Foundational Typed Assembly Language for Grid Computing

Joseph C. Vanderwaart and Karl Crary

February 3, 2004

CMU-CS-04-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report describes a type theory for certified code, called TALT-R, in which type safety guarantees cooperation with a mechanism to limit the CPU usage of untrusted code. At its core is the foundational typed assembly language TALT, extended with an instruction-counting mechanism, or "virtual clock", intended to bound the number of non-yielding instructions a program may execute in a row. The type theory also contains a form of dependent refinement that allows reasoning about integer values to be reflected in the typing of a program; in particular, the refinement system enables a simple but effective dynamic checking scheme for the clock, which we predict will greatly improve the performance of TALT-R programs. We exhibit a translation from a clock-ignorant source language into a form of TALT-R, demonstrating that the type system is expressive enough to write general programs in.

This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

Keywords: Certified code, type theory, typed assembly language, distributed computing, resource bound certification.

1 Introduction

The ubiquity of more and more powerful personal computers with connections to the Internet has given rise to the paradigm of *grid computing*, in which the idle cycles of large numbers of machines around the world are tapped to cooperatively solve large computational problems. Examples of this increasingly popular phenomenon include SETI@Home [36], Folding@Home [15], and distributed.net. In each of these projects, hundreds of thousands of participants download and install specialized software on their network-attached computers; this software runs when the machine is idle (either as a screensaver or as a low-priority background process), downloading problem instances from a central location, solving them and sending back the results.

The grid computing paradigm is thought of as providing a means for low-cost supercomputing, since the network of participants ("The Grid") essentially functions as one entity, a parallel "computer" with a very large number of processors. Unfortunately, the effective cost of using the Grid is still high, due to the fact that problem-specific software must be installed on every node that is to participate in a particular task. The use of mobile code technology can ease this problem: instead of requiring the owner of each node to download and install new software for every problem, a framework can be set up in which code is distributed to participating machines automatically and executed under the control of a supervising program running on each host. However, automatically executing code received over the network is one of the most obvious sources of security problems imaginable, so this form of grid computing must be conducted with great care. More importantly, it is still very costly in the sense that it is available only to those persons and organizations that are in a position to gain the trust of the required number of participants.

The ConCert project [4] aims to lower this barrier to entry, and realize the vision of the Grid as a supercomputer available for anyone's use, by removing the need for established trust relationships between the programmers of the Grid and the owners of the machines of which it is comprised. An important part of the solution lies in *certified code*: any piece of problem-specific code received by a host for execution is accompanied by a "certificate" whose validity implies, with the force of a mathematical proof, that the code will not compromise the security of the host. More precisely, the host owner specifies a *safety policy* that all incoming code must satisfy. The grid node implementation (called the *conductor* in ConCert terminology) then rejects all untrusted programs whose certificates do not prove they satisfy the safety policy. The most well-known variants of certified code are the Java Virtual Machine [22], Proof-Carrying Code (PCC) [29, 30] and Typed Assembly Language (TAL) [26, 24].

ConCert is work in progress, and there is a great deal remaining. This report describes our plans for addressing three particular observations about the use of certified code for grid computing. First, the safety policy should not be overly tailored to a particular source programming language or, worse, a particular compiler. Second, the more direct the logical connection between the certificate's validity and the satisfaction of the host owner's intended safety policy, the better. Third, the safety policy that grid programs are certified to satisfy must go beyond the traditional territory of type safety, memory safety, and respect for OS abstractions. Among other things, it must ensure that the consumption of host resources by untrusted programs is limited.

The first two of these observations have already led to the invention of *foundational certified code*. Roughly speaking, foundational certified code systems are characterized by operating at a very low level of abstraction. Since a concrete processor represents programs as bytes encoding sequences of instructions, the safety policy in a foundational system specifies those sequences of bytes that result in safe behavior when loaded and executed on the particular processor architecture being used. This safety policy is generally expressed in a formal logic, so that proofs of the safety of programs can be mechanically checked; these proofs play an important role in the certification process, though the details vary from one variant of foundational certified code to the next. In the foundational PCC systems of Appel *et al.* [2, 1] and Hamid *et al.* [18], the certificate for a program is just such a proof of the safety policy, encoded in a checkable logic. In the foundational framework described by Crary and Sarkar [7], the form of a certificate is less constrained, but there is required to be a machine-checkable proof that certificate validity implies safety.

The third observation, that a safety policy must include bounded resource consumption, motivates the need for what has come to be called *resource bound certification*. This report describes our plans for adding support for resource bound certification to the TALT assembly language [6], with the goal of producing a complete foundational certified code system—certifying compiler, verifier, and runtime system—that is capable of guaranteeing bounds on CPU usage of certified programs, and that is suitable for creating applications to run on the ConCert grid computing framework. To do this, we will assume that the (trusted) runtime system provides a "yield" operation to relinquish the CPU, and incorporate into the safety policy a maximum number of instructions that may be performed in a row without yielding. Even though we only intend to directly address this small subproblem of resource bound certification, however, elements of the type theory we develop may be useful for guaranteeing bounds on other resources as well; stack space bounds and network bandwith limiting seem particularly promising extensions.

The starting point of the work outlined in this report is the framework for foundational certified code described and (partially) implemented by Crary and Sarkar. Our type theory is formulated in this report as an extension of TALT, and we intend to construct our certification and verification tools and safety proofs by modifying and extending those already done for TALT. This implementation work, as well as a Popcorn compiler targeting the new assembly language, is ongoing.

1.1 Overview

The main technical content of this report consists of a type theory (based on TALT) for CPU usage bounds as hinted above, an informal discussion of some techniques for writing certified programs in this language, and a formal translation from a fairly general low-level language that is ignorant of resource bounds into our bounded assembly language. The remainder of this section gives an overview of these components, which comprise Sections 2 through 4. Section 5 describes related work and concludes.

A Type Theory for Resource Bounds In Section 2 of this report, we describe a type system for certified code with resource bounds based on the foundational typed assembly language TALT. Our new typed assembly language, which we call TALT-R, contains a `yield` instruction that must be performed at least once every Y instructions in order for a program to be considered safe. (The number Y is a parameter of the type theory and is determined by the safety policy). Concretely, we will modify the safety policy to include an imaginary clock that is decremented by one for every instruction executed; if the clock reaches zero and the next instruction is not `yield`, then the machine is stuck. So that well-typed programs can never get stuck, the TALT-R type system keeps track of (a conservative approximation of) the value of the clock at every point in the program.

In addition to simple instruction counting, our type theory for resource bound certification incorporates what is essentially a dependent refinement system in the style of DML [42, 44] and DTAL [43]. A major difference between our system and those of Xi *et al.,* however, is that we will explicitly specify a set of inference rules for the constraint domain. This is for the sake of the foundational safety proof: in the metatheorem language of Twelf (the meta-proof checker used by the TALT framework) it is impossible to state a theorem of the form "if the integer formula φ is valid..." but very convenient to state one of the form "given a proof of the formula φ..." Consequently, the safety of a well-certified program must depend not on the *truth* of formulas, but on their *provability*, and the set of rules for constructing these proofs must be fixed ahead of time.

Another (less important) aspect of foundationality is that we do not want the success of our system to depend on the inclusion of a constraint solver in the trusted computing base; we rather view the use of a trusted constraint solver as an optimization that may be used to trade some verification complexity for a reduction in certificate size. For the implementation we envision, proofs of any constraints required for a program will be included in the certificate, and there will be no need for any special support from the verifier. Of course, this means that the certifying compiler must come up with the necessary proofs.

A Generic Intermediate Language In order to show that TALT-R's type system is expressive enough for it to serve as a general-purpose typed assembly language, we will exhibit a translation from a resource-bound-ignorant language into TALT-R. The source language of this translation, which we call Lilt, is a rather low-level programming language based loosely on the higher-order polymorphic lambda calculus (F_ω). Lilt is intended to be generic in the sense that it could serve as an intermediate language in a compiler for a number of different source languages. In particular, the Popcorn-to-TALT-R compiler we plan to implement will be based on the Lilt-to-TALT-R translation given in this report. Because of its importance to our implementation, Section 3 of the report describes the Lilt language in some detail.

Resource-Bounded Compilation As far as safety is concerned, the problem of ensuring bounded yield latency during compilation to TALT-R is essentially that of placing `yield` instructions in the generated program such that the requirements of the TALT-R type system (and hence those of the safety policy) are satisfied. The inherent difficulty of this task is low: it could be accomplished easily (but pathologically) by placing a `yield` before every instruction in the program, or (somewhat more realistically) at the beginning of every basic block and every Y instructions thereafter. The real problem, then, is devising a strategy for placing yields so that as few as possible are executed by the program, minimizing the execution cost they introduce.

In the setting of the ConCert grid computing framework, the `yield` instruction is likely to be very expensive indeed. In particular, it will involve a function call to the runtime system and, in at least some cases, interprocess communication with the supervising ConCert node implementation. In recognition of the large cost, the yield frequency required by the safety policy will be low (that is, Y will be large). In Section 4 we describe a number of heuristics and strategies for placing `yield` instructions in TALT-R programs that we expect will yield good performance. In addition, we give a complete type-preserving translation from Lilt into TALT-R that incorporates some of these ideas.

2 TALT-R: Resource-Bounded Foundational TAL

In this section we give the syntax of our resource-bounded typed assembly language and an overivew of its static semantics, and discuss its unusual features. This assembly language is based very closely on TALT [6], with a small number of new constructs; we call it TALT-R (for "Resource"). We plan to implement the certification, verification, and formal safety proof for our system by adapting the prior work on TALT [7]. This section assumes the reader is familiar with TALT.

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

- TALT-R itself, which is the language for which the safety metatheorem is directly proved. This is an implicitly typed language (also known as a Curry-style or type-assignment system), meaning that programs are completely free of type annotations; type-checking in this language (as in TALT) is presumed undecidable, because it involves polymorphic type inference [41].

- XTALT-R (analogous to XTALT), which is to be the input language of the TALT-R assembler (and therefore also the direct target of the high-level language compiler we plan to implement). XTALT-R differs from TALT-R in two important ways: first, XTALT-R is explicitly typed, making type-checking feasible; and second, XTALT-R programs do not allow explicit pc-relative operands for control transfer instructions. Instead, an XTALT-R program is divided into *blocks*, each having a *label*; labels may be used as operands, and the assembler translates them into pc-relative addressing.

The primary design criterion of TALT and TALT-R is that the operational semantics mirror the concrete IA-32 architecture closely enough to make the foundational safety proof feasible. The primary design criterion of XTALT and XTALT-R is that they be suitable languages for generation by a compiler or by a human programmer.

Unfortunately, neither TALT-R nor XTALT-R is a particularly convenient language to use when formally describing a compiler as we must do in this report. Therefore, the typed assembly language we will describe and use in this document is a third variant, which we will call BTALT-R. BTALT-R is implicitly typed, like TALT-R, but incorporates XTALT-R's syntax for blocks and labels (as well as syntax for operands and destinations that is XTALT-like and hence more familiar to IA-32 programmers). Implicit typing reduces the notational overhead necessary for describing the compilation process; blocks and labels allow us to ignore pc-relative addressing.[1]

It is common in colloquial usage to use the name TALT to refer to the entire certified code system based on TALT, including the XTALT language. This is not entirely inappropriate, since XTALT is designed to correspond to TALT very closely; the theory of subtyping in TALT, for example, is exactly mirrored by the theory of coercions in XTALT. Regarding the resource-bounded versions we are about to describe, we consider XTALT-R and BTALT-R as merely two different "views" of the underlying type theory TALT-R. We will (unapologetically) use the name TALT-R when making statements intended to apply to all three.

2.1 Basic Syntax

Some of the architecture-specific notation we will use when discussing TALT-R is defined in Figure 1. The syntax for BTALT-R (not including the syntax of types) is shown in Figure 2. Aside from

[1] In order for our translation to output pc-relative displacements, we would have to be able to reason about instruction encoding lengths, which TALT leaves undefined. In any case it is perfectly usual for these calculations to be left to the assembler.

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

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

Figure 1: TALT-R Notation for IA-32

$$
\begin{array}{llcl}
\mathit{Operands} & o & ::= & B \mid \ell \mid \bar{r} \mid i`[o + j] \mid i`[o_1 + j + j' \cdot o_2] \\
\mathit{Destinations} & d & ::= & \bar{r} \mid i`[o + j] \mid i`[o_1 + j + j' \cdot o_2] \\
\mathit{Conditions} & \kappa & ::= & \mathtt{e} \mid \mathtt{ne} \mid \mathtt{b} \mid \mathtt{be} \mid \mathtt{a} \mid \mathtt{ae} \\
\mathit{Instruction\ Sequences} & I & ::= & \epsilon \\
 & & \mid & \mathtt{add}\ d, o_1, o_2\ I \\
 & & \mid & \mathtt{addsptr}\ d, o, n\ I \\
 & & \mid & \mathtt{call}\ o\ I \\
 & & \mid & \mathtt{cmp}\ o_1, o_2\ I \\
 & & \mid & \mathtt{cmpjcc}\ o_1, o_2, \kappa, o_3\ I \\
 & & \mid & \mathtt{jcc}\ \kappa, o\ I \\
 & & \mid & \mathtt{jmp}\ o\ I \\
 & & \mid & \mathtt{malloc}\ d, o, n\ I \\
 & & \mid & \mathtt{mallocarr}\ d, n, o_1, o_2\ I \\
 & & \mid & \mathtt{mov}\ d, o\ I \\
 & & \mid & \mathtt{pop}\ n, d\ I \\
 & & \mid & \mathtt{push}\ o\ I \\
 & & \mid & \mathtt{ret}\ I \\
 & & \mid & \mathtt{salloc}\ n\ I \\
 & & \mid & \mathtt{sfree}\ n\ I \\
 & & \mid & \mathtt{sub}\ d, o_1, o_2\ I \\
 & & \mid & \mathtt{subjae}\ \mathtt{r}_d, o_1, o_2, o_3\ I \\
 & & \mid & \mathtt{yield}\ I \\
\mathit{Programs} & P & ::= & \ell_1 = I_1, \ldots, \ell_n = I_n
\end{array}
$$

Figure 2: BTALT-R Syntax (Except Type System)

