A Linearly Typed Assembly Language

James Cheney Greg Morrisett
Cornell University
Ithaca, NY 14853

Abstract

Today's type-safe low-level languages rely on garbage collection to recycle heap-allocated objects safely. We present LTAL, a safe, low-level, yet simple language that "stands on its own": it guarantees safe execution within a fixed memory space, without relying on external run-time support. We demonstrate the expressiveness of LTAL by giving a type-preserving compiler for the functional core of ML. But this independence comes at a steep price: LTAL's type system imposes a draconian discipline of linearity that ensures that memory can be reused safely, but prohibits any useful kind of sharing. We present the results of experiments with a prototype LTAL system that show just how high the price of linearity can be.

1 Background and Motivation

Safety certification systems such as Java or MSIL bytecode verification make it possible to verify the safety of code obtained from an untrusted provider or over an untrusted network [13, 7]. Refinements like proof-carrying code and typed assembly language [19, 18] make it possible to check and execute machine code directly rather than through interpretation. However, all widely used systems for verifying low-level code require a trusted run-time environment to provide safe memory management. Furthermore, each of these systems takes a rather *ad hoc* approach to initialization of heap-allocated objects.

Recently, Wang and Appel [32] and Monnier, Saha, and Shao [16] have shown how to build a type-safe garbage collector based on the ideas of the Capability Calculus [29], thus eliminating most of the memory management from the trusted computing base. However, some trusted code is still required to implement the region primitives, and the region calculi on which these systems are based are relatively complicated. Furthermore, the region-based approaches do not address the initialization problem.

In this paper, we step back and examine a more foundational approach to the issues of initialization and memory management for low-level code. In particular, we use a *linear* type system to provide a clean, elegant solution to both problems. More specifically, we present:

1. A linear type system for a conventional, MIPS-style assembly language called LTAL.
2. Theorems that show every well-typed LTAL program is sound and "leak-free" (*i.e.*, uses bounded memory).
3. An encoding of memory management operations `malloc` and `free` *within* LTAL.
4. Techniques for compiling unrestricted (*i.e.*, non-linear) high-level functional languages to LTAL in a type-preserving fashion.

It is important to note that we do not consider the resulting system to be practical. The price of LTAL's simplicity and elegance is that it does not support shared data structures. At first, such a restriction seems to preclude the use of LTAL as a target for high-level languages. However, we show that there is a type-preserving translation for a high-level ML-like language to LTAL based on explicit copying. Unfortunately, our experiments show that this naive translation is far

from practical. Nonetheless, we think that LTAL can serve as an important core for more realistic systems.

Of course, the idea of employing linearity is not new—many researchers have proposed linear languages and implementation techniques for implementing functional languages without garbage collection or using bounded space [12, 4, 8, 11]. But none of these approaches carry type information all the way down to a realistic assembly language as we do. Recently, Aspinall and Compagnoni [2] have developed Heap Bounded Assembly Language (HBAL), a variant of TAL that employs linearity to guarantee finite heap usage with direct memory management. HBAL was tailored to serve as a safe target language for Hofmann's first-order linear functional programming language LFPL [11]. However, HBAL includes many pseudo-instructions for memory and data structure management and assumes the presence of an unbounded stack, but supports neither polymorphism nor higher-order functions.

Previous proposals for typed resource-conscious intermediate and low-level languages include Walker, Crary, and Morrisett's Capability Calculus [29], Smith, Walker and Morrisett's Alias Types [25, 30], and Walker and Watkins' linear region calculus [31]. These systems are clearly more powerful than LTAL while permitting some form of direct control over memory. However, all these techniques are disappointingly complex: each involves some combination of type-level named memory locations, singleton name types, and bounded quantification. Yet none of the above provide an unconditional guarantee of safety: all rely on some outside run-time support for memory management, such as a trusted implementation of regions. In contrast, with LTAL we aim for simplicity while still obtaining strong memory management and safety guarantees.

