Automated and Certified Conformance to Responsiveness Policies

Joseph C. Vanderwaart

joev@cs.cmu.edu

Karl Crary

crary@cs.cmu.edu

Computer Science Department, Carnegie Mellon University

ABSTRACT

Certified code systems protect computers from faulty or malicious code by requiring untrusted software to be accompanied by checkable evidence of its safety. This paper presents a certified code solution to a problem in grid computing, namely, controlling the CPU usage of untrusted programs. Specifically, we propose to endow the runtime system supervising local execution of grid programs with a trusted "yield" operation, and require the untrusted code to execute this operation with at least a certain frequency. Compliance with this requirement is enforced by a special typed assembly language, which we describe.

We also describe a compilation strategy for a general-purpose programming language that can enforce and certify conformance to such policies automatically without any sophisticated program analyses. This means that owners of hosts participating in the computation network can be confident that executing foreign code will not compromise the availability of their machines for running their own processes, and application programmers do not need to modify their coding style in order to produce compliant software.

Categories and Subject Descriptors: F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D.3.2 [Programming Languages]: Language classifications—macro and assembly languages; D.3.4 [Programming Languages]: Processors—compilers.

General Terms: Languages, Reliability, Security

Keywords: Certified code, typed assembly language, grid computing

1. INTRODUCTION

Certified code provides a powerful and flexible approach to protecting computers from untrusted and potentially dangerous mobile code they execute. In a certified code framework, any untrusted software must be accompanied by some additional information, called a *certificate*, whose validity is known to imply that the software is safe to run. The most obvious manifestation of this idea is Proof-Carrying Code (PCC) [14, 15], in which the certificate is simply a proof that the code satisfies some safety criterion. Systems based on variations of Typed Assembly Language (TAL) [12, 11, 4] also qualify: in these systems, the certificate consists of typing annotations that demonstrate that the accompanying code is a well-typed TAL program.

An application of certified code in which we are particularly interested is *grid computing*. The ConCert project [3] aims to build a scalable, secure framework for distributed computing in which the owners of the machines performing the computation need not trust, nor indeed know anything about, the producers of the code their machines run. If deployed on a global scale, with many thousands of participants, this framework would provide a platform for grid computing in the style of SETI@Home [16], with the difference that *anyone* could submit jobs to the grid for distributed execution, but nevertheless the owners of participating hosts could be assured that no harm would come to their machines by way of faulty or malicious mobile code.

Of course, certified code can only be practically useful if the notion of "safety" guaranteed by a valid certificate is appropriate to the application. The usual criteria of type and memory safety are a good start, but many applications, including grid computing, demand stronger safety policies. One important aspect of safety for grid computing is that foreign code must respond to the presence of higher-priority local processes in a timely fashion, either by slowing down their CPU usage or by quitting altogether. Furthermore, it is important to us that compliance with this responsiveness requirement not place an undue burden on grid application programmers.

In the ConCert grid computing model [2], each host on the grid runs a process (referred to in this paper as the *supervisor*) that downloads mobile software from the network, schedules jobs for local execution, and communicates the results of those jobs to other hosts that need them. The supervisor is also responsible for ensuring that participation in the grid does no harm to the host. It must therefore be able to respond to increased load on the host system by local programs by "throttling" the foreign programs under its control. To do this, the supervisor must be able to communicate with its subordinate processes and must have some assurance that they will pay attention to its instructions.

We ensure responsiveness by providing programs with a trusted library routine that communicates synchronously

with the supervisor, and requiring that they call this routine with at least a certain frequency. This behavior, which we call "yielding", ensures that the supervisor has ample opportunities for communication with the runtime system supporting the untrusted process and therefore can prevent the mobile code from running when local processes compete for the CPU. The behavior we have in mind for the trusted yield operation is simply to send a message to the supervisor and wait for an acknowledgement. If the machine is idle, then the supervisor replies immediately, but if it is busy, the supervisor may leave the foreign process suspended for some time. Since ConCert programs don't perform any explicit I/O or network operations (and the grid is designed to tolerate delayed results or failures transparently), this delay is not observable by the grid application. This simple strategy allows the supervisor to control the CPU usage of untrusted processes over a very wide range, and to adjust its policy at any time—either automatically or at the host user's request—without any dependency on the flexibility or fairness of the operating system's scheduler.

In this paper, we present a certified code system for enforcing our yielding policy. Our solution is an extension of the typed assembly language TALT [4], augmented with a "yield" instruction that must be performed with at least a certain frequency. More precisely, we require that no more than Y non-yield instructions are ever executed in a row, where the number Y is a parameter of the type system whose value is determined by the safety policy. The type system for our variant of TALT, which we call TALT-R, is such that well-typed programs necessarily obey this additional safety criterion.

In addition, we show that generating well-typed TALT-R code places essentially no burden on the grid application programmer. We present an approach for compiling arbitrary programs in a safe C-like language to TALT-R. Essentially, we show that a compiler can turn any valid source program into well-typed TALT-R by inserting enough yield instructions; we describe a fairly simple strategy for doing this, and then show how to improve it using a dynamic instruction counter. This involves showing how to *type* the code for instruction counting in a way that certifies its correctness, so programs that rely on the accuracy of this run-time information can still be certifiably safe. We have implemented both these strategies in a compiler, and we present experimental results indicating that the performance penalty associated with the dynamic counting approach is small.

The paper is organized as follows. Section 2 presents our typed assembly language for responsiveness certification. Section 3 describes our strategy for generating well-typed assembly code from arbitrary source programs. Section 4 assesses the performance of the compiler we have implemented according to this strategy. Finally, Section 5 concludes by discussing future directions for this research and its relationship to other work.

2. TALT-R

In this section we present our typed assembly language for responsiveness certification. Our language TALT-R is an extension of TALT, which itself is a typed version of Intel IA-32 (also known as x86) assembly language. Due to concerns for space and clarity, we will limit ourselves in this paper to a minimal subset of the language, which we call MiniTALT-R.

$$W = 4 \text{ (word size in bytes)}$$

Words	B	$\in$	$\{0, \ldots, 2^{8W} - 1\}$
Registers	r	$::=$	$\texttt{eax} \mid \texttt{ebx} \mid \texttt{ecx} \mid \texttt{edx}$
			$\mid \;\; \texttt{esi} \mid \texttt{edi} \mid \texttt{ebp}$
	$\bar{r}$	$::=$	$r \mid \texttt{esp}$
Operands	o	$::=$	$B \mid \ell \mid \bar{r} \mid [o + j]$
Destinations	d	$::=$	$\bar{r} \mid [o + j]$
Conditions	κ	$::=$	$\texttt{e} \mid \texttt{ne} \mid \texttt{b} \mid \texttt{be} \mid \texttt{a} \mid \texttt{ae}$
Instr. Sequences	I	$::=$	ϵ
			$\mid \;\; \texttt{add } d, o_1, o_2 \; I$
			$\mid \;\; \texttt{call } o \; I \mid \texttt{cmp } o_1, o_2 \; I$
			$\mid \;\; \texttt{jcc } \kappa, o \; I \mid \texttt{jmp } o \; I$
			$\mid \;\; \texttt{mov } d, o \; I \mid \texttt{ret } I$
			$\mid \;\; \texttt{pop } d \; I \mid \texttt{push } o \; I$
			$\mid \;\; \texttt{salloc } n \; I \mid \texttt{sfree } n \; I$
			$\mid \;\; \texttt{sub } d, o_1, o_2 \; I$
			$\mid \;\; \texttt{subjae } \mathbf{r}_d, o_1, o_2, o_3 \; I$
			$\mid \;\; \texttt{yield } I$
Programs	P	$::=$	$\ell_1 = I_1, \ldots, \ell_n = I_n$

Figure 1: Untyped MiniTALT-R Syntax

MiniTALT-R contains only those instructions needed for the examples of Section 3, and only the types and typing rules needed to type those examples. We also allow ourselves the luxury of a somewhat more readable syntax than the original presentation of TALT [4] by glossing over the details of instruction decoding and pc-relative addressing. A more complete treatment of TALT-R can be found in the technical report [17].

