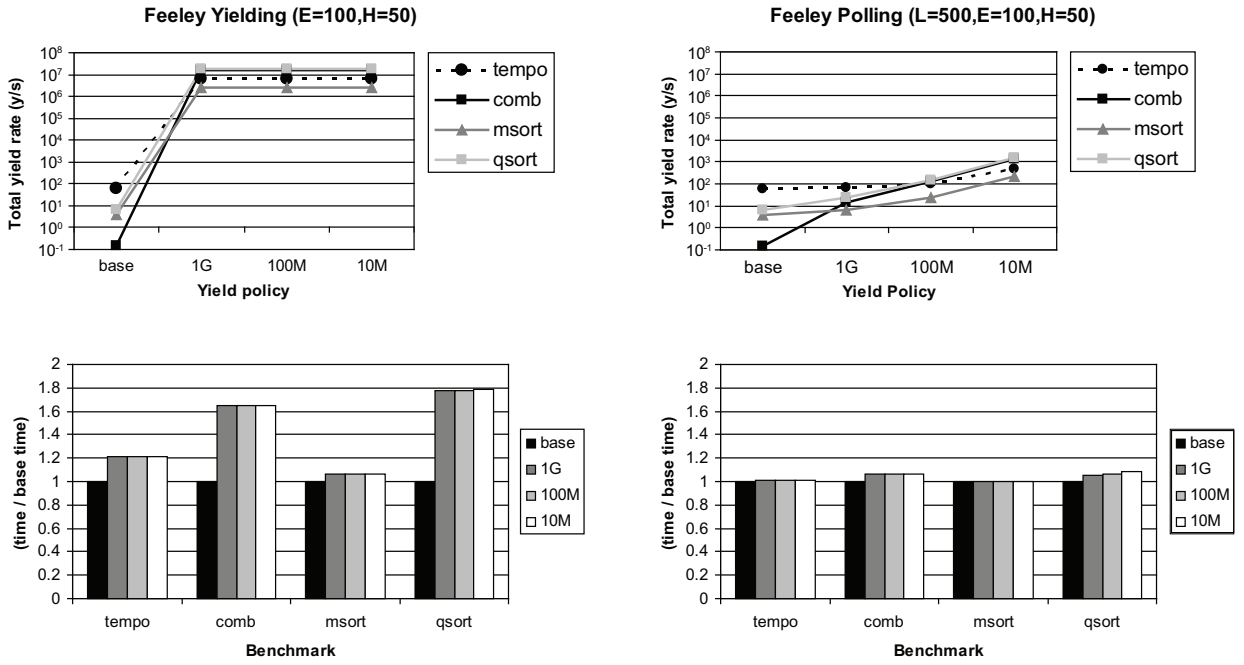


Figure 9.2: Effect of Y on Yielding Performance

Next, I looked at the effect of the policy yield period Y on the observed yield rate and running time of the benchmark programs. Each program was compiled for three different safety policies, with Y equal to 10 million, 100 million and one billion, assuming a smart `malloc` with a non-collecting allocation cost of 400 instructions. Figure 9.2 shows the results. The graphs on the left are for Feeley yielding, and those on the right are for Feeley polling. The yield frequency graphs show "base" values obtained by counting the number of garbage collections performed by non-yielding versions of each program and dividing by the elapsed time; since garbage collections are counted as implicit yields, this gives an idea of the amount of programmatically necessary yielding in each benchmark. As the figure shows, the performance of Feeley yielding is insensitive to the choice of Y. In fact, for each program, the three different yielding versions performed the *exact* same number of yields per run, suggesting that the compiler generated exactly the same code regardless of Y. This is not surprising, since even the smallest value of Y tested is much, much larger than the longest basic block length in any of the programs. For Feeley polling, the yield rate of the programs seemed to be more or less inversely proportional to Y, as one would expect. The overhead introduced by Feeley polling was also greater for smaller values of Y, but was still never measured to add more than 10% to the running time of any benchmark.

The next experiment, whose results are shown in Figure 9.3, looked at the effect of the minor yield period L on yielding performance. As before, the "base" numbers in both graphs were obtained by measuring the running time of non-yielding versions of the benchmarks and counting the number of garbage collections they performed. The total yield frequencies and normalized running times of the benchmarks were measured for versions compiled with Feeley polling with minor yield periods of 500, 700 and 1000 instructions. The impact of this parameter on running time is small and does not exhibit any universal trend; however, the graphs on the left clearly show that yield frequency increases as L increases. This effect is probably a different manifestation of

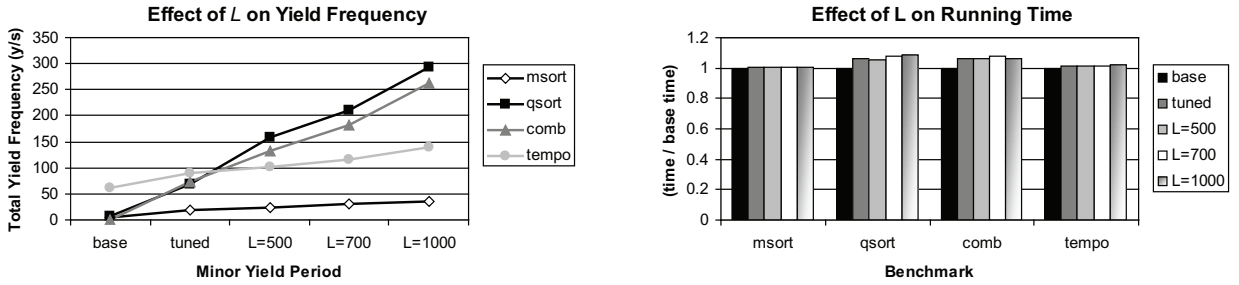


Figure 9.3: Effect of L on Yielding Performance (Y=100M, E=100, H=50)

the same phenomenon that accounts for the insensitivity of the direct Feeley yielding strategy to changes in Y: most basic blocks in most programs are sufficiently shorter than 500 instructions that the compiler places minor yields at exactly the same program points for each of these values of L, so the same number of minor yields occur per run of the program. Since each minor yield decrements the clock register by about L, larger values of L permit fewer minor yields per major yield, leading to a proportional increase in major yields. This indicates that smaller values of L are better than large values; unfortunately, it would have been very inconvenient to test values of L smaller than 500, since the assumed cost of the simulated smart `malloc` was 400 instructions.

Instead, I tested the hypothesis that the minor-to-major yield ratio was the determining factor of yield frequency by hand-tuning the minor yields in a select few extremely commonly used functions in the library shared by all TALT-R programs.[1] The tuning consisted merely of adjusting the quantity subtracted from the clock register in each minor yield so that it corresponded exactly to the requirements of the ensuing basic block (and correcting the relevant typing annotations). No minor yields were added or removed, and no other changes were made to the code. Non-tuned portions of the program were identical to those compiled with $L = 500$. The performance figures for the resulting "tuned" versions of the benchmarks are shown alongside the others in Figure 9.3. As the graphs show, this improvement of the precision of minor yielding, even when applied to a rather small portion of each program's TALT-R code, produced noticeable improvements in yield frequency for all four benchmarks. I take this as evidence that any serious implementation of yielding with TALT-R should use the precise yield-on-jump strategy described at the end of Chapter 6 rather than a simple minor yield strategy that treats all blocks in a program the same.

Figure 9.4 shows the results of a very similar experiment to measure the effect of the Feeley function "cost" E on yielding performance. Benchmarks were compiled using function costs of 25, 50, 80 and 100 instructions; the minor yield period was 500. The results are mixed: the `comb` and `qsort` benchmarks, which use arrays and iteration more than they use allocation or recursion, seemed to perform best with E equal to 80. For `tempo`, probably the benchmark most representative of typical Popcorn code, 100 was better. The `msort` benchmark yielded infrequently regardless of E. These results are less impressive than the effect of Y or of L, probably because there is a smaller range (zero to L) over which E can be varied; the fact that they are highly program-dependent suggests that a compiler capable of varying the value of E between functions (which, remember, requires *inter*procedural flow analysis) would perform better than one that treats all functions the same. Again, using precise yield-on-jump instead of a Feeley strategy is perhaps the easiest solution.

[1] Namely, the functions implementing multiplication and division in terms of shifts and addition, needed to work around the absence of the native instructions for these operations in the implementation of TALT.

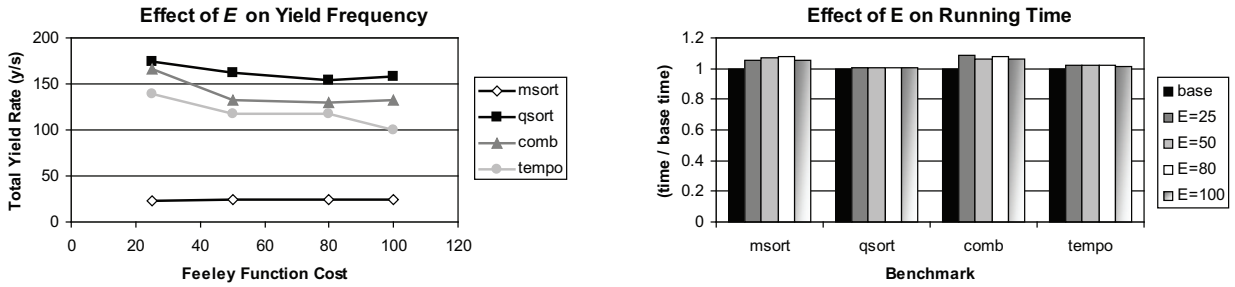


Figure 9.4: Effect of E on Yielding Performance (Y=100M, L=500, H=50)

A major confounding factor in these experiments is that they were run under Linux, a traditional preemptive kernel; the preemption of the processes by the kernel was completely unrelated to their executing the trivial stub implementation of the `yield` instruction. For the measurements shown in Figure 9.5, I replaced that dummy `yield` with one that yielded the CPU using the Linux `sched_yield` system call. Each program tested was run in parallel with a CPU-hungry "drone" process that also yielded in a manner consistent with the TALT-R policy.[2] The figure shows, for three choices of L and for "tuned" versions produced as described earlier, the elapsed time and the percentage of that time allocated to the benchmark process as measured by the Linux `time` utility. The base times with respect to which the elapsed times are normalized were measured by running a non-yielding version of the benchmark in parallel with a non-yielding version of the drone — that is, by allowing the Linux kernel to manage the competition between them in its usual way. The `qsort` benchmark was not used in this experiment because it did not run long enough (less than half a second) for the CPU allocation measurements to be meaningful.

As expected, benchmark processes compiled with larger values of L, which produced higher yield rates in the experiment described earlier, took longer to run and fared less well in competition with the drone process. Also predictably, the `msort` benchmark, which tended to yield the least often in other experiments, did the best when competing for the CPU, with the hand-tuned version even out-competing the drone; `comb`, which generally yielded more often, performed the worst. What is also worth noting is that the impact of L on elapsed time was much greater in this experiment, where each major yield forces a context switch, than in the earlier experiment where the yield consisted of a function call and little else.

It does seem discouraging that explicit yielding seems to produce such inequitable allocation of the CPU. Fortunately, I do not think this necessarily represents an inherent flaw in the idea of allowing processes to participate in scheduling. Rather, one must remember that the Linux kernel is accustomed to being in complete control of the allocation of time to processes. The `sched_yield` call defeats this careful design: it unconditionally yields the remainder of the current process's time slice, and if another runnable process exists, the caller is guaranteed to be suspended. A kernel that expected processes to yield rather than be preempted would have to recognize that their yielding patterns would probably be erratic and differ between programs, and would have to take this into account when scheduling threads for execution. The yielding operation provided by such a kernel would represent an *opportunity* for a task switch but would not *force* one if the calling process deserved more time.

[2]To be precise, the drone was a small C program, not requiring much memory, with an inner loop that performed some integer arithmetic and yielded every m iterations, where m was calculated such that yields occurred on the order of every $Y/2$ instructions.

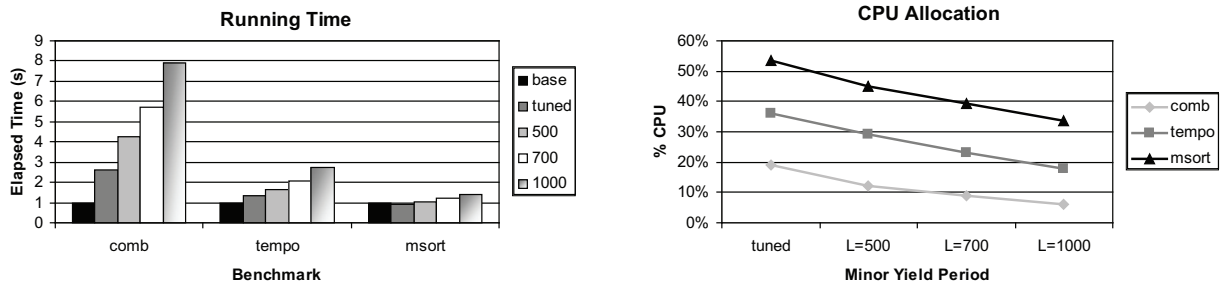


Figure 9.5: Effect of L on Competition (Y=10M, E=100, H=50)

Another flaw in this attempt to simulate a cooperative scheduling environment is that even though both the drone and the benchmark process in each trial yielded the CPU very frequently, each of them was still vulnerable to preemption between yields, making the actual impact of context switches difficult to measure. Creating the necessary testing environment to explore the true effects of static enforcement on cooperative scheduling and compare the results to preemptive scheduling would be a significant undertaking.

9.2 Discussion and Future Directions

In this section, I discuss some of the interesting features of the work I have done that suggest potentially worthwhile topics for further study. The areas I see for potential future work in this area fall into two categories: those that amount to improving the particular safety policy I have studied and system I have implemented, and those that further explore the potential capabilities of the techniques I have used here.

9.2.1 Improvements to Implemented System

In spite of the groundwork laid by earlier researchers at CMU and elsewhere, writing a type-preserving compiler from scratch is still far from easy, even for a very simple source language. The level of sophistication I was able to achieve in the time allotted to me for this project was quite low. Many improvements are needed if the system I built is to be useful, or if measurements of its performance are to be taken seriously. Most of these improvements, however, are not particularly interesting research avenues.

TALT External Syntax Quite apart from the challenges of timing certification, I found the task of generating well-typed EXTALT output from my compiler surprisingly difficult. The main culprit was "coercions", the reified subtyping derivations with which operands must often be annotated. EXTALT's coercions tend to be verbose, repetitive and finicky, especially those that apply to the type of the stack. Nearly every control transfer instruction output by my compiler must coerce the stack so that its type matches up exactly with the type expected by the target code block. Since the stack type is a long, right-associated product of types most of which are not changing, the necessary coercion is a long, right-associated "product" of coercions most of which are the identity (and often, those that are not the identity are the `forget` coercion from some type to

nonsense). The structure of the coercion must mirror the structure of the stack type exactly, but failures to do so often result in almost incomprehensible error messages. Coercions that must "reassociate" a large product of types (*e.g.* turn $(\tau_1 \times (\tau_2 \times (\cdots \times \tau_n)))$ into $(((\tau_1 \times \tau_2) \times \cdots) \times \tau_n))$ are also extremely annoying to generate.

Another locus of painful coercion is the compilation of Lilt's `case` construct for eliminating variant types. Since Lilt's variants can have any number of summands but TALT's union types are binary, the coercions witnessing the subtyping premises of the `cmpjcc` instruction typing rule (not to mention the coercions required to give the case subject a binary union type in the first place) are large and hard to get right. All in all, I estimate that more than half of the time it took to write the back end of the compiler was spent debugging coercions. Something must be done about this.