Kinds	K	$::=$	$\mathsf{T} \mid \mathsf{T}i \mid \mathsf{TD} \mid \mathsf{Word} \mid \mathsf{N} \mid \mathsf{P}$
Static Terms	c, t, φ, τ, x	$::=$	α
		$\mid$	$\mathtt{ns} \mid \mathtt{B0} \mid \mathtt{B1} \mid \tau_1 \times \tau_2 \mid \tau \uparrow x \mid \mathtt{box}(\tau) \mid \mathtt{mbox}(\tau) \mid \mathtt{sptr}(\tau)$
		$\mid$	$\Gamma \to 0 \mid \mathtt{set}_{=}(x) \mid \mathtt{set}_{<}(x) \mid \mathtt{set}_{>}(x) \mid \forall\alpha{:}K.\tau \mid \exists\alpha{:}K.\tau$
		$\mid$	$\tau_1 \wedge \tau_2 \mid \tau_1 \vee \tau_2 \mid \mathtt{void} \mid \mu\alpha.\tau \mid B \mid \mathtt{got}$
		$\mid$	$\varphi \Rightarrow \tau \mid \mathcal{S}(t)$
		$\mid$	$\overline{n} \mid t_1 + t_2$
		$\mid$	$t_1 \leq t_2 \mid t_1 = t_2$
Static Contexts	Δ	$::=$	$\cdot \mid \Delta, \alpha{:}K \mid \Delta, \varphi\ \mathsf{true}$
Register File Types	Γ	$::=$	$\{\mathtt{eax}{:}\tau_{\mathtt{ax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{bp}}, \mathtt{esp}{:}\tau, \mathtt{ck}{:}t\}$
Memory Types	Ψ	$::=$	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$

Figure 3: BTALT-R Type System Syntax

the syntax of operands and destinations, which we have changed to more closely resemble the Intel syntax, the syntax of programs in BTALT-R differs from TALT in two ways. The first is that a program in BTALT-R is a list of labeled instruction sequences as opposed to an initial machine state. The labels attached to these blocks may be used as operands, and are particularly useful in control transfer instructions. The second difference is that two new instructions have been added. The `yield` instruction is the key addition: it is this instruction that must be executed with at least a certain frequency to guarantee bounded CPU usage. The other new instruction is `subjae`: the instruction sequence $\mathtt{subjae}\ \mathtt{r}_d, o_1, o_2, o_3\ I$ has the same operational behavior as $(\mathtt{sub}\ \mathtt{r}_d, o_1, o_2; \mathtt{jcc}\ \mathtt{ae}, o_3; I)$, but a special typing rule that reflects the result of the conditional jump into the type system. In this sense it is related to the `cmpjcc` instruction inherited from TALT.

2.2 Type System

The syntax of the TALT-R type system is given in Figure 3. At the top level of the system are six *kinds*, which classify the terms at the second level, which we call *static terms*. The kinds T, $\mathsf{T}i$, TD and Word are inherited from TALT (but Word used to be called Num); N and P are new. The class of static terms is comprised of the *types* (of kinds $\mathsf{T}i$, TD and T), the *word terms* (of kind Word), the *constraint terms* (of kind N), and the the *constraint formulas* (of kind P). By convention, we will use the metavariables τ, x, t and φ in place of the general metavariable c to indicate that the static term referred to is a type, a word term, a constraint term, or a formula, respectively. Furthermore, we will use the letter a instead of α for variables intended to be of kind N. We have chosen to call the syntactic category containing the types "static terms" rather than the more usual "type constructors" (or simply "constructors") because although constraint terms and formulas may appear in types, they cannot really be said to *construct* anything. The name "static terms" also highlights our intention that these terms are part of the (static) type assignment system only; they do not appear in raw BTALT-R programs.

The role of the types and word terms is the same as in TALT: types classify values, and word terms appear in array types ($\tau \uparrow x$) and subrange types ($\mathtt{set}_{<}(x), \mathtt{set}_{=}(x), \mathtt{set}_{>}(x)$). The constraint terms and constraint formulas together form a logic of constraints that allows the integration of arithmetic reasoning into the type system. We will discuss the role of this reasoning in detail later on. The types of TALT-R are just those of TALT, plus singleton types $\mathcal{S}(t)$ and constraint-

Judgment	Meaning
$\Delta \vdash c : k$	c has kind k
$\Delta \vdash \Gamma$	Γ is well-formed
$\Delta \vdash \varphi\ \mathsf{true}$	Formula φ is true
$\Delta \vdash \tau_1 \leq \tau_2$	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
$\Delta; \Psi; \Gamma \vdash d : \tau \rightarrow \Gamma'$	Propagating a value of type τ to d yields Γ'
$\Delta; \Psi; \Gamma \vdash I$	I is well-typed
$\Delta; \Psi \vdash I : \tau\ \mathsf{block}$	I constitutes a block of type τ
$\vdash P$	P is well-typed

Figure 4: BTALT-R typing judgment forms

$$\frac{((\alpha{:}K) \in \Delta)}{\Delta \vdash \alpha : K} \qquad \frac{}{\Delta \vdash \overline{n} : \mathsf{N}}\ (n \geq 0) \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 + t_2 : \mathsf{N}}$$

$$\frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 \leq t_2 : \mathsf{P}} \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 = t_2 : \mathsf{P}}$$

Figure 5: Kinding for Constraint Terms and Formulas

guarded types $\varphi \Rightarrow \tau$. The sole value of type $\mathcal{S}(t)$ is the word-size encoding of the natural number denoted by t (provided such a value exists); a value of type $\varphi \Rightarrow \tau$ is a value that belongs to type τ provided that the formula φ is satisfied.

2.2.1 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 we will describe here is engineered mostly for clarity of presentation.

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 ($\wedge$, $\vee$, $\supset$, $\perp$) to the constraint logic; somewhat surprisingly, we have not found that this is necessary to accomplish our task. We therefore leave them out of this presentation for simplicity. The formation rules for constraint terms and formulas are given in Figure 5; note that a formula need not be "true" in order to be well-formed.

The notion of "truth" for constraint formulas is captured by a 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, Δ may contain both kinding assumptions of the form $\alpha{:}K$ and hypotheses of the form $\varphi\ \mathsf{true}$. The rules defining the truth judgment are given in Figure 6. They are intended to capture a useful, if naïve, theory of addition of natural numbers that will allow (at least) the output of our compiler to be certified. They include reflexivity, symmetry, transitivity and compatibility

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

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

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

$$\frac{\Delta \vdash t_1 = t_2\ \mathsf{true}}{\Delta \vdash t_1 \leq t_2\ \mathsf{true}} \qquad \frac{\Delta \vdash t_1 \leq t_3\ \mathsf{true} \quad \Delta \vdash t_3 \leq t_2\ \mathsf{true}}{\Delta \vdash t_1 \leq t_2\ \mathsf{true}} \qquad \frac{}{\Delta \vdash \overline{m} \leq \overline{n}\ \mathsf{true}}\ (m \leq n)$$

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

Figure 6: Truth of Formulas

rules for equality; an axiom for addition of natural number constants; identity, commutativity and associativity rules for addition; reflexivity and transitivity for $\leq$, and an axiom for ordering of constants; monotonicity of addition; and finally a rule allowing cancellation of an addend on both sides of an inequality.

2.2.2 The Virtual Clock

The key to the CPU usage bound capability of TALT-R is the *virtual clock*. To ensure that the `yield` instruction is performed at least every Y instructions, we add to the dynamic semantics of our language an imaginary "clock" register that is decremented for every instruction executed. No instruction (other than `yield`) may execute unless the counter is at least 1. This method of instruction counting is not new: Necula and Lee [30] proposed the use of a virtual clock for proof-carrying code, and Crary and Weirich [10] used one in their languages LXres and TALres. Unlike these other efforts, however, we are not attempting to bound total running time here; we am only interested in bounding the time until the next yield. The `yield` instruction, therefore, resets this counter to Y.

Accounting for the virtual clock in the type system is not difficult. 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) $\Gamma(\mathtt{ck})$. For example, the typing rule for the `add` instruction is:

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

Similarly, a code pointer of type $\Gamma' \to 0$ 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 $1 + \Gamma'(\texttt{ck})$ in order for the jump instruction itself to be safe:

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

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

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

The fact that $\Gamma(\texttt{ck})$ is an *approximation* of the virtual clock comes from TALT-R's rule for register file subtyping, which allows the constraint term assigned to `ck` to vary:

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

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 $\texttt{ck}{:}t$ 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.

2.2.3 Guarded and Singleton Types

There are two forms of type in TALT-R that are not present in TALT: *guarded types* $(\varphi \Rightarrow \tau)$ and *singleton types* $(\mathcal{S}(t))$. The intuitive meanings of these types are simple, but their usefulness may not be obvious until we discuss yield-placement strategies in Section 4. Basically, we 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 allow more efficient code to be written. They are not strictly necessary in the sense that it is possible to write a compiler whose output is well-typed without them, but we expect they will deliver significant performance benefits for a reasonably small metatheoretic investment.

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

Figure 7: Elementary Rules for Singletons

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:

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

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.

The introduction mechanism for guarded types differs slightly between TALT-R and BTALT-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 BTALT-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-R play a role similar to that of singletons in DTAL [43] and LTT [8]. 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. In TALT-R, the 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 7.

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 we will have for singleton types in TALT-R is to describe a counter which is repeatedly decremented until it reaches zero. Consequently, the only form of arithmetic we 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 we have already mentioned, the instruction sequence ($\texttt{subjae}\ \mathtt{r}_d, o_1, o_2, o_3\ I$) subtracts the value of o_2 from o_1 and stores the result in $\mathtt{r}_d$; if this result is greater than zero, control jumps to the address in o_3; otherwise, execution continues with I. The `subjae` instruction has a special singleton-aware typing rule:

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

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 positive 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 our 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.

While it appears that `subjae` is the only singleton arithmetic instruction necessary for efficient TALT-R programming, it would be nice to include others to make TALT-R as general as possible. Unfortunately, it is inconvenient to do so. The problem is that the results of singleton arithmetic ought to be reflected in the constraint logic, but the 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. An alternative solution is for all the singleton operations to be "double" instructions like `subjae`, so that they automatically detect when their results are inconsistent with natural number arithmetic. Unfortunately, the current TALT implementation does not yet support checking for integer overflow; we therefore make the addition of more singleton operations a low priority.

2.3 Certification and Verification

An implementation of all the tools necessary to use TALT-R for code certification in the context of the ConCert grid computing framework is currently under development, based on the certification machinery for TALT. The TALT implementation performs certification and verification using the Twelf logic programming and meta-proof checking software. The semantics of the IA-32 architecture (which constitutes the safety policy), the type system and abstract semantics of TALT, and the safety meta-proof stating that any well-typed TALT program is safe to run are all encoded in Twelf [7].

The basic structure of the system is as follows. Certifying compilers that wish to target TALT output programs in XTALT, an explicitly-typed variant of the language in which type-checking is possible. The relationship between XTALT and TALT is sufficiently tight that a metatheorem can be proven in Twelf stating that any binary obtained by assembling a well-typed XTALT program is also the representation of a well-typed TALT program. A certificate is simply the LF encoding of an XTALT program; a Twelf program (the "checker") verifies the correspondence between this XTALT program and an LF representation of the untrusted binary, and a string of several metatheorems relates the success of the checker to the safety of the binary.

It is critical to this methodology that type-checking of XTALT programs is tractable. Because of this requirement, XTALT does away with most uses of subtyping in favor of a calculus of *coercions*, which correspond closely to the subtyping derivations possible in the TALT type theory. (Not all uses of TALT subtyping require coercions in XTALT: certain very common subtyping idioms are "baked in" to the typing rules of XTALT to make programs easier to read and write.) Since TALT-R has some extra subtyping rules, it would be reasonable to add corresponding new forms of coercion to XTALT-R. A challenge arises, however, due to the role of the constraint logic in several of the new rules. In the most natural design, a coercion from $\varphi \Rightarrow \tau$ to τ would contain a proof of the constraint formula φ. However, we believe this would make XTALT-R too intimidating for human programmmers, who are generally not accustomed to writing machine-checkable proofs.

Therefore, as a concession to the human user, we plan to allow proofs of constraints to be elided; it will be the job of the XTALT-R assembler (which transforms its XTALT-R input into the LF representation that serves as a certificate) to reconstruct appropriate proofs when they are left out by the programmer. This means that the assembler will have to include a proof-generating constraint solver. Since the constraint-proving problems the assembler encounters are likely to be small, we believe a simple heuristic approach will suffice. If this turns out not to be the case, we will investigate middle-ground solutions, such as requiring the programmer to provide "hints"—but if at all possible we will stop short of requiring constraint proofs in XTALT-R programs.

3 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 [23]

So that we may formalize the process of resource-bound certifying compilation, this section presents a low-level typed language that will serve as the source of a translation into BTALT-R. Our intention is that this language, which we call Lilt[2], will serve as the intermediate language in a certifying compiler for the Popcorn language; the Lilt-to-TALT translation in Section 4.2 will form the back-end of that compiler.

Lilt is designed to be completely ignorant of resource bound 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 (probably) used in the majority of compilers for C-like languages, and described in many if not most traditional compiler design texts [27]: 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.[3] Unfortunately, the decision to adopt this compilation model complicates

[2]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.

[3]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

the intermediate language, since it introduces a distinction between local (*intra*procedural) and non-local (*inter*procedural) transfers of control, and forces us to deal with mutable local variables.

3.1 Syntax

The syntax of Lilt is given in Figure 8. (The static semantics is discussed in the next section.) 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; *labels* (ranged over by ℓ), which stand for code blocks within a function and are meaningful only inside that function, and *local variables* (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 [16] augmented with some 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 ($\lambda\alpha{:}k.c$)

(IA-32) has few registers and we do not plan to implement any interprocedural optimizations, so we will not discuss these matters any further.

$$
\begin{array}{llcl}
\textit{Operands} & v & ::= & s \mid n \mid \texttt{tt} \mid \texttt{ff} \mid \star \mid f \mid q@v \\
\textit{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] \\
\textit{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\} \\
\textit{Conditions} & cond & ::= & v_1 = v_2 \mid v_1 < v_2 \\
\\
\textit{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 \\
\\
\textit{Functions} & F & ::= & \texttt{func}(\Delta; \Gamma; \tau).(\texttt{enter}(s_1, \ldots, s_n).e, \ell_1 = B_1, \ldots, \ell_m = B_m) \\
\textit{Blocks} & B & ::= & \texttt{block}(\Delta; \Xi; \Gamma).e \mid \texttt{hndl}(\Delta; \Xi; \Gamma; s).e \\
\textit{Programs} & P & ::= & f_1 = F_1, \ldots, f_n = F_n \\
\\
\textit{Kinds} & k & ::= & T \mid k_1 \to k_2 \\
\textit{Type Constructors} & c, \tau & ::= & \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 \\
\textit{Type Contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}k \\
\textit{Block Types} & \gamma & ::= & \textit{lbl}(\Delta; \Xi; \Gamma) \mid \textit{hnd}(\Delta; \Xi; \Gamma) \\
\textit{Local Contexts} & \Gamma & ::= & [s_1{:}\tau_1, \ldots, s_n{:}\tau_n] \\
\textit{Exception Stack Types} & \Xi & ::= & \cdot \mid \Xi, \Gamma \\
\textit{Label Contexts} & \Lambda & ::= & \ell_1{:}\gamma_1, \ldots, \ell_n{:}\gamma_n \\
\textit{Function Contexts} & \Phi & ::= & f_1{:}\tau_1, \ldots, f_n{:}\tau_n
\end{array}
$$

Figure 8: Lilt Syntax

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
$\Phi; \Delta; \Gamma \vdash \mathit{cond}\ \mathsf{cond}$	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

Figure 9: Lilt typing judgment forms

and applied in the usual way ($c_1\, c_2$).