As mentioned earlier, TALT-R has a yield instruction, implemented as a call to the trusted runtime system. The precise behavior of this instruction is not our concern for the bulk of this paper—on the contrary, we assume that the effects of the yield instruction are not directly observable by the program, so that our compiler will be free to insert them wherever necessary to satisfy the safety policy. What is most important to us at the moment is that at least one yield must be executed for every Y other instructions, where Y is a fixed large integer. The TALT-R type system extends that of TALT with facilities for tracking the number of instructions remaining until the next yield is due, including a mechanism for arithmetic reasoning and a form of singleton type that we use to implement run-time clock checks.

Like TALT, TALT-R is a type assignment (or Curry-style) system. Programs contain no typing annotations, and typing is presumed undecidable. An explicitly-typed version of the language that enjoys tractable type-checking plays a central role in the certification mechanism, but we will not discuss it in this paper.

2.1 Syntax

The syntax of MiniTALT-R programs is given in Figure 1. The first few definitions in the figure (namely the word size W and the sets of words and register names) are specific to the Intel IA-32 architecture; the rest of the syntax is also fairly Intel-like but should be familiar to programmers in other assembly languages. A MiniTALT-R program con-

sists of a sequence of *blocks*, each consisting of a label and a sequence of instructions. Most instructions read from one or more *operands* and many store a result in a *destination*. An operand may be a literal word value (B), a label denoting an address in the code segment (ℓ), a register ($\bar{r}$, which may be the stack pointer register $\mathtt{esp}$), or a memory operand of the form $[o+j]$, which refers to the word-sized (*i.e.*, 4-byte) value in memory starting at the location j bytes after the one pointed to by the operand o. A destination may be a register or a memory location. Most of the instructions shown in the figure are presumably familiar to IA-32 assembly programmers. The exceptions are the $\mathtt{salloc}$ and $\mathtt{sfree}$ instructions we inherit from TALT, which allocate and deallocate space on the stack by subtracting from or adding to the stack pointer, respectively, and $\mathtt{subjae}$ and $\mathtt{yield}$, which are new in TALT-R.

The $\mathtt{yield}$ instruction is the most important addition: it is this instruction that the new safety policy requires the program to execute with at least a certain frequency to ensure responsiveness to the supervisor. The meaning of this will be made more precise shortly. The $\mathtt{subjae}$ instruction is a combination of a subtraction and a conditional jump. That is, the operational behavior of the instruction sequence $\mathtt{subjae}\ \mathtt{r}_d, o_1, o_2, o_3\ I$ is the same as that of $\mathtt{sub}\ \mathtt{r}_d, o_1, o_3\ \mathtt{jcc}\ \mathtt{ae}, o_3\ I$. The purpose of this apparently redundant "compound" instruction will become clearer when we consider its typing rule later on.

2.2 Safety Policy

In the TALT methodolgy, the safety policy takes the form of an operational semantics for the target architecture (in the form of a transition relation between machine states) that does not include any "unsafe" transitions. The standard type preservation and progress lemmas are then proved, implying that a concrete IA-32 processor will not perform any unsafe actions when executing a well-typed program. We refer the reader to [5] for more details on the TALT approach to certification.

The safety policy of ordinary TALT provides a fairly basic form of type safety. For TALT-R, we also want to require that the $\mathtt{yield}$ instruction is executed with at least a certain frequency; concretely, we choose a number Y and require that a $\mathtt{yield}$ be performed at least every Y instructions. The way to enforce this in the safety policy is to add a *virtual clock* to the operational semantics as proposed by Necula and Lee [15]. The virtual clock is an imaginary register that is decremented with every instruction executed; no instruction (other than $\mathtt{yield}$) may be executed when the virtual clock is zero. Consequently, the virtual clock represents an upper bound on the number of instructions that may be executed before the next $\mathtt{yield}$. The $\mathtt{yield}$ instruction itself resets the virtual clock to Y.

With the safety policy thus enriched, we will give a type system for MiniTALT-R such that well-typed programs execute without getting stuck. The main additions to the system are types that track the time remaining until the next yield and a logic of constraints for reasoning about the clock.

2.3 Type System

The syntax of the type system for MiniTALT-R is given in Figure 2. The judgment forms of the system are summarized in Figure 3. As with the syntax of programs, the syntax we have shown here is only a fragment of the full TALT-

Kinds	K	$::=$	$\mathsf{T} \mid \mathsf{T}i \mid \mathsf{N} \mid \mathsf{P}$
Static Terms	c, t, φ, τ	$::=$	α
(e.g., types)			$\mid\ \mathtt{ns}i \mid \mathtt{B}i \mid \tau_1 \times \tau_2$
			$\mid\ \mathtt{sptr}(\tau) \mid \Gamma \to 0$
			$\mid\ \forall \alpha{:}K.\tau \mid \exists \alpha{:}K.\tau$
			$\mid\ \varphi \Rightarrow \tau \mid \mathcal{S}(t)$
			$\mid\ \overline{n} \mid t_1 + t_2$
			$\mid\ t_1 \leq t_2 \mid t_1 = t_2$
Static Contexts	Δ	$::=$	$\cdot \mid \Delta, \alpha{:}K \mid \Delta, \varphi\ \mathsf{true}$
Reg. File Types	Γ	$::=$	$\{\overline{r{:}\tau_r}, \mathtt{esp}{:}\tau, \mathtt{ck}{:}t\}$
Memory Types	Ψ	$::=$	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$

Figure 2: MiniTALT-R Type System Syntax

Judgment	Meaning
$\Delta \vdash c : K$	Static term c has kind K
$\Delta \vdash \Gamma$	Reg. file type Γ is well-formed
$\Delta \vdash \varphi\ \mathsf{true}$	Formula φ is true
$\Delta \vdash \tau \leq \tau'$	τ is a subtype of τ'
$\Psi; \Delta; \Gamma \vdash I$	Instr. seq. I is well-typed
$\Psi; \Delta; \Gamma \vdash o : \tau$	Operand o produces a value of type τ
$\Psi; \Delta; \Gamma \vdash$ $d : \tau \to \Gamma'$	Storing a value of type τ in dest. d yields a reg. file of type Γ
$\Psi; \Delta \vdash I : \tau\ \mathsf{block}$	I is a block of type τ
$\vdash \Psi$	Heap type Ψ is well-formed.
$\vdash P$	Program P is well-typed.

Figure 3: MiniTALT-R Judgment Forms

R, sufficient for the examples and discussion later in the paper; the missing features are exactly analogous to TALT and are covered in the technical report [17]. Due to space constraints we will discuss only a few key typing rules later in this section; a complete set of rules for MiniTALT-R can be found in Appendix A.

At the top level of the system are four *kinds*, which classify the terms at the second level, which we call *static terms*. The kinds T and $\mathsf{T}i$ are inherited from TALT; N and P are new. The class of static terms is comprised of the *types* (of kinds $\mathsf{T}i$ and T), the *constraint terms* (of kind N), and the *constraint formulas* (of kind P). By convention, we will use the metavariables τ, t and φ in place of the general metavariable c to indicate that the static term referred to is a type, 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 MiniTALT-R programs.

An unusual feature of TALT, inherited by TALT-R, is that values are not all the same size. For each natural number $n \geq 0$, $\mathsf{T}n$ is the kind of types whose values are exactly n bytes in size; T is the kind of any type whatsoever (thus any type of kind $\mathsf{T}i$ also has kind T). Most of the values manipulated by MiniTALT-R code will have types of kind $\mathsf{T}W$, where W is the word size of the architecture (for IA-

$$\frac{((\alpha{:}K) \in \Delta)}{\Delta \vdash \alpha : K} \qquad \frac{}{\Delta \vdash \mathtt{ns}i : \mathsf{T}i} \qquad \frac{}{\Delta \vdash \mathtt{B}i : \mathsf{T}i} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T} \quad \Delta \vdash \tau_2 : \mathsf{T}}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{T}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{T}i \quad \Delta \vdash \tau_2 : \mathsf{T}j}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{T}(i+j)} \qquad \frac{\Delta \vdash \tau : \mathsf{T}}{\Delta \vdash \mathtt{sptr}(\tau) : \mathsf{TW}} \qquad \frac{\Delta \vdash \Gamma}{\Delta \vdash \Gamma \to 0 : \mathsf{TW}}$$

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

$$\frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \mathcal{S}(t) : \mathsf{TW}} \qquad \frac{}{\Delta \vdash \overline{n} : \mathsf{N}} \ (n \geq 0) \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 + t_2 : \mathsf{N}} \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 \leq t_2 : \mathsf{P}} \qquad \frac{\Delta \vdash t_1 : \mathsf{N} \quad \Delta \vdash t_2 : \mathsf{N}}{\Delta \vdash t_1 = t_2 : \mathsf{P}} \qquad \frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau : \mathsf{T}}$$

Figure 4: Type Formation Rules for MiniTALT-R ($\Delta \vdash c : K$)

32, $W = 4$). The notable exception is the stack, which is permitted to vary greatly in size and hence usually has a type of kind T.