Some of the problems I encountered could be solved by adding support for some set of non-trivial utility coercions to the front end of the EXTALT assembler. These could take care of things like reassociating large products, permuting large unions or intersections, and so on. Allowing EXTALT programs to contain "coercion definitions" akin to type definitions would also probably result in shorter, more readable assembly code. But it is not clear whether such techniques would be particularly useful for the problem of stack coercions. It just seems unfair that, although IA-32 programmers are all but forced to treat the first several words of the stack as if they were registers, the EXTALT elaborator can automatically insert `forget` coercions for actual registers but not for stack slots. I predict that very few people will be willing to program in EXTALT as long as this is the case.

Responsiveness Implementation Improvements Some interesting questions are raised by the effect the `malloc` operation was observed to have on yield rate earlier in this chapter. In particular, the "smart `malloc`" simulated in the experiments is fictitious; the question of how such a thing might be implemented reveals a larger issue, namely: *When separately compiled program modules and libraries must cooperate on time management, what protocols should govern the interfaces between them to maximize both flexibility and performance?* An even broader issue is, *how far do the implications of a particular choice of yielding strategy reach?* Can the decision to use, say, Feeley versus call-return yielding, or the decision to reserve a register for instruction counting be viewed as an implementation detail of a module and hidden behind a timing-agnostic interface? Or does the abstraction boundary introduce an "impedance mismatch" that hurts performance unacceptably?

A related question is whether the Feeley placement strategy (whether for direct yield placement or minor yield placement) discards too much useful information by assigning all functions the same fictitious "cost". Recall that the number E is an upper bound on three quantities: the "cost" of a function as seen by callers, the number of instructions between function entry and the first yield, and the number of instructions between the last yield and function exit. For many functions, the latter two are not very sensitive to the initial value of the clock; thus rather than choosing a single cost for all functions ahead of time, one should usually be able to *compute* the cost for each function in a program separately. A whole-program analysis would be needed in order to take advantage of this more precise information — is it worth it? Only further study can answer that question.

When I began my investigation of responsiveness certification, it was expected that simple strategies like Feeley yielding and dynamic checking would be insufficient to produce acceptable performance. My performance measurements seem to suggest that this is not the case, but as I suggested in my discussion of those results, it is possible that an aggressive optimizing compiler would suffer more from the cost of dynamic checking than my naïve compiler. If this is the case,

then further improvement of the constraint reasoning in TALT-R is called for, along with investigation of program analyses to detect opportunities for moving, hoisting or eliminating yields or clock checks. The static computation typical of LXres [18] is a good place to start.

The yield placement strategies I have studied were all designed with the goal of insulating programmers from the issue of responsiveness. This is an important thing to be able to do, because it allows experienced programmers who are accustomed to a preemptive setting (where the operating system insulates them from responsiveness) can also work on certification-based systems without having to learn anything new. More importantly, it means that code written for traditional operating systems (in type-safe languages) will be portable to certification-based ones — it will only need to be recompiled. Finally, the ability of TALT-R to account for dynamic instruction-counting schemes (like my minor yields) means that the difficulty of porting *compilers* to a certification-based setting is also small and need not increase their complexity by much. Realistically, however, the benefits of certification over preemption for timing policy enforcement cannot be fully realized without help from programmers. Here, again, the work of Crary and Weirich on PopCron and TALres may provide a useful starting point.

9.2.2 Applicability

Real-Time Programming An obvious application for a system that certifies timing properties is in real-time programming. As discussed in Chapter 8, TALT-R could in principle be modified to certify policies based on real time rather than on instruction counting. The suggestions for doing this given in that chapter, however, were mostly speculative. Further study is needed to determine whether the level of imprecision resulting from conservative estimates of instruction cost or the cost of frequent checks of the real clock is too great for such an approach to be practical. A major obstacle to improving the precision of static reasoning about time is, of course, that the cost of any instruction is highly dependent on the dynamic context in which it is executed, that is, the recent history of the process that determines the state of the processor pipelines, caches and virtual memory. It is conceivable that usable levels of precision can only be achieved by reasoning about the execution time of *sequences* of instructions rather than individual ones. How to integrate such reasoning into a type system is, to my knowledge, an open problem.

Software-Based Process Isolation I have hinted throughout this thesis that the responsiveness policy of TALT-R is the kind of timing policy that an operating system kernel might want to enforce, but I have not put this claim to the test. It will not be clear without further experimentation just how realistic a yielding policy like TALT-R's is for something so central to everyday computing life as the process scheduler of an operating system. More expressive policies may be needed in order to create the right balance between safety and flexibility that preserves system reliability without compromising efficiency.

This thesis has been primarily concerned with the type-theoretic and language-related issues involved in certifying responsiveness, and consequently has largely ignored the concrete semantics of the `yield` operation. Where concrete intuition has been required, I have assumed that "yielding" necessarily always involves a context switch and/or interprocess communication — but this is nowhere reflected in the operational semantics of the TALT-R abstract machine or the static semantics of TALT-R. From the point of view of the formalism, the only thing that matters about the `yield` instruction is that it must, as a matter of safety, be performed periodically with a certain frequency. Any such operation can easily take the place of `yield` without changing the type theory — but the performance characteristics of that operation probably *will* affect may im-

plementation decisions, including the choice of "yield" placement strategy. Fortunately, TALT-R is flexible enough to support several reasonable choices and extensible enough to support many more.

Indeed, from an efficiency point of view, forcing processes to surrender control of the machine after *every* Y instructions (or nanoseconds) is not a very good policy. It might very well be better, if the hardware supports it, to require merely that a program examine some flag periodically, and yield control whenever it finds the flag set. This is the behavior widely known as "polling", although I have used that word in this thesis for something slightly different. Although their typing properties are identical, this "poll" operation differs from a true yield in that it is fast in the common case: usually, no event will have occurred to set the flag, so no context switch will be needed and the program will be able to proceed. In fact, it was for this kind of polling that Feeley designed his placement strategy [23]. The positive results of his experiments indicate that direct placement of polling operations using the Feeley strategy should be a viable approach to compilation of timing-ignorant programs in this setting. (In particular, since the poll operation is fast, one would expect little or nothing to be gained from dynamic instruction counting using a clock register.) Importantly, this is no different from TALT-R with yields as far as typing, certification and safety are concerned; it has only to do with mapping the TALT-R abstract machine to concrete hardware in the most useful way possible for the application at hand.

The difference between yielding and polling (in this more usual sense) is a manifestation of a deep question opened by the availability of certification-based enforcement for timing policies, namely: *Who* ***should*** *decide when a process yields?* More broadly, *what should the respective roles of the operating system and a user process be in managing resources?* The answer is not obvious: on the one hand, only the operating system knows enough about the state of all the computer's hardware and software to know when a yield is needed; on the other, only the user process knows when a yield will disrupt it the least. If user processes are allowed to yield on their own terms, the cost of saving useless state can be reduced and yields can be arranged not to fall in the middle of cache-sensitive inner loops; but unless they are guaranteed to yield soon enough after a yield becomes necessary, the OS will have no choice but to incur the cost of preemptive management. The tradeoffs between preemptive and non-preemptive scheduling, not unknown territory to operating systems experts, must be reexamined in light of the fact that with certification, preemptive systems no longer have a monopoly on stability, reliability or fairness.

9.3 Conclusion

The world of code certification is at a turning point. We have more or less mastered the type theory, compilation techniques and logics needed to ensure a baseline of type and memory safety in small to medium-sized chunks of untrusted code. The focus of much of the work so far has been on mobile code applications, including applets, mobile agents, and large-scale distributed computing; these applications have in common the fact that they run on top of modern desktop or server operating systems on which the host environments can rely for many aspects of safety, such as resource usage, that the certification does not cover. Our attention is now turning toward increasingly expressive safety policies and increasingly powerful certification technologies that can take over more and more of these duties traditionally assigned to operating systems.

There are a number of rewards to be found just around this bend. Just as statically type-safe programming languages admit more compiler optimizations and allow programs to run with less

safety-related overhead than dynamically typed languages, so too can static enforcement of safety policies by code certification improve the performance of software. By making it safe to lower the hardware-based barriers between processes, between components of a process, and even between the kernel and user processes, static enforcement creates the potential for tighter coupling between components, resulting in leaner and more efficient computer systems. This potential, if realized, will benefit not only conventional desktop and server systems but also the growing number of smaller devices, such as mobile phones, handheld computers and smart cards, for which power and memory are still scarce resources.

I have argued in this thesis that **timing policies** play critical roles in the security and reliability of real systems. If we are to take full advantage of the power of certified code, we must be able to enforce such policies statically. To show that this is possible, I have studied the certification of an extremely common timing policy that I call *responsiveness*. I have exhibited some strategies for compiling programs in a general intermediate language so that they satisfy this policy, and I have shown that a surprisingly simple type theory suffices to prove that programs compiled using these techniques are responsive. My theory, called TALT-R, fits well into an existing certification framework and is easily extended to a wide range of timing and resource management policies. I hope that the work I have described here constitutes some first steps toward static enforcement strategies for these policies that can truly compete with the currently popular dynamic approach.

Appendix A

Complete MiniTALT Semantics

Where possible, the inference rules in this appendix are labeled with the names of the corresponding rules in the Twelf formalization of TALT type safety; these labels are in `typewriter font`. Due to differences between MiniTALT and full TALT, and between the media of typeset inference rules and Twelf code, this correspondence is rough. The Twelf version of a rule may differ significantly from that presented here. Rules in this appendix that do not correspond to anything in the Twelf codebase are given ad hoc names which are written in SMALL CAPS.

A.1 Static Semantics

A.1.1 $\boxed{\Delta \vdash c : K, \Delta \vdash \Gamma}$ Static Term Formation

$$\frac{((\alpha{:}K) \in \Delta)}{\Delta \vdash \alpha : K}\ (\text{KOF\_VAR}) \qquad \frac{}{\Delta \vdash \mathsf{ns}i : \mathsf{T}i}\ (\texttt{kof\_ns}) \qquad \frac{}{\Delta \vdash \mathtt{Bi} : \mathsf{T}i}\ (\text{KOF\_B}) \qquad \frac{}{\Delta \vdash B : \mathsf{N}}\ (\text{KOF\_NUMLIT})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{T}}\ (\texttt{kof\_prod}) \qquad \frac{\Delta \vdash \tau_1 : \mathsf{TD} \quad \Delta \vdash \tau_2 : \mathsf{TD}}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{TD}}\ (\text{KOF\_PROD\_D})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}j}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{T}(i+j)}\ (\text{KOF\_PROD\_I}) \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathtt{box}(\tau) : \mathsf{T}W}\ (\texttt{kof\_box}) \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathtt{mbox}(\tau) : \mathsf{T}W}\ (\texttt{kof\_mbox})$$

$$\frac{\Delta \vdash \tau : \mathsf{TD}}{\Delta \vdash \mathtt{sptr}(\tau) : \mathsf{T}W}\ (\texttt{kof\_sptr}) \qquad \frac{\Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash x : \mathsf{N}}{\Delta \vdash \tau \uparrow x : \mathsf{T}}\ (\texttt{kof\_exp}) \qquad \frac{\Delta \vdash \tau : \mathsf{TD} \quad \Delta \vdash x : \mathsf{N}}{\Delta \vdash \tau \uparrow x : \mathsf{TD}}\ (\text{KOF\_EXP\_D})$$

$$\frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau \uparrow B : \mathsf{T}(i \cdot B)}\ (\text{KOF\_EXP\_I}) \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \tau \uparrow 0 : \mathsf{T}0}\ (\text{KOF\_EXP\_Z}) \qquad \frac{\Delta \vdash \Gamma}{\Delta \vdash \Gamma \to 0 : \mathsf{T}W}\ (\texttt{kof\_arrow})$$

$$\frac{\Delta \vdash x : \mathsf{N}}{\Delta \vdash \mathtt{set}_{=}(x) : \mathsf{T}W}\ (\texttt{kof\_seteq}) \qquad \frac{\Delta \vdash x : \mathsf{N}}{\Delta \vdash \mathtt{set}_{<}(x) : \mathsf{T}W}\ (\texttt{kof\_setlt}) \qquad \frac{\Delta \vdash x : \mathsf{N}}{\Delta \vdash \mathtt{set}_{>}(x) : \mathsf{T}W}\ (\texttt{kof\_setgt})$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T}}{\Delta \vdash \forall \alpha{:}K.\tau : \mathsf{T}}\ (\texttt{kof\_forall}) \qquad \frac{(K' \in \{\mathsf{TD}, \mathsf{T}i\}) \quad \Delta \vdash c : K \quad \Delta, \alpha{:}K \vdash \tau : \mathsf{T} \quad \Delta \vdash \tau[c/\alpha] : K'}{\Delta \vdash \forall \alpha{:}K.\tau : K'}\ (\text{KOF\_FORALL\_DI})$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T}}{\Delta \vdash \exists \alpha{:}K.\tau : \mathsf{T}}\ (\texttt{kof\_exists}) \qquad \frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T}i}{\Delta \vdash \exists \alpha{:}K.\tau : \mathsf{T}i}\ (\text{KOF\_EXISTS\_I}) \qquad \frac{}{\Delta \vdash \mathtt{void} : \mathsf{T}i}\ (\texttt{kof\_void})$$

$$\frac{\Delta, \alpha{:}\mathsf{T} \vdash \tau : \mathsf{T}}{\Delta \vdash \mu\alpha.\tau : \mathsf{T}}\ (\texttt{kof\_rec}) \qquad \frac{(K \in \{\mathsf{TD}, \mathsf{T}i\}) \quad \Delta, \alpha{:}\mathsf{T} \vdash \tau : \mathsf{T} \quad \Delta \vdash \tau[\mu\alpha.\tau/\alpha] : K}{\Delta \vdash \mu\alpha.\tau : K}\ (\text{KOF\_REC\_DI})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{T}}\ (\texttt{kof\_meet}) \qquad \frac{\Delta \vdash \tau_1 : \mathsf{TD} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{TD}}\ (\text{KOF\_MEET\_D1})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{TD}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{TD}}\ (\text{KOF\_MEET\_D2}) \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{T}i}\ (\text{KOF\_MEET\_I1})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}i}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{T}i}\ (\text{KOF\_MEET\_I2}) \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \vee \tau_2 : \mathsf{T}}\ (\texttt{kof\_join})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}i}{\Delta \vdash \tau_1 \vee \tau_2 : \mathsf{T}i}\ (\text{KOF\_JOIN\_I}) \qquad \frac{\Delta \vdash \tau : \mathsf{TD}}{\Delta \vdash \tau : \mathsf{T}}\ (\text{KOF\_D}) \qquad \frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau : \mathsf{TD}}\ (\text{KOF\_I})$$

$$\frac{\Delta \vdash \tau_{\mathtt{sp}} : \mathsf{TD} \quad \Delta \vdash \tau_r : \mathsf{T}W \textit{ for } r \in \{\mathtt{ax}, \ldots, \mathtt{bp}\}}{\Delta \vdash \{\mathtt{eax}{:}\tau_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{bp}}, \mathtt{esp}{:}\tau_{\mathtt{sp}}\}}\ (\texttt{rtpok\_})$$