3.2 Static Semantics

The judgment forms of the Lilt type system are listed in Figure 9. The complete set of rules defining these judgments may be found in Appendix B; we 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:

- 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 storage locations (arguments and local variables) of the current function.
- 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:

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

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:

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

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 $\Delta \vdash \Xi$ $\mathsf{handles}$ Γ:

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

The auxiliary judgment $\Delta \vdash \Xi$ $\mathsf{handles}$ Γ (defined in Appendix B) 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 $\Delta \vdash \Xi$ $\mathsf{handles}$ Γ 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:

$$\frac{\Delta \vdash \tau = [\ldots, j{:}\tau_j, \ldots] \quad \Phi; \Delta; \Gamma \vdash v : \tau_j}{\Phi; \Delta; \Gamma \vdash \mathtt{inj}_\tau(j, v) : \tau}$$

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:

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

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 $\mathtt{outj}(v)$ 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 we have done, except that we 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 not particularly surprising either. 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 $\mathtt{pack}[\tau, c_1, \ldots, c_n]$, is similar:

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

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

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

Roughly speaking, Lilt uses coercions for operations whose TALT equivalents are subtyping rules rather than value forms or instructions. 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

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

Lilt

Figure 10: Lilt Example: Recursive Fibonacci

3.3 Lilt Examples

A very simple Lilt function, illustrating the use of local variables, is shown in Figure 10. 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 11, 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 12. This example uses the polymorphic type constructor *list*, defined as follows:

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

(Note that the type *list* τ 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* τ is simply the unrolling of the recursive type *list* τ. 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* α) and an application of the coercion $\mathtt{roll}_{\mathit{list}\ \alpha}$ 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.

```
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 11: 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:

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

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

Lilt

Figure 12: Lilt Example: List Reversal

4 Resource-Bound Certifying Compilation

4.1 Yield Placement

The major novel element in compiling Lilt to BTALT-R is, naturally, the placement of `yield` instructions so that the typing conditions regarding the virtual clock are satisfied. One possible strategy is to place a `yield` at the beginning of every basic block in the program; this idea, while sound, is not very appealing because we expect that yielding is very expensive. We will describe a number of simple yield placement heuristics in this section, 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 (as we expect it will be). Later on in this section, we 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 and which (we conjecture) will greatly reduce the number of actual yields performed. However, even these checks are not free, so we would like 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, we will structure the discussion as follows: first, we will explain some strategies for placing yield instructions with no dynamic checks; then, we will explain how dynamic checking is possible. The translation of Lilt to BTALT-R we give later will combine these ideas, using the placement strategies we discuss 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 into 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 of exception handling. We will discuss each of these subproblems of yield placement in turn.

4.1.1 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 likely to be tricky. For our prototype compiler, we desire a method of local yield placement that does not require complicated analysis of a function before code generation. We will describe three simple heuristics here; after discussing dynamic checks we will be able to formulate a fourth. The initial version of our compiler will implement one or two of these.

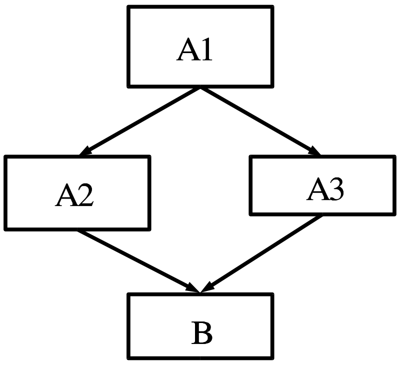

Figure 13: A Flow Graph With a Join

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 13, 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 the global yield placement strategy 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 discussed here, it is necessary to distinguish between *forward* and *backward* jumps. Specifically, we 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 our system (probably) lacks the expressive power to avoid yielding at least once per iteration, so we expect that our 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, we will simply use the order in which the blocks appear in the Lilt representation of the function.

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 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 we intend to use. We therefore do not expect to implement backward propagation.

4.1.2 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, we 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. We will describe these two strategies, which we call *call-return yielding* and *Feeley yielding*.

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 [10]. 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 $0 \le k_1 < Y$): 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 $0 \le k_2 < Y$), *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 we 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 (which we expect to be the case). 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.

4.1.3 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 we discussed in Section 2.2.3, code blocks in BTALT-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 we decide the type of a function should be

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

(the same type as the previous attempt at a function of cost k, except for the guard), then we 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, we 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) we 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 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.

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 [14]. Feeley, whose motivation was placing checkpoints in a program to detect interrupts, named his strategy *balanced polling*. (Feeley also inspired our use of the term 'call-return yielding'.) We choose to refer to the yielding scheme we have just described as *Feeley yielding*, and we follow

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

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

Figure 14: Fibonacci using Feeley Yielding

Feeley in using the letter E to denote the fixed value chosen for k. The major advantage of Feeley yielding is that functions (more accurately, loop-free leaf functions) 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 BTALT-R program fragment using the Feeley yielding strategy is shown in Figure 14. The function in the figure is a recursive function to compute Fibonacci numbers; it was hand-coded in BTALT-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).

4.1.4 Exceptional Placement

We believe it is sensible to adopt a simple heuristic 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 simply H plus the cost of raising the exception (probably 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 due to the added complexity it would introduce to the type aspects of compilation we do not plan to investigate it.

4.1.5 Clocks and Polling

The yield placement strategies discussed so far are straightforward and easy to implement, but (we predict) they fall far 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, we cannot expect that increasing it any more will continue to lower the actual frequency with which the program will yield under these strategies.

The next level of refinement is based on the following idea. Assume that $Y = M \cdot L$, where L is close to, but safely larger than, the length of most extended basic blocks in the program. Each yield period (of Y instructions) can then be thought of as M *minor yield periods* of L instructions each. If the language had a *minor yield* operation such that every M'th minor yield performs an ordinary yield (which we hereafter call a *major yield* for the sake of contrast), then a new sufficient condition for safety is that the program performs a *minor* yield every L instructions. Since L is

```
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−3‾), ck:‾Y−1‾ = L̄+(2̄+a′)
end:
     // a′:N, rck:S(a′), ck:L̄+(2̄+a′)
```

Figure 15: Code for a Minor Yield

much closer to the lengths of actual basic blocks than Y, each join point will introduce less waste; provided the cost of the remaining $M-1$ out of M minor yields is small enough compared to the one major yield, we believe programs will be more efficient this way.

Using the singleton and guarded types of TALT-R, minor yielding can actually be implemented *within the language* and does not need to be added as a new primitive. This is very important, because it means that different compilers targeting TALT-R, or human programmers working directly in TALT-R, are free to choose whether they wish to use a minor yielding strategy or not. If they do choose to use minor yields, they are still free to choose the value of L however they wish—the choice may differ between compilers or even between individual TALT-R programs. It also turns out that the yield placement strategies we have already discussed work just as well (in principle) for placing minor yields every L instructions as for placing major yields every Y instructions, but there are a couple of "tricks" one can do with the implementation of minor yields that are impossible with major yields. We will discuss these shortly.

First, however, we must explain how minor yields can be implemented in the TALT-R type system. The most obvious way to implement the intended behavior for the minor yield operation itself is probably to have a counter, stored in a register or global variable, representing the number of minor yields remaining before the next major yield. The counter is decremented for every minor yield; when the counter reaches zero, a major yield is performed and the counter is reset to M. We prefer, however, to have a register count down from Y to zero in increments of L; a minor yield that finds the counter less than L performs a major yield and resets it. Counting down L at a time instead of one at a time makes the arithmetic reasoning simpler—in particular, it means we do not need multiplication in TALT-R's constraint language—and permits some useful tricks which we will discuss later on. Now we are in a position to explain how minor yields work.

Clocks In what follows we will assume that a particular register is reserved for timing purposes. We 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 we 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.

Minor Yields A BTALT-R implementation of a minor yield is shown in Figure 15. 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 major 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 $\overline{L} + (\overline{2} + a)$. We will use the name `YIELD` to refer to this code sequence.

The Minor Clock In a register file Γ with $\Gamma(\texttt{rck}) = \mathcal{S}(t)$ and $\Gamma(\texttt{ck}) = (t' + (\overline{2} + t))$, we will say that t' is the value of the *minor clock*. Intuitively, t' 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:

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

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'. We conjecture that similar "minor clock rules" can be derived for all the instructions of BTALT-R except for `yield`. Furthermore, the typing annotations in Figure 15 suggest that (if one ignores the fact that it involves multiple blocks in BTALT-R), `YIELD` essentially acts like an instruction with a typing rule like the following:

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

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 we have mentioned, the fact that `YIELD` behaves so much like `yield` means that the local, global and exceptional placement strategies we have already 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, we call it a *polling strategy*. For example, recalling the type of a function under Feeley yielding,

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

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*:

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

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

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

fib:
    // $a, b_0 : \mathsf{N}$, rck:$\mathcal{S}(b_0)$,
    // ck : $(\overline{E} + a) + (\overline{2} + b_0)$, $(\overline{E} + a \;\leq\; \overline{L-1})$ true
    cmp eax,1
    ja L1   // $n \;\leq\; 1$?
    mov eax,1
    // ck : $(\overline{E-3} + a) + (\overline{2} + b_0)$
    ret   //   Return 1
L1:
    // ck : $(\overline{E-2} + a) + (\overline{2} + b_0)$
    push eax
    sub eax,1
    // ck : $(\overline{E-4} + a) + (\overline{2} + b_0)$
    YIELD
    // $b_1 : \mathsf{N}$, rck:$\mathcal{S}(b_1)$
    // ck : $\overline{L} + (\overline{2} + b_1)$
    call fib   // Compute fib(n-1)
    // $b_2 : \mathsf{N}$, rck:$\mathcal{S}(b_2)$
    // ck : $\overline{L-E-1} + (\overline{2} + b_2)$
    pop ecx
    push eax
    mov eax,ecx
    sub eax,2
    // ck : $\overline{L-E-5} + (\overline{2} + b_2)$
    call fib   // Compute fib(n-2)
    // $b_3 : \mathsf{N}$, rck:$\mathcal{S}(b_3)$
    // ck : $\overline{L-2E-6} + (\overline{2} + b_3)$
    pop ecx
    add eax,ecx   // eax := fib(n-1)+fib(n-2)
    // ck : $\overline{L-2E-8} + (\overline{2} + b_3)$
    YIELD
    // $b_4 : \mathsf{N}$, rck:$\mathcal{S}(b_4)$
    // ck : $\overline{L} + (\overline{2} + b_4)$
    ret   // Return

Figure 16: 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 + 2)‾ true,
    //                ck:(F̄ + a) = F̄ + (L − F + 2)‾ + a′ = L̄ + (2̄ + a′)
    // otherwise: rck:int, ck:F̄ + a
    yield
    // ck:Ȳ
    mov rck,(Y-L-3)
    // a′  ↦  (Y − L − 3)‾; rck: S((Y − L − 3)‾), ck:(Y − 1)‾  =  L̄ + (2̄ + a′)
end:
    // a′:N, rck:S(a′), ck:L̄ + (2̄ + a′)
```