Most of the types of TALT-R are analogous to those of TALT. For $i > 0$, the type $\mathtt{ns}i$ may be given to any value whatsoever of size i; any type of kind $\mathsf{T}i$ is therefore a subtype of $\mathtt{ns}i$. For $i \geq 0$, $\mathtt{B}i$ is the type of integer values i bytes in width. Values of the product type $\tau_1 \times \tau_2$ consist of a value of type τ_1 and one of type τ_2 appended together; hence if $\tau_1 : \mathsf{T}i$ and $\tau_2 : \mathsf{T}j$ then the product has kind $\mathsf{T}(i + j)$. There are subtyping rules that make the product constructor associative and $\mathtt{B}0$ a unit.

Pointers to code (and in particular the labels associated with MiniTALT-R instruction blocks) have arrow types of the form $\Gamma \to 0$, where Γ is a *register file type*. It is safe to jump to a pointer of type $\Gamma \to 0$ if the current register state has type Γ. The type $\mathtt{sptr}(\tau)$ describes a pointer into the stack. Universal quantification $\forall \alpha{:}K.\tau$ is crucial, as it usually is in typed assembly languages; we will not have much use for existential types $\exists \alpha{:}K.\tau$ in this paper. The special new types in TALT-R are the *guarded types* ($\varphi \Rightarrow \tau$) and the *singleton types* ($\mathcal{S}(t)$), which we will discuss later.

Due to space considerations, we do not address the issue of type safety in this paper—to do so would require a definition of the language's operational semantics. We are currently working on extending the formal and mechanically checkable proof of type safety for TALT [4] to cover TALT-R; we do not anticipate any difficulties, since the new features are largely orthogonal to the existing language and analogous constructs have been studied in connection with LXres/TALres [7] and DTAL [18].

2.3.1 The Constraint Subsystem

The purpose of the constraint terms and formulas is to allow the type system to reason about the time remaining before the next yield instruction must be performed. This constraint logic is largely separable from the rest of the type system; in fact, there is a certain degree of flexibility in its design. The version 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 ($\wedge$, $\vee$, $\supset$, $\bot$) to the constraint logic, but we have not found this necessary to accomplish our task. A formula such as $t_1 \leq t_2$ is well-formed (with kind P) if t_1 and t_2 have kind N; 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 $\Delta \vdash \varphi \ \mathtt{true}$ means that the truth of the formula φ follows from the assumptions in Δ. Note that according to Figure 2, Δ may contain both kinding assumptions of the form $\alpha{:}K$ and hypotheses of the form $\varphi \ \mathtt{true}$. The truth judgment is defined inductively by a set of rules 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. These rules are given in Appendix A and include: reflexivity, symmetry, transitivity and compatibility rules for equality; an axiom for addition of natural number constants; identity, commutativity and associativity rules for addition; reflexivity, transitivity and anti-symmetry 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.

2.3.2 Instruction Typing and the Virtual Clock

Since the virtual clock behaves like an extra register, the type system accounts for it in the types of code pointers and in the typing rules for instructions. Again, the complete set of instruction typing rules can be found in Appendix A, along with the typing rules for operands and destinations on which they depend.

According to Figure 2, a TALT-R register file type Γ contains types for the machine's registers (just like in TALx86 or TALT) as well as a constraint term that represents a conservative approximation of the virtual clock. A register file type Γ where $\Gamma(\mathtt{ck}) = t$ specifies a machine state where the virtual clock reads *at least* (the number denoted by) t.

The decrementing of the virtual clock with every instruction is accomplished straightforwardly in the typing rules. The rule for the $\mathtt{add}$ instruction is typical:

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

This rule stipulates that the operands to be added must be 4-byte integers, and determines the register file type Γ' produced by propagating the result of the addition (of type $\mathtt{B4}$) to the destination. The side condition that $\Gamma(\mathtt{ck}) = \overline{1}+t$, where t is the virtual clock assumption under which the remaining code must be typed, accomplishes two things: it captures the requirement that the virtual clock read at least one in order to perform an $\mathtt{add}$, and it tracks the fact that performing the instruction decrements the clock. The other instruction typing rules follow the same pattern, except for $\mathtt{yield}$ (which does not restrict the clock's initial value and resets it to $\overline{Y}$), $\mathtt{subjae}$ (which must decrement the clock by

two), and the instructions that manipulate code addresses, which we will discuss next.

As usual, the code pointer type $\Gamma \to 0$ describes code that may be executed when the register file has type Γ. The implications for the typing of control-transfer instructions (`jmp`, `jcc`, `call` and `ret`) are straightforward and are easiest to see by considering the rule for the `jmp` instruction:

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

This says that it is safe to jump to a code address of type $\Gamma' \to 0$ only if the clock reads $\overline{1} + \Gamma'(\mathtt{ck})$ *before the jump*; that is, it requires that it be possible to decrement the virtual clock in the current register file type to get the register file type expected by the jump target. The rules for `jcc`, `call` and `ret` are a bit more complicated than for `jmp`, but the clock manipulations they perform are the same (and in all other respects they are similar to their counterparts in TALx86 and TALT).

A few important rules having to do with subtyping allow a program to "forget" about some of the time remaining on the virtual clock. The key rule is the one for register file subtyping:

$$\frac{\begin{array}{c}\Delta \vdash \tau \leq \tau' \quad \Delta \vdash t' \leq t\ \mathsf{true}\\ \Delta \vdash \tau_r \leq \tau'_r \ \text{for each register } r\end{array}}{\begin{array}{c}\Delta \vdash \{\mathtt{eax}{:}\tau_{\mathtt{eax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{ebp}}, \mathtt{esp}{:}\tau, \mathtt{ck}{:}t\}\\ \leq \{\mathtt{eax}{:}\tau'_{\mathtt{eax}}, \ldots, \mathtt{ebp}{:}\tau'_{\mathtt{ebp}}, \mathtt{esp}{:}\tau', \mathtt{ck}{:}t'\}\end{array}}$$

As usual, this rule requires that the type assigned to any register $\bar{r}$ by the register file type on the left must be a subtype of the type assigned to $\bar{r}$ on the right. For the clock, it additionally requires that the approximation of the clock on the right be provably less than or equal to the one on the left. To see why this is correct, recall that $\mathtt{ck}{:}t$ on the left means that the clock reads *at least* t, and $\mathtt{ck}{:}t'$ means the clock reads at least t'. If $t' \leq t$ is provable, then anything that is at least t will also be at least t'; thus the register file type on the left is a more restrictive specification of the machine state, consistent with the usual meaning of subtyping.

The key consequences of this register file subtyping rule arise from its use in two other rules inherited from TALT:

$$\frac{\Delta; \Psi; \Gamma' \vdash I \quad \Delta \vdash \Gamma \leq \Gamma'}{\Delta; \Psi; \Gamma \vdash I} \qquad \frac{\Delta \vdash \Gamma' \leq \Gamma}{\Delta \vdash \Gamma \to 0 \leq \Gamma' \to 0}$$

The first of these, called *register file weakening*, allows an instruction sequence to "forget" about some of the time on the clock or, even more usefully, to rewrite the `ck` term in the register file using the equality relation of the constraint logic. (In particular, the side condition in most of the typing rules that $\Gamma(\mathtt{ck})$ be *syntactically* equal to $\overline{1} + t$ for some t is not overly restrictive.) The second, the subtyping rule for arrow types, allows a pointer to code expecting a smaller clock value to stand in for one to code expecting a larger value.