A.1.2 $\boxed{c_1 \equiv c_2, \Gamma \equiv \Gamma'}$ Static Term Equivalence

$$\frac{}{c \equiv c}\ (\texttt{equiv\_reflex}) \qquad \frac{c_2 \equiv c_1}{c_1 \equiv c_2}\ (\texttt{equiv\_symm}) \qquad \frac{c_1 \equiv c_2 \quad c_2 \equiv c_3}{c_1 \equiv c_3}\ (\texttt{equiv\_trans})$$

$$\frac{}{(\lambda\alpha{:}K_2.c_1)c_2 \equiv c_1[c_2/\alpha]}\ (\texttt{equiv\_beta}) \qquad \frac{(\alpha \textit{ not free in } c)}{(\lambda\alpha{:}K_1.c\alpha) \equiv c}\ (\texttt{equiv\_eta})$$

$$\frac{c_1 \equiv c_2}{\lambda\alpha{:}K_1.c_1 \equiv \lambda\alpha{:}K_1.c_2}\ (\texttt{equiv\_lam}) \qquad \frac{c_1 \equiv c_1' c_2 \equiv c_2'}{c_1\ c_2 \equiv c_1'\ c_2'}\ (\texttt{equiv\_app})$$

$$\frac{x_1 \equiv x_2}{\mathtt{set}_{=}(x_1) \equiv \mathtt{set}_{=}(x_2)}\ (\texttt{equiv\_seteq}) \qquad \frac{x_1 \equiv x_2}{\mathtt{set}_{<}(x_1) \equiv \mathtt{set}_{<}(x_2)}\ (\texttt{equiv\_setlt})$$

$$\frac{x_1 \equiv x_2}{\mathtt{set}_{>}(x_1) \equiv \mathtt{set}_{>}(x_2)}\ (\texttt{equiv\_setgt}) \qquad \frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2'}\ (\texttt{equiv\_prod}) \qquad \frac{\tau \equiv \tau' \quad x \equiv x'}{\tau \uparrow x \equiv \tau' \uparrow x'}\ (\texttt{equiv\_exp})$$

$$\frac{\Gamma \equiv \Gamma'}{\Gamma \to 0 \equiv \Gamma' \to 0}\ (\texttt{equiv\_arrow}) \qquad \frac{\tau \equiv \tau'}{\mathtt{box}(\tau) \equiv \mathtt{box}(\tau')}\ (\texttt{equiv\_box}) \qquad \frac{\tau \equiv \tau'}{\mathtt{mbox}(\tau) \equiv \mathtt{mbox}(\tau')}\ (\texttt{equiv\_mbox})$$

$$\frac{\tau \equiv \tau'}{\mathtt{sptr}(\tau) \equiv \mathtt{sptr}(\tau')}\ (\texttt{equiv\_sptr}) \qquad \frac{\tau \equiv \tau'}{\forall\alpha{:}K.\tau \equiv \forall\alpha{:}K.\tau'}\ (\texttt{equiv\_forall})$$

$$\frac{\tau \equiv \tau'}{\exists\alpha{:}K.\tau \equiv \exists\alpha{:}K.\tau'}\ (\texttt{equiv\_exists}) \qquad \frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1 \wedge \tau_2 \equiv \tau_1' \wedge \tau_2'}\ (\texttt{equiv\_meet}) \qquad \frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1 \vee \tau_2 \equiv \tau_1' \vee \tau_2'}\ (\texttt{equiv\_join})$$

$$\frac{\tau \equiv \tau' \quad \tau_r \equiv \tau_r' \textit{ for } r \in \{\mathtt{ax}, \ldots, \mathtt{bp}\}}{\{\mathtt{eax}{:}\tau_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{bp}}, \mathtt{esp}{:}\tau_{\mathtt{sp}}\} \equiv \{\mathtt{eax}{:}\tau_{\mathtt{ax}}', \ldots, \mathtt{ebp}{:}\tau_{\mathtt{bp}}', \mathtt{esp}{:}\tau_{\mathtt{sp}}'\}}\ (\texttt{equivr\_})$$

A.1.3 $\boxed{\Delta \vdash \tau_1 \leq \tau_2, \Delta \vdash \Gamma \leq \Gamma'}$ Subtyping

$$\frac{}{\Delta \vdash \tau \leq \tau}\ (\texttt{reflex}) \qquad \frac{\tau_1 \equiv \tau_2}{\Delta \vdash \tau_1 \leq \tau_2}\ (\texttt{reflexeq}) \qquad \frac{\Delta \vdash \tau_1 \leq \tau_3 \quad \Delta \vdash \tau_3 \leq \tau_2}{\Delta \vdash \tau_1 \leq \tau_2}\ (\texttt{trans})$$

$$\frac{\Delta \vdash \tau_1 \leq \tau_1' \quad \Delta \vdash \tau_2 \leq \tau_2'}{\Delta \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}\ (\texttt{prod\_sub}) \qquad \frac{\Delta \vdash \tau \leq \tau'}{\Delta \vdash \tau \uparrow x \leq \tau' \uparrow x}\ (\texttt{exp\_sub}) \qquad \frac{\Delta \vdash \Gamma' \leq \Gamma}{\Delta \vdash \Gamma \to 0 \leq \Gamma' \to 0}\ (\texttt{arrow\_sub})$$

$$\frac{\Delta \vdash \tau \leq \tau'}{\Delta \vdash \mathtt{box}(\tau) \leq \mathtt{box}(\tau')}\ (\texttt{box\_sub}) \qquad \frac{\Delta \vdash \tau \leq \tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta \vdash \mathtt{mbox}(\tau) \leq \mathtt{mbox}(\tau')}\ (\texttt{mbox\_sub})$$

$$\frac{}{\Delta \vdash \mathtt{mbox}(\tau) \leq \mathtt{box}(\tau)}\ (\texttt{forgetm}) \qquad \frac{\Delta \vdash \tau \leq \tau' \quad \Delta \vdash \tau : \mathsf{TD} \quad \Delta \vdash \tau' : \mathsf{TD}}{\Delta \vdash \mathtt{sptr}(\tau) \leq \mathtt{sptr}(\tau')}\ (\texttt{sptr\_sub})$$

$$\frac{\Delta, \alpha{:}K \vdash \tau \leq \tau'}{\Delta \vdash \forall\alpha{:}K.\tau \leq \forall\alpha{:}K.\tau'}\ (\texttt{forall\_sub}) \qquad \frac{\Delta, \alpha{:}K \vdash \tau \leq \tau'}{\Delta \vdash \exists\alpha{:}K.\tau \leq \exists\alpha{:}K.\tau'}\ (\texttt{exists\_sub})$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T} \quad \Delta \vdash c : K}{\Delta \vdash \forall\alpha{:}K.\tau \leq \tau[c/\alpha]}\ (\texttt{forall\_elim}) \qquad \frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T} \quad \Delta \vdash c : K}{\Delta \vdash \tau[c/\alpha] \leq \exists\alpha{:}K.\tau}\ (\texttt{exists\_intro})$$

$$\frac{(\alpha \notin \tau)}{\Delta \vdash \tau \leq \forall\alpha{:}K.\tau}\ (\texttt{gen}) \quad \frac{(\alpha \notin \tau)}{\Delta \vdash \exists\alpha{:}K.\tau \leq \tau}\ (\texttt{cogen}) \quad \frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau \leq \mathsf{ns}i}\ (\text{NS\_SUB}) \quad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathtt{void} \leq \tau}\ (\texttt{void\_sub})$$

$$\frac{\Delta, \alpha{:}\mathsf{T} \vdash \tau : \mathsf{T}}{\Delta \vdash \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau}\ (\texttt{rec\_intro}) \quad \frac{\Delta, \alpha{:}\mathsf{T} \vdash \tau : \mathsf{T}}{\Delta \vdash \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha]}\ (\texttt{rec\_elim})$$

$$\frac{\Delta \vdash \tau \leq \tau_1 \quad \Delta \vdash \tau \leq \tau_2}{\Delta \vdash \tau \leq \tau_1 \wedge \tau_2}\ (\texttt{meet\_intro}) \qquad \frac{\Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 \leq \tau_1}\ (\texttt{meet\_elim1})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 \leq \tau_2}\ (\texttt{meet\_elim2}) \quad \frac{\Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \leq \tau_1 \vee \tau_2}\ (\texttt{join\_intro1}) \quad \frac{\Delta \vdash \tau_1 : \mathsf{T}}{\Delta \vdash \tau_2 \leq \tau_1 \vee \tau_2}\ (\texttt{join\_intro2})$$

$$\frac{}{\Delta \vdash \tau \wedge (\tau_1 \vee \tau_2) \leq (\tau \wedge \tau_1) \vee (\tau \wedge \tau_2)}\ (\texttt{meet\_dist\_join})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}i}{\Delta \vdash (\tau_1 \times \tau_2) \wedge (\tau_1' \times \tau_2') \leq (\tau_1 \wedge \tau_1') \times (\tau_2 \wedge \tau_2')}\ (\texttt{meet\_dist\_prod})$$

$$\frac{}{\Delta \vdash \tau \times (\tau_1 \vee \tau_2) \leq (\tau \times \tau_1) \vee (\tau \times \tau_2)}\ (\texttt{prod\_dist\_join1})$$

$$\frac{}{\Delta \vdash (\tau_1 \vee \tau_2) \times \tau \leq (\tau_1 \times \tau) \vee (\tau_2 \times \tau)}\ (\texttt{prod\_dist\_join2})$$

$$\frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \tau \times \mathtt{void} \leq \mathtt{void}}\ (\texttt{prod\_dist\_void1}) \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathtt{void} \times \tau \leq \mathtt{void}}\ (\texttt{prod\_dist\_void2})$$

$$\frac{}{\Delta \vdash \tau_1 \times (\tau_2 \times \tau_3) \leq (\tau_1 \times \tau_2) \times \tau_3}\ (\texttt{lassoc}) \qquad \frac{}{\Delta \vdash (\tau_1 \times \tau_2) \times \tau_3 \leq \tau_1 \times (\tau_2 \times \tau_3)}\ (\texttt{rassoc})$$

$$\frac{}{\Delta \vdash \tau \leq \mathtt{B0} \times \tau}\ (\texttt{luniti}) \quad \frac{}{\Delta \vdash \mathtt{B0} \times \tau \leq \tau}\ (\texttt{lunite}) \quad \frac{}{\Delta \vdash \tau \leq \tau \times \mathtt{B0}}\ (\texttt{runiti})$$

$$\frac{}{\Delta \vdash \tau \times \mathtt{B0} \leq \tau}\ (\texttt{runite}) \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \tau \uparrow B \leq \tau^B}\ (\texttt{explode}) \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \tau^B \leq \tau \uparrow B}\ (\texttt{implode})$$

$$\frac{}{\Delta \vdash \mathtt{set}_{=}(B) \leq \mathtt{BW}}\ (\texttt{seteq\_forget}) \qquad \frac{}{\Delta \vdash \mathtt{set}_{<}(B) \leq \mathtt{BW}}\ (\texttt{setlt\_forget})$$

$$\frac{}{\Delta \vdash \mathtt{set}_{>}(B) \leq \mathtt{BW}}\ (\texttt{setgt\_forget}) \qquad \frac{}{\Delta \vdash \mathtt{set}_{=}(B) \wedge \mathtt{set}_{<}(B) \leq \mathtt{void}}\ (\texttt{raa\_lt})$$

$$\frac{}{\Delta \vdash \mathtt{set}_{=}(B) \wedge \mathtt{set}_{>}(B) \leq \mathtt{void}}\ (\texttt{raa\_gt}) \qquad \frac{}{\Delta \vdash \mathtt{set}_{<}(B) \wedge \mathtt{set}_{>}(B) \leq \mathtt{void}}\ (\texttt{raa\_ltgt})$$

$$\frac{}{\Delta \vdash \mathtt{BW} \leq \exists\alpha{:}\mathsf{N}.\mathtt{set}_{=}(\alpha)}\ (\texttt{focus}) \qquad \frac{(B_1 \leq B_2)}{\Delta \vdash \mathtt{set}_{<}(B_1) \leq \mathtt{set}_{<}(B_2)}\ (\texttt{subrange\_lt})$$

$$\frac{(B_1 \geq B_2)}{\Delta \vdash \mathtt{set}_{>}(B_1) \leq \mathtt{set}_{>}(B_2)}\ (\texttt{subrange\_gt}) \qquad \frac{(B_1 < B_2)}{\Delta \vdash \mathtt{set}_{=}(B_1) \leq \mathtt{set}_{<}(B_2)}\ (\texttt{subrange\_eqlt})$$

$$\frac{(B_1 > B_2)}{\Delta \vdash \mathtt{set}_{=}(B_1) \leq \mathtt{set}_{>}(B_2)}\ (\texttt{subrange\_eqgt})$$

$$\frac{\Delta \vdash \tau \leq \tau' \quad \Delta \vdash \tau_r \leq \tau_r' \textit{ for } r \in \{\mathtt{ax}, \ldots, \mathtt{bp}\}}{\Delta \vdash \{\mathtt{eax}{:}\tau_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{bp}}, \mathtt{esp}{:}\tau_{\mathtt{sp}}\} \leq \{\mathtt{eax}{:}\tau'_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau'_{\mathtt{bp}}, \mathtt{esp}{:}\tau'_{\mathtt{sp}}\}}\ (\text{SUBRTYPE})$$

A.1.4 $\boxed{\Delta;\Psi;\Gamma \vdash o : \tau}$ Operand Typing

$$\frac{}{\Delta;\Psi;\Gamma \vdash B : \mathtt{set}_{=}(B)}\ (\text{OOF\_SETEQ}) \qquad \frac{}{\Delta;\Psi;\Gamma \vdash \ell : \Psi(\ell)}\ (\text{OOF\_POINTER})$$

$$\frac{}{\Delta;\Psi;\Gamma \vdash \mathtt{esp} : \mathtt{sptr}(\Gamma(\mathtt{esp}))}\ (\mathtt{oof\_spco}) \qquad \frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \mathtt{sptr}(\tau_1 \times \tau_2 \times \tau_3) \\ \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \\ \Delta \vdash \Gamma(\mathtt{esp}) \leq \tau \times \tau_1 \times \tau_2 \times \tau_3\end{array}}{\Delta;\Psi;\Gamma \vdash m`[o+n] : \tau_2}\ (\mathtt{oof\_zco})$$

$$\frac{\begin{array}{l}\Delta;\Psi;\Gamma \vdash o_1 : \mathtt{box}((\tau_1 \times \tau_2 \times \tau_3)\uparrow x) \\ \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{set}_{<}(x) \\ \Delta \quad\ \vdash \tau_1 : \mathsf{T}n \\ \Delta \quad\ \vdash \tau_2 : \mathsf{T}m \\ \Delta \quad\ \vdash \tau_1 \times \tau_2 \times \tau_2 : \mathsf{T}k\end{array}}{\Delta;\Psi;\Gamma \vdash m`[o_1 + n + k \cdot o_2] : \tau_2}\ (\mathtt{oof\_imco}) \qquad \frac{\begin{array}{c}\Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \\ \Delta;\Psi;\Gamma \vdash o : \mathtt{box}(\tau_1 \times \tau_2 \times \tau_3)\end{array}}{\Delta;\Psi;\Gamma \vdash m`[o+n] : \tau_2}\ (\mathtt{oof\_mco})$$

$$\frac{}{\Delta;\Psi;\Gamma \vdash r : \Gamma(r)}\ (\mathtt{oof\_rco}) \qquad \frac{\Delta;\Psi;\Gamma \vdash o : \mathtt{BW} \quad \Delta \vdash x : \mathsf{N}}{\Delta;\Psi;\Gamma \vdash o : \mathtt{set}_{<}(x) \vee \mathtt{set}_{=}(x) \vee \mathtt{set}_{>}(x)}\ (\text{OOF\_TRICHOTOMY})$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta;\Psi;\Gamma \vdash o : \tau}\ (\mathtt{oof\_subsume})$$