Figure 17: 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 + 2)‾ true,
    //                ck:(F̄ + a) = F̄ + (R − F + 2)‾ + a′ = R̄ + (2̄ + a′)
    // otherwise: rck:int, ck:F̄ + a
    yield
    // ck:Ȳ
    mov rck,(Y-F-3)
    // a′  ↦  (Y − R − 3)‾; rck: S((Y − R − 3)‾), ck:(Y − 1)‾  =  R̄ + (2̄ + a′)
end:
    // a′:N, rck:S(a′), ck:R̄ + (2̄ + a′)
```

Figure 18: Resetting the Clock from F to R

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 16 shows the Fibonacci function from Figure 14 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.

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 17 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 $\texttt{YIELD}(F, R)$ defined in Figure 18. Note that the first two forms of minor yield are really special cases of this last one: $\texttt{YIELD}(F)$ is simply $\texttt{YIELD}(F, L)$, and the `YIELD` from Figure 15 is $\texttt{YIELD}(0, L)$. 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 we may 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. It would be interesting to investigate the trade-off between precision and time spent on polling.

4.2 Compilation of Lilt

In this section, we will finally give a formal translation from Lilt to BTALT-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 our 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 us to argue that the type system we propose for BTALT-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 we described in Section 4.1.5 is flexible enough that resource bound certification need not get in the programmer's way.

The translation we give here uses Feeley polling for global 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 BTALT-R code. An actual implementation of this translation, such as the one we plan to create, would resolve the nondeterminism using a heuristic such as the ones we have already described.

Although the implications of polling are the main point of this paper, the formal translation we give in this section addresses *all* aspects of type-directed compilation of Lilt. In particular, we give a complete translation from Lilt types to TALT-R types, and we 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, we must take some time to introduce some conventions and notation.

4.2.1 Type-Directedness

Formal translations between languages generally come in two flavors: syntax-directed and type-directed. 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 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 (we conjecture) be translated to BTALT-R by a syntax-directed translation. This is so because BTALT-R (as opposed to XTALT-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 we give in this section is supposed to be the basis for the one we plan to implement, and that implementation must target XTALT-R, not BTALT-R; because of the explicit typing annotations (and coercions) needed in XTALT-R, our actual Lilt compiler must be type-directed. Therefore, we will give a type-directed translation in this paper even though doing so renders the presentation a good deal less concise. We will use the context and typing information available in the setting of a type-directed translation to annotate the BTALT-R output with typing information for labels, even though such annotations are not officially part of BTALT-R. This will hopefully help make the intended meaning of the generated code more clear.

4.2.2 Variable Naming

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

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

Second, we 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 we 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 we will show shortly how the TALT operand 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 we will write

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

instead of

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

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

Figure 19: Translation of kinds and types (except function types)

when we define the translation.

4.2.3 Types and Data Representation

The translation of Lilt kinds and type constructors is defined in Figures 19 and 20. The translation of kinds is nearly trivial; the only point of interest is that the Lilt kind T is translated as $\mathsf{T4}$, which 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, our translation will not require any run-time type constructor analysis (as in [11, 9, 33]) 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

$$
\begin{array}{rcl}
|(\tau_1, \ldots, \tau_m) \to \tau| & = & \forall \rho_1{:}\mathsf{TD}.\forall \rho_2{:}\mathsf{TD}.\forall \alpha_f{:}\mathsf{T4}.\forall \alpha_h{:}\mathsf{T4}.\forall a{:}\mathsf{N}.\forall b{:}\mathsf{N}.\ (\overline{E} + a \leq \overline{L-1}) \Rightarrow \\
& & \quad \{\mathtt{ebx}{:}\mathtt{got}, \mathtt{esi}{:}\mathcal{S}(b), \mathtt{edi}{:}\tau_e, \mathtt{ebp}{:}\alpha_f, \mathtt{esp}{:}\tau_r \times \sigma_0, \\
& & \quad\ \ \mathtt{ck}{:}(\overline{E} + a) + (\overline{2} + b)\} \to 0
\end{array}
$$

$$
\begin{array}{ll}
\textit{where:} & \sigma_0 = |\tau_1| \times \cdots \times |\tau_m| \times \rho_1 \times \tau_h \times \rho_2 \\
& \tau_h = \alpha_h \wedge \forall b'{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau_{\text{exn}}|, \mathtt{ebx}{:}\mathtt{got}, \mathtt{esi}{:}\mathcal{S}(b'), \mathtt{esp}{:}\rho_2, \mathtt{ck}{:}\overline{H} + (\overline{2} + b')\} \to 0 \\
& \tau_e = \mathtt{sptr}(\tau_h \times \rho_2) \\
& \tau_r = \forall b''{:}\mathsf{N}.\{\mathtt{eax}{:}|\tau|, \mathtt{ebx}{:}\mathtt{got}, \mathtt{esi}{:}\mathcal{S}(b''), \mathtt{edi}{:}\tau_e, \mathtt{ebp}{:}\alpha_f, \mathtt{esp}{:}\sigma_0, \\
& \qquad\qquad \mathtt{ck}{:}a + (\overline{2} + b)\} \to 0
\end{array}
$$

Figure 20: Translation of function types

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, but also the (interprocedural) exception handling mechanism. (Our treatment of exception handling is very similar to that of the TALx86 Popcorn compiler [24], which in turn appears to be based on the canonical translation into STAL [25].) As the translation in Figure 20 indicates, 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 actual type of the exception handler is a bit more complicated, however, because the stack type expected by the handler may in general be a *supertype* of ρ_2. The function that installed the handler might rely on this fact, so the more precise type of the handler must be tracked through all function calls. The usual way to do this sort of thing is with bounded quantification; rather than add this feature to TALT-R we use a known trick for simulating it using ordinary universal quantification and intersection types [34, 5]. Intuitively, the parameter α_h is the "real" type 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.

The translation of function types also reveals that the register `ebp` is treated as callee-saves: The function is polymorphic in the initial type of `ebp` and the type of the return address requires that a value of the same type be in `ebp` when the function returns. The register `ebx` contains the *global offset table* pointer; this special value contains the addresses of the functions provided by the runtime system, and must be provided to the `malloc` instruction. There is no particular reason (other than convention) why the GOT pointer must stay in `ebx`; however for simplicity our compiler will always leave it there. Every code type in the translation will specify the type `got` for `ebx`.

Finally, observe that the translation of function types assumes a dynamic polling discipline for yielding, as described in Section 4.1.5. The clock register is `esi`; a function expects this register to have a singleton type $\mathcal{S}(b)$, where b is a static term parameter. 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 $\overline{E} + a$, 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. Note, though, that just like in our earlier discussion of polling, the return address and exception handler must not care about the exact value of the clock register.

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

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

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

Figure 21: Clock Specifiers

4.2.4 Clock Specifiers

In BTALT-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 21 introduces *clock specifiers*, which are a more abstract way of describing the minor clock. The clock specifier just n corresponds to n on the minor clock; 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 numbers L and E in the definition of the ordering are parameters of the translation: L is the *minor yield period*, and E is cost assumed by the Feeley yielding strategy for every function. A third parameter, not appearing in the figure, is H, the minor clock requirement of every exception handler.

4.2.5 Stacks, Register Files and Labels

In order to give typing annotations for the labels in the output of our translation, we 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 our translation is type-preserving, we must be able to specify the types we intend for the register file and stack at any point in the BTALT-R program we 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 22. 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 set up to point to the bottom of the stack frame instead of into the middle as is more customary. This is because we wish

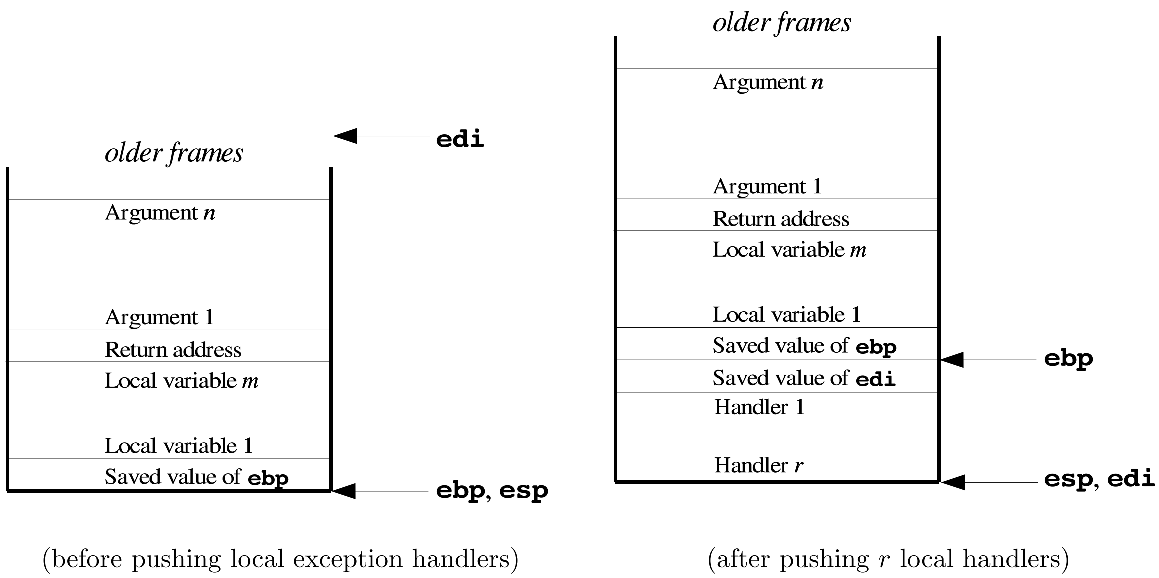

Figure 22: A Lilt function's stack frame

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 we 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 22 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`.

Since we are using a polling strategy for yielding, all the code pointers in the BTALT-R output of our translation must make assumptions about the minor clock that are reflected in their types. To write these types, we 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 BTALT-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; $\Gamma[\mathtt{mck}_u \mapsto t]$ 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. We 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.

The type of the stack at any point in a Lilt program can be determined using the function ST in Figure 23. Intuitively, $ST_{\rho_1,\rho_2,\alpha_f,\alpha_h,a}(\Xi, \Gamma, \tau)$ is the type of the stack type corresponding to a

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

Figure 23: Determining the Stack Type

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

Figure 24: Determining the Register File Type

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

$$
\begin{array}{rcl}
|\mathit{lbl}(\Delta'; \Xi; \Gamma)|_{\Delta,\tau,\kappa} & = & LL((\Delta, \Delta'); \Xi; \Gamma, \tau, \kappa, [\,]) \\[2ex]
|\mathit{hnd}(\Delta'; \Xi; \Gamma)|_{\Delta,\tau,\kappa} & = & \forall \alpha_1{:}|k_1|. \ldots \forall \alpha_m{:}|k_m|. \forall \rho_1{:}\mathsf{TD}. \forall \rho_2{:}\mathsf{TD}. \forall \alpha_f{:}\mathsf{T4}. \forall \alpha_h{:}\mathsf{T4}. \\
& & \forall a{:}\mathsf{N}. \forall b{:}\mathsf{N}.\ (\overline{E} + a \leq \overline{L-1}) \Rightarrow \\
& & \quad \{\mathtt{eax}{:}|\tau_{\mathrm{exn}}|, \mathtt{ebx}{:}\mathtt{got}, \mathtt{esp}{:}ST_{\rho_1,\rho_2,\alpha_f,\alpha_h,a}(\Xi, \Gamma, \tau), \mathtt{mck}_b{:}\overline{H}\} \to 0 \\
\text{where} & & (\Delta, \Delta') = \alpha_1{:}k_1, \ldots, \alpha_m{:}k_m
\end{array}
$$

Figure 25: Label and Block Types

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 24 shows how to find the types of the registers for any point in a compiled Lilt program, and the type of any local label occurring inside the BTALT-R version of a Lilt function. 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 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 25 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`, the global offset table pointer in `ebx`, and H on the minor clock.

4.2.6 Compiling Expressions

Because of our assumptions about the names of local storage locations, if the total number L of local variables allocated by the current function is known then the operand corresponding to location s

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

Figure 26: Translation Contexts

(denoted by $|s|_L$) can be determined from the name s as follows:

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

In the BTALT-R syntax used in this document, 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 we will write $|s|_L^d$.

We 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:

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

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

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

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

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, we collect all the context information for a Lilt expression into one *translation context*, ranged over by $\mathcal{C}$ as shown in Figure 26. 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 $\mathrm{dom}(\Gamma) = \{\texttt{arg}(1), \ldots, \texttt{arg}(A), \texttt{loc}(1), \ldots, \texttt{loc}(B)\}$ (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 $\Phi; \Delta; \Gamma \vdash v : \tau'$