2.3.3 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 3. Essentially, they comprise

an index refinement system similar to the dependent types of DTAL [18]. We will use this form of refinement to construct more precise types for functions than would otherwise be possible, and the constraint reasoning built into the type system can allow more efficient code to be written. In particular, singleton types will play a central role in our dynamic checking scheme.

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, a value 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 such a value.

It turns out to be more important (for our compilation strategy) to be able to give guarded types to code pointers than to any other kind of value. Thus, we have an introduction rule for guarded types at the level of code blocks:

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

This rule states that (the address of) a code block I may be given type $\varphi \Rightarrow \tau$ if it can be shown to have the type τ under the assumption that φ is true. Importantly, the derivation of $I : \tau\ \mathsf{block}$ may depend on the hypothesis $\varphi\ \mathsf{true}$; I 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.

Singleton types in TALT-R play a role similar to that of singletons in DTAL [18] and LTT [6]. 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.)

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 (`subjae` $\mathbf{r}_d, o_1, o_2, o_3\ I$) subtracts the value of o_2 from o_1 and stores the result in $\mathbf{r}_d$; if this result is greater than or equal to zero, control jumps to the address in o_3; otherwise, execution continues with I. The `subjae` instruction has a special singleton-aware typing rule:

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

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. If the conditional branch is taken, then the result is nonnegative 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 $u = v + a$. 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 following the `subjae` must be well-formed assuming only that the destination register contains an integer. Finally, note that the virtual clock is decremented by two instead of by one; this is because `subjae` is implemented by a sequence of two instructions on a concrete IA-32 machine.

3. COMPILATION

As mentioned earlier, we have implemented a compiler for a general-purpose high-level language that produces well-typed (and therefore yield-latency certified) TALT-R code without any explicit help from the programmer. The source language is the safe C-like language Popcorn [11], which is completely ignorant of our responsiveness requirement; it is the job of the compiler to insert `yield` instructions wherever necessary to make the resulting TALT-R code well-typed. This is in contrast to, for example, the PopCron language of Crary and Weirich [7], in which functions must be annotated with cost information. In this section we will give some details of our compilation strategy.

Assuming one knows how to compile Popcorn into TALT (which is not too different from compiling it into TAL [11]), targeting TALT-R instead does not present much additional difficulty in principle: one could simply place a `yield` after every instruction in a TALT program (and add `ck:`$\overline{Y-1}$ to every code pointer type) to produce a well-typed TALT-R program. Of course, this is unsatisfactory because the `yield` instruction involves synchronous communication with a supervising process and is therefore very expensive. The goal when generating TALT-R, then, is to maximize the time between successive yields while keeping it less than Y.

A less pathological, but still naïve, approach is to place a `yield` at the beginning of every basic block. Under this strategy, each function call and each return must incur a `yield`. It is possible to improve the situation using a strategy based on Feeley's *balanced polling* [8], so that some function calls do not have to yield. Unfortunately, it is still the case that every iteration of a loop must yield. Since loops are common and their bodies are often short, lifting this requirement seems critical for performance.

Hoisting yield instructions out of loops altogether would seem to require that we be able to statically predict the number of iterations a loop will perform. This would require either programmer assistance or sophisticated program analyses to discover this information, as well as sophisticated static reasoning at the typed assembly language level to certify its correctness. We have chosen another option: programs will use a dynamic instruction counter to decide when to yield, and the type system will enforce the relationship between this dynamic counter and our static knowledge about the virtual clock. This means that our compiler must insert instructions to consult and decrement the counter.

It turns out that the problem of placing these run-time checks is analogous to that of placing `yield` instructions in-sofar as each dynamic check permits the program to execute a certain number of instructions, and the time between dynamic checks must be bounded by Y. Our solution is to use the Feeley placement strategy to determine where the checks should occur. In this section, we will first describe our Feeley-inspired approach for "direct" placement of yields into the instruction stream; after we have described the machinery of dynamic checks, we will show how to adapt the Feeley placement strategy to place checks instead of yields.

3.1 Direct Yield Placement

Placement of yield operations in straight-line code (or tree-structured code) is uninteresting; the only challenges arise when translating program constructs that involve non-trivial control flow. There are three kinds of control flow in Popcorn: *intra*procedural control flow, associated with conditionals and loops; *inter*procedural control flow, associated with function call and return; and exceptional control flow, associated with raising and handling exceptions. Due to space considerations, we will ignore exceptions here.

We also have very little to say about *intra*procedural control flow; since the target of every *intra*procedural jump is statically known, this does not take much insight. The compiler simply decides on an initial virtual clock value for each basic block in a function; then, whenever it emits an *intra*procedural jump instruction, it generates a yield first iff the initial clock value of the target block is greater than the current value, which it computes from the current block's initial value and its length.

*Inter*procedural control flow presents a greater challenge, because Popcorn allows programmers to write higher-order functions. (It provides function pointers like those in C.) Because of this capability, the function being invoked in a function call expression is not always statically known, and so the *cost* of a particular function call cannot be statically calculated either. We therefore need a way of translating functions and function calls *uniformly*—that is, so that any two functions that have the same type in Popcorn have the same clock behavior and hence the same type in TALT-R. The remainder of this section describes how this is done.

To begin, consider this attempt at a type for a function from integers to integers (passing both argument and result in `eax`):

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

This type attempts to describe a function with "cost" n: it says that the function may be called if the clock reads $n + a$, and returns with the clock reading a, where a is a universally quantified constraint term variable. This idiom for specifying a function's cost is essentially that used in TALres [7], but there is a problem: 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.

To solve this problem, we note that in reality the function's initial clock value of $n + a$ will always be less than or equal to $Y - 1$. Hence, if the function yields, the resulting clock value of Y is guaranteed to be greater than or equal

to $n+a+1$, allowing the function to return—in fact, it may execute up to n more instructions before doing so.

The constraint logic of TALT-R is powerful enough to perform this simple reasoning, if only the function "knows" that $n+a \leq Y-1$; in other words, a function that yields and returns up to n instructions later can be well-typed, but *only under the constraint hypothesis* $\overline{n} + a \leq \overline{Y-1}$. This is the purpose of the guarded types introduced in Section *2.3.3*; if we compile a function so that its body is well-typed under this hypothesis, we can discharge the hypothesis and give it the type:

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

It is important to note that (assuming $n \leq Y-1$) the clock information in this function type does not significantly limit the points in a program where such a function may be called. If the current clock value is at least n, then a call to this function is clearly well-typed; if the clock value is less than n, the caller can simply yield, resetting the clock to Y. Similarly, *any* source language function of the appropriate type may be compiled to TALT-R code of the above type. If the function body needs to execute more than n instructions, then it will have to yield before returning, but functions less than n instructions long may not have to yield at all.

It seems reasonable, therefore, to use the same constant value of n for *every* function in a program. By doing so, we achieve our goal of uniformity in the translation of functions: Popcorn functions of the same type will translate to MiniTALT-R functions of the same type. In fact, in this special case, the induced yield-placement strategy is essentially the one described by Feeley [8]. Feeley, whose motivation was placing checkpoints in a program to detect interrupts, named his strategy *balanced polling*. We choose to refer to the yielding scheme we have just described as *Feeley yielding*, and we follow Feeley in using the letter E to denote the fixed value chosen for n.

The major advantage of Feeley yielding over a more naïve strategy in which yields occur at every function call and return is that many short functions (specifically, those functions shorter than E instructions that contain no loops or function calls) never need to 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.

3.2 Dynamic Checks

While the Feeley yielding strategy is fairly simple and easy to implement, it falls well short of our goal of yielding as infrequently as the safety policy will allow, especially if Y is large. In particular, it seems that the greatest actual inter-yield time we can achieve is on the order of the size of the largest basic block in the program being compiled. We now present a dynamic checking scheme that gets around this difficulty. It is important to note that we are not adding dynamic instruction counting primitives to TALT-R; rather, have endowed our type system with enough expressive power to code them up within the language. Thus, the issue of whether to use a direct yield placement strategy such as Feeley yielding or a dynamic one such as we are about to describe is left up to individual compilers or programmers.