A.1.5 $\boxed{\Delta;\Psi;\Gamma \vdash d : \tau \to \Gamma'}$ Destination Typing

$$\frac{\Delta \vdash \tau : \mathsf{T}W}{\Delta;\Psi;\Gamma \vdash r : \tau \to \Gamma\{r{:}\tau\}}\ (\mathtt{update\_rdest}) \qquad \frac{\Delta \vdash \tau \leq \mathtt{sptr}(\tau_2) \quad \Delta \vdash \Gamma(\mathtt{esp}) \leq \tau_1 \times \tau_2}{\Delta;\Psi;\Gamma \vdash \mathtt{esp} : \tau \to \Gamma\{\mathtt{esp}{:}\tau_2\}}\ (\mathtt{update\_spdest})$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \mathtt{mbox}(\tau_1 \times \tau_2 \times \tau_3) \quad \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m}{\Delta;\Psi;\Gamma \vdash m`[o+n] : \tau_2 \to \Gamma}\ (\mathtt{update\_mdest})$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma(r) \leq \mathtt{sptr}(\tau_1 \times \tau_2 \times \tau_3) \\ \Delta \vdash \Gamma(\mathtt{esp}) \leq \tau \times \tau_1 \times \tau_2 \times \tau_3 \\ \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \quad \Delta \vdash \tau_2' : \mathsf{T}m\end{array}}{\Delta;\Psi;\Gamma \vdash m`[r+n] : \tau_2' \to \Gamma\{\mathtt{esp}{:}\tau \times \tau_1 \times \tau_2' \times \tau_3, r{:}\mathtt{sptr}(\tau_1 \times \tau_2' \times \tau_3)\}}\ (\mathtt{update\_zdest})$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \quad \Delta \vdash \tau_1 \times \tau_2 \times \tau_2 : \mathsf{T}k \\ \Delta;\Psi;\Gamma \vdash o_1 : \mathtt{mbox}((\tau_1 \times \tau_2 \times \tau_3)\uparrow x) \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{set}_{<}(x)\end{array}}{\Delta;\Psi;\Gamma \vdash m`[o_1 + n + k \cdot o_2] : \tau_2 \to \Gamma}\ (\mathtt{update\_imdest})$$

A.1.6 $\boxed{\Delta;\Psi;\Gamma \vdash I}$ Instruction Typing

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o_1 : \mathtt{B4} \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{B4} \\ \Delta;\Psi;\Gamma \vdash d : \mathtt{B4} \to \Gamma' \quad \Delta;\Psi;\Gamma' \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{add}\ d, o_1, o_2\ I}\ (\mathtt{ok\_add})$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \mathtt{sptr}(\tau_1 \times \tau_2) \quad \Delta \vdash \tau_1 : \mathsf{T}i \\ \Delta \vdash \tau_2 : \mathsf{TD} \quad \Delta;\Psi;\Gamma \vdash d : \mathtt{sptr}(\tau_2) \to \Gamma' \quad \Delta;\Psi;\Gamma \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{addsptr}\ d, o, i\ I}\ (\mathtt{ok\_addsptr})$$

$$\frac{(\Gamma(\mathtt{esp}) = \tau_s) \quad \Delta;\Psi;\Gamma \vdash \ell : \Gamma_r \to 0 \quad \Delta;\Psi;\Gamma \vdash o : \Gamma\{\mathtt{esp} : (\Gamma_r \to 0) \times \tau_s\} \to 0}{\Delta;\Psi;\Gamma \vdash \mathtt{call}\, o\, \ell}\ (\mathtt{ok\_call}) \qquad \frac{\Delta;\Psi;\Gamma \vdash I \quad \Delta;\Psi;\Gamma \vdash o_1 : \mathtt{BW} \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{BW}}{\Delta;\Psi;\Gamma \vdash \mathtt{cmp}\, o_1, o_2\, I}\ (\mathtt{ok\_cmp})$$

$$\frac{\begin{array}{ll} & (\Gamma(r) = \tau_1 \vee \tau_2) \\ \Delta;\Psi;\Gamma & \vdash o_1 : \mathtt{BW} \\ \Delta;\Psi;\Gamma & \vdash o_2 : \mathtt{set}_=(x) \\ \Delta & \vdash \tau_1 \vee \tau_2 : \mathsf{T}W \\ \Delta;\Psi;\Gamma\{r{:}\tau_1\} & \vdash o_1 : \tau_1' \\ \Delta;\Psi;\Gamma\{r{:}\tau_2\} & \vdash o_1 : \tau_2' \\ \Delta & \vdash \tau_1' \wedge \tau_{unsat}^{\kappa,x} \leq \mathtt{void} \\ \Delta & \vdash \tau_2' \wedge \tau_{sat}^{\kappa,x} \leq \mathtt{void} \\ \Delta;\Psi;\Gamma & \vdash o_3 : \Gamma\{r{:}\tau_1, \mathtt{ck}{:}t\} \to 0 \\ \Delta;\Psi;\Gamma\{r{:}\tau_2, \mathtt{ck}{:}t\} & \vdash I \end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{cmpjcc}\, o_1, o_2, \kappa, o_3\, I}\ (\mathtt{ok\_cmpjcc}) \qquad \frac{\Delta \vdash \Gamma(\mathtt{esp}) \leq \mathtt{B0}}{\Delta;\Psi;\Gamma \vdash \mathtt{halt}}\ (\mathtt{ok\_halt})$$

$$\frac{\Delta;\Psi;\Gamma \vdash I \quad \Delta;\Psi;\Gamma \vdash o : \Gamma \to 0}{\Delta;\Psi;\Gamma \vdash \mathtt{jcc}\, \kappa, o; I}\ (\mathtt{ok\_jcc}) \qquad \frac{\Delta;\Psi;\Gamma \vdash o : \Gamma \to 0}{\Delta;\Psi;\Gamma \vdash \mathtt{jmp}\, o\, I}\ (\mathtt{ok\_jmp})$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \tau \quad \Delta;\Psi;\Gamma \vdash d : \tau \to \Gamma' \quad \Delta;\Psi;\Gamma' \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{mov}\, d, o}\ (\mathtt{ok\_mov}) \qquad \frac{\Delta;\Psi;\Gamma\{r{:}\mathtt{nsw}\} \vdash I \text{ inits } r{:}\mathtt{mbox}(\mathtt{ns}^n)}{\Delta;\Psi;\Gamma \vdash \mathtt{malloc}\, n, r\, I}\ (\mathtt{ok\_malloc})$$

$$\frac{\Delta;\Psi;\Gamma \vdash o_2 : \mathtt{set}_=(x) \quad \Delta;\Psi;\Gamma\{r : \mathtt{nsw}\} \vdash o_3 : \tau \quad \Delta \vdash \tau : \mathsf{T}n \quad \Delta;\Psi;\Gamma\{r{:}\mathtt{mbox}(\tau \uparrow x)\} \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{mallocarr}\, o_1, r, n, o_2, o_3\, I}\ (\mathtt{ok\_mallocarr})$$

$$\frac{\Delta \vdash \Gamma(\mathtt{esp}) \leq \tau_1 \times \tau_2 \quad \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{TD} \quad \Delta;\Psi;\Gamma\{\mathtt{esp}{:}\tau_2\} \vdash d : \tau_1 \to \Gamma' \quad \Delta;\Psi;\Gamma' \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{pop}\, n, d\, I}\ (\mathtt{ok\_pop})$$

$$\frac{(\Gamma(\mathtt{esp}) = \tau_s) \quad \Delta;\Psi;\Gamma \vdash o : \tau \quad \Delta;\Psi;\Gamma\{\mathtt{esp}{:}\tau \times \tau_s\} \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{push}\, o\, I}\ (\mathtt{ok\_push}) \qquad \frac{\Delta \vdash \tau : \mathsf{TD} \quad \Delta \vdash \Gamma(\mathtt{sp}) \leq (\Gamma\{sp{:}\tau\} \to 0) \times \tau}{\Delta;\Psi;\Gamma \vdash \mathtt{ret}}\ (\mathtt{ok\_ret})$$

$$\frac{\Delta;\Psi;\Gamma\{\mathtt{esp} : \mathtt{ns}n \times \Gamma(\mathtt{esp})\} \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{salloc}\, n\, I}\ (\mathtt{ok\_salloc}) \qquad \frac{\Delta \vdash \Gamma(\mathtt{esp}) \leq \tau_1 \times \tau_2 \quad \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{TD} \quad \Delta;\Psi;\Gamma\{\mathtt{esp}{:}\tau_2\} \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{sfree}\, n\, I}\ (\mathtt{ok\_sfree})$$

$$\frac{\Delta;\Psi;\Gamma \vdash o_1 : \mathtt{BW} \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{BW} \quad \Delta;\Psi;\Gamma \vdash d : \mathtt{BW} \to \Gamma' \quad \Delta;\Psi;\Gamma' \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{sub}\, d, o_1, o_2\, I}\ (\mathtt{ok\_sub}) \qquad \frac{\Delta;\Psi;\Gamma' \vdash I \quad \Delta \vdash \Gamma \leq \Gamma'}{\Delta;\Psi;\Gamma \vdash I}\ (\mathtt{ok\_coerce})$$

A.1.7 $\boxed{\Psi; \Delta \vdash I : \tau \text{ block}, \Delta \vdash P}$ Block and Program Typing

$$\frac{\Psi; (\Delta, \alpha{:}K) \vdash I : \tau \text{ block}}{\Psi; \Delta \vdash I : \forall \alpha{:}K.\tau \text{ block}} \text{ (BLOCKOK\_FORALL)} \qquad \frac{\Psi; \Delta; \Gamma \vdash I}{\Psi; \Delta \vdash I : \Gamma \to 0 \text{ block}} \text{ (BLOCKOK\_ARROW)}$$

$$\frac{\begin{array}{c}(\text{dom}(\Psi) = \{\ell_1, \ldots, \ell_n\}) \\ \vdash \Psi \quad (\Psi(\ell_1) = \{\texttt{esp:B0}\} \to 0) \\ \Psi; \cdot \vdash I_i : \Psi(\ell_i) \text{ block for } 1 \le i \le n\end{array}}{\vdash \ell_1 = I_1, \ldots, \ell_n = I_n} \text{ (PROGOK)}$$

A.2 Operational Semantics

A.2.1 $\boxed{H, V_s, R \vdash o \rightsquigarrow v}$ Operand Resolution

$$\frac{}{H, V_s, R \vdash v \rightsquigarrow v} \text{ (resolve\_im)} \qquad \frac{}{H, V_s, R \vdash r \rightsquigarrow R(r)} \text{ (resolve\_rco)}$$

$$\frac{}{H, V_s, R \vdash \texttt{esp} \rightsquigarrow \texttt{sptr}(|V_s|)} \text{ (resolve\_spco)} \qquad \frac{\begin{array}{c}H, V_s, R \vdash o \rightsquigarrow \ell \quad (|V_1| = n) \\ (H(\ell) = V_1@V_2@V_3) \quad (|V_2| = m)\end{array}}{H, V_s, R \vdash m`[o + n] \rightsquigarrow V_2} \text{ (resolve\_mco)}$$

$$\frac{H, V_s, R \vdash o \rightsquigarrow \texttt{sptr}(s) \quad (V_s = V'@V,\ |V| = s,\ V = V_1@V_2@V_3,\ |V_1| = n,\ |V_2| = m)}{H, V_s, R \vdash m`[o + n] \rightsquigarrow V_2} \text{ (resolve\_zco)}$$

$$\frac{\begin{array}{c}H, V_s, R \vdash o_1 \rightsquigarrow \ell \quad H, V_s, R \vdash o_2 \rightsquigarrow B \\ (H(\ell) = V_1@V_2@V_3,\ |V_1| = n + n'B,\ |V_2| = m)\end{array}}{H, V_s, R \vdash m`[o_1 + n + n' \cdot o_2] \rightsquigarrow V_2} \text{ (resolve\_imco)}$$

A.2.2 $\boxed{H, V_s, R \vdash d(v) \rightsquigarrow H', V_s', R'}$ Destination Propagation

$$\frac{}{H, V_s, R \vdash r(v) \rightsquigarrow H, V_s, R\{r \mapsto v\}} \text{ (propagate\_rdest)}$$

$$\frac{(V_s = V_0@V_s', |V_s'| = n)}{H, V_s, R \vdash \texttt{esp}(\texttt{sptr}(n)) \rightsquigarrow H, V_s', R} \text{ (propagate\_spdest)}$$

$$\frac{H, V_s, R \vdash o \rightsquigarrow \ell \quad (H(\ell) = V_1@V_2@V_3,\ |V_1| = n,\ |V_2| = W)}{H, V_s, R \vdash W`[o + n](v) \rightsquigarrow H\{\ell \mapsto V_1@v@V_3\}, V_s, R} \text{ (propagate\_mdest)}$$

$$\frac{\begin{array}{c}H, V_s, R \vdash o \rightsquigarrow \texttt{sptr}(s) \\ (V_s = V_0@V_1@V_2@V_3,\ |V_1@V_2@V_3| = s,\ |V_1| = n,\ |V_2| = W)\end{array}}{H, V_s, R \vdash W`[o + n](v) \rightsquigarrow H, V_0@V_1@v@V_3, R} \text{ (propagate\_zdest)}$$

$$\frac{\begin{array}{c}H, V_s, R \vdash o_1 \rightsquigarrow \ell \quad H, V_s, R \vdash o_2 \rightsquigarrow B \\ (H(\ell) = V_1@V_2@V_3,\ |V_1| = m + m'B,\ |V_2| = W)\end{array}}{H, V_s, R \vdash W`[o_1 + m + m' \cdot o2](v) \rightsquigarrow H\{\ell \mapsto V_1@v@V_3\}, V_s, R} \text{ (propagate\_imdest)}$$

Appendix B

Complete MiniTALT-R Typing Rules

To save space, this appendix focuses on the differences between MiniTALT and MiniTALT-R. Unless otherwise noted, all of the rules listed in Appendix A are retained in MiniTALT-R. Any rule in this appendix with the same label as a rule in Appendix A supercedes the MiniTALT rule. MiniTALT-R rules for which no MiniTALT analogues exist are given *ad hoc* names and labeled in SMALL CAPS, even if there *is* an analogue in the LF implementation of TALT.