- $\mathcal{C} \vDash \Gamma'$ iff $\Delta \vdash \Gamma \leq \Gamma'$
- $\mathcal{C} \vDash \Xi'$ iff $\Delta \vdash \Xi \leq \Xi'$
- $\mathcal{C} \vDash \mathsf{canraise}$ iff $\Delta \vdash \Xi\ \mathsf{handles}\ \Gamma$

The complete translation rules are in Section 4.2.7. 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:

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

Note that this rule takes advantage of the clock register "tricks" discussed earlier, 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, or it may perform some analysis to determine good values for m at each minor yield it generates.

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

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

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 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:

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

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:

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

(Note, though, that a simple addition takes three BTALT-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 in this paper ignores these issues, but there is no reason a real compiler cannot be designed to deal with wide tuples and high-arity functions.

4.2.7 Complete Translation Rules

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

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
$\text{dom}(\Gamma_i) = \text{dom}(\Gamma)$ for each i

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

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

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

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

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

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

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

$$\frac{\mathcal{C} \vdash v : (\tau_1' \ldots, \tau_n') \to \tau'' \quad \mathcal{C} \vDash \textsf{canraise} \quad \mathcal{C} \vdash v_i : \tau_i' \text{ for } 1 \le i \le n \quad \mathcal{C}[s \mapsto \tau'']; \mathcal{T}; (\kappa - (n+3+E)) \vdash e \mapsto I\, S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let } s = v(v_1, \ldots, v_n) \texttt{ in } e \rightsquigarrow \\ \quad \begin{array}{l} \quad \texttt{push } |v_n|_{\mathcal{C}} \\ \quad \vdots \\ \quad \texttt{push } |v_1|_{\mathcal{C}} \\ \quad \texttt{call } |v|_{\mathcal{C}} \\ \quad \texttt{mov } |s|^d_{\mathcal{C}}, \texttt{eax} \\ \quad \texttt{sfree } 4n \\ \quad I \\ S \end{array}\end{array}}$$

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

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

$$
\frac{\begin{array}{c}\mathcal{C} \vdash v : [\overline{j{:}\tau}, i{:}\tau', \overline{j{:}\tau}'] \\ \mathcal{C}[s \mapsto [i{:}\tau']]; \mathcal{T}; (\kappa - 4) \vdash e_1 \rightsquigarrow I_1\, S_1 \quad \mathcal{C}[s \mapsto [\overline{j{:}\tau}, \overline{j{:}\tau}']]; \mathcal{T}; (\kappa - 4) \vdash e_2 \rightsquigarrow I_2\, S_2\end{array}}{\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{case}\ v\ \texttt{of}\ \texttt{inj}(i, s) \Rightarrow e_1\ \texttt{else}\ e_2 \rightsquigarrow}
$$

$$
\begin{array}{l}
\quad \texttt{mov eax}, |v|_{\mathcal{C}} \\
\quad \texttt{cmpje}\ [\texttt{eax}], i, \ell_{match} \\
\quad \texttt{mov}\ |s|^{d}_{\mathcal{C}}, \texttt{eax} \\
\quad I_2 \\
S_2 \\
\ell_{match} : LL(\mathcal{C}, [\texttt{eax} \mapsto |[i{:}\tau']|]) = \\
\quad \texttt{mov}\ |s|^{d}_{\mathcal{C}}, \texttt{eax} \\
\quad I_1 \\
S_1
\end{array}
$$

$$
\frac{\begin{array}{c}\mathcal{C} \vdash v_i : \texttt{int}\ \text{for}\ i = 1, 2 \\ \mathcal{C}; \mathcal{T}; (\kappa - 3) \vdash e_1 \rightsquigarrow I_1\, S_1 \quad \mathcal{C}; \mathcal{T}; (\kappa - 3) \vdash e_2 \rightsquigarrow I_2\, S_2\end{array}}{\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{if}\ v_1 = v_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow}
$$

$$
\begin{array}{l}
\quad \texttt{mov eax}, |v_1|_{\mathcal{C}} \\
\quad \texttt{cmp eax}, |v_2|_{\mathcal{C}} \\
\quad \texttt{jne}\ \ell_{else} \\
\quad I_1 \\
S_1 \\
\ell_{else} : LL(\mathcal{C}, [\,]) = \\
\quad I_2 \\
S_2
\end{array}
$$

$$
\frac{\begin{array}{c}\mathcal{C} \vdash v : \langle \tau_0, \ldots, \tau_m \rangle \\ \mathcal{C} \vdash v : \tau_i \quad \mathcal{C}; \mathcal{T}; \kappa - 3 \vdash e \rightsquigarrow I\, S\end{array}}{\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ \pi_i\, v := v'\ \texttt{in}\ e \rightsquigarrow}
$$

$$
\begin{array}{l}
\quad \texttt{mov eax}, |v|_{\mathcal{C}} \\
\quad \texttt{mov ecx}, |v'|_{\mathcal{C}} \\
\quad \texttt{mov}\ [\texttt{eax} + 4i], \texttt{ecx} \\
\quad I \\
S
\end{array}
$$

$$
\frac{\begin{array}{c}\mathcal{C} \vdash v_i : \texttt{int}\ \text{for}\ i = 1, 2 \\ \mathcal{C}; \mathcal{T}; \kappa - 3 \vdash e_1 \rightsquigarrow I_1\, S_1 \quad \mathcal{C}; \mathcal{T}; \kappa - 3 \vdash e_2 \rightsquigarrow I_2\, S_2\end{array}}{\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{if}\ v_1 < v_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow}
$$

$$
\begin{array}{l}
\quad \texttt{mov eax}, |v_1|_{\mathcal{C}} \\
\quad \texttt{cmp eax}, |v_2|_{\mathcal{C}} \\
\quad \texttt{ja}\ \ell_{else} \\
\quad I_1 \\
S_1 \\
\ell_{else} : LL(\mathcal{C}, [\,]) = \\
\quad I_2 \\
S_2
\end{array}
$$

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

$$
\begin{array}{l}
\quad \texttt{mov eax}, |v|_{\mathcal{C}} \\
\quad \texttt{mov}\ |s|^{d}_{\mathcal{C}}, \texttt{eax} \\
\quad I \\
S
\end{array}
$$

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\mathcal{C} \vdash \tau' = [\ldots, j{:}\tau_j, \ldots] : T \\ \mathcal{C} \vdash v : \tau_j \quad \mathcal{C}[s \mapsto \tau']; \mathcal{T}; (\kappa - 5) \vdash e \leadsto I\, S\end{array}}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = \texttt{inj}'_{\tau}(j, v)\ \texttt{in}\ e \leadsto \\ \quad\quad \begin{array}{l} \texttt{push}\ |v|_{\mathcal{C}} \\ \texttt{malloc eax, ebx}, 8 \\ \texttt{mov}\ [\texttt{eax}], j \\ \texttt{pop}\ [\texttt{eax} + 4] \\ \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ I \end{array} \\ \quad S \end{array}}
\qquad
\frac{\mathcal{C} \vdash v : [i{:}\tau'] \quad \mathcal{C}[s \mapsto \tau']; \mathcal{T}; (\kappa - 3) \vdash e \leadsto I\, S}{\begin{array}{l}\mathcal{C}; \mathcal{T}; \kappa \vdash \texttt{let}\ s = \texttt{outj}(v)\ \texttt{in}\ e \leadsto \\ \quad\quad \begin{array}{l} \texttt{mov eax}, |v|_{\mathcal{C}} \\ \texttt{mov eax}, [\texttt{eax} + 4] \\ \texttt{mov}\ |s|^d_{\mathcal{C}}, \texttt{eax} \\ I \end{array} \\ \quad S \end{array}}$$

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

5 Conclusion

5.1 Related Work

The idea of a certified code format that counts instructions to bound running time is not new. Necula and Lee [30] proposed just this idea, and Crary and Weirich [10] based the languages LXres and TALres on it. Our work is largely inspired by their techniques; however, to the best of our knowledge, neither of these previous efforts resulted in an implementation of sufficient generality to support real applications. Crary and Weirich's work in particular suffers from the fact that one must specify the running time of a function in terms of its input, all in a theory whose complexity must be carefully controlled to keep typing decidable. We avoid this difficulty by changing the problem: since we are concerned with bounding the time between yields, rather than the total running time of functions or programs, we do not need to reason about the dependency of a function's cost on its arguments. On the other hand, some optimization of yield placement might be enabled if the TALT-R type theory were extended along the lines of LXres so that function costs could be more precisely described.

The integration of arithmetic or logical reasoning into a type system is not new, either. Xi and Harper's DTAL [43] is a typed assembly language in which *dependent types*, singleton types, and constraints are used to track knowledge of integer values. As an example, they show how a typed assembly language can safely support unchecked array primitives—essentially by forcing the programmer to perform bounds checks explicity when they are needed. The array operations in TALT are more or less based on DTAL. Other type systems that allow the integration of logical reasoning include LTT [8], in which function preconditions and proofs that they hold are represented via the Linear LF type theory [3, 39], and a system described by Shao *et al.* [37] which accomplishes a similar feat using the Calculus of Inductive Constructions [32].

Speaking more broadly, the entire issue of "resource bounds" is essentially the problem of ensuring that a program cooperates with other software executing concurrently on the same computer. In particular, the idea of requiring a program to "yield" is fundamental to multitasking or multithreading in any setting where preemptive scheduling is not a possibility. This problem is probably as old as the very notion of an operating system, and has been well studied over the years. Of course, we are interested in producing *foundational proofs* that programs cooperate; in this respect our work differs from much of the systems literature. For example, the "engine" abstraction in Scheme captures the notion of a computation that is allowed to run for a specific amount of time, which may or may not be enough for it to finish. (For a good introduction to engines we refer the reader to Dybvig's book on Scheme [12].) Haynes and Friedman [19] have shown that the engine abstraction may be used to implement user-level threads; Dybvig and Hieb [13] have shown that engines may in turn be implemented using `call/cc` and a timer interrupt. However, none of this work explains how to *guarantee* that an engine is stopped at the end of its allotted time, unless the operating system can be counted on to deliver an asynchronous interrupt at the right moment. (The implementation of engines using a timer is intended to work regardless of how the timer is implemented; Dybvig and Hieb suggest using an explicit counter if true preemption is not available.) Our TALT-R type theory provides some insight for how such guarantees may be achieved.

Many security properties can be specified using *security automata* [35]. Briefly, a security automaton has a set of states, one of which is designated as the initial state and another of which is the "bad" state; transitions between states are labeled by actions the program might perform, and there is no transition from the bad state to any other state. Any such automaton defines a security policy, namely the one in which a sequence of actions is permissible iff it does not lead from the initial state to the bad state. Schneider [35] describes a safety mechanism called *execution monitoring*, in which the actions of a program are observed at run time and the corresponding transitions of some security automaton are simulated; if the automaton ever enters the bad state, the program is terminated. Schneider argues that only *safety properties* can be enforced in this way, and points out that liveness is not a safety property. However, properties such as liveness can be conservatively approximated by specifying a "maximum waiting time"; in particular, our safety requirement that any TALT-R program must yield after at most Y instructions can be seen as an approximation of the policy that any program must *eventually* yield—although for the purpose of bounding CPU usage, it is important for us to have a specific upper bound on latency. Schneider attributes the idea of a maximum waiting time to Gligor [17].

Based on the idea of execution monitoring, Walker [40] developed a type system in which conformance to the policy defined by a security automaton can be certified. He also exhibited a program transformation that automatically instruments code with safety checks, and shows that the output of this transformation is well-formed according to his type system. Walker's type system, like ours, is inspired by the dependent refinement types of DML and DTAL; he uses singleton types and provides the means to integrate knowledge of the safety policy into the type system.

Thiemann [38] has proposed an implementation of execution monitoring based on partial evaluation, in which the instrumented version of a program is produced by specializing an instrumented interpreter to the untrusted program. This approach is general, in the sense that it can handle a wide variety of forms of instrumentation, but it does not produce certified output. Instead, Thiemann claims that the partial evaluation algorithm is simple enough that the entire instrumentation process can occur within the trusted computing base. This point of view is essentially incompatible with our commitment to foundational certified code.