3.2.1 The Clock Register

We choose one of the machine's registers to serve as our dynamic clock. We will refer to this register as the *clock register* and by the suggestive name $\mathtt{rck}$, but this is only for clarity of presentation; there is nothing special about this register. In fact, it is not necessary to keep the dynamic instruction counter in a register at all; in particular, there is no reason a compiler could not choose to spill it to the stack and use the register for some other purpose.

To make use of the clock register, we will maintain the invariant that its value is always less than or equal to the number of instructions remaining until the next $\mathtt{yield}$ instruction must occur (*i.e.,* the value of the clock register never exceeds that of the virtual clock). This alone is not very useful information, since it does not rule out the possibility of both quantities being zero; it is much more helpful to know, at a given point in a program, that the virtual clock exceeds the clock register *by (at least) a certain known amount*. This difference (or the best available static approximation thereof) we call the *clock difference*, and it plays a key role: one seemingly trivial but important property is that any instruction (other than $\mathtt{yield}$) that does not change the clock register effectively decrements the clock difference by exactly as much as it decrements the virtual clock itself. Assuming the value of the clock register is statically unknown, it is when the clock difference is (not provably greater than) zero that some special action must be taken.

Thus, the difference between the two "clocks" behaves much like a clock itself. In fact, it is possible to "set" this key quantity to any value (less than Y) at any point in a program. Roughly speaking, to set it to n, one may simply perform a $\mathtt{yield}$ instruction and then move the value $Y - n - 1$ into the clock register. Since the virtual clock reads $Y-1$ after the move, it now exceeds the clock register value by n. Usually, however, it is not necessary to perform a $\mathtt{yield}$ in order to accomplish this feat: if the clock register's current value is greater than n, then subtracting a little more than n from the register (the "little more" accounts for the cost of the subtraction) will suffice. Of course, the exact value of the clock register will almost never be statically known: we need a way of examining the clock register *at run time* to determine whether a yield is necessary to produce the desired clock difference. That is precisely what we will exhibit in the next section, where we make all this reasoning precise.

3.2.2 The Minor Clock

In the next section we will exhibit a sequence of instructions capable of "setting" the clock difference to any constant value less than Y even when the value of the clock register is not statically known. Since the clock difference represents the number of instructions that may safely be executed before examining the clock register, we will call the sequence of instructions that resets it to a predetermined constant value a *minor yield*. It will turn out that the minor yield itself requires 2 instructions, and so it can only be executed when the clock difference is at least 2. (In terms of types, this means that the clock register must have a singleton type $\mathcal{S}(u)$ and the virtual clock value must be provably at least $\overline{2} + u$.) Thus if the clock difference is δ, the number of (non-$\mathtt{yield}$) instructions that may safely be executed before a minor yield becomes impossible is $\delta - 2$; we call

this quantity the *minor clock*. Clearly, "setting" the clock difference to some fixed value D is equivalent to setting the minor clock to $D - 2$. From now on we will consider the minor clock to be the more basic notion.

To be precise: if, at a given point in the program, the clock register has the singleton type $\mathcal{S}(u)$ and the static approximation of the virtual clock (*i.e.*, the term assigned to ck in the register file type) is $t + (\overline{2} + u)$, then we say that t is the current value of the minor clock. By defining some special notation, we may pretend that the minor clock is an additional virtual register: for a given register file type Γ we will write $\Gamma(\texttt{mck}) = t$ to mean that for some u we have $\Gamma(\texttt{rck}) = \mathcal{S}(u)$ and $\Gamma(\texttt{ck}) = t + (\overline{2} + u)$. Additionally, we overload our notation for register file update and write $\Gamma\{\texttt{mck}_u{:}t\}$ to mean $\Gamma\{\texttt{rck}{:}\mathcal{S}(u), \texttt{ck}{:}t + (\overline{2} + u)\}$.

It should be clear that for non-yielding instructions that do not change the clock register, the minor clock locally behaves like the virtual clock: each such instruction decrements the minor clock by one. For example, the following typing rule for the add instruction is derivable:

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

This rule shows how to type an add instruction when the 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 $\mathcal{S}(u)$, meaning that the new minor clock is just t. Similar "minor clock rules" can be derived for all the instructions of MiniTALT-R except for yield. The minor clock also obeys a similar subtyping principle to the virtual clock: in particular, $\Gamma\{\texttt{mck}_u{:}t\} \leq \Gamma\{\texttt{mck}_u{:}t'\}$ if the formula $t' \leq t$ is provable, meaning that as for the virtual clock itself, the static representation of the minor clock is a conservative approximation. We will see in a moment that the minor yield behaves like a minor-clock version of the yield instruction, completing the analogy.

3.2.3 The Minor Yield

Figure 5 shows the code for a minor yield to set the minor clock to a constant number L. For the remainder of this paper we will assume that *all* minor yields set the minor clock to the same value L; this highlights the analogy with the yield instruction, which always sets the virtual clock to Y. Ignoring the typing annotations for the moment, the intent of this code is clear: it attempts to subtract $L - 2$ from the clock register; if this subtraction produces a nonnegative result, control jumps to the label end; if the result is negative, then a yield instruction is performed, the clock register is reset to slightly less than Y, and again control is transferred to end. (In the full TALT-R language this is a fall-through rather than a jump, but the simple MiniTALT-R of this paper does not have this capability.)

The typing annotations in the figure assume that the minor clock is zero before the subjae, and say that the code following the end label may assume that the minor clock is L but the singleton type of the clock register, and hence its value, has probably changed in the process. That is, if the initial register file type is Γ, then the minor yield will be well-typed if the label end has the type $\forall a{:}\mathsf{N}.(\Gamma\{\texttt{mck}_a{:}\overline{L}\} \to 0)$. It is not hard to see that the yielding path through the code is well-typed, as this follows straightforwardly from the def-

```
        // minor clock = 0
        // i.e., rck:S(u), ck:2̄ + u
        subjae rck,rck,(L + 2),end
        // rck:int, ck:u
        yield
        // ck:Ȳ
        mov rck,(Y-L-4)
        jmp end
        // a ↦ Y − L − 4; rck: S(Y − L − 4);
        // ck:Y − 2  =  L̄ + (2̄ + a)
end:
        // a:N, rck:S(a), ck:L̄ + (2̄ + a)
        // i.e., minor clock = L
```

Figure 5: Code for a Minor Yield

inition of the minor clock. It is a little harder to see why end is an appropriate target for the subjae instruction. According to the typing rule for subjae given earlier, the jump operand in this case should be of type

$$\forall a{:}\mathsf{N}.(u = \overline{L+2} + a) \Rightarrow (\Gamma\{\texttt{rck}{:}\mathcal{S}(a), \texttt{ck}{:}u\} \to 0)$$

But the type of end is a subtype of this, reasoning as follows:

$$\begin{aligned} \forall a{:}\mathsf{N}.&(\Gamma\{\texttt{mck}_a{:}\overline{L}\} \to 0) \\ &= \forall a{:}\mathsf{N}.(\Gamma\{\texttt{rck}{:}a, \texttt{ck}{:}\overline{L} + (\overline{2} + a)\} \to 0) \\ &\leq \forall a{:}\mathsf{N}.(u = \overline{L+2} + a) \Rightarrow \\ &\qquad (\Gamma\{\texttt{rck}{:}\mathcal{S}(a), \texttt{ck}{:}\overline{L} + (\overline{2} + a)\} \to 0) \\ &\leq \forall a{:}\mathsf{N}.(u = \overline{L+2} + a) \Rightarrow \\ &\qquad (\Gamma\{\texttt{rck}{:}\mathcal{S}(a), \texttt{ck}{:}u\} \to 0) \end{aligned}$$

Thus, end is an appropriate operand for the subjae instruction, and so the entire minor yield is well-typed.

If we ignore the syntactic inconvenience that a minor yield spans multiple code blocks in MiniTALT-R, it is possible to think of it as an instruction with the followng typing rule:

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

(Here, and from now on, we use the name MYIELD to refer to the minor yield instruction sequence.) This rule states that MYIELD 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.

3.2.4 Compilation Using Minor Yields

Because the minor clock behaves so much like the virtual clock, it turns out that all of the strategies we have found for placing yield instructions directly can be turned into dynamic checking strategies by tracking the minor clock and placing minor yields instead. This is the approach we have implemented in our compiler: in particular, we adapt the Feeley yielding strategy of Section 3.1 and refer to the resultant dynamic checking scheme as *Feeley polling*.