B.1 $\boxed{\Delta \vdash c : K}$ Static Term Formation

$$\frac{(n \geq 0)}{\Delta \vdash \overline{n} : \mathsf{N}}\ (\texttt{kof\_numlit}) \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 + t_2 : \mathsf{N}}\ (\text{KOF\_NUMADD})$$

$$\frac{(K \in \{\mathsf{T}, \mathsf{T}i, \mathsf{TD}\}) \quad \Delta \vdash \varphi : \mathsf{P} \quad \Delta \vdash \tau : K}{\Delta \vdash \varphi \Rightarrow \tau : K}\ (\text{KOF\_GUARD})$$

B.2 $\boxed{\Delta \vdash \varphi\ \mathsf{prop}}$ Constraint Formula Formation

$$\frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 \leq t_2\ \mathsf{prop}}\ (\text{PROPOK\_LEQ}) \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 = t_2 :\ \mathsf{prop}}\ (\text{PROPOK\_EQ})$$

B.3 $\boxed{c_1 \equiv c_2, \varphi_1 \equiv \varphi_2}$ Static Term and Formula Equivalence

$$\frac{\varphi \equiv \varphi' \quad \tau \equiv \tau'}{\varphi \Rightarrow \tau \equiv \varphi' \Rightarrow \tau'}\ (\text{EQUIV\_GUARD}) \qquad \frac{t_1 \equiv t_1' \quad t_2 \equiv t_2'}{t_1 + t_2 \equiv t_1' + t_2'}\ (\text{EQUIV\_NUMADD})$$

$$\frac{t_1 \equiv t_1' \quad t_2 \equiv t_2'}{(t_1 \leq t_2) \equiv (t_1' \leq t_2')}\ (\text{FEQUIV\_LEQ}) \qquad \frac{t_1 \equiv t_1' \quad t_2 \equiv t_2'}{(t_1 = t_2) \equiv (t_1' = t_2')}\ (\text{FEQUIV\_EQ})$$

$$\frac{t \equiv t' \quad \tau \equiv \tau' \quad \tau_r \equiv \tau_r' \textit{ for } r \in \{\mathtt{ax}, \ldots, \mathtt{bp}\}}{\{\mathtt{eax}{:}\tau_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{bp}}, \mathtt{esp}{:}\tau_{\mathtt{sp}}, \mathtt{ck}{:}t\} \equiv \{\mathtt{eax}{:}\tau'_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau'_{\mathtt{bp}}, \mathtt{esp}{:}\tau'_{\mathtt{sp}}, \mathtt{ck}{:}t'\}}\ (\texttt{equivr\_})$$

B.4 $\boxed{\Delta \vdash \varphi \text{ true}}$ Constraint Truth

$$\frac{((\varphi \text{ true}) \in \Delta)}{\Delta \vdash \varphi \text{ true}} \text{ (TR\_HYP)} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash t = t \text{ true}} \text{ (TR\_EQ\_REFL)} \qquad \frac{\Delta \vdash t_2 = t_1 \text{ true}}{\Delta \vdash t_1 = t_2 \text{ true}} \text{ (TR\_EQ\_SYMM)}$$

$$\frac{\Delta \vdash t_1 = t_3 \text{ true} \quad \Delta \vdash t_3 = t_2 \text{ true}}{\Delta \vdash t_1 = t_2 \text{ true}} \text{ (TR\_EQ\_TRANS)} \qquad \frac{\Delta \vdash t_1 = t_1' \text{ true} \quad \Delta \vdash t_2 = t_2' \text{ true}}{\Delta \vdash t_1 + t_2 = t_1' + t_2' \text{ true}} \text{ (TR\_ADD\_COMPAT)}$$

$$\frac{}{\Delta \vdash \overline{m} + \overline{n} = \overline{m+n} \text{ true}} \text{ (TR\_ADD\_LIT)} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \overline{0} + t = t \text{ true}} \text{ (TR\_ADD\_IDENT)}$$

$$\frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 + t_2 = t_2 + t_1 \text{ true}} \text{ (TR\_ADD\_COMMUTE)} \qquad \frac{\Delta \vdash t_i : \mathsf{N} \text{ (for } i = 1, 2, 3)}{\Delta \vdash (t_1 + t_2) + t_3 = t_1 + (t_2 + t_3) \text{ true}} \text{ (TR\_ADD\_ASSOC)}$$

$$\frac{\Delta \vdash t_1 = t_2 \text{ true}}{\Delta \vdash t_1 \leq t_2 \text{ true}} \text{ (TR\_LEQ\_REFL)} \qquad \frac{\Delta \vdash t_1 \leq t_3 \text{ true} \quad \Delta \vdash t_3 \leq t_2 \text{ true}}{\Delta \vdash t_1 \leq t_2 \text{ true}} \text{ (TR\_LEQ\_TRANS)}$$

$$\frac{(m \leq n)}{\Delta \vdash \overline{m} \leq \overline{n} \text{ true}} \text{ (TR\_LEQ\_LIT)} \qquad \frac{\Delta \vdash t_1 \leq t_1' \text{ true} \quad \Delta \vdash t_2 \leq t_2' \text{ true}}{\Delta \vdash t_1 + t_2 \leq t_1' + t_2' \text{ true}} \text{ (TR\_ADD\_MONO)}$$

$$\frac{\Delta \vdash t + t_1 \leq t + t_2 \text{ true}}{\Delta \vdash t_1 \leq t_2 \text{ true}} \text{ (TR\_ADD\_INJ)} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \overline{0} \leq t \text{ true}} \text{ (TR\_LEQ\_Z)}$$

$$\frac{\Delta \vdash t_1 \leq t_2 \text{ true} \quad \Delta \vdash t_2 \leq t_1 \text{ true}}{\Delta \vdash t_1 = t_2 \text{ true}} \text{ (TR\_LEQ\_ANTISYMM)}$$

B.4.1 Rule For Rational Extension

$$\frac{\Delta \vdash^+ \overbrace{t + \cdots + t}^{n} \leq \overbrace{u + \cdots + u}^{n} \text{ true} \quad (n \in \{1, 2, \ldots\})}{\Delta \vdash^+ t \leq u \text{ true}} \text{ (TR\_ADD\_REPEAT)}$$

B.5 $\boxed{\Delta \vdash \tau_1 \leq \tau_2, \Delta \vdash \Gamma \leq \Gamma'}$ Subtyping

$$\frac{\Delta \vdash \varphi \text{ prop}}{\Delta \vdash \tau \leq (\varphi \Rightarrow \tau)} \text{ (GEN\_GUARD)} \qquad \frac{\Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash \varphi \text{ true}}{\Delta \vdash (\varphi \Rightarrow \tau) \leq \tau} \text{ (GUARD\_ELIM)}$$

$$\frac{\Delta, \varphi \text{ true} \vdash \tau \leq \tau'}{\Delta \vdash (\varphi \Rightarrow \tau) \leq (\varphi \Rightarrow \tau')} \text{ (GUARD\_SUB)}$$

$$\frac{\Delta \vdash t_1 = t_2 \text{ true}}{\Delta \vdash \mathcal{S}(t_1) \leq \mathcal{S}(t_2)} \text{ (SETEQ\_SUB)} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \mathcal{S}(t) \leq \mathsf{B}W} \text{ (\texttt{seteq\_forget})}$$

$$\frac{\Delta \vdash t' \leq t \text{ true} \quad \Delta \vdash \tau \leq \tau' \quad \Delta \vdash \tau_r \leq \tau_r' \textit{ for } r \in \{\mathtt{ax}, \ldots, \mathtt{bp}\}}{\Delta \vdash \{\mathsf{eax}{:}\tau_{\mathtt{ax}}, \ldots, \mathsf{ebp}{:}\tau_{\mathtt{bp}}, \mathsf{esp}{:}\tau_{\mathtt{sp}}, \mathsf{ck}{:}t\} \leq \{\mathsf{eax}{:}\tau_{\mathtt{ax}}', \ldots, \mathsf{ebp}{:}\tau_{\mathtt{bp}}', \mathsf{esp}{:}\tau_{\mathtt{sp}}', \mathsf{ck}{:}t'\}} \text{ (SUBRTYPE)}$$

B.5.1 Unofficial Rules

$$\frac{\Delta \vdash t \leq u \text{ true}}{\Delta \vdash \mathcal{S}(u) \leq \exists a : \mathsf{N}.\mathcal{S}(t + a)} \text{ (FOCUS-LEQ)}$$

B.6 $\boxed{\Delta;\Psi;\Gamma \vdash o : \tau}$ Operand Typing

$$\frac{(\Delta, \varphi\ \mathsf{true});\Psi \vdash v : \tau}{\Delta;\Psi \vdash v : \varphi \Rightarrow \tau}\ (\text{OOF\_GUARD\_INTRO})$$

B.7 $\boxed{\Delta;\Psi;\Gamma \vdash I}$ Instruction Typing

$$\frac{\begin{array}{c}(\Gamma(\mathsf{ck}) = \overline{1} + t)\\ \Delta;\Psi;\Gamma \vdash o_1 : \mathsf{B4} \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathsf{B4}\\ \Delta;\Psi;\Gamma \vdash d : \mathsf{B4} \to \Gamma' \quad \Delta;\Psi;\Gamma'\{\mathsf{ck}{:}t\} \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{add}\ d, o_1, o_2; I}\ (\mathtt{ok\_add})$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \mathtt{sptr}(\tau_1 \times \tau_2) \quad \Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{TD}\\ \Delta;\Psi;\Gamma \vdash d : \mathtt{sptr}(\tau_2) \to \Gamma' \quad \Delta;\Psi;\Gamma'\{\mathsf{ck}{:}t\} \vdash I \quad (\Gamma(\mathsf{ck}) = 1 + t)\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{addsptr}\ d, o, i\ I}\ (\mathtt{ok\_addsptr})$$

$$\frac{\begin{array}{c}\Delta';\Psi;\Gamma_{ret} \vdash I \quad \Delta' \vdash \Gamma_{ret} \quad \Delta;\Psi;\Gamma' \vdash o : \Gamma' \to 0\\ (\Gamma(\mathsf{ck}) = 1 + t) \quad (\Delta' = \Delta, \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n)\\ (\Gamma' = \Gamma\{\mathsf{sp}{:}(\forall\alpha_1{:}K_1 \ldots \forall\alpha_n{:}K_n.\Gamma_{ret} \to 0) \times \Gamma(\mathsf{sp}), \mathsf{ck}{:}t\})\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{call}\ o; I}\ (\mathtt{ok\_call!!!!!})$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma\{\mathsf{ck}{:}t\} \vdash I \quad (\Gamma(\mathsf{ck}) = 1 + t)\\ \Delta;\Psi;\Gamma \vdash o_1 : \mathtt{int} \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{int}\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{cmp}\ o_1, o_2\ I}\ (\mathtt{ok\_cmp})$$

$$\frac{\begin{array}{c}(\Gamma(\mathsf{ck}) = 2 + t) \quad (\Gamma(r) = \tau_1 \vee \tau_2)\\ \begin{array}{ll}\Delta;\Psi;\Gamma & \vdash o_1 : \mathtt{int}\\ \Delta;\Psi;\Gamma & \vdash o_2 : \mathtt{set}_{=}(x)\\ \Delta & \vdash \tau_1 \vee \tau_2 : \mathsf{T}W\\ \Delta;\Psi;\Gamma\{r{:}\tau_1\} & \vdash o_1 : \tau_1'\\ \Delta;\Psi;\Gamma\{r{:}\tau_2\} & \vdash o_1 : \tau_2'\\ \Delta & \vdash \tau_1' \wedge \tau_{unsat}^{\kappa,x} \leq \mathtt{void}\\ \Delta & \vdash \tau_2' \wedge \tau_{sat}^{\kappa,x} \leq \mathtt{void}\\ \Delta;\Psi;\Gamma & \vdash o_3 : \Gamma\{r{:}\tau_1, \mathsf{ck}{:}t\} \to 0\\ \Delta;\Psi;\Gamma\{r{:}\tau_2, \mathsf{ck}{:}t\} & \vdash I\end{array}\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{cmpjcc}\ o_1, o_2, \kappa, o_3\ I}\ (\mathtt{ok\_cmpjcc})$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : (\Gamma\{\mathsf{ck}{:}t\}) \to 0 \quad \Delta;\Psi;\Gamma\{\mathsf{ck}{:}t\} \vdash I \quad (\Gamma(\mathsf{ck}) = 1 + t)}{\Delta;\Psi;\Gamma \vdash \mathtt{jcc}\ \kappa, o; I}\ (\mathtt{ok\_jcc}) \qquad \frac{(\Gamma(\mathsf{ck}) = 1 + t) \quad \Delta;\Psi;\Gamma \vdash o : (\Gamma\{\mathsf{ck}{:}t\}) \to 0}{\Delta;\Psi;\Gamma \vdash \mathtt{jmp}\ o; I}\ (\mathtt{ok\_jmp})$$

$$\frac{\begin{array}{c}(\Gamma(\mathsf{ck}) = 1 + t) \quad \Delta;\Psi;\Gamma \vdash o : \tau\\ \Delta;\Psi;\Gamma \vdash d : \tau \to \Gamma' \quad \Delta;\Psi;\Gamma'\{\mathsf{ck}{:}t\} \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{mov}\ d, o\ I}\ (\mathtt{ok\_mov})$$

$$\frac{(\Gamma(\mathsf{ck}) = 1 + t)}{\Delta;\Psi;\Gamma\{r{:}\mathtt{nsw}, \mathsf{ck}{:}t\} \vdash I \text{ inits } r{:}\mathtt{mbox}(\mathtt{ns}^n)}\ (\mathtt{ok\_malloc})$$

$$\frac{\begin{array}{c}(\Gamma(\mathsf{ck}) = 1 + t) \\ \Delta;\Psi;\Gamma \vdash o_2 : \mathsf{set}_{=}(x) \quad \Delta;\Psi;\Gamma\{r : \mathsf{nsw}\} \vdash o_3 : \tau \\ \Delta \vdash \tau : \mathsf{T}n \quad \Delta;\Psi;\Gamma\{r{:}\mathsf{mbox}(\tau \uparrow x), \mathsf{ck}{:}t\} \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{mallocarr}\ o_1, r, n, o_2, o_3\ I}\ (\mathtt{ok\_mallocarr})$$

$$\frac{\begin{array}{c}(\Gamma(\mathsf{ck}) = 1 + t) \\ \Delta \vdash \Gamma(\mathsf{esp}) \leq \tau_1 \times \tau_2 \quad \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{TD} \\ \Delta;\Psi;\Gamma\{\mathsf{esp}{:}\tau_2\} \vdash d : \tau_1 \to \Gamma' \quad \Delta;\Psi;\Gamma'\{\mathsf{ck}{:}t\} \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{pop}\ n, d\ I}\ (\mathtt{ok\_pop})$$

$$\frac{(\Gamma(\mathsf{ck}) = 1 + t) \quad \Delta;\Psi;\Gamma \vdash o : \tau \quad \Delta \vdash \tau : \mathsf{TD} \quad \Delta;\Psi;\Gamma\{\mathsf{esp}{:}\tau \times \Gamma(\mathsf{esp}), \mathsf{ck}{:}t \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{push}\ o\ I}\ (\mathtt{ok\_push})$$

$$\frac{\Delta \vdash \tau : \mathsf{TD} \quad (\Gamma(\mathsf{ck}) = 1 + t) \quad \Delta \vdash \Gamma(\mathsf{esp}) \leq (\Gamma\{\mathsf{esp}{:}\tau, \mathsf{ck}{:}t\} \to 0) \times \tau}{\Delta;\Psi;\Gamma \vdash \mathtt{ret}}\ (\mathtt{ok\_ret})$$

$$\frac{(\Gamma(\mathsf{ck}) = 1 + t) \quad \Delta;\Psi;\Gamma\{\mathsf{esp}{:}\mathsf{ns}n \times \Gamma(\mathsf{esp}), \mathsf{ck}{:}t\} \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{salloc}\ n\ I}\ (\mathtt{ok\_salloc}) \qquad \frac{\begin{array}{c}(\Gamma(\mathsf{ck}) = 1 + t) \\ \Delta \vdash \Gamma(\mathsf{esp}) \leq \tau_1 \times \tau_2 \quad \Delta \vdash \tau_1 : \mathsf{T}n \\ \Delta \vdash \tau_2 : \mathsf{TD} \quad \Delta;\Psi;\Gamma\{\mathsf{esp}{:}\tau_2, \mathsf{ck}{:}t\} \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{sfree}\ n\ I}\ (\mathtt{ok\_sfree})$$

$$\frac{\begin{array}{c}(\Gamma(\mathsf{ck}) = 1 + t) \\ \Delta;\Psi;\Gamma \vdash o_1 : \mathtt{int} \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{int} \\ \Delta;\Psi;\Gamma \vdash d : \mathtt{int} \to \Gamma' \quad \Delta;\Psi;\Gamma'\{\mathsf{ck}{:}t\} \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{sub}\ d, o_1, o_2\ I}\ (\mathtt{ok\_sub}) \qquad \frac{\Delta;\Psi;\Gamma\{\mathsf{ck}{:}\overline{Y}\} \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{yield}; I}\ (\text{OK\_YIELD})$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma\{\mathsf{r}_d{:}\mathsf{B}W, \mathsf{ck}{:}t\} \vdash I \\ \Delta;\Psi;\Gamma \vdash o_3 : \forall a{:}\mathsf{N}.(u = v + a) \Rightarrow \Gamma\{\mathsf{r}_d{:}\mathcal{S}(a), \mathsf{ck}{:}t\} \to 0 \\ \Delta;\Psi;\Gamma \vdash o_1 : \mathcal{S}(u) \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathcal{S}(v) \quad (\Gamma(\mathsf{ck}) = 2 + t)\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{subjae}\ \mathsf{r}_d, o_1, o_2, o_3\ I}\ (\text{OK\_SUBJAE})$$