The interrupt calculus of Palsberg and Ma [31] is a type system for interrupt-driven programs that ensures bounded stack usage. Recently, Naik [28] gave a variant of the interrupt calculus that is capable of ensuring interrupts will always be handled within a certain amount of time after they occur. As in TALT-R, singleton types play a key role; Naik combines them with extensive use of intersection and union types to give statements and interrupt handlers types that precisely capture the possible state transitions of the program. As a result, a program is typable whenever it passes a certain model-checking analysis. This means that model checking can be used to perform type inference. Naik argues that this relationship between type checking and model checking is useful, since model checking systems are good at explaining failures (*i.e.,* by providing counterexamples), while type checking is better at explaining successes (because type annotations provide useful information for understanding a well-typed program).

Liblit *et al.* [21] have implemented a system that randomly samples program behavior for the purpose of detecting bugs. Like TALT-R, their technique is based on forcing a program to do something with a certain frequency, and they make similar use of a dynamic counter to determine when a "sample" must occur. There are two major differences from our work. First, unlike our virtual clock which is decremented for every instruction, their counter only tracks the number of designated sampling points that are encountered. Second, in order for their sampling to have the desired statistical properties, their counter must be precise; our virtual clock is merely an upper bound on the time to the next yield. It is possible that a type system similar to ours could allow the sampling behavior of a system like that of Liblit *et al.* to be certified correct; however, the statistical properties of data collection are usually not safety-critical, so it is unclear whether there is any incentive to do this.

Hofmann [20] has presented a language in which any definable function may be computed in polynomial time. This seems superficially related to our goal of bounding CPU usage, but is really quite different. One difference is that Hofmann's complexity bounds apply to entire functions, whereas in TALT-R a program may run for arbitrarily long provided it yields often enough. Another difference is that the time between yields in TALT-R is bounded by a fixed constant, while Hofmann's results "bound" running time only by restricting to a certain computational complexity class, meaning that a function in Hofmann's language can actually take an arbitrarily long time if given a large enough input. Finally, Hofmann's results are for a fairly high-level language; it is not at all clear how well they could be extended to provide foundational proofs of similar guarantees for programs at the assembly language level.

5.2 Continuing and Future Work

As mentioned earlier, We are currently undertaking a complete end-to-end implementation of code certification based on the TALT-R type theory described in this report. Our implementation, from certifying Popcorn compiler to verifying loader and runtime system, is intended to be suitable for use in the ConCert grid computing framework. As a result, the ConCert node implementation will be able to monitor and regulate foreign code easily, expanding the options available to host owners for allocation of their system resources.

Of course, there are a number of other possible applications for TALT-R and for the intuitions motivating its design. The relatively impoverished environments and lightweight operating systems of handheld devices and smart cards, for example, make resource bound certification seem particularly useful in these domains. Certification of resources other than CPU time is also a possibility. Heap allocation in a garbage-collected setting, for example, can be treated in much the same way as we have viewed time in this report: in place of the "yield" operation that must happen after at most Y instructions, one has an instruction that calls the garbage collector and a limit on how much space can be allocated before the collector must run again. Another possible direction is to change the rules of the type system so that certain special instructions are executed with *at least* a certain number of other instructions in between (that is, providing lower rather than upper bounds on certain latencies). This could give a useful mechanism for bandwidth limiting, applicable to either local disk or network access. Many of these possibilities will be considered in future work.

References

[1] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, June 2001.

[2] 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.

[3] Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Eleventh IEEE Symposium on Logic in Computer Science*, pages 264–275, July 1996.

[4] The ConCert project home page. `http://www.cs.cmu.edu/~concert/`.

[5] Karl Crary. Typed compilation of inclusive subtyping. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, MA, 2000.

[11] 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.

[12] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[13] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[14] 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.

[15] Folding@home. `http://folding.stanford.edu`.

[16] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[17] Virgil D. Gligor. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering*, SE-10(3):320–324, May 1984.

[18] 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.

[19] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):102–121, 1987.

[20] 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.

[21] Ben Liblit, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Bug isolation via remote program sampling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[23] *Merriam-Webster's Collegiate Dictionary*. Merriam-Webster, Springfield, MA, tenth edition, 1994.

[24] 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.

[25] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[26] 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.

[27] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[28] Mayur Naik. A type system equivalent to model checking. Master's thesis, Purdue University, 2003.

[29] George Necula. Proof-carrying code. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.

[30] 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.

[31] Jens Palsberg and Di Ma. A typed interrupt calculus. In *Seventh International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 291–310, Oldenburg, Germany, September 2002.

[32] 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.

[33] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University, 2000.

[34] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1991.

[35] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[36] SETI@home. `http://setiathome.ssl.berkeley.edu`.

[37] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Proceedings of the Twenty-Ninth ACM Symposium on Principles of Programming Languages*, pages 217–232, Portland, OR, January 2002.

[38] Peter Thiemann. Program specialization for execution monitoring. *Journal of Functional Programming*, 13(3):573–600, May 2003.

[39] Joseph C. Vanderwaart and Karl Crary. A simplified account of the metatheory of Linear LF. In Frank Pfenning, editor, *Proceedings of the Third International Workshop on Logical Frameworks and Meta-Languages (LFM)*, volume 70, issue 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[40] David Walker. A type system for expressive security policies. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, MA, 2000.

[41] 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.

[42] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[43] 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.

[44] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the Twenty-Sixth ACM Symposium on Principles of Programming Languages*, 1999.

A Rules for BTALT-R

$\boxed{\Delta \vdash c : K}$

$$\frac{((\alpha{:}K) \in \Delta)}{\Delta \vdash \alpha : K} \qquad \frac{}{\Delta \vdash \mathtt{ns} : \mathsf{T1}} \qquad \frac{}{\Delta \vdash \mathtt{B0} : \mathsf{T0}} \qquad \frac{}{\Delta \vdash \mathtt{B1} : \mathsf{T1}} \qquad \frac{}{\Delta \vdash B : \mathsf{Word}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{T}}$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{TD} \quad \Delta \vdash \tau_2 : \mathsf{TD}}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{TD}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}j}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{T}(i+j)} \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathtt{box}(\tau) : \mathsf{T}W} \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathtt{mbox}(\tau) : \mathsf{T}W}$$

$$\frac{\Delta \vdash \tau : \mathsf{TD}}{\Delta \vdash \mathtt{sptr}(\tau) : \mathsf{T}W} \qquad \frac{\Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash x : \mathsf{Word}}{\Delta \vdash \tau \upharpoonright x : \mathsf{T}} \qquad \frac{\Delta \vdash \tau : \mathsf{TD} \quad \Delta \vdash x : \mathsf{Word}}{\Delta \vdash \tau \upharpoonright x : \mathsf{TD}} \qquad \frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau \upharpoonright B : \mathsf{T}(i \cdot B)}$$

$$\frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \tau \upharpoonright 0 : \mathsf{T0}} \qquad \frac{\Delta \vdash \Gamma}{\Delta \vdash \Gamma \to 0 : \mathsf{T}W} \qquad \frac{\Delta \vdash x : \mathsf{Word}}{\Delta \vdash \mathtt{set}_{=}(x) : \mathsf{T}W} \qquad \frac{\Delta \vdash x : \mathsf{Word}}{\Delta \vdash \mathtt{set}_{<}(x) : \mathsf{T}W} \qquad \frac{\Delta \vdash x : \mathsf{Word}}{\Delta \vdash \mathtt{set}_{>}(x) : \mathsf{T}W}$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T}}{\Delta \vdash \forall \alpha{:}K.\tau : \mathsf{T}} \qquad \frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T} \quad \Delta \vdash c : K \quad \Delta \vdash \tau[c/\alpha] : K'}{\Delta \vdash \forall \alpha{:}K.\tau : K'} \; (K' \in \{\mathsf{TD}, \mathsf{T}i\}) \qquad \frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T}}{\Delta \vdash \exists \alpha{:}K.\tau : \mathsf{T}}$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T}i}{\Delta \vdash \exists \alpha{:}K.\tau : \mathsf{T}i} \qquad \frac{}{\Delta \vdash \mathtt{void} : \mathsf{T}i} \qquad \frac{\Delta, \alpha{:}\mathsf{T} \vdash \tau : \mathsf{T}}{\Delta \vdash \mu\alpha.\tau : \mathsf{T}} \qquad \frac{\Delta, \alpha{:}\mathsf{T} \vdash \tau : \mathsf{T} \quad \Delta \vdash \tau[\mu\alpha.\tau/\alpha] : K}{\Delta \vdash \mu\alpha.\tau : K} \; (K \in \{\mathsf{TD}, \mathsf{T}i\})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{T}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{TD} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{TD}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{TD}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{TD}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{T}i}$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}i}{\Delta \vdash \tau_1 \wedge \tau_2 : \mathsf{T}i} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \vee \tau_2 : \mathsf{T}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}i}{\Delta \vdash \tau_1 \vee \tau_2 : \mathsf{T}i} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \mathcal{S}(t) : \mathsf{T}W}$$

$$\frac{\Delta \vdash \varphi : \mathsf{P} \quad \Delta \vdash \tau : K}{\Delta \vdash \varphi \Rightarrow \tau : K} \; (K \in \{\mathsf{T}, \mathsf{T}i, \mathsf{TD}\}) \qquad \frac{}{\Delta \vdash \overline{n} : \mathsf{N}} \; (n \geq 0) \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 + t_2 : \mathsf{N}}$$

$$\frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 \leq t_2 : \mathsf{P}} \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 = t_2 : \mathsf{P}} \qquad \frac{\Delta \vdash \tau : \mathsf{TD}}{\Delta \vdash \tau : \mathsf{T}} \qquad \frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau : \mathsf{TD}}$$

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

$$\frac{\begin{array}{c}\Delta \vdash \tau : \mathsf{TD} \quad \Delta \vdash t : \mathsf{N} \\ \Delta \vdash \tau_i : \mathsf{T}W \text{ for } 1 \leq i \leq N\end{array}}{\Delta \vdash \{\mathtt{r1}{:}\tau_1, \ldots, \mathtt{rN}{:}\tau_N, \mathtt{sp}{:}\tau, \mathsf{ck}{:}t\}}$$

$\boxed{\Delta \vdash \varphi \ \mathsf{true}}$

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

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

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

$$\frac{\Delta \vdash t_1 = t_2 \ \mathsf{true}}{\Delta \vdash t_1 \leq t_2 \ \mathsf{true}} \qquad \frac{\Delta \vdash t_1 \leq t_3 \ \mathsf{true} \quad \Delta \vdash t_3 \leq t_2 \ \mathsf{true}}{\Delta \vdash t_1 \leq t_2 \ \mathsf{true}} \qquad \frac{}{\Delta \vdash \overline{m} \leq \overline{n} \ \mathsf{true}} \ (m \leq n)$$

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

$\boxed{\Delta \vdash \tau_1 \leq \tau_2 \quad \Delta \vdash \Gamma \leq \Gamma'}$

$$\frac{}{\Delta \vdash \tau \leq \tau} \qquad \frac{\Delta \vdash \tau_1 \leq \tau_3 \quad \Delta \vdash \tau_3 \leq \tau_2}{\Delta \vdash \tau_1 \leq \tau_2} \qquad \frac{\Delta \vdash \tau_1 \leq \tau_1' \quad \Delta \vdash \tau_2 \leq \tau_2'}{\Delta \vdash \tau_1 \times \tau_1 \leq \tau_1' \leq \tau_2'} \qquad \frac{\Delta \vdash \tau \leq \tau'}{\Delta \vdash \tau \uparrow x \leq \tau' \uparrow x}$$