Conceptually, to derive Feeley polling from Feeley yielding, we replace all yield instructions with minor yields (written MYIELD as in the typing rule above), replace all assertions about the virtual clock (*e.g.*, ck:t) with assertions about the minor clock (mck$_b$:t), and replace Y with L since MYIELD resets the minor clock to L. (Of course it is not quite this simple, since L is smaller than Y and therefore very long extended basic blocks may require more minor yields than they

```
int fib( int x ) {
  if ( x <= 1 ) {
    return 1;
  }
  return fib(x-1)+fib(x-2);
}
```

Figure 6: A Simple Popcorn Function

would have needed ordinary yields.) Thus we get a compilation strategy which, instead of placing a yield every Y instructions, places a minor yield every L instructions.

The type of a function from integers to integers under Feeley polling is:

$$\forall a{:}\mathsf{N}.\forall b{:}\mathsf{N}.\forall \rho{:}\mathsf{T}.(\overline{E} + a \leq \overline{L-1}) \Rightarrow$$
$$\{\mathtt{eax}{:}\mathsf{B4}, \mathtt{esp}{:}(\forall b'{:}\mathsf{N}.\{\mathtt{eax}{:}\mathsf{B4}, \mathtt{esp}{:}\rho, \mathtt{mck}_{b'}{:}a\} \to 0)$$
$$\times \rho, \mathtt{mck}_b{:}\overline{E} + a\} \to 0$$

The parameter b is the static term describing the initial value of the clock register (recall that the notation $\mathtt{mck}_b{:}t$ gives the type $\mathcal{S}(b)$ to $\mathtt{rck}$ as well as assigning a constraint term to $\mathtt{ck}$). Because the value, and hence the type, of the clock register may change during the execution of the function, the return address passed on the stack must be able to execute no matter what the register's new value is. The term describing this value is therefore universally quantified in the type of the return address.

An example of the Feeley polling strategy is shown in Figures 6 and 7. The Popcorn function in Figure 6 is a recursive function to compute Fibonacci numbers; the assembly version in Figure 7 was hand-coded in MiniTALT-R and has the type discussed above. Note that the function has a "short path" corresponding to the case where the argument is less than or equal to one, and a "long path" that performs two recursive calls if it is not. The short path does not need to yield (of course, this depends on E being chosen large enough). The long path must perform a minor yield before the first recursive call, and between the last call and the final return instruction. This is typical of the Feeley strategies, since a function might start out with as little as E on the clock, but every callee requires at least E. Similarly, no callee can be assumed to return with more than $L - E - 1$ 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, L and E). Also, notice that every minor yield and every function call might change the value of the clock register; thus each of these events introduces a new constraint term variable (the b_i's) to describe the altered clock register seen by the instructions following it.

4. EVALUATION

We measured the effects of our yielding strategies on four different programs. The benchmarks range in complexity and qualitative behavior: msort applies a polymorphic merge-sort procedure to a pseudorandomly-generated linked list of integers; qsort applies quicksort to an array; comb computes a row of Pascal's triangle; and tempo is a port of the grid-based chess player developed by the ConCert project. Figure 8 shows the impact on execution time: for each benchmark we show the execution time using Feeley yielding and

```
fib:
      // a,b0:N, rck:S(b0), ck:(E + a) + (2 + b0),
      // (E + a  ≤  L - 1)
      cmp eax,1
      ja L1                         // if (x <= 1)
      mov eax,1
      ret                           //   return 1;
L1:
      // ck:(E - 2 + a) + (2 + b0)
      push eax
      sub eax,1
      MYIELD
      // b1:N, rck:S(b1), ck:L + (2 + b1)
      call fib                      // fib(x-1)
      // b2:N, rck:S(b2), ck:L - E - 1 + (2 + b2)
      pop ecx
      push eax
      mov eax,ecx
      sub eax,2
      // ck:L - E - 5 + (2 + b2)
      call fib                      // fib(x-2)
      // b3:N, rck:S(b3), ck:L - 2E - 6 + (2 + b3)
      pop ecx
      add eax,ecx
      // ck:L - 2E - 8 + (2 + b3)
      MYIELD
      // b4:N, rck:S(b4), ck:L + (2 + b4)
      ret     // return fib(x-1)+fib(x-2);
```

Figure 7: Fibonacci using Feeley Polling

Feeley polling, normalized with respect to the running time in ordinary TALT with no yielding requirements. The test programs were linked against a version of the runtime system in which the yield operation does nothing other than count the number of times it is called; thus the increases in running time are due only to function call overhead and/or clock register operations. All timing experiments were performed on a 3.06 GHz Pentium 4. As the chart shows, Feeley yielding slowed down programs by up to 80%, while Feeley polling never altered execution time by more than 5%.

An important issue faced by the implementation but not apparent in our discussion of MiniTALT-R is the timing behavior of TALT-R's malloc instruction, which allocates space in a garbage-collected heap. It is difficult to predict how long an invocation of malloc will take: those that trigger a garbage collection run much longer than those that do not. For our initial experiments we assumed that the runtime system would conservatively yield at every allocation. Figure 9 shows that for msort and tempo, which do a lot of allocation, these "implicit yields" dominate the "explicit yields" introduced by our compiler: msort performs some 7.8 million yielding operations per second on average, only 3.5 of which are yield instructions. The qsort and comb benchmarks were carefully written to allocate as little as possible, and perform only a constant number of implicit yields per run.

These results clearly indicate a need for a better treatment of allocation. One possibility is to provide a version of malloc that has access to the program's clock register and yields only when needed. This "smart malloc" poses no problems in principle, but the implementation effort required is nontrivial and we have not attempted it yet. To estimate the performance improvement, we modified our implementation to assume a fixed cost for malloc (to simulate fast, non-collecting allocations) and instrumented the
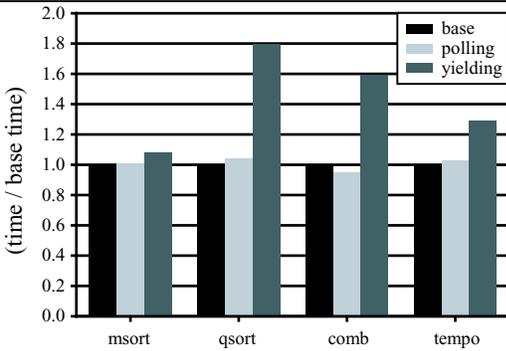
Figure 8: Normalized execution time ($Y = 1$ billion, $L = 500$, $E = 100$, $H = 50$)

	Implicit Y/s	Explicit Y/s
msort	7 800 000	3.5
qsort	44	44
comb	75	50
tempo	840 000	0

Figure 9: Breakdown of Yields under Feeley Polling

	FY	FP	FPsm (est.)
msort	1.8×10^7	7.9×10^6	3.4×10^1
qsort	5.8×10^7	8.7×10^1	5.6×10^1
comb	6.0×10^7	1.3×10^2	6.4×10^1
tempo	3.5×10^7	8.5×10^5	1.5×10^3

Figure 10: Yield Frequencies (Yields/sec)

runtime system to count the number of garbage collections, which presumably would still have to yield. Figure 10 shows the estimated yield rate for the smart `malloc` along with the rates we measured for Feeley yielding and polling. In the case of `msort`, the smart `malloc` reduces the total yield rate to tens of yields per second. The `tempo` benchmark still performs badly; this is likely due to its heavy use of arrays, which our compiler does not handle well.

5. DISCUSSION

There are a number of directions in which we believe the work we have described can be readily extended. First of all, although we have implemented a Popcorn compiler and described our compilation strategy in terms of Popcorn, it should be clear that there is nothing particularly source-language-specific about our strategy. In fact, our compiler is based on a relatively low-level typed intermediate language and we believe that front-ends for other source languages could easily be developed. It would be interesting to see how the implications of the yield placement strategy for performance would differ between, say, Popcorn programs and ML programs.