B.7.1 Unofficial Rules

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma\{\mathsf{r}_d{:}\mathsf{B}W, \mathsf{ck}{:}t\} \vdash I \\ \Delta;\Psi;\Gamma \vdash o_3 : \Gamma\{\mathsf{r}_d{:}\mathcal{S}(u + v), \mathsf{ck}{:}t\} \to 0 \\ \Delta;\Psi;\Gamma \vdash o_1 : \mathcal{S}(u) \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathcal{S}(v) \quad (\Gamma(\mathsf{ck}) = 2 + t)\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{addjno}\ \mathsf{r}_d, o_1, o_2, o_3\ I}\ (\text{OK\_ADDJNO})$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o_1 : \mathcal{S}(u + v) \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathcal{S}(u) \\ \Delta;\Psi;\Gamma'\{\mathsf{ck}{:}t\} \vdash I \quad \Delta;\Psi;\Gamma \vdash d : \mathcal{S}(v) \to \Gamma' \quad (\Gamma(\mathsf{ck}) = 1 + t)\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{sub}\ d, o_1, o_2\ I}\ (\text{OK\_SUB\_SETEQ})$$

B.8 $\boxed{\Delta;\Psi;\Gamma \vdash I \text{ inits } r{:}\mathbf{mbox}(\tau)}$ Object Initialization

$$\frac{\begin{array}{c}\Delta \vdash \tau \leq \tau_1 \times \tau_2 \times \tau_3 \\ \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \quad \Delta \vdash \tau_2' : \mathsf{T}m \quad (\Gamma(\mathsf{ck}) = 1 + t) \\ \Delta;\Psi;\Gamma \vdash o : \tau_2' \quad \Delta;\Psi;\Gamma\{\mathsf{ck}{:}t\} \vdash I \text{ inits } r{:}\mathtt{mbox}(\tau_1 \times \tau_2' \times \tau_3)\end{array}}{\Delta \vdash \mathtt{mov}\ m`[r + n], o\ I \text{ inits } r{:}\mathtt{mbox}(\tau)}\ (\mathtt{ok\_init\_mov})$$

$$\frac{\begin{array}{c}\Delta \vdash \tau \leq \tau_1 \times \tau_2 \times \tau_3 \\ \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \quad (\Gamma(\mathsf{ck}) = 1 + t) \\ \Delta \vdash \Gamma(\mathsf{esp}) \leq \tau_2' \times \tau' \quad \Delta \vdash \tau_2' : \mathsf{T}m \quad \Delta \vdash \tau' : \mathsf{TD} \\ \Delta; \Psi; \Gamma\{\mathsf{esp}{:}\tau', \mathsf{ck}{:}t\} \vdash I \text{ inits } r{:}\mathtt{mbox}(\tau_1 \times \tau_2' \times \tau_3)\end{array}}{\Delta \vdash \mathsf{pop}\ m, m`[o+n]\ I \text{ inits } r{:}\mathtt{mbox}(\tau)}\ (\mathtt{ok\_init\_pop})$$

$$\frac{\Delta; \Psi; \Gamma\{r{:}\mathtt{mbox}(\tau)\} \vdash I}{\Delta; \Psi; \Gamma \vdash I \text{ inits } r{:}\mathtt{mbox}(\tau)}\ (\mathtt{ok\_init\_done})$$

B.9 $\boxed{\Psi;\Delta \vdash I : \tau\ \mathsf{block}}$ Block Typing

$$\frac{\Psi; (\Delta, \varphi\ \mathsf{true}) \vdash I : \tau\ \mathsf{block}}{\Psi; \Delta \vdash I : \varphi \Rightarrow \tau\ \mathsf{block}}\ (\textsc{blockok\_guard})$$

Appendix C

Rational Semantic Proofs

This appendix continues the discussion of Chapter 4 to describe a somewhat more powerful variant of the TALT-R constraint logic. The result that there exists a proof within this peculiar logic of a given formula under given hypotheses if and only if there exists a feasible solution to a certain integer program (in the proof of Theorem 4.1) raises some questions. Among the most obvious is, what happens if this linear program is feasible over the rationals but not the integers? It is not hard to convince oneself that when the program is feasible over $\mathbb{Q}$, the constraint problem from which it was derived is valid over $\mathbb{Q}$ (and hence also $\mathbb{Z}$), for a feasible solution provides a set of multipliers by which any high school algebra student may scale the hypotheses, add them together and conclude the truth of the goal formula — and to the high school student, it makes little difference whether these multipliers are integers or not.

By allowing the possibility of noninteger coefficients in a linear combination, in fact, we all but exhaust the high school algebra student's repertoire of techniques for deriving such inequalities. Indeed, it turns out that if the restriction to integers in the characteristic linear program is dropped, then an interesting completeness property can be shown to hold. It further turns out that only one more rule must be added to the TALT-R constraint logic to get a proof system of equivalent power — that is, capable of deriving any constraint judgment that denotes a valid entailment over the rational numbers (with one technical restriction). Those two results are the subject of this appendix.

Other than a few notational definitions, surprisingly little work is needed to prove that there exists a (rational) feasible solution to the linear program in the proof of Theorem 4.1 (hereafter called the ***characteristic linear program*** of the judgment $\Delta \vdash t \leq u$ true) whenever, and only when, the constraints Δ imply $t \leq u$ over the nonnegative rationals. In fact, this is a simple corollary of so-called ***linear programming duality,*** a concept well understood by numerical optimization experts if not by programming language designers. For an overview of the topic I refer the reader to the popular algorithms textbook from which the following definitions, notations, and statement of the key theorem are adapted [10].

C.1 Linear Programming Duality

A linear program is an optimization problem in which the goal is to maximize (or minimize) the value of one linear polynomial (the ***objective function***) subject to a set of constraints, each of which is a linear equation or inequality. A linear program in ***standard form*** can be written like so:

$$\begin{array}{ll} \text{Maximize:} & c_1 x_1 + \cdots + c_n x_n \\ \\ \text{subject to:} & a_{11} x_1 + \cdots + a_{1n} x_n \leq b_1 \\ & \vdots \qquad\qquad \vdots \qquad \vdots \\ & a_{m1} x_m + \cdots + a_{mn} x_n \leq b_m \end{array}$$

and $x_i \geq 0$ for $1 \leq i \leq n$.

The nonnegativity constraints for all the variables are usually left implicit. Every linear program has an equivalent standard form.

The ***dual*** of the linear program above is the following one (not in standard form):

$$\begin{array}{ll} \text{Minimize:} & b_1 y_1 + \cdots + b_m y_m \\ \\ \text{subject to:} & a_{11} y_1 + \cdots + a_{m1} y_m \geq c_1 \\ & \vdots \qquad\qquad \vdots \qquad \vdots \\ & a_{1n} y_1 + \cdots + a_{mn} y_m \geq c_n \end{array}$$

and $y_i \geq 0$ for $1 \leq i \leq m$.

In other words, to obtain the dual linear program from the original (or ***primal***), we:

1. transpose the matrix of coefficients in the constraints, so that there are now m variables and n constraints;

2. interchange the roles of the constant terms (the b's in the primal) and the objective function coefficients (the c's in the primal);

3. replace maximization with minimization; and

4. reverse the sense of each inequality constraint (except for the nonnegativity constraints).

(The last of these is the reason the dual program as given above is not in standard form.) The linear programming duality theorem states that the primal and dual linear programs have the same optimal objective value.

Theorem C.1 (Linear-programming duality) *For primal and dual linear programs as given above, if $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$ is an optimal solution to the primal linear program and $\bar{y} = (\bar{y}_1, \ldots, \bar{y}_m)$ is an optimal solution to the dual linear program, then*

$$\sum_{i=1}^{n} c_i x_i = \sum_{j=1}^{m} b_j y_j .$$

Proof: See [10], Chap. 29, Thm. 29.10.

C.2 Characteristic Linear Programs

An ***assignment*** over a set A is a function $\eta : \mathrm{Var} \to A$. If P is a linear polynomial (in the sense of Definition 4.2), then the ***application*** of P to an assignment η over $\mathbb{Q}$ is the rational number given by

$$P@\eta = P(\overline{1}) + \sum_{x \in \mathrm{Var}} P(x)\eta(x).$$

(If η is an assignment over $\mathbb{Z}$, then $P@\eta$ is an integer.) An assignment η over the nonnegative rationals (or over the nonnegative integers) is a ***rational model*** (or an ***integer model***, respectively) of the polynomial constraint $(P \leq 0)$ if $P@\eta \leq 0$. A model of a set of polynomial constraints is an assignment that is a model of every constraint in the set. A set of polynomial constraints is ***consistent*** over $\mathbb{Q}$ or $\mathbb{Z}$ if it has at least one rational or integer model, respectively, and ***inconsistent*** otherwise. A constraint judgment $\Delta \vdash t \leq u$ true is ***valid*** over $\mathbb{Q}$ or over $\mathbb{Z}$ if every rational or integer model, respectively, of $[\![\Delta]\!]$ is a model of $[\![t \leq u]\!]$.

Theorem C.2 *If Δ is finite and $[\![\Delta]\!]$ is consistent over $\mathbb{Q}$, then $\Delta \vdash t \leq u$ true is valid over $\mathbb{Q}$ if and only if the linear program in the proof of Theorem 4.1 is feasible over $\mathbb{Q}$.*

Proof: Suppose, in all that follows, that $[\![\Delta]\!]$ is consistent over $\mathbb{Q}$. Let $\mathcal{J}$ stand for the judgment $\Delta \vdash t \leq u$ true. The structure of this proof is as follows: we construct a feasible linear program L whose maximum objective value is nonpositive if and only if $\mathcal{J}$ is valid over $\mathbb{Q}$; by the duality theorem, it follows that L's dual program L^* can take on a negative objective value iff $\mathcal{J}$ is valid; finally we show that L^* can take on a negative objective value if and only if the characteristic linear program of Theorem 4.1 is feasible.

First, we construct a linear program corresponding directly to the validity of $\mathcal{J}$. Suppose $\Delta = \{H_1, \ldots, H_n\}$. Any assignment η that is a model of $[\![\Delta]\!]$ must by definition satisfy the following:

$$\begin{array}{ccccccccc} H_1(\overline{1}) & + & H_1(a_1)\eta(a_1) & + \cdots + & H_1(a_m)\eta(a_m) & \leq 0 \\ \vdots & & \vdots & \vdots & \vdots & \vdots \\ H_n(\overline{1}) & + & H_n(a_1)\eta(a_1) & + \cdots + & H_n(a_m)\eta(a_m) & \leq 0 \end{array}$$

where $a_1, \ldots, a_m$ are all of the constraint term variables appearing in $\mathcal{J}$. $\mathcal{J}$ is valid over the rationals if and only if for every such η, $[\![t \leq u]\!]@\eta \leq 0$, that is, if the objective value of the linear program

$$\begin{array}{ll} \text{Maximize:} & P(\overline{1}) + P(a_1)x_1 + \cdots + P(a_m)x_m \\ \\ \text{subject to:} & H_1(\overline{1}) + H_1(a_1)x_1 + \cdots + H_1(a_m)x_m \leq 0 \\ & \vdots \qquad\qquad\qquad \vdots \qquad\qquad \vdots \\ & H_n(\overline{1}) + H_n(a_1)x_1 + \cdots + H_n(a_m)x_m \leq 0 \\ \\ & \text{and } x_i \geq 0 \text{ for } 1 \leq i \leq m \end{array}$$

is always less than or equal to zero, where $P = [\![t \leq u]\!]$. Because $[\![\Delta]\!]$ is consistent, this linear program is feasible, so $\mathcal{J}$ is valid if and only if the optimal objective value of the program is nonpositive.

This program is not quite in standard form as defined above, because of the constant terms on the left-hand sides of the constraints. One way to obtain an equivalent program in standard form

is to add a new variable, $x_{\overline{1}}$, to play the role of unity, and constrain it to be equal to 1. In this way we obtain the linear program L:

$$\begin{array}{ll} \text{Maximize:} & P(\overline{1})x_{\overline{1}} + P(a_1)x_1 + \cdots + P(a_m)x_m \\ \\ \text{subject to:} & H_1(\overline{1})x_{\overline{1}} + H_1(a_1)x_1 + \cdots + H_1(a_m)x_m \leq 0 \\ & \vdots \qquad\qquad\qquad\qquad \vdots \qquad\qquad \vdots \\ & H_n(\overline{1})x_{\overline{1}} + H_n(a_1)x_1 + \cdots + H_n(a_m)x_m \leq 0 \\ & x_{\overline{1}} \qquad\qquad\qquad\qquad\qquad\qquad\quad \leq 1 \\ & -x_{\overline{1}} \qquad\qquad\qquad\qquad\qquad\qquad\quad \leq -1 \\ \\ & \text{and all variables} \geq 0. \end{array}$$

This program has a nonpositive optimum if and only if the judgment $\mathcal{J}$ is valid over $\mathbb{Q}$.

The dual of the program above is:

$$\begin{array}{ll} \text{Minimize:} & y_{\overline{1}} - y_{-\overline{1}} \\ \\ \text{subject to:} & H_1(\overline{1})y_1 + \cdots + H_n(\overline{1})y_n + y_{\overline{1}} - y_{-\overline{1}} \geq P(\overline{1}) \\ & H_1(a_1)y_1 + \cdots + H_n(a_1)y_n \qquad\qquad \geq P(a_1) \\ & \vdots \qquad\qquad\qquad\quad \vdots \qquad\qquad\qquad\quad \vdots \\ & H_1(a_m)y_1 + \cdots + H_n(a_m)y_n \qquad\qquad \geq P(a_m) \\ \\ & \text{and all variables} \geq 0. \end{array}$$

The constraints of this linear program are almost exactly the same as those of the characteristic linear program of Theorem 4.1. The only difference is the presence of the two extra terms in the first line of this dual linear program, which are missing in the characteristic linear program. Therefore, we can say that the characteristic linear program is feasible if and only if there is a feasible solution to the present program with objective value zero.

Now, if the judgment $\mathcal{J}$ is valid over the rationals, then the objective value of the primal linear program is bounded above by zero, so its maximum objective value is nonpositive. This means that the minimum objective value of the dual linear program is also nonpositive. But because the only occurrence of the variables in the objective function in the constraints is in the first line, where the objective function appears positively on the left of a $\geq$, if there is a feasible solution with negative objective value, then there is one with objective value zero. This latter solution corresponds to a solution to the characteristic linear program.