$$\frac{\Delta \vdash \Gamma' \leq \Gamma}{\Delta \vdash \Gamma \to 0 \leq \Gamma' \to 0} \qquad \frac{\Delta \vdash \tau \leq \tau'}{\Delta \vdash \texttt{box}(\tau) \leq \texttt{box}(\tau')} \qquad \frac{\Delta \vdash \tau \leq \tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta \vdash \texttt{mbox}(\tau) \leq \texttt{mbox}(\tau')} \qquad \frac{}{\Delta \vdash \texttt{mbox}(\tau) \leq \texttt{box}(\tau)}$$

$$\frac{\Delta \vdash \tau \leq \tau' \quad \Delta \vdash \tau : \mathsf{TD} \quad \Delta \vdash \tau' : \mathsf{TD}}{\Delta \vdash \texttt{sptr}(\tau) \leq \texttt{sptr}(\tau')} \qquad \frac{\Delta, \alpha{:}K \vdash \tau \leq \tau'}{\Delta \vdash \forall\alpha{:}K.\tau \leq \forall\alpha{:}K.\tau'} \qquad \frac{\Delta, \alpha{:}K \vdash \tau \leq \tau'}{\Delta \vdash \exists\alpha{:}K.\tau \leq \exists\alpha{:}K.\tau'}$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T} \quad \Delta \vdash c : K}{\Delta \vdash \forall\alpha{:}K.\tau \leq \tau[c/\alpha]} \qquad \frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T} \quad \Delta \vdash c : K}{\Delta \vdash \tau[c/\alpha] \leq \exists\alpha{:}K.\tau} \qquad \frac{(\alpha \notin \tau)}{\Delta \vdash \tau \leq \forall\alpha{:}K.\tau} \qquad \frac{(\alpha \notin \tau)}{\Delta \vdash \exists\alpha{:}K.\tau \leq \tau}$$

$$\frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau \leq \texttt{ns}^i} \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \texttt{void} \leq \tau} \qquad \frac{\Delta, \alpha{:}\mathsf{T} \vdash \tau : \mathsf{T}}{\Delta \vdash \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau} \qquad \frac{\Delta, \alpha{:}\mathsf{T} \vdash \tau : \mathsf{T}}{\Delta \vdash \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha]}$$

$$\frac{\Delta \vdash \tau \leq \tau_1 \quad \Delta \vdash \tau \leq \tau_2}{\Delta \vdash \tau \leq \tau_1 \wedge \tau_2} \qquad \frac{\Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 \leq \tau_1} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T}}{\Delta \vdash \tau_1 \wedge \tau_2 \leq \tau_2} \qquad \frac{\Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \leq \tau_1 \vee \tau_2}$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{T}}{\Delta \vdash \tau_2 \leq \tau_1 \vee \tau_2} \qquad \frac{}{\Delta \vdash \tau \wedge (\tau_1 \vee \tau_2) \leq (\tau \wedge \tau_1) \vee (\tau \wedge \tau_2)} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}i}{\Delta \vdash (\tau_1 \times \tau_2) \wedge (\tau_1' \times \tau_2') \leq (\tau_1 \wedge \tau_1') \times (\tau_2 \wedge \tau_2')}$$

$$\frac{}{\Delta \vdash \tau \times (\tau_1 \vee \tau_2) \leq (\tau \times \tau_1) \vee (\tau \times \tau_2)} \qquad \frac{}{\Delta \vdash (\tau_1 \vee \tau_2) \times \tau \leq (\tau_1 \times \tau) \vee (\tau_2 \times \tau)}$$

$$\frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \tau \times \texttt{void} \leq \texttt{void}} \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \texttt{void} \times \tau \leq \texttt{void}} \qquad \frac{}{\Delta \vdash \tau_1 \times (\tau_2 \times \tau_3) \leq (\tau_1 \times \tau_2) \times \tau_3}$$

$$\frac{}{\Delta \vdash (\tau_1 \times \tau_2) \times \tau_3 \leq \tau_1 \times (\tau_2 \times \tau_3)} \qquad \frac{}{\Delta \vdash \tau \leq \texttt{B0} \times \tau} \qquad \frac{}{\Delta \vdash \texttt{B0} \times \tau \leq \tau} \qquad \frac{}{\Delta \vdash \tau \leq \tau \times \texttt{B0}} \qquad \frac{}{\Delta \vdash \tau \times \texttt{B0} \leq \tau}$$

$$\frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \tau \!\uparrow\! B \leq \tau^B} \quad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \tau^B \leq \tau \!\uparrow\! B} \quad \frac{}{\Delta \vdash \mathtt{set}_=(B) \leq \mathtt{int}} \quad \frac{}{\Delta \vdash \mathtt{set}_<(B) \leq \mathtt{int}} \quad \frac{}{\Delta \vdash \mathtt{set}_>(B) \leq \mathtt{int}}$$

$$\frac{}{\Delta \vdash \mathtt{set}_=(B) \wedge \mathtt{set}_<(B) \leq \mathtt{void}} \quad \frac{}{\Delta \vdash \mathtt{set}_=(B) \wedge \mathtt{set}_>(B) \leq \mathtt{void}} \quad \frac{}{\Delta \vdash \mathtt{set}_<(B) \wedge \mathtt{set}_>(B) \leq \mathtt{void}}$$

$$\frac{}{\Delta \vdash \mathtt{int} \leq \exists \alpha{:}\mathsf{Word}.\mathtt{set}_=(\alpha)} \quad \frac{(B_1 \leq B_2)}{\Delta \vdash \mathtt{set}_<(B_1) \leq \mathtt{set}_<(B_2)} \quad \frac{(B_1 \geq B_2)}{\Delta \vdash \mathtt{set}_>(B_1) \leq \mathtt{set}_>(B_2)}$$

$$\frac{(B_1 < B_2)}{\Delta \vdash \mathtt{set}_=(B_1) \leq \mathtt{set}_<(B_2)} \quad \frac{(B_1 > B_2)}{\Delta \vdash \mathtt{set}_=(B_1) \leq \mathtt{set}_>(B_2)} \quad \frac{\Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash \varphi \ \mathsf{true}}{\Delta \vdash \varphi \Rightarrow \tau \leq \tau} \quad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \mathcal{S}(t) \leq \mathtt{BW}}$$

$$\frac{\Delta \vdash t_1 = t_2 \ \mathsf{true}}{\Delta \vdash \mathcal{S}(t_1) \leq \mathcal{S}(t_2)}$$

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

$\boxed{\Delta; \Psi; \Gamma \vdash o : \tau}$

$$\frac{}{\Delta; \Psi; \Gamma \vdash B : \mathtt{set}_=(B)} \quad \frac{}{\Delta; \Psi; \Gamma \vdash B : \mathcal{S}(\overline{B})} \quad \frac{}{\Delta; \Psi; \Gamma \vdash \ell : \Psi(\ell)} \quad \frac{}{\Delta; \Psi; \Gamma \vdash \mathsf{esp} : \mathtt{sptr}(\Gamma(\mathsf{esp}))}$$

$$\frac{\begin{array}{c}\Delta; \Psi; \Gamma \vdash o : \mathtt{sptr}(\tau_1 \times \tau_2 \times \tau_3) \\ \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \\ \Delta \vdash \Gamma(\mathsf{esp}) \leq \tau \times \tau_1 \times \tau_2 \times \tau_3\end{array}}{\Delta; \Psi; \Gamma \vdash m`[o+n] : \tau_2} \quad \frac{\begin{array}{ll}\Delta; \Psi; \Gamma & \vdash o_1 : \mathtt{box}((\tau_1 \times \tau_2 \times \tau_3) \!\uparrow\! x) \\ \Delta; \Psi; \Gamma & \vdash o_2 : \mathtt{set}_<(x) \\ \Delta & \vdash \tau_1 : \mathsf{T}n \\ \Delta & \vdash \tau_2 : \mathsf{T}m \\ \Delta & \vdash \tau_1 \times \tau_2 \times \tau_2 : \mathsf{T}k\end{array}}{\Delta; \Psi; \Gamma \vdash m`[o_1 + n + k \cdot o_2] : \tau_2}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \\ \Delta; \Psi; \Gamma \vdash o : \mathtt{box}(\tau_1 \times \tau_2 \times \tau_3)\end{array}}{\Delta; \Psi; \Gamma \vdash m`[o+n] : \tau_2} \quad \frac{}{\Delta; \Psi; \Gamma \vdash r : \Gamma(r)} \quad \frac{\Delta; \Psi; \Gamma \vdash o : \tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta; \Psi; \Gamma \vdash o : \tau}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \mathtt{int} \quad \Delta \vdash x : \mathsf{Word}}{\Delta; \Psi; \Gamma \vdash o : \mathtt{set}_<(x) \vee \mathtt{set}_=(x) \vee \mathtt{set}_>(x)}$$

$\boxed{\Delta; \Psi; \Gamma \vdash d : \tau \to \Gamma'}$

$$\frac{\Delta \vdash \tau : \mathsf{T}W}{\Delta; \Psi; \Gamma \vdash r : \tau \to \Gamma\{r{:}\tau\}} \quad \frac{\Delta \vdash \tau \leq \mathtt{sptr}(\tau_2) \quad \Delta \vdash \Gamma(\mathsf{esp}) \leq \tau_1 \times \tau_2}{\Delta; \Psi; \Gamma \vdash \mathsf{esp} : \tau \to \Gamma\{\mathsf{esp}{:}\tau_2\}}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \mathtt{mbox}(\tau_1 \times \tau_2 \times \tau_3) \quad \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m}{\Delta; \Psi; \Gamma \vdash m`[o+n] : \tau_2 \to \Gamma}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma(r) \leq \mathtt{sptr}(\tau_1 \times \tau_2 \times \tau_3) \\ \Delta \vdash \Gamma(\mathsf{esp}) \leq \tau \times \tau_1 \times \tau_2 \times \tau_3 \\ \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \quad \Delta \vdash \tau_2' : \mathsf{T}m\end{array}}{\Delta;\Psi;\Gamma \vdash m\text{‘}[r+n] : \tau_2' \to \Gamma\{\mathtt{esp}{:}\tau \times \tau_1 \times \tau_2' \times \tau_3, r{:}\mathtt{sptr}(\tau_1 \times \tau_2' \times \tau_3)\}}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \quad \Delta \vdash \tau_1 \times \tau_2 \times \tau_2 : \mathsf{T}k \\ \Delta;\Psi;\Gamma \vdash o_1 : \mathtt{mbox}((\tau_1 \times \tau_2 \times \tau_3)\uparrow x) \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{set}_<(x)\end{array}}{\Delta;\Psi;\Gamma \vdash m\text{‘}[o_1 + n + k \cdot o_2] : \tau_2 \to \Gamma}$$

$\boxed{\Delta;\Psi;\Gamma \vdash I}$

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

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \mathtt{sptr}(\tau_1 \times \tau_2) \quad \Delta \vdash \tau_1 : \mathsf{T}i \\ \Delta \vdash \tau_2 : \mathsf{TD} \quad \Delta;\Psi;\Gamma \vdash d : \mathtt{sptr}(\tau_2) \to \Gamma' \quad \Delta;\Psi;\Gamma'\{\mathtt{ck}{:}t\} \vdash I\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{addsptr}\ d, o, i\ I} \ (\Gamma(\mathtt{ck}) = 1 + t)$$

$$\frac{\begin{array}{c}\Delta';\Psi;\Gamma_{ret} \vdash I \quad \Delta' \vdash \Gamma_{ret} \quad \Delta;\Psi;\Gamma' \vdash o : \Gamma' \to 0 \\ (\Gamma(\mathtt{ck}) = 1 + t) \quad (\Delta' = \Delta, \alpha_1{:}K_1, \ldots, \alpha_n{:}K_n) \\ (\Gamma' = \Gamma\{\mathtt{sp}{:}(\forall \alpha_1{:}K_1 \ldots \forall \alpha_n{:}K_n.\Gamma_{ret} \to 0) \times \Gamma(\mathtt{sp}), \mathtt{ck}{:}t\})\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{call}\, o; I}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o_1 : \mathtt{int} \quad \Delta;\Psi;\Gamma \vdash o_2 : \mathtt{int} \quad \Delta;\Psi;\Gamma\{\mathtt{ck}{:}t\} \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{cmp}\ o_1, o_2\ I} \ (\Gamma(\mathtt{ck}) = 1 + t)$$

$$\frac{\begin{array}{ll} & (\Gamma(\mathtt{ck}) = 2 + t) \\ & (\Gamma(r) = \tau_1 \vee \tau_2) \\ \Delta;\Psi;\Gamma & \vdash o_1 : \mathtt{int} \\ \Delta;\Psi;\Gamma & \vdash o_2 : \mathtt{set}_=(x) \\ \Delta & \vdash \tau_1 \vee \tau_2 : \mathsf{T}W \\ \Delta;\Psi;\Gamma\{r{:}\tau_1\} & \vdash o_1 : \tau_1' \\ \Delta;\Psi;\Gamma\{r{:}\tau_2\} & \vdash o_1 : \tau_2' \\ \Delta & \vdash \tau_1' \wedge \tau_{unsat}^{\kappa,x} \leq \mathtt{void} \\ \Delta & \vdash \tau_2' \wedge \tau_{sat}^{\kappa,x} \leq \mathtt{void} \\ \Delta;\Psi;\Gamma & \vdash o_3 : \Gamma\{r{:}\tau_1, \mathtt{ck}{:}t\} \to 0 \\ \multicolumn{2}{l}{\Delta;\Psi;\Gamma\{r{:}\tau_2, \mathtt{ck}{:}t\} \vdash I}\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{cmpjcc}\ o_1, o_2, \kappa, o_3\ I}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : (\Gamma\{\mathtt{ck}{:}t\}) \to 0 \quad \Delta;\Psi;\Gamma\{\mathtt{ck}{:}t\} \vdash I}{\Delta;\Psi;\Gamma \vdash \mathtt{jcc}\ \kappa, o; I} \ (\Gamma(\mathtt{ck}) = 1 + t)$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : (\Gamma\{\mathtt{ck}{:}t\}) \to 0}{\Delta;\Psi;\Gamma \vdash \mathtt{jmp}\ o} \ (\Gamma(\mathtt{ck}) = 1 + t) \qquad \frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \mathtt{got} \\ \Delta;\Psi;\Gamma\{r{:}\mathtt{nsw}, \mathtt{ck}{:}t\} \vdash I \text{ inits } r{:}\mathtt{mbox}(\mathtt{ns}^n)\end{array}}{\Delta;\Psi;\Gamma \vdash \mathtt{malloc}\ o, n, r\ I} \ (\Gamma(\mathtt{ck}) = 1 + t)$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma\vdash o_1:\mathtt{got}\\ \Delta;\Psi;\Gamma\vdash o_2:\mathtt{set}_=(x)\quad \Delta;\Psi;\Gamma\{r:\mathtt{nsw}\}\vdash o_3:\tau\\ \Delta\vdash\tau:\mathsf{T}n\quad \Delta;\Psi;\Gamma\{r{:}\mathtt{mbox}(\tau\uparrow x),\mathtt{ck}{:}t\}\vdash I\end{array}}{\Delta;\Psi;\Gamma\vdash\mathtt{mallocarr}\ o_1,r,n,o_2,o_3\ I}\ (\Gamma(\mathtt{ck})=1+t)$$

$$\frac{\begin{array}{c}(\Gamma(\mathtt{ck})=1+t)\\ \Delta\vdash\Gamma(\mathtt{esp})\le\tau_1\times\tau_2\quad\Delta\vdash\tau_1:\mathsf{T}n\quad\Delta\vdash\tau_2:\mathsf{TD}\\ \Delta;\Psi;\Gamma\{\mathtt{esp}{:}\tau_2\}\vdash d:\tau_1\to\Gamma'\quad\Delta;\Psi;\Gamma'\{\mathtt{ck}{:}t\}\vdash I\end{array}}{\Delta;\Psi;\Gamma\vdash\mathtt{pop}\ n,d\ I}\qquad\frac{\begin{array}{c}(\Gamma(\mathtt{ck})=1+t)\quad\Delta;\Psi;\Gamma\vdash o:\tau\\ \Delta\vdash\tau:\mathsf{TD}\quad\Delta;\Psi;\Gamma\{\mathtt{esp}{:}\tau\times\Gamma(\mathtt{esp}),\mathtt{ck}{:}t\vdash I\end{array}}{\Delta;\Psi;\Gamma\vdash\mathtt{push}\ o\ I}$$

$$\frac{\Delta\vdash\Gamma(\mathtt{sp})\le(\Gamma\{sp{:}\tau,\mathtt{ck}{:}t\}\to 0)\times\tau\quad\Delta\vdash\tau:\mathsf{TD}\quad(\Gamma(\mathtt{ck})=1+t)}{\Delta;\Psi;\Gamma\vdash\mathtt{ret}}$$

$$\frac{\begin{array}{c}(\Gamma(\mathtt{ck})=1+t)\\ \Delta;\Psi;\Gamma\{\mathtt{esp}{:}\mathtt{ns}^n\times\Gamma(\mathtt{esp}),\mathtt{ck}{:}t\}\vdash I\end{array}}{\Delta;\Psi;\Gamma\vdash\mathtt{salloc}\ n\ I}\qquad\frac{\begin{array}{c}(\Gamma(\mathtt{ck})=1+t)\\ \Delta\vdash\Gamma(\mathtt{esp})\le\tau_1\times\tau_2\quad\Delta\vdash\tau_1:\mathsf{T}n\\ \Delta\vdash\tau_2:\mathsf{TD}\quad\Delta;\Psi;\Gamma\{\mathtt{esp}{:}\tau_2,\mathtt{ck}{:}t\}\vdash I\end{array}}{\Delta;\Psi;\Gamma\vdash\mathtt{sfree}\ n\ I}$$

$$\frac{\begin{array}{c}(\Gamma(\mathtt{ck})=1+t)\\ \Delta;\Psi;\Gamma\vdash o_1:\mathtt{int}\quad\Delta;\Psi;\Gamma\vdash o_2:\mathtt{int}\\ \Delta;\Psi;\Gamma\vdash d:\mathtt{int}\to\Gamma'\quad\Delta;\Psi;\Gamma'\{\mathtt{ck}{:}t\}\vdash I\end{array}}{\Delta;\Psi;\Gamma\vdash\mathtt{sub}\ d,o_1,o_2\ I}\qquad\frac{\begin{array}{c}\Delta;\Psi;\Gamma\vdash o:\mathtt{got}\\ \Delta;\Psi;\Gamma\{\mathtt{ck}{:}\overline{Y}\}\vdash I\end{array}}{\Delta;\Psi;\Gamma\vdash\mathtt{yield}\ o;I}$$

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

$\boxed{\Delta;\Psi;\Gamma\vdash I\ \text{inits}\ r{:}\mathtt{mbox}(\tau)}$

$$\frac{\begin{array}{c}\Delta\vdash\tau\le\tau_1\times\tau_2\times\tau_3\\ \Delta\vdash\tau_1:\mathsf{T}n\quad\Delta\vdash\tau_2:\mathsf{T}m\quad\Delta\vdash\tau_2':\mathsf{T}m\\ \Delta;\Psi;\Gamma\vdash o:\tau_2'\quad\Delta;\Psi;\Gamma\{\mathtt{ck}{:}t\}\vdash I\ \text{inits}\ r{:}\mathtt{mbox}(\tau_1\times\tau_2'\times\tau_3)\end{array}}{\Delta\vdash\mathtt{mov}\ m`[r+n],o\ I\ \text{inits}\ r{:}\mathtt{mbox}(\tau)}\ (\Gamma(\mathtt{ck})=1+t)$$