In the remainder of this paper, we first give an overview of the LTAL language, emphasizing the departures from TAL. Section 3 describe a simple compiler from a *non-linear*, ML-like functional language to LTAL via a linear intermediate language. We summarize the proofs of the relevant soundness and memory preservation properties of LTAL in Section 4. Section 5 describes several extensions to LTAL such as polymorphism, recursion, and datatypes, as well as non-extensions like references and laziness, at an informal level. In Section 6 we describe an implementation of a typechecker for LTAL and a compiler based on the translation in Section 3. which serves as a proof of concept, and present microbenchmark results. Finally, we give an overview of related work and future directions for safe low-level memory management.

2 Overview of Linear TAL

2.1 Syntax

The syntax for LTAL code is as follows:

$$
\begin{array}{rrcl}
\text{operands} & op & ::= & r \mid i \mid f \\
\text{instructions} & \iota & ::= & \texttt{add}\ r, r', op \mid \texttt{bnz}\ r, op \mid \texttt{ld}\ r, s[i] \mid \texttt{mov}\ r, op \mid \\
 & & & \texttt{mul}\ r, r', op \mid \texttt{st}\ r[i], r' \mid \texttt{sub}\ r, r', op \\
\text{blocks} & I & ::= & \iota; I \mid \texttt{jmp}\ op \mid \texttt{halt}
\end{array}
$$

Operands include register names r, integer values i, and code labels f. The instructions are a representative subset of MIPS assembly language with the usual interpretation. For instance, $\texttt{ld}\ r, s[i]$ loads the word in memory at the effective address computed by adding the contents of register s with the offset i, and places the word into the destination register r. We give a formal operational semantics for this instruction and the others in Section 2.2 where we introduce a suitable abstract machine. Following TAL, we group instructions into blocks, which are `jmp`- or `halt`-terminated sequences of instructions.

LTAL does not include any pseudo-instructions such as `alloc`, `free`, `cons`/`nil`, or `case`. LTAL programs also cannot refer to heap data (as opposed to code) via labels. Some support for global data can be added to LTAL programs, with the references to linear types do not "escape" to global types (see Section 5.1.5).

2.2 Operational semantics

We call the sets of integer constants, register names, data labels, code labels, and instruction blocks by the names Int, Reg, Lab, $CLab$, and $Block$ respectively. We write $A \uplus B$ for disjoint union, and $A \rightharpoonup B$ for (finite) partial maps from A to B. If F is a partial map, we write $F\{x \mapsto y\}$ for the partial map resulting from updating F at x to y, $F\#G$ to indicate that $dom(F) \cap dom(G) = \emptyset$, and $F \uplus G$ for $F \cup G$ if $F\#G$ holds. The other components of the operational semantics are defined as follows:

$$\begin{array}{rrcll} \text{values} & v & \in & Val & = & Int \uplus CLab \uplus Lab \\ \text{heap values} & h & \in & HVal & = & \{0,1\} \to Val \\ \text{heaps} & H & \in & Heap & = & Lab \rightharpoonup HVal \\ \text{register files} & R & \in & RegFile & = & Reg \rightharpoonup Val \\ \text{code sections} & C & \in & CodeSec & = & CLab \rightharpoonup Block \end{array}$$

Given an operand op (that is, a register, code label, or integer), we write $\hat{R}(op)$ for the value of op in register file R.

The operational semantics of LTAL is essentially the same as that of TAL, except for the omission of a built-in `malloc` instruction. A program state P is a triple (H, R, I) consisting of the current heap, current register file contents, and current remaining instruction sequence. We write $P \longrightarrow_C P'$ to indicate that a program with code section C steps from state P to P' in one step. This relation is defined by:

$(H, R, I) \longrightarrow_C P$ where	
if $I =$	then $P =$
`add` $r, s, op; I'$	$(H, R\{r \mapsto R(s) + \hat{R}(op)\}, I')$
`bnz` $r, op; I'$	$\begin{cases} (H, R, I') & \text{if } R(r) = 0 \\ (H, R, C(\hat{R}(op))) & \text{if } R(r) \neq 0 \end{cases}$
`ld` $r, s[i]; I'$	$(H, R\{r \mapsto H(R(s))[i]\}, I')$
`mov` $r, op; I'$	$(H, R\{r \mapsto \hat{R}(op)\}, I')$
`mul` $r, s, op; I'$	$(H, R\{r \mapsto R(s) * \hat{R}(op)\}, I')$
`st` $r[i], s; I'$	$(H\{R(r) \mapsto H(R(r))\{i \mapsto R(s)\}\}, R, I')$
`sub` $r, s, op; I'$	$(H, R\{r \mapsto R(s) - \hat{R}(op)\}, I')$
`jmp` op	$(H, R, C(\hat{R}(op)))$

None of the instructions affect the domains of the heap, register file, or code section. That is, memory, code, and registers are neither created nor destroyed during execution.

A program state P is *stuck* if the instruction sequence remaining is not `halt` and there is no transition that the program can take. This can only happen in two ways. First, a register, code label, or data label may not be in the domain of the register file, code section, or data section. In a real machine, this would result in a hardware exception or memory protection fault. Second, the contents of a register or memory location may be of an unexpected sort. In a real machine, the sorts may not be distinguishable at run time, so the program might put memory into an inconsistent state rather than failing immediately. Thus, stuckness in our abstract machine corresponds to undesirable behavior in real machines.

2.3 Type system

LTAL includes *operand types* ω, τ which describe the contents of a register or instruction operand.

$$\begin{array}{rrcl} \text{unrestricted types} & \tau & ::= & \forall\alpha.\tau \mid \mathsf{int} \mid \mathsf{word} \mid \mathtt{code}(\Gamma) \\ \text{operand types} & \omega & ::= & \alpha \mid \tau \mid ?\langle\sigma\rangle \mid @\langle\sigma\rangle \mid \exists\alpha.\omega \mid \mu\alpha.\omega \\ \text{memory types} & \sigma & ::= & \omega_1 \otimes \omega_2 \\ \text{register contexts} & \Gamma & ::= & \{r_1 : \omega_1, \ldots, r_n : \omega_n\} \end{array}$$

Machine words interpreted as integers have type int. The cell type word indicates an uninterpreted machine word. The difference between int and word is that only int may be used in arithmetic or

conditional branch instructions; word values may *only* be overwritten. Words interpreted as code addresses have types of the form $\forall\overline{\alpha}.\texttt{code}(\Gamma)$, where Γ is a register file typing context. Such a type indicates that a value is an address which can be called when the current register context matches Γ, for appropriate instantiations of the type variables $\overline{\alpha}$. Code labels of procedures typically include a register with another code label type, indicating that the register contains the return address. These types are *unrestricted*; that is, their values may be copied or ignored (that is, cast to word) as desired.

The restricted (or linear) types include type variables, reference types, existential, and recursive types. Type variables are restricted because they might be instantiated with restricted types. References come in two flavors: $@\langle\sigma\rangle$ indicates a reference to a memory block having type σ, and $?\langle\sigma\rangle$ indicates a reference that might be NULL (that is, zero). Memory types include only simple pairs, written using $\otimes$ to emphasize their linear nature. Existential types have their usual form, except that abstract types are restricted. Recursive types are also standard. We do not include explicit pack/unpack or roll/unroll forms; instead we leave type reconstruction or annotation design choices to implementations.