Conversely, if the characteristic system of inequalities is satisfiable, then by setting $y_{\overline{1}} = y_{-\overline{1}} = 0$ we obtain a feasible solution to the dual program with objective value zero. There are now two cases: either the dual program is unbounded, or it has a minimum objective value that is nonpositive. In the latter case, the maximum objective value of the primal program is nonpositive and so $\mathcal{J}$ is valid over the rationals. As for the unbounded case, it is a fact that for any pair of dual linear programs, the objective value of any feasible solution to the maximization problem is less than or equal to that of any feasible solution to the minimization problem; it follows that if either one is unbounded, then other is infeasible. Therefore, if the dual program is unbounded, then $[\![\Delta]\!]$ is inconsistent, contradicting our assumption to the contrary.

We conclude that $\mathcal{J}$ is valid over $\mathbb{Q}$ if and only if the characteristic system of inequalities has a rational solution.

End of Proof.

C.3 Rational Semantic Proofs

Now that I have shown the connection between the validity of a truth judgment and the satisfiability of is characteristic inequalities, we can work backwards through the development of Section 4.2 to discover an extension of the TALT-R constraint logic capable of deriving all those judgments that are true in that sense. The first step is to define a new notion of semantic proof that admits a rational version of the proof of Theorem 4.1.

Definition C.1 *A **rational semantic proof** of $(P \leq 0)$ in context Δ is a semantic proof of $(qP \leq 0)$ in Δ for some positive integer q.*

Lemma C.1 *There exists a rational semantic proof of $[\![t \leq u]\!]$ in context Δ if and only if there is a rational solution to the characteristic inequalities of $\Delta \vdash t \leq u$ true.*

Proof Sketch: Since the left-hand sides of those inequalities are homogeneous and the right-hand sides consist of the coefficients in P, scaling any solution to the inequalities for P by an integer q gives a solution to the inequalities for qP.
End of Sketch.

Lemma C.2 *If there exist rational semantic proofs of $(P \leq 0)$ and $(Q \leq 0)$ in context Δ, then there exists a rational semantic proof of $(P + Q \leq 0)$ in Δ.*

Proof: Suppose $\Delta \vdash M : pP \leq 0$ and $\Delta \vdash N : qQ \leq 0$ where p and q are positive integers. If $M = (A, F)$, then define qM to consist of the multiset qA containing q copies of A and the polynomial qF, and define pN similarly. Then $\Delta \vdash qM : pqP \leq 0$ and $\Delta \vdash pN : pqQ \leq 0$. By Lemma 4.12, there is a semantic proof of $pq(P + Q) \leq 0$ in Δ, which is a rational semantic proof of $(P + Q) \leq 0$.
End of Proof.

C.4 Augmented Syntactic Proof System

Finally, I modify the TALT-R constraint logic to correspond to this new semantic proof theory by adding one additional rule schema:

$$\frac{\Delta \vdash^{+} \overbrace{t + \cdots + t}^{n} \leq \overbrace{u + \cdots + u}^{n} \text{ true} \quad (n \in \{1, 2, \ldots\})}{\Delta \vdash^{+} t \leq u \text{ true}}$$

(I distinguish the augmented system from the original by writing $\vdash^{+}$ in place of $\vdash$.)

The new rule essentially allows for the high-school algebra operation of dividing both sides of an inequality by a constant positive integer, provided this does not produce any nonintegral coefficients on either side. Of course, there is no multiplication in the constraint term language, so the "division" is really the removal of repeated addition. Together with the ability to add inequalities (the monotonicity rule), this allows hypotheses in a proof to be scaled by any rational factor so long as the resulting formula is expressible using integer coefficients. The augmented system of syntactic proof rules is equivalent to the semantic proof theory just defined.

Lemma C.3 *For any terms t and u and positive integer n,*

$$[\![\underbrace{t + \cdots + t}_{n} \leq \underbrace{u + \cdots + u}_{n}]\!] = n[\![t \leq u]\!].$$

Proof: Omitted.

Lemma C.4 (Soundness of Rational Semantic Proof) *If there is a rational semantic proof of $[\![t \leq u]\!]$ in context Δ, then $\Delta \vdash^{+} t \leq u$ true.*

Proof: Suppose $\Delta \models M : q[\![t \leq u]\!]$. By Lemma C.3, $\Delta \models M : [\![t + \cdots + t \leq u + \cdots + u]\!]$, where each sum has q copies of the term. By Lemma 4.11, $\Delta \vdash t + \cdots + t \leq u + \cdots + u$ true and thus $\Delta \vdash^{+} t + \cdots + t \leq u + \cdots + u$ true. By the new rule, $\Delta \vdash^{+} t \leq u$ true.
End of Proof.

Lemma C.5 (Completeness of Rational Semantic Proof) *If $\Delta \vdash^{+} t \leq u$ true, then there is a rational semantic proof M of $[\![t \leq u]\!]$ in context Δ.*

Proof Sketch: Analogous to Lemma 4.13, using Lemma C.2 in place of Lemma 4.12, and with one new case.

Case:

$$\frac{\Delta \vdash^{+} \overbrace{t + \cdots + t}^{n} \leq \overbrace{u + \cdots + u}^{n} \text{ true} \quad (n \in \{1, 2, \ldots\})}{\Delta \vdash^{+} t \leq u \text{ true}}$$

By the induction hypothesis, $\Delta \models M : q[\![t + \cdots + t \leq u + \cdots + u]\!]$.
By Lemma C.3, $\Delta \models M : qn[\![t \leq u]\!]$.
Thus $[\![t \leq u]\!]$ is rationally semantically provable, as desired.
End of Sketch.

Theorem C.3 (Characterization of Augmented System)

1. *It is decidable whether or not $\Delta \vdash^{+} \varphi$ true.*
2. *If $\Delta \vdash^{+} \varphi$ true, then $\Delta \vdash \varphi$ true is valid over $\mathbb{Q}$.*
3. *If Δ is consistent over $\mathbb{Q}$ and $\Delta \vdash \varphi$ true is valid over $\mathbb{Q}$, then $\Delta \vdash^{+} \varphi$ true.*

Proof:

1. By Lemmas C.1, C.4 and C.5, it suffices to decide whether there is a rational solution to the characteristic inequalities. This can be accomplished using any linear programming algorithm.
2. Suppose $\Delta \vdash^{+} \varphi$ true. By Lemma C.5, there is a rational semantic proof of $[\![\varphi]\!]$ in Δ. By Lemma C.1, there is a rational solution to the characteristic inequalities. By Theorem C.2, either $\Delta \vdash \varphi$ true is valid over $\mathbb{Q}$ or Δ is inconsistent over $\mathbb{Q}$. But if Δ has no rational model, then $\Delta \vdash \varphi$ true is vacuously valid over $\mathbb{Q}$.
3. Suppose Δ is consistent over $\mathbb{Q}$ and $\Delta \vdash \varphi$ true is valid over $\mathbb{Q}$. By Theorem C.2, there is a rational solution to the characteristic inequalities; by Lemma C.1, there is a rational semantic proof of $[\![\varphi]\!]$. By Lemma C.4, $\Delta \vdash^{+} \varphi$ true.

End of Proof.

Appendix D

Typing Rules for Lilt

$\boxed{\Delta \vdash \Phi \quad \Delta \vdash \Lambda \quad \Delta \vdash \Gamma \quad \Delta \vdash \Xi}$

$$\frac{\Delta \vdash \tau_i : T \text{ for } 1 \leq i \leq n}{\Delta \vdash (f_1{:}\tau_1, \ldots, f_n{:}\tau_n)} \qquad \frac{\Delta \vdash \gamma_i \text{ btype for } 1 \leq i \leq n}{\Delta \vdash (ell_1{:}\gamma_1, \ldots, \ell_n{:}\gamma_n)}$$

$$\frac{\Delta \vdash \tau_i : T \text{ for } 1 \leq i \leq n}{\Delta \vdash [s_1{:}\tau_1, \ldots, s_n{:}\tau_n]} \qquad \frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Xi \quad \Delta \vdash \Gamma}{\Delta \vdash \Xi, \Gamma}$$

$$\frac{}{\Delta \vdash \cdot \text{ handles } \Gamma} \qquad \frac{\Delta \vdash \Gamma \leq \Gamma'}{\Delta \vdash (\Xi, \Gamma') \text{ handles } \Gamma}$$

$\boxed{\Delta \vdash \Xi \text{ handles } \Gamma}$

$\boxed{\Delta \vdash \tau_1 \leq \tau_2 \quad \Delta \vdash \Gamma_1 \leq \Gamma_2 \quad \Delta \vdash \Xi_1 \leq \Xi_2}$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : T}{\Delta \vdash \tau_1 \leq \tau_2} \qquad \frac{\Delta \vdash \tau_1 : T \quad \Delta \vdash \tau_2 = \mathsf{ns} : T}{\Delta \vdash \tau_1 \leq \tau_2}$$

$$\frac{\Delta \vdash \tau_i \leq \tau_i' \text{ for } 1 \leq i \leq n}{\Delta \vdash [s_1{:}\tau_1, \ldots, s_n{:}\tau_n] \leq [s_1{:}\tau_1', \ldots, s_n{:}\tau_n']} \qquad \frac{}{\Delta \vdash \cdot \leq \cdot} \qquad \frac{\Delta \vdash \Xi_1 \leq \Xi_2 \quad \Delta \vdash \Gamma_2 \leq \Gamma_1}{\Delta \vdash (\Xi_1, \Gamma_1) \leq (\Xi_2, \Gamma_2)}$$

$\boxed{\Delta \vdash c : k}$

$$\frac{((\alpha{:}k) \in \Delta)}{\Delta \vdash \alpha : k} \qquad \frac{}{\Delta \vdash \mathsf{ns} : T} \qquad \frac{}{\Delta \vdash \mathtt{int} : T} \qquad \frac{}{\Delta \vdash \mathtt{bool} : T} \qquad \frac{}{\Delta \vdash \mathtt{unit} : T}$$

$$\frac{\Delta \vdash \tau_i : T \text{ for } 1 \leq i \leq k}{\Delta \vdash \langle \tau_1, \ldots, \tau_k \rangle : T} \qquad \frac{(i_j \neq i_k \text{ for } j \neq k) \quad \Delta \vdash \tau_i : T \text{ for } 1 \leq i \leq k}{\Delta \vdash [i_1{:}\tau_1, \ldots, i_n{:}\tau_n] : T} \qquad \frac{\Delta \vdash \tau : T \quad \Delta \vdash \tau_i : T \text{ for } 1 \leq i \leq n}{\Delta \vdash (\tau_1, \ldots, \tau_n) \to \tau : T}$$

$$\frac{\Delta \vdash \tau : T}{\Delta \vdash \tau\ \texttt{array} : T} \qquad \frac{\Delta, \alpha{:}T \vdash \tau : T}{\Delta \vdash \mu\alpha.\tau : T} \qquad \frac{\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n \vdash \tau : T}{\Delta \vdash \forall\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau : T}$$

$$\frac{\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n \vdash \tau : T}{\Delta \vdash \exists\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau : T} \qquad \frac{\Delta, \alpha{:}k_1 \vdash c : k_2}{\Delta \vdash \lambda\alpha{:}k_1.c : k_1 \to k_2} \qquad \frac{\Delta \vdash c_1 : k_2 \to k \quad \Delta \vdash c_2 : k_2}{\Delta \vdash c_1\, c_2 : k}$$

$\boxed{\Delta \vdash \gamma\ \mathsf{btype}}$

$$\frac{(\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \emptyset) \quad (\Delta, \Delta') \vdash \Xi \quad (\Delta, \Delta') \vdash \Gamma}{\Delta \vdash \mathit{lbl}(\Delta'; \Xi; \Gamma)\ \mathsf{btype}} \qquad \frac{(\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \emptyset) \quad (\Delta, \Delta') \vdash \Xi \quad (\Delta, \Delta') \vdash \Gamma}{\Delta \vdash \mathit{hnd}(\Delta'; \Xi; \Gamma)\ \mathsf{btype}}$$

$\boxed{\Delta \vdash c_1 = c_2 : k}$

$$\frac{((\alpha{:}k) \in \Delta)}{\Delta \vdash \alpha = \alpha : k} \qquad \frac{}{\Delta \vdash \mathsf{ns} = \mathsf{ns} : T} \qquad \frac{}{\Delta \vdash \mathtt{int} = \mathtt{int} : T} \qquad \frac{}{\Delta \vdash \mathtt{bool} = \mathtt{bool} : T}$$

$$\frac{}{\Delta \vdash \mathtt{unit} = \mathtt{unit} : T} \qquad \frac{\Delta \vdash \tau_i = \tau_i' : T \text{ for } 1 \le i \le k}{\Delta \vdash \langle \tau_1, \ldots, \tau_k \rangle = \langle \tau_1', \ldots, \tau_2' \rangle : T}$$

$$\frac{(i_j \neq i_k \text{ for } j \neq k) \quad \Delta \vdash \tau_i = \tau_i' : T \text{ for } 1 \le i \le k}{\Delta \vdash [i_1{:}\tau_1, \ldots, i_n{:}\tau_n] = [i_1{:}\tau_1', \ldots, i_n{:}\tau_n'] : T} \qquad \frac{\Delta \vdash \tau = \tau' : T \quad \Delta \vdash \tau_i = \tau_i' : T \text{ for } 1 \le i \le n}{\Delta \vdash (\tau_1, \ldots, \tau_n) \to \tau = (\tau_1', \ldots, \tau_n') \to \tau' : T}$$

$$\frac{\Delta \vdash \tau = \tau' : T}{\Delta \vdash \tau\ \texttt{array} = \tau'\ \texttt{array} : T} \qquad \frac{\Delta, \alpha{:}T \vdash \tau = \tau' : T}{\Delta \vdash \mu\alpha.\tau = \mu\alpha.\tau' : T}$$

$$\frac{\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n \vdash \tau = \tau' : T}{\Delta \vdash \forall\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau = \forall\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' : T} \qquad \frac{\Delta, \alpha{:}k_1 \vdash c = c' : k_2}{\Delta \vdash \lambda\alpha{:}k_1.c = \lambda\alpha{:}k_1.c' : k_1 \to k_2}$$

$$\frac{\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n \vdash \tau = \tau' : T}{\Delta \vdash \exists\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau = \exists\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' : T} \qquad \frac{\Delta \vdash c_2 = c_2' : k_2 \quad \Delta \vdash c_1 = c_1' : k_2 \to k}{\Delta \vdash c_1\, c_2 = c_1'\, c_2' : k}$$

$$\frac{\Delta, \alpha{:}k_2 \vdash c_1 : k \quad \Delta \vdash c_2 : k_2}{\Delta \vdash (\lambda\alpha{:}k_2.c_1)\, c_2 = c_1[c_2/\alpha] : k}$$

$\boxed{\Delta \vdash q : \tau_1 \Rightarrow \tau_2}$

$$\frac{}{\Delta \vdash \mathtt{id} : \tau \Rightarrow \tau} \qquad \frac{\Delta \vdash c_i : k_i \text{ for } 1 \le i \le n}{\Delta \vdash [c_1, \ldots, c_n] : \forall\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau \Rightarrow \tau[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n]}$$

$$\frac{\Delta \vdash \tau = \mu\alpha.\tau' : T}{\Delta \vdash \texttt{roll}_\tau : \tau'[\tau/\alpha] \Rightarrow \tau} \qquad \frac{\Delta \vdash \mu\alpha.\tau : T}{\Delta \vdash \texttt{unroll} : \mu\alpha.\tau \Rightarrow \tau[\mu\alpha.\tau/\alpha]}$$

$$\frac{\Delta \vdash \tau = \exists\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' : T \quad \Delta \vdash c_i : k_i \text{ for } 1 \le i \le n}{\Delta \vdash \texttt{pack}[\tau, c_1, \ldots, c_n] : \tau'[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n] \Rightarrow \tau} \qquad \frac{\begin{array}{c}\Delta \vdash q : \tau_1' \Rightarrow \tau_2' \\ \Delta \vdash \tau_i = \tau_i' : T \text{ for } i = 1, 2\end{array}}{\Delta \vdash q : \tau_1 \Rightarrow \tau_2}$$