Interestingly, although guaranteeing bounds on the total running time of programs was not our goal, the core type system of TALT-R may be used for this purpose. To do so, we simply remove the `yield` instruction so that there is no way to reset the virtual clock; now, the clock effectively represents the number of instructions that may be executed before the program must halt. From the point of view of the certification mechanism, this bounds the total running time of every program by a constant. A dynamic checking scheme akin to minor yielding could be used in a compiler, producing programs that would run for approximately the maximum allowed time and then quit.

Certification of latencies may well have applications outside of grid computing. A suitable application of TALT-R could be used to certify compliance with scheduling constraints in real-time applications, or to enforce cooperation among processes in embedded settings without the overhead of a preemptive scheduler. Furthermore, although we have presented TALT-R as a type system for instruction counting, the same basic system could be applied to enforcing other properties as well. For example, copying garbage collectors provide a contiguous block of memory into which new objects are written; when this area is full, the collector must be notified. A slightly modified version of TALT-R's type system could describe this behavior: instead of counting down the number of instructions until the next yield, it could count the number of bytes that may be allocated until the next call to the garbage collector. The same ideas could also be used to track the amount of stack space remaining; a program could be forced to use some special operation to actively compare the stack pointer to the end of the available space, rather than relying on the virtual memory system to trap overflows and terminate the program as the TALT implementation currently does.

Some system resources, such as network bandwidth, can be controlled by placing *upper bounds* on the frequency with which a program performs certain actions. It seems likely that a system like TALT-R can certify bounds of this kind: by reversing the sense of approximation of the virtual clock so that `ck`:t means the clock reads *at most t*, one would be able to show that *at least Y* instructions are executed in between, say, calls to network library functions. It even appears possible to design a language in which both upper and lower bounds on latencies can be certified in the same system, but we have not investigated the implications for compilation in this setting.

As another direction for improvement, we note that as far as the TALT-R type system is concerned, it would be a simple matter to replace the constraint logic used for reasoning about the virtual clock with a more sophisticated one. It is easy to imagine, for example, adding multiplication or primitive recursion to the language of constraint terms, or enhancing the constraint truth judgment with a rule for induction over natural numbers. Such extensions might be useful in that they could allow some clock computations to be hoisted out of loops or functions. On the other hand, the more sophisticated the logic, the more difficult and expensive the theorem-proving tasks involved in certification are likely to become.

5.1 Related Work

The most closely related work to ours is in the area of *resource bound certification*. The study of this topic, and the use of virtual clocks in certified code, began with Necula and Lee's observation [15] that Proof-Carrying Code could guarantee bounded running time of programs. Crary and Weirich's type theory LXres and assembly language TALres [7] allowed programmers to use the type system to specify the running time of a function based on the structure (most

importantly the size) of its argument. To our knowledge, neither of these approaches has produced a workable system for general programming.

Hofmann [9] has shown that a type system based on linear logic can limit both the space usage and time complexity of functional programs; in particular, any well-typed program in Hofmann's linear λ-calculus denotes a polynomial-time function. This is a weaker guarantee than the safety policy of TALT-R since, after all, even a constant-time program might take longer to finish than a user is willing to wait. Also, as a source language, the linear λ-calculus is less programmer-friendly than Popcorn, and we conjecture that it would be a fair amount more difficult for grid application programmers to learn. As far as space is concerned, Hofmann's language forbids programs to allocate new storage, and the linear type system is used to allow already-allocated space to be reused in a type-safe way. Hofmann and Jost [10] continued in this direction, showing how to allow programs to perform some allocation while keeping total heap usage under control.

Aspinall *et al.* [1] have presented a program logic for a fragment of JVM bytecode that is capable of reasoning about resource usage. The compiler and PCC system they have implemented based on this logic requires programs to be written in a special resource-aware language, and to our knowledge they do not treat any resources other than space.

Naik [13] has described a type system for interrupt-driven programming that ensures all interrupts will be handled within appropriate deadlines. This is similar to ensuring that programs yield with a certain frequency, and so his type system bears some resemblance to ours. In particular, it has a notion of a virtual clock and uses singleton types in a critical way, but it does not support dynamic checking. It would be interesting to explore the relationship between these type systems in detail.

6. REFERENCES

[1] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Proc. of the Intl. Conf. on Theorem Proving in Higher-Order Logics*, 2004.
[2] B.-Y. E. Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning. Trustless grid computing in ConCert. In *Proceedings of the GRID 2002 Workshop*, 2002.
[3] The ConCert project home page. http://www.cs.cmu.edu/~concert/.
[4] K. Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th POPL*, 2003.
[5] K. Crary and S. Sarkar. Foundational certified code in a metalogical framework. In *Proceedings of the Conference on Automated Deduction (CADE-19)*, Miami, FL, July 2003.
[6] K. Crary and J. C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proceedings of ICFP'02*, Pittsburgh, PA, Oct. 2002.
[7] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, MA, 2000.
[8] M. 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.
[9] M. Hofmann. Linear types and non-size increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.
[10] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th POPL*, 2003.
[11] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, 1999.
[12] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
[13] M. Naik. A type system equivalent to model checking. Master's thesis, Purdue University, 2003.
[14] G. Necula. Proof-carrying code. In *Proceedings of the 24th POPL*, pages 106–119, Paris, Jan. 1997.
[15] G. Necula and P. 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, Oct. 1997.
[16] SETI@home. http://setiathome.ssl.berkeley.edu.
[17] J. C. Vanderwaart and K. Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, Carnegie Mellon University, Pittsburgh, PA, Feb. 2004.
[18] H. Xi and R. Harper. A dependently typed assembly language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Florence, Italy, Sept. 2001.

APPENDIX

A. MINITALT-R TYPING RULES

Static Term Formation ($\Delta \vdash c : K$)

These rules are given in Figure 4.

Context Formation ($\Delta \vdash \Gamma, \vdash \Psi$)

$$\frac{\begin{array}{c} \Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash t : \mathsf{N} \\ \Delta \vdash \tau_r : \mathsf{TW} \text{ for each register } r \end{array}}{\Delta \vdash \{\mathtt{eax}{:}\tau_{\mathtt{eax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{ebp}}, \mathtt{esp}{:}\tau, \mathtt{ck}{:}t\}} \qquad \frac{\vdash \tau_i : \mathsf{T} \text{ for } 1 \leq i \leq n}{\vdash \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}}$$

Constraint Truth ($\Delta \vdash \varphi$ true)

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

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

$$\frac{\Delta \vdash t_1 = t_1' \text{ true} \quad \Delta \vdash t_2 = t_2' \text{ true}}{\Delta \vdash t_1 + t_2 = t_1' + t_2' \text{ true}} \qquad \frac{}{\Delta \vdash \overline{m} + \overline{n} = \overline{m+n} \text{ true}}$$

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

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

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

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

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

Subtyping ($\Delta \vdash \tau_1 \leq \tau_2$)

$$\frac{}{\Delta \vdash \tau \leq \tau} \qquad \frac{\Delta \vdash \tau_1 \leq \tau_3 \quad \Delta \vdash \tau_3 \leq \tau_2}{\Delta \vdash \tau_1 \leq \tau_2}$$

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

$$\frac{\Delta, \alpha{:}K \vdash \tau \leq \tau'}{\Delta \vdash \forall\alpha{:}K.\tau \leq \forall\alpha{:}K.\tau'} \qquad \frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T} \quad \Delta \vdash c : K}{\Delta \vdash \tau[c/\alpha] \leq \exists\alpha{:}K.\tau}$$

$$\frac{\Delta, \alpha{:}K \vdash \tau : \mathsf{T} \quad \Delta \vdash c : K}{\Delta \vdash \forall\alpha{:}K.\tau \leq \tau[c/\alpha]} \qquad \frac{\Delta, \alpha{:}K \vdash \tau \leq \tau'}{\Delta \vdash \exists\alpha{:}K.\tau \leq \exists\alpha{:}K.\tau'}$$

$$\frac{(\alpha \notin \tau)}{\Delta \vdash \tau \leq \forall\alpha{:}K.\tau} \qquad \frac{(\alpha \notin \tau)}{\Delta \vdash \exists\alpha{:}K.\tau \leq \tau} \qquad \frac{\Delta \vdash \tau : \mathsf{T}i}{\Delta \vdash \tau \leq \mathtt{ns}^i}$$

$$\frac{}{\Delta \vdash \tau_1 \times (\tau_2 \times \tau_3) \leq (\tau_1 \times \tau_2) \times \tau_3} \qquad \frac{\Delta \vdash t : \mathsf{N}}{\Delta \vdash \mathcal{S}(t) \leq \mathtt{BW}}$$