An LTAL program is a collection of labeled blocks $C = \{l_1 = I_1, \ldots, l_n = I_n\}$ together with a code typing context $\Psi = \{l_1 : \tau_i, \ldots, l_n : \tau_n\}$. As in TAL, we typecheck a program by typechecking each block against the context of all blocks, and typecheck a block by updating the typing context with the effect of each instruction on the register types. Here we present some special cases of LTAL's typing judgments in a simplified form. We write $\{\Gamma\}i\{\Gamma'\}$ to indicate that in register context Γ, instruction i is well-formed and changes the context to Γ'. The general rules are given in the appendix. The typing rules for arithmetic instructions are straightforward:

$$\{\Gamma, r_d : \tau, r_s : \mathsf{int}, r_t : \mathsf{int}\} \quad \texttt{arith}\ r_d, r_s, r_t \quad \{\Gamma, r_d : \mathsf{int}, r_s : \mathsf{int}, r_t : \mathsf{int}\}$$

where `arith` denotes one of `add`, `mul`, `sub`. The source registers must be of type int and the target register must have some unrestricted type (and in this case may be one of the source registers). After the operation, all three registers have type int.

The control flow instruction typings are also similar to those for TAL. When we encounter a jump, we check if the branch target has a type that matches the current context, possibly modified to take into account any new type information arising from a conditional branch. It is possible to perform a `bnz` instruction with either a word or reference type as an argument. Branching on a reference type lets us distinguish at run-time whether a nullable reference is actually null or not.

$$\begin{array}{rcl} \{\Gamma, r : \mathsf{int}, s : \texttt{code}\ \Gamma\} & \texttt{bnz}\ r, s & \{\Gamma\} \\ \{\Gamma, r : ?\langle\sigma\rangle, s : \texttt{code}(\Gamma, r : @\langle\sigma\rangle)\} & \texttt{bnz}\ r, s & \{\Gamma, r : \mathsf{int}\} \end{array}$$

The main novelty is in the typing of `mov` instructions and memory accesses `ld`, `st`. For move instructions, we have

$$\{\Gamma, r_1 : \mathsf{word}, r_2 : \omega\} \quad \texttt{mov}\ r_1, r_2 \quad \{\Gamma, r_1 : \omega, r_2 : \mathsf{word}\}$$

if ω is linear (otherwise $r_2 : \omega$ still after the move). Loads must load into a register with unrestricted type, and render the loaded component of the memory cell unusable as an alias by giving it type word.

$$\begin{array}{lcl} \{\Gamma, r_1 : \mathsf{word}, r_2 : @\langle\omega_0 \otimes \omega_1\rangle\} & \texttt{ld}\ r_2[0], r_1 & \{\Gamma, r_1 : \omega_0, r_2 : @\langle\mathsf{word} \otimes \omega_1\rangle\} \\ \{\Gamma, r_1 : \omega_0, r_2 : @\langle\mathsf{word} \otimes \omega_1\rangle\} & \texttt{st}\ r_1, r_2[0] & \{\Gamma, r_1 : \mathsf{word}, r_2 : @\langle\omega_0 \otimes \omega_1\rangle\} \end{array}$$

Although these rules swap the *types* of the memory cell and register, the actual operational behavior does not swap the *values*; that is, the behavior of the move, load and store instructions is as usual. Only the typing forbids reusing the source value. We could clearly introduce a more CISC-like `swap` instruction subsuming the behavior of `ld` and `st` as well as supporting exchange of types besides word.

Previous versions of TAL included initialization flags on types, junk values, and a simple subtyping system to bridge the gap between allocation and initialization. In LTAL, this machinery is

not necessary. Instead, uninitialized-but-allocated memory cells have type word, which can only be initialized.[1]

Using recursive types we can define a *freelist* type $\mathsf{flist} = \mu\alpha.?\langle\mathsf{word} \otimes \alpha\rangle$, and we can *implement* allocation and deallocation operations rather than taking them as primitive. For example, the following code constructs the pair $\langle 1, 2\rangle$, allocating from the freelist, terminating the program if no more memory is available:

$$\begin{array}{rl} & \{r_0 : \mathsf{word}, r_f : \mathsf{flist}\} \\ & \texttt{bnz}\ r_f, l_1 \\ & \texttt{halt} \\ l_1 : & \{r_0 : \mathsf{word}, r_f : @\langle\mathsf{word} \otimes \mathsf{flist}\rangle\} \\ & \texttt{mov}\ r_0, r_f \\ & \texttt{ld}\ r_f, r_0[1] \\ & \texttt{st}\ r_0[0], 1 \\ & \texttt{st}\ r_0[1], 2 \\ & \{r_0 : @\langle\mathsf{int} \otimes \mathsf{int}\rangle, r_f : \mathsf{flist}\} \end{array}$$