$\boxed{\Phi; \Delta; \Gamma \vdash r : \tau}$

$$\frac{(\Gamma(s) = \tau)}{\Phi; \Delta; \Gamma \vdash s : \tau} \qquad \frac{}{\Phi; \Delta; \Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Phi; \Delta; \Gamma \vdash \texttt{tt} : \texttt{bool}} \qquad \frac{}{\Phi; \Delta; \Gamma \vdash \texttt{ff} : \texttt{bool}}$$

$$\frac{}{\Phi; \Delta; \Gamma \vdash \star : \texttt{unit}} \qquad \frac{(\Phi(f) = \tau)}{\Phi; \Delta; \Gamma \vdash f : \tau} \qquad \frac{\Phi; \Delta; \Gamma \vdash v : \tau_2 \quad \Delta \vdash q : \tau_2 \Rightarrow \tau}{\Phi; \Delta; \Gamma \vdash q@v : \tau}$$

$$\frac{\Phi; \Delta; \Gamma \vdash r : \tau' \quad \Delta \vdash \tau' = \tau}{\Phi; \Delta; \Gamma \vdash r : \tau} \qquad \frac{(op : (\tau_1, \ldots, \tau_k) \to \tau) \quad \Phi; \Delta; \Gamma \vdash v_i : \tau_i \text{ for } 1 \le i \le k}{\Phi; \Delta; \Gamma \vdash op(v_1, \ldots, v_k) : \tau}$$

$$\frac{\Phi; \Delta; \Gamma \vdash v_i : \tau_i \text{ for } 0 \le i \le k}{\Phi; \Delta; \Gamma \vdash \langle v_0, \ldots, v_k \rangle : \langle \tau_0, \ldots, \tau_k \rangle} \qquad \frac{\Phi; \Delta; \Gamma \vdash v : \langle \tau_0, \ldots, \tau_k \rangle}{\Phi; \Delta; \Gamma \vdash \pi_i v : \tau_i} \qquad \frac{\Phi; \Delta; \Gamma \vdash v_i : \tau \text{ for } 1 \le i \le n}{\Phi; \Delta; \Gamma \vdash \{v_1, \ldots, v_n\} : \tau\ \texttt{array}}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau = [\ldots, j{:}\tau_j, \ldots] \\ \Phi; \Delta; \Gamma \vdash v : \tau_j\end{array}}{\Phi; \Delta; \Gamma \vdash \texttt{inj}_\tau(j, v) : \tau} \qquad \frac{\Phi; \Delta; \Gamma \vdash v : [i : \tau]}{\Phi; \Delta; \Gamma \vdash \texttt{outj}(v) : \tau}$$

$\boxed{\Phi; \Delta; \Gamma \vdash \mathit{cond}\ \mathsf{cond}}$

$$\frac{\Phi; \Delta; \Gamma \vdash v_i : \texttt{int} \text{ for } i = 1, 2}{\Phi; \Delta; \Gamma \vdash v_1 = v_2\ \mathsf{cond}} \qquad \frac{\Phi; \Delta; \Gamma \vdash v_i : \texttt{int} \text{ for } i = 1, 2}{\Phi; \Delta; \Gamma \vdash v_1 < v_2\ \mathsf{cond}}$$

$\boxed{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash e}$

$$\frac{\Phi; \Delta; \Gamma \vdash v : \tau}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{return}\ v} \qquad \frac{\Phi; \Delta; \Gamma \vdash v : \tau_{\text{exn}} \quad \Delta \vdash \Xi\ \mathsf{handles}\ \Gamma}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{raise}\ v}$$

$$\frac{\begin{array}{c}(\Lambda(\ell) = \mathit{lbl}(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \\ \Delta \vdash c_i : k_i \quad \Delta \vdash \Gamma \le \Gamma'[\vec{c}/\vec{\alpha}] \quad \Delta \vdash \Xi \le \Xi'[\vec{c}/\vec{\alpha}]\end{array}}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{goto}\ \ell[c_1, \ldots, c_n]} \qquad \frac{\Phi; \Delta; \Gamma \vdash r : \tau' \quad \Phi; \Delta; \Lambda; \Xi; \Gamma[s \mapsto \tau']; \tau \vdash e}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{let}\ s = r\ \texttt{in}\ e}$$

$$\frac{\begin{array}{c}\Phi; \Delta; \Gamma \vdash v : (\tau_1' \ldots, \tau_n') \to \tau'' \quad \Delta \vdash \Xi\ \mathsf{handles}\ \Gamma \\ \Delta \vdash v_i : \tau_i' \text{ for } 1 \le i \le n \quad \Phi; \Delta; \Lambda; \Xi; \Gamma[s \mapsto \tau'']; \tau \vdash e\end{array}}{\Phi; \Delta; \Lambda; \Xi; \Gamma; \tau \vdash \texttt{let}\ s = v(v_1, \ldots, v_n)\ \texttt{in}\ e}$$

$$\frac{\Phi;\Delta;\Gamma \vdash v : \langle \tau_0, \ldots, \tau_m \rangle \quad \Phi;\Delta;\Gamma \vdash v : \tau_i \quad \Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash e}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}\ \pi_i\, v := v'\ \texttt{in}\ e} \qquad \frac{\Phi;\Delta;\Gamma \vdash v : \tau'\ \texttt{array} \quad \Phi;\Delta;\Gamma \vdash v' : \texttt{int} \quad \Delta \vdash \Xi\ \mathsf{handles}\ \Gamma \quad \Phi;\Delta;\Lambda;\Xi;\Gamma[s \mapsto \tau'];\tau \vdash e}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}\ s = \texttt{sub}(v, v')\ \texttt{in}\ e}$$

$$\frac{\Phi;\Delta;\Gamma \vdash v_1 : \tau'\ \texttt{array} \quad \Phi;\Delta;\Gamma \vdash v_2 : \texttt{int} \quad \Phi;\Delta;\Gamma \vdash v_3 : \tau' \quad \Delta \vdash \Xi\ \mathsf{handles}\ \Gamma \quad \Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash e}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}\ \texttt{sub}(v_1, v_2) := v_3\ \texttt{in}\ e}$$

$$\frac{\Phi;\Delta;\Gamma \vdash v : [\overline{j{:}\tau}, i{:}\tau', \overline{j{:}\tau}'] \quad \Phi;\Delta;\Lambda;\Xi;\Gamma[s \mapsto [i{:}\tau']];\tau \vdash e_1 \quad \Phi;\Delta;\Lambda;\Xi;\Gamma[s \mapsto [\overline{j{:}\tau}, \overline{j{:}\tau}']];\tau \vdash e_2}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{case}\ v\ \texttt{of}\ \texttt{inj}(i, s) \Rightarrow e_1\ \texttt{else}\ e_2}$$

$$\frac{\Phi;\Delta;\Gamma \vdash v : \exists \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' \quad \Phi;(\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n);\Lambda;\Xi;\Gamma[s \mapsto \tau'];\tau \vdash e}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}(\alpha_1, \ldots, \alpha_n, s) = \texttt{unpack}\ v\ \texttt{in}\ e}$$

$$\frac{\Phi;\Delta;\Gamma \vdash \mathit{cond}\ \mathsf{cond} \quad \Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash e_1 \quad \Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash e_2}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{if}\ \mathit{cond}\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \qquad \frac{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash e}{\Phi;\Delta;\Lambda;(\Xi, \Gamma');\Gamma;\tau \vdash \texttt{pophandler}\ \texttt{in}\ e}$$

$$\frac{(\Lambda(\ell) = \mathit{hnd}(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \quad \Delta \vdash c_i : k_i \quad \Delta \vdash \Xi \leq \Xi'[\vec{c}/\vec{\alpha}] \quad \Phi;\Delta;\Lambda;(\Xi, \Gamma'[\vec{c}/\vec{\alpha}]);\Gamma;\tau \vdash e}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{pushhandler}\ \ell[c_1, \ldots, c_n]\ \texttt{in}\ e}$$

$\boxed{\Phi;\Delta;\Lambda;\tau \vdash B : \gamma}$

$$\frac{\Delta, \Delta' \vdash \Xi \quad \Delta, \Delta' \vdash \Gamma \quad \Phi;(\Delta, \Delta');\Lambda;\Xi;\Gamma;\tau \vdash e}{\Phi;\Delta;\Lambda;\tau \vdash \texttt{block}(\Delta';\Xi;\Gamma).e : \mathit{lbl}(\Delta';\Xi;\Gamma)} \qquad \frac{\Delta, \Delta' \vdash \Xi \quad \Delta, \Delta' \vdash \Gamma \quad \Phi;(\Delta, \Delta');\Lambda;\Xi;\Gamma[s \mapsto \tau_{\mathrm{exn}}];\tau \vdash e}{\Phi;\Delta;\Lambda;\tau \vdash \texttt{hndl}(\Delta';\Xi;\Gamma;s).e : \mathit{hnd}(\Delta';\Xi;\Gamma)}$$

$\boxed{\Phi \vdash F : \tau}$

$$\frac{\vdash \Delta \quad \Delta \vdash \Gamma_{\mathrm{arg}} \quad \Delta \vdash \tau : T \quad \Delta \vdash \Lambda \quad \Phi;\Delta;\Lambda;\cdot;\Gamma;\tau \vdash e \quad \Phi;\Delta;\Lambda;\tau \vdash B_i : \Lambda(\ell_i)\ \text{for}\ 1 \leq i \leq m}{\Phi \vdash \texttt{func}(\Delta;\Gamma_{\mathrm{arg}};\tau).(\texttt{enter}(s_1, \ldots, s_n).e, \ell_1 = B_1, \ldots, \ell_m = B_m) : \forall\Delta.(\tau_1, \ldots, \tau_p) \to \tau}$$

where

$\Gamma_{\mathrm{arg}} = [s'_1{:}\tau_1, \ldots, s'_p{:}\tau_p]$
$\Gamma = [s'_1{:}\tau_1, \ldots, s'_p{:}\tau_p, s_1{:}\mathsf{ns}, \ldots, s_n{:}\mathsf{ns}]$
each B_i is either $\texttt{block}(\Delta_i;\Xi_i;\Gamma_i).e$ or $\texttt{hndl}(\Delta_i;\Xi_i;\Gamma_i;s).e$, and
$\mathrm{dom}(\Gamma_i) = \mathrm{dom}(\Gamma)$ for each i

$\boxed{\vdash P}$

$$\frac{\vdash \Phi \quad \Phi \vdash F_i : \Phi(f_i)\ \text{for}\ 1 \leq i \leq n \quad (\mathrm{dom}(\Phi) = \{f_1, \ldots, f_n\})}{\vdash f_1 = F_1, \ldots, f_n = F_n}$$

Bibliography

[1] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. Technical Report MSR-TR-2006-43, Microsoft Research, Redmond, WA, 2006.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical Report 86-727, Cornell University, Ithaca, NY, 1986.

[3] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, June 2001.

[4] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, MA, 2000.

[5] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resource verification. In *Proceedings of the International Conference on Theorem Proving in Higher-Order Logics*, September 2004.

[6] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[7] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical Report RC 19887 (December 21, 1994 – Declassified March 16, 1995), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1994.

[8] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, June 2000.

[9] The ConCert project home page. `http://www.cs.cmu.edu/~concert/`.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001. Fourth printing, 2003.

[11] Karl Crary. Personal communication.

[12] Karl Crary. Typed compilation of inclusive subtyping. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[13] Karl Crary. Toward a foundational typed assembly language. Technical Report CMU-CS-02-196, Carnegie Mellon University, Pittsburgh, PA, December 2002.

[14] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the Thirtieth ACM Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, January 2003.

[15] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *Proceedings of the Conference on Automated Deduction (CADE-19)*, Miami, FL, July 2003.

[16] Karl Crary and Joseph C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 191–205, Pittsburgh, PA, October 2002.

[17] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 233–248, September 1999.

[18] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, MA, 2000.

[19] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 301–312, 1998.

[20] Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe — Probably. In *Proceedings of the Eleventh European Conference on Object-Oriented Programming (ECOOP 1997)*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer-Verlag, June 1997.

[21] R. Kent Dybvig. *The Scheme Programming Language*. The MIT Press, third edition, 2003. Full text online at `http://www.scheme.com/tspl3/`.

[22] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[23] Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming and Computer Architecture*, pages 179–187, Copenhagen, Denmark, June 1993.

[24] Folding@home. `http://folding.stanford.edu`.

[25] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, November 1999.

[26] Stephen N. Freund and John C. Mitchell. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning*, 2003.

[27] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[28] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2002.

[29] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31:191–229, December 2003.

[30] Robert Harper and Karl Crary. How to believe a Twelf proof. Online at `http://www.cs.cmu.edu/~rwh/papers/how/believe-twelf.pdf`, 2005.

[31] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[32] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Carnegie Mellon University, July 2000.

[33] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[34] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):102–121, 1987.

[35] M. Hofmann. Linear types and non-size increasing polynomial time computation. In *14th Annual IEEE Symposium on Logic in Computer Science (LICS)*, Trento, Italy, July 1999.

[36] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the Thirtieth ACM Symposium on Principles of Programming Languages*, pages 185–197, New Orleans, LA, January 2003.

[37] Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Stephen Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, 2005.

[38] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 1999.

[39] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*, 2006. Order Number 248966-013US.

[40] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[41] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[42] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[43] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[44] Frederick C. Mish, editor. *Merriam-Webster's Collegiate Dictionary*. Merriam-Webster, Springfield, MA, tenth edition, 1994.

[45] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, 1999.

[46] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[47] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[48] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[49] Mayur Naik. A type system equivalent to model checking. Master's thesis, Purdue University, 2003.

[50] George Necula. Proof-carrying code. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.

[51] George Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1997.

[52] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[53] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 142–154, London, United Kingdom, January 2001.

[54] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, September 1998. Technical report CMU-CS-98-154.

[55] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[56] Leaf Petersen. *Certifying Compilation for Standard ML in a Type Analysis Framework*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2005. Published as CMU Technical Report CMU-CS-05-135.

[57] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University, 2000.

[58] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *Proceedings of the Thirtieth ACM Symposium on Principles of Programming Languages*, New Orleans, LA, January 2003.

[59] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the Conference on Automated Deduction (CADE-16)*, pages 202–206, July 1999.

[60] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1991.

[61] Vijay Saraswat. Java is not type-safe, 1997. Available at http://citeseer.ist.psu.edu/saraswat97java.html.

[62] SETI@home. `http://setiathome.ssl.berkeley.edu`.

[63] Dorai Sitaram. Teach yourself Scheme in fixnum days. Online at `http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`, 2004.

[64] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. Technical Report 158, Digital Systems Research Center (SRC), 1998.

[65] Guy L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12:221–236, 1999.

[66] Sun Microsystems. Javasoft ships Java 1.0. Press release, January 1996.

[67] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, May 2006.

[68] Joseph C. Vanderwaart and Karl Crary. A typed interface for garbage collection. In *Proceedings of the Workshop on Types in Language Design and Implementation (TLDI)*, New Orleans, LA, January 2003.

[69] Joseph C. Vanderwaart and Karl Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, Carnegie Mellon University, 2004.

[70] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

[71] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[72] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Florence, Italy, September 2001.