$$\frac{\begin{array}{c}\Delta\vdash\tau\le\tau_1\times\tau_2\times\tau_3\\ \Delta\vdash\tau_1:\mathsf{T}n\quad\Delta\vdash\tau_2:\mathsf{T}m\\ \Delta\vdash\Gamma(\mathtt{esp})\le\tau_2'\times\tau'\quad\Delta\vdash\tau_2':\mathsf{T}m\quad\Delta\vdash\tau':\mathsf{TD}\\ \Delta;\Psi;\Gamma\{\mathtt{esp}{:}\tau',\mathtt{ck}{:}t\}\vdash I\ \text{inits}\ r{:}\mathtt{mbox}(\tau_1\times\tau_2'\times\tau_3)\end{array}}{\Delta\vdash\mathtt{pop}\ m,m`[o+n]\ I\ \text{inits}\ r{:}\mathtt{mbox}(\tau)}\ (\Gamma(\mathtt{ck})=1+t)\qquad\frac{\Delta;\Psi;\Gamma\{r{:}\mathtt{mbox}(\tau)\}\vdash I}{\Delta;\Psi;\Gamma\vdash I\ \text{inits}\ r{:}\mathtt{mbox}(\tau)}$$

$\boxed{\Psi;\Delta\vdash I:\tau\ \mathsf{block}\quad\vdash P}$

$$\frac{\Psi; (\Delta, \alpha{:}K) \vdash I : \tau \text{ block}}{\Psi; \Delta \vdash I : \forall \alpha{:}K.\tau \text{ block}} \qquad \frac{\Psi; (\Delta, \varphi \text{ true}) \vdash I : \tau \text{ block}}{\Psi; \Delta \vdash I : \varphi \Rightarrow \tau \text{ block}} \qquad \frac{\Psi; \Delta; \Gamma \vdash I}{\Psi; \Delta \vdash I : \Gamma \to 0 \text{ block}}$$

$$\frac{\begin{array}{c}(\mathrm{dom}(\Psi) = \{\ell_1, \ldots, \ell_n\}) \\ \vdash \Psi \quad (\Psi(\ell_1) = \{\mathtt{ebx}{:}\mathtt{got}, \mathtt{esp}{:}\mathtt{B0}\} \to 0) \\ \Psi; \cdot \vdash I_i : \Psi(\ell_i) \text{ block for } 1 \le i \le n\end{array}}{\vdash \ell_1{:}\tau_1 = I_1, \ldots, \ell_n{:}\tau_n = I_n}$$

B Typing Rules for Lilt

$\boxed{\Delta \vdash \Gamma \quad \Delta \vdash \Xi \quad \Delta \vdash \Xi \text{ handles } \Gamma}$

$$\frac{\Delta \vdash \tau_1 : T \text{ for } 1 \le i \le n}{\Delta \vdash [s_1{:}\tau_1, \ldots, s_n{:}\tau_n]} \qquad \frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Xi \quad \Delta \vdash \Gamma}{\Delta \vdash \Xi, \Gamma}$$

$$\frac{}{\Delta \vdash \cdot \text{ handles } \Gamma} \qquad \frac{\Delta \vdash \Gamma \le \Gamma'}{\Delta \vdash (\Xi, \Gamma') \text{ handles } \Gamma}$$

$\boxed{\Delta \vdash \tau_1 \le \tau_2 \quad \Delta \vdash \Gamma_1 \le \Gamma_2 \quad \Delta \vdash \Xi_1 \le \Xi_2}$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : T}{\Delta \vdash \tau_1 \le \tau_2} \qquad \frac{\Delta \vdash \tau_1 : T \quad \Delta \vdash \tau_2 = \mathtt{ns} : T}{\Delta \vdash \tau_1 \le \tau_2}$$

$$\frac{\Delta \vdash \tau_i \le \tau_i' \text{ for } 1 \le i \le n}{\Delta \vdash [s_1{:}\tau_1, \ldots, s_n{:}\tau_n] \le [s_1{:}\tau_1', \ldots, s_n{:}\tau_n']} \qquad \frac{}{\Delta \vdash \cdot \le \cdot} \qquad \frac{\Delta \vdash \Xi_1 \le \Xi_2 \quad \Delta \vdash \Gamma_2 \le \Gamma_1}{\Delta \vdash (\Xi_1, \Gamma_1) \le (\Xi_2, \Gamma_2)}$$

$\boxed{\Delta \vdash c : k}$

$$\frac{((\alpha{:}k) \in \Delta)}{\Delta \vdash \alpha : k} \qquad \frac{}{\Delta \vdash \mathtt{ns} : T} \qquad \frac{}{\Delta \vdash \mathtt{int} : T} \qquad \frac{}{\Delta \vdash \mathtt{bool} : T} \qquad \frac{}{\Delta \vdash \mathtt{unit} : T}$$

$$\frac{\Delta \vdash \tau_i : T \text{ for } 1 \le i \le k}{\Delta \vdash \langle \tau_1, \ldots, \tau_k \rangle : T} \qquad \frac{\begin{array}{c}(i_j \ne i_k \text{ for } j \ne k) \\ \Delta \vdash \tau_i : T \text{ for } 1 \le i \le k\end{array}}{\Delta \vdash [i_1{:}\tau_1, \ldots, i_n{:}\tau_n] : T} \qquad \frac{\begin{array}{c}\Delta \vdash \tau : T \\ \Delta \vdash \tau_i : T \text{ for } 1 \le i \le n\end{array}}{\Delta \vdash (\tau_1, \ldots, \tau_n) \to \tau : T}$$

$$\frac{\Delta \vdash \tau : T}{\Delta \vdash \tau\ \mathtt{array} : T} \qquad \frac{\Delta, \alpha{:}T \vdash \tau : T}{\Delta \vdash \mu\alpha.\tau : T} \qquad \frac{\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n \vdash \tau : T}{\Delta \vdash \forall \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau : T}$$

$$\frac{\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n \vdash \tau : T}{\Delta \vdash \exists \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau : T} \qquad \frac{\Delta, \alpha{:}k_1 \vdash c : k_2}{\Delta \vdash \lambda\alpha{:}k_1.c : k_1 \to k_2} \qquad \frac{\Delta \vdash c_1 : k_2 \to k \quad \Delta \vdash c_2 : k_2}{\Delta \vdash c_1\, c_2 : k}$$

$\boxed{\Delta \vdash c_1 = c_2 : k}$

$$\frac{((\alpha{:}k) \in \Delta)}{\Delta \vdash \alpha = \alpha : k} \qquad \frac{}{\Delta \vdash \texttt{ns} = \texttt{ns} : T} \qquad \frac{}{\Delta \vdash \texttt{int} = \texttt{int} : T} \qquad \frac{}{\Delta \vdash \texttt{bool} = \texttt{bool} : T}$$

$$\frac{}{\Delta \vdash \texttt{unit} = \texttt{unit} : T} \qquad \frac{\Delta \vdash \tau_i = \tau_i' : T \text{ for } 1 \leq i \leq k}{\Delta \vdash \langle \tau_1, \ldots, \tau_k \rangle = \langle \tau_1', \ldots, \tau_2' \rangle : T}$$

$$\frac{\begin{array}{c}(i_j \neq i_k \text{ for } j \neq k) \\ \Delta \vdash \tau_i = \tau_i' : T \text{ for } 1 \leq i \leq k\end{array}}{\Delta \vdash [i_1{:}\tau_1, \ldots, i_n{:}\tau_n] = [i_1{:}\tau_1', \ldots, i_n{:}\tau_n'] : T} \qquad \frac{\begin{array}{c}\Delta \vdash \tau = \tau' : T \\ \Delta \vdash \tau_i = \tau_i' : T \text{ for } 1 \leq i \leq n\end{array}}{\Delta \vdash (\tau_1, \ldots, \tau_n) \to \tau = (\tau_1', \ldots, \tau_n') \to \tau' : T}$$

$$\frac{\Delta \vdash \tau = \tau' : T}{\Delta \vdash \tau \ \texttt{array} = \tau' \ \texttt{array} : T} \qquad \frac{\Delta, \alpha{:}T \vdash \tau = \tau' : T}{\Delta \vdash \mu\alpha.\tau = \mu\alpha.\tau' : T}$$

$$\frac{\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n \vdash \tau = \tau' : T}{\Delta \vdash \forall \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau = \forall \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' : T} \qquad \frac{\Delta, \alpha{:}k_1 \vdash c = c' : k_2}{\Delta \vdash \lambda\alpha{:}k_1.c = \lambda\alpha{:}k_1.c' : k_1 \to k_2}$$

$$\frac{\Delta, \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n \vdash \tau = \tau' : T}{\Delta \vdash \exists \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau = \exists \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' : T} \qquad \frac{\begin{array}{c}\Delta \vdash c_2 = c_2' : k_2 \\ \Delta \vdash c_1 = c_1' : k_2 \to k\end{array}}{\Delta \vdash c_1\, c_2 = c_1'\, c_2' : k}$$

$$\frac{\Delta, \alpha{:}k_2 \vdash c_1 : k \quad \Delta \vdash c_2 : k_2}{\Delta \vdash (\lambda\alpha{:}k_2.c_1)\, c_2 = c_1[c_2/\alpha] : k}$$

$\boxed{\Delta \vdash q : \tau_1 \Rightarrow \tau_2}$

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

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

$$\frac{\Delta \vdash \tau = \exists \alpha_1{:}k_1, \ldots, \alpha_n{:}k_n.\tau' : T \quad \Delta \vdash c_i : k_i \text{ for } 1 \leq i \leq n}{\Delta \vdash \texttt{pack}[\tau, c_1, \ldots, c_n] : \tau'[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n] \Rightarrow \tau} \qquad \frac{\begin{array}{c}\Delta \vdash q : \tau_1' \Rightarrow \tau_2' \\ \Delta \vdash \tau_i = \tau_i' : T \text{ for } i = 1, 2\end{array}}{\Delta \vdash q : \tau_1 \Rightarrow \tau_2}$$

$\boxed{\Phi; \Delta; \Gamma \vdash r : \tau}$

$$\frac{(\Gamma(s) = \tau)}{\Phi; \Delta; \Gamma \vdash s : \tau} \qquad \frac{}{\Phi; \Delta; \Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Phi; \Delta; \Gamma \vdash \texttt{tt} : \texttt{bool}} \qquad \frac{}{\Phi; \Delta; \Gamma \vdash \texttt{ff} : \texttt{bool}}$$

$$\frac{}{\Phi;\Delta;\Gamma \vdash \star : \texttt{unit}} \qquad \frac{(\Phi(f) = \tau)}{\Phi;\Delta;\Gamma \vdash f : \tau} \qquad \frac{\Phi;\Delta;\Gamma \vdash v : \tau_2 \quad \Delta \vdash q : \tau_2 \Rightarrow \tau}{\Phi;\Delta;\Gamma \vdash q@v : \tau}$$

$$\frac{\Phi;\Delta;\Gamma \vdash r : \tau' \quad \Delta \vdash \tau' = \tau}{\Phi;\Delta;\Gamma \vdash r : \tau} \qquad \frac{(\mathit{op} : (\tau_1, \ldots, \tau_k) \to \tau) \quad \Phi;\Delta;\Gamma \vdash v_i : \tau_i \text{ for } 1 \le i \le k}{\Phi;\Delta;\Gamma \vdash \mathit{op}(v_1, \ldots, v_k) : \tau}$$

$$\frac{\Phi;\Delta;\Gamma \vdash v_i : \tau_i \text{ for } 0 \le i \le k}{\Phi;\Delta;\Gamma \vdash \langle v_0, \ldots, v_k\rangle : \langle \tau_0, \ldots, \tau_k\rangle} \qquad \frac{\Phi;\Delta;\Gamma \vdash v : \langle \tau_0, \ldots, \tau_k\rangle}{\Phi;\Delta;\Gamma \vdash \pi_i v : \tau_i} \qquad \frac{\Phi;\Delta;\Gamma \vdash v_i : \tau \text{ for } 1 \le i \le n}{\Phi;\Delta;\Gamma \vdash \{v_1, \ldots, v_n\} : \tau\ \texttt{array}}$$

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

$\boxed{\Phi;\Delta;\Gamma \vdash \mathit{cond}\ \mathsf{cond}}$

$$\frac{\Phi;\Delta;\Gamma \vdash v_i : \texttt{int} \text{ for } i = 1, 2}{\Phi;\Delta;\Gamma \vdash v_1 = v_2\ \mathsf{cond}} \qquad \frac{\Phi;\Delta;\Gamma \vdash v_i : \texttt{int} \text{ for } i = 1, 2}{\Phi;\Delta;\Gamma \vdash v_1 < v_2\ \mathsf{cond}}$$

$\boxed{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash e}$

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

$$\frac{\begin{array}{c}(\Lambda(\ell) = \mathit{lbl}(\alpha_1{:}k_1, \ldots, \alpha_n{:}k_n; \Xi'; \Gamma')) \\ \Delta \vdash c_i : k_i \quad \Delta \vdash \Gamma \le \Gamma'[\vec{c}/\vec{\alpha}] \quad \Delta \vdash \Xi \le \Xi'[\vec{c}/\vec{\alpha}]\end{array}}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{goto}\ \ell[c_1, \ldots, c_n]} \qquad \frac{\Phi;\Delta;\Gamma \vdash r : \tau' \quad \Phi;\Delta;\Lambda;\Xi;\Gamma[s \mapsto \tau'];\tau \vdash e}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}\ s = r\ \texttt{in}\ e}$$

$$\frac{\begin{array}{c}\Phi;\Delta;\Gamma \vdash v : (\tau_1' \ldots, \tau_n') \to \tau'' \quad \Delta \vdash \Xi\ \mathsf{handles}\ \Gamma \\ \Delta \vdash v_i : \tau_i' \text{ for } 1 \le i \le n \quad \Phi;\Delta;\Lambda;\Xi;\Gamma[s \mapsto \tau''];\tau \vdash e\end{array}}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}\ s = v(v_1, \ldots, v_n)\ \texttt{in}\ e}$$

$$\frac{\begin{array}{c}\Phi;\Delta;\Gamma \vdash v : \langle \tau_0, \ldots, \tau_m\rangle \\ \Phi;\Delta;\Gamma \vdash v : \tau_i \quad \Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash e\end{array}}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}\ \pi_i\, v := v'\ \texttt{in}\ e} \qquad \frac{\begin{array}{c}\Phi;\Delta;\Gamma \vdash v : \tau'\,\texttt{array} \quad \Phi;\Delta;\Gamma \vdash v' : \texttt{int} \\ \Delta \vdash \Xi\ \mathsf{handles}\ \Gamma \quad \Phi;\Delta;\Lambda;\Xi;\Gamma[s \mapsto \tau'];\tau \vdash e\end{array}}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}\ s = \texttt{sub}(v, v')\ \texttt{in}\ e}$$

$$\frac{\begin{array}{c}\Phi;\Delta;\Gamma \vdash v_1 : \tau'\,\texttt{array} \quad \Phi;\Delta;\Gamma \vdash v_2 : \texttt{int} \\ \Phi;\Delta;\Gamma \vdash v_3 : \tau' \quad \Delta \vdash \Xi\ \mathsf{handles}\ \Gamma \quad \Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash e\end{array}}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau \vdash \texttt{let}\ \texttt{sub}(v_1, v_2) := v_3\ \texttt{in}\ e}$$

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

$$\frac{\Phi;\Delta;\Gamma\vdash v:\exists\alpha_1{:}k_1,\ldots,\alpha_n{:}k_n.\tau'\quad \Phi;(\Delta,\alpha_1{:}k_1,\ldots,\alpha_n{:}k_n);\Lambda;\Xi;\Gamma[s\mapsto\tau'];\tau\vdash e}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau\vdash \texttt{let}(\alpha_1,\ldots,\alpha_n,s)=\texttt{unpack}\ v\ \texttt{in}\ e}$$

$$\frac{\begin{array}{c}\Phi;\Delta;\Gamma\vdash \mathit{cond}\ \mathsf{cond}\\ \Phi;\Delta;\Lambda;\Xi;\Gamma;\tau\vdash e_1\quad \Phi;\Delta;\Lambda;\Xi;\Gamma;\tau\vdash e_2\end{array}}{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau\vdash \texttt{if}\ \mathit{cond}\ \texttt{then}\ e_1\ \texttt{else}\ e_2}\qquad \frac{\Phi;\Delta;\Lambda;\Xi;\Gamma;\tau\vdash e}{\Phi;\Delta;\Lambda;(\Xi,\Gamma');\Gamma;\tau\vdash \texttt{pophandler in}\ e}$$

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

$\boxed{\Phi;\Delta;\Lambda;\tau\vdash B:\gamma}$

$$\frac{\begin{array}{c}\Delta,\Delta'\vdash\Xi\\ \Delta,\Delta'\vdash\Gamma\quad \Phi;(\Delta,\Delta');\Lambda;\Xi;\Gamma;\tau\vdash e\end{array}}{\Phi;\Delta;\Lambda;\tau\vdash \texttt{block}(\Delta';\Xi;\Gamma).e:\mathit{lbl}(\Delta';\Xi;\Gamma)}\qquad \frac{\begin{array}{c}\Delta,\Delta'\vdash\Xi\quad \Delta,\Delta'\vdash\Gamma\\ \Phi;(\Delta,\Delta');\Lambda;\Xi;\Gamma[s\mapsto\tau_{\text{exn}}];\tau\vdash e\end{array}}{\Phi;\Delta;\Lambda;\tau\vdash \texttt{hndl}(\Delta';\Xi;\Gamma;s).e:\mathit{hnd}(\Delta';\Xi;\Gamma)}$$

$\boxed{\Phi\vdash F:\tau}$

$$\frac{\begin{array}{c}\vdash\Delta\quad \Delta\vdash\Gamma_{\text{arg}}\quad \Delta\vdash\tau:T\quad \Delta\vdash\Lambda\\ \Phi;\Delta;\Lambda;\cdot;\Gamma;\tau\vdash e\quad \Phi;\Delta;\Lambda;\tau\vdash B_i:\Lambda(\ell_i)\ \text{for}\ 1\le i\le m\end{array}}{\Phi\vdash \texttt{func}(\Delta;\Gamma_{\text{arg}};\tau).(\texttt{enter}(s_1,\ldots,s_n).e,\ell_1=B_1,\ldots,\ell_m=B_m):\forall\Delta.(\tau_1,\ldots,\tau_p)\to\tau}$$

where

$\Gamma_{\text{arg}}=[s'_1{:}\tau_1,\ldots,s'_p{:}\tau_p]$
$\Gamma=[s'_1{:}\tau_1,\ldots,s'_p{:}\tau_p,s_1{:}\texttt{ns},\ldots,s_n{:}\texttt{ns}]$
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
$\text{dom}(\Gamma_i)=\text{dom}(\Gamma)$ for each i

$\boxed{\vdash P}$

$$\frac{\vdash\Phi\quad \Phi\vdash F_i:\Phi(f_i)\ \text{for}\ 1\le i\le n\quad (\text{dom}(\Phi)=\{f_1,\ldots,f_n\})}{\vdash f_1=F_1,\ldots,f_n=F_n}$$