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

$$\frac{}{\Delta \vdash \mathtt{B0} \times \tau \leq \tau} \qquad \frac{}{\Delta \vdash \tau \leq \tau \times \mathtt{B0}} \qquad \frac{}{\Delta \vdash \tau \times \mathtt{B0} \leq \tau}$$

$$\frac{}{\Delta \vdash \mathtt{ns}(i+j) \leq \mathtt{ns}i \times \mathtt{ns}j} \qquad \frac{\Delta \vdash t_1 = t_2 \; \mathsf{true}}{\Delta \vdash \mathcal{S}(t_1) \leq \mathcal{S}(t_2)}$$

$$\frac{\Delta \vdash \Gamma' \leq \Gamma}{\Delta \vdash \Gamma \to 0 \leq \Gamma' \to 0} \qquad \frac{\Delta \vdash \tau : \mathsf{T} \quad \Delta \vdash \varphi \; \mathsf{true}}{\Delta \vdash (\varphi \Rightarrow \tau) \leq \tau}$$

$$\frac{\Delta \vdash \varphi : \mathsf{P}}{\Delta \vdash \tau \leq (\varphi \Rightarrow \tau)} \qquad \frac{\Delta, \varphi \; \mathsf{true} \vdash \tau \leq \tau'}{\Delta \vdash (\varphi \Rightarrow \tau) \leq (\varphi \Rightarrow \tau')}$$

Register File Subtyping ($\Delta \vdash \Gamma \leq \Gamma'$)

$$\frac{\begin{array}{c}\Delta \vdash \tau \leq \tau' \quad \Delta \vdash t' \leq t \; \mathsf{true} \\ \Delta \vdash \tau_r \leq \tau_r' \text{ for each register } r\end{array}}{\begin{array}{c}\Delta \vdash \{\mathtt{eax}{:}\tau_{\mathtt{eax}}, \ldots, \mathtt{ebp}{:}\tau_{\mathtt{ebp}}, \mathtt{esp}{:}\tau, \mathtt{ck}{:}t\} \\ \leq \{\mathtt{eax}{:}\tau'_{\mathtt{eax}}, \ldots, \mathtt{ebp}{:}\tau'_{\mathtt{ebp}}, \mathtt{esp}{:}\tau', \mathtt{ck}{:}t'\}\end{array}}$$

Operand Typing ($\Delta; \Psi; \Gamma \vdash o : \tau$)

$$\frac{}{\Delta; \Psi; \Gamma \vdash B : \mathcal{S}(\overline{B})} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash \ell : \Psi(\ell)} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash r : \Gamma(r)}$$

$$\frac{\begin{array}{c}\Delta; \Psi; \Gamma \vdash o : \mathtt{sptr}(\tau_1 \times \tau_2 \times \tau_3) \\ \Delta \vdash \tau_1 : \mathsf{T}n \quad \Delta \vdash \tau_2 : \mathsf{T}m \quad \Delta \vdash \Gamma(\mathtt{esp}) \leq \tau \times \tau_1 \times \tau_2 \times \tau_3\end{array}}{\Delta; \Psi; \Gamma \vdash m^{\mathsf{c}}[o+n] : \tau_2}$$

$$\frac{}{\Delta; \Psi; \Gamma \vdash \mathtt{esp} : \mathtt{sptr}(\Gamma(\mathtt{esp}))} \qquad \frac{\Delta; \Psi; \Gamma \vdash o : \tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta; \Psi; \Gamma \vdash o : \tau}$$

Destination Propagation ($\Delta; \Psi; \Gamma \vdash d : \tau \to \Gamma'$)

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

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

Instruction Typing ($\Psi; \Delta; \Gamma \vdash I$)

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

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

$$\frac{(\Gamma(\mathtt{ck}) = \overline{1} + t) \quad \Delta; \Psi; \Gamma \vdash o : (\Gamma\{\mathtt{ck}{:}t\}) \to 0}{\Delta; \Psi; \Gamma \vdash \mathtt{jmp}\ o} \qquad \frac{(\Gamma(\mathtt{ck}) = \overline{1} + t) \quad \Delta; \Psi; \Gamma\{\mathtt{esp}{:}\mathtt{ns}^n \times \Gamma(\mathtt{esp}), \mathtt{ck}{:}t\} \vdash I}{\Delta; \Psi; \Gamma \vdash \mathtt{salloc}\ n\ I}$$

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

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

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

$$\frac{\begin{array}{c}\Delta; \Psi; \Gamma \vdash o_1 : \mathtt{B4} \quad \Delta; \Psi; \Gamma \vdash o_2 : \mathtt{B4} \\ \Delta; \Psi; \Gamma\{\mathtt{ck}{:}t\} \vdash I \quad (\Gamma(\mathtt{ck}) = \overline{1} + t)\end{array}}{\Delta; \Psi; \Gamma \vdash \mathtt{cmp}\ o_1, o_2\ I}$$

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

$$\frac{\begin{array}{c}(\Gamma(\mathtt{ck}) = \overline{1} + t) \quad \Delta; \Psi; \Gamma \vdash o : \tau \\ \Delta \vdash \tau : \mathsf{T} \quad \Delta; \Psi; \Gamma\{\mathtt{esp}{:}\tau \times \Gamma(\mathtt{esp}), \mathtt{ck}{:}t\} \vdash I\end{array}}{\Delta; \Psi; \Gamma \vdash \mathtt{push}\ o\ I}$$

$$\frac{\begin{array}{c}(\Gamma(\mathtt{ck}) = \overline{1} + t) \\ \Delta \vdash \Gamma(\mathtt{esp}) \leq \tau_1 \times \tau_2 \quad \Delta \vdash \tau_1 : \mathsf{T}n \\ \Delta \vdash \tau_2 : \mathsf{T} \quad \Delta; \Psi; \Gamma\{\mathtt{esp}{:}\tau_2, \mathtt{ck}{:}t\} \vdash I\end{array}}{\Delta; \Psi; \Gamma \vdash \mathtt{sfree}\ n\ I} \qquad \frac{\Delta; \Psi; \Gamma\{\mathtt{ck}{:}\overline{Y}\} \vdash I}{\Delta; \Psi; \Gamma \vdash \mathtt{yield}\ I}$$

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

$$\frac{\begin{array}{c}\Gamma(r) = \exists\alpha{:}K.\tau \\ (\Delta, \alpha{:}K); \Psi; \Gamma\{r{:}\tau\} \vdash I\end{array}}{\Delta; \Psi; \Gamma \vdash I} \qquad \frac{\Delta; \Psi; \Gamma' \vdash I \quad \Delta \vdash \Gamma \leq \Gamma'}{\Delta; \Psi; \Gamma \vdash I}$$

Block and Program Typing ($\Psi; \Delta \vdash I : \tau \; \mathsf{block}, \vdash P$)

$$\frac{\Psi; (\Delta, \varphi \; \mathsf{true}) \vdash I : \tau \; \mathsf{block}}{\Psi; \Delta \vdash I : \varphi \Rightarrow \tau \; \mathsf{block}} \qquad \frac{\Psi; \Delta; \Gamma \vdash I}{\Psi; \Delta \vdash I : \Gamma \to 0 \; \mathsf{block}}$$

$$\frac{\Psi; (\Delta, \alpha{:}K) \vdash I : \tau \; \mathsf{block}}{\Psi; \Delta \vdash I : \forall\alpha{:}K.\tau \; \mathsf{block}} \qquad \frac{\begin{array}{c}(\Psi = \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}) \\ \vdash \Psi \quad (\tau_1 = \{\mathtt{esp}{:}\mathtt{B0}, \mathtt{ck}{:}\overline{Y}\} \to 0) \\ \Psi; \cdot \vdash I_i : \tau_i \; \mathsf{block} \text{ for } 1 \leq i \leq n\end{array}}{\vdash \ell_1{:}\tau_1 = I_1, \ldots, \ell_n{:}\tau_n = I_n}$$