In typing the branch instruction, we first unroll the definition of flist to $?\langle\mathsf{word} \otimes \mathsf{flist}\rangle$ in order to expose the nullable reference. The type of l_1 reflects the change in the typing information arising from the fact that r_f is non-NULL. The rest of the block containing the branch is typed with $r_f : \mathsf{word}$. Frequently, allocation happens right after deallocation, and we can optimize away the branch since we can tell from the type that it will always succeed.

3 Compiling to Linear TAL

We begin with a primitive source language, the call-by-value, simply-typed λ-calculus with integers and pairs (abbreviated $\lambda^{\to\times}$). The syntax of the language is given by:

$$\begin{array}{rrcl} \text{types} & \tau & ::= & \mathsf{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\ \text{terms} & e & ::= & x \mid i \mid \lambda x{:}\tau.e \mid e_1\, e_2 \mid \langle e_1, e_2\rangle \mid \pi_1\, e \mid \pi_2\, e \\ \text{values} & v & ::= & i \mid \lambda x{:}\tau.e \mid \langle v_1, v_2\rangle \\ \text{contexts} & \Gamma & ::= & \bullet \mid \Gamma, x{:}\tau \end{array}$$

where i ranges over integer constants, and x ranges over a denumerable set of variables. As usual, the free occurrences of x within e are bound in $\lambda x{:}\tau.e$, and we consider terms to be the same modulo α-equivalence. We write $e[e_1/x]$ for the capture-avoiding substitution of e_1 for the free occurrences of x within e. We omit integer operations and conditionals, but they can easily be added.

A large-step semantics for the programming language is given by the following rules which define a judgment $e \Downarrow v$ meaning the expression e evaluates to the value v.

$$v \Downarrow v \qquad \frac{e \Downarrow \langle v_1, v_2\rangle}{\pi_1\, e \Downarrow v_1} \qquad \frac{e \Downarrow \langle v_1, v_2\rangle}{\pi_2\, e \Downarrow v_2}$$

$$\frac{e_1 \Downarrow \lambda x{:}\tau.e \qquad e_2 \Downarrow v_2 \qquad e[v_2/x] \Downarrow v}{e_1\, e_2 \Downarrow v}$$

The typing judgment $\Gamma \vdash e : \tau$ is defined below.

$$\Gamma \vdash x : \Gamma(x) \qquad \Gamma \vdash i : \mathsf{int}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, e : \tau_i} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

[1] Arguably, initialization flags and subtyping have not really disappeared, only been replaced by slightly cleaner implementations: initialization flags and junk values the word type, and general register-file subtyping by rules like $\Delta; \Gamma, r : \tau \vdash r : \mathit{cell}$. We are willing to concede this point, but believe that it is still an improvement

3.1 Linear Closure Conversion

We introduce a linear intermediate language (LIL) whose type system requires precise accounting for dynamically allocated data structures. In particular, values of pair type cannot be freely duplicated or forgotten. Rather, each such value must be used exactly once. However, integer or function values may be copied and forgotten without restrictions. This corresponds to the fact that we do not need to dynamically allocate space for primitive values that fit into registers, only for pairs.

Following Minamide et al. [15], we represent closures as a data structure consisting of closed code and an environment. The type of the environment is held abstract using an existential to ensure that the translation of closures is uniform. Source-level closures with the same type might have different environments, so we use an existential type to abstract these environment types so that the target language, which makes environments explicit, can have a uniform type for the closures.

The source language supports arbitrary duplication or forgetting of resources such as pairs or closures. Our translation to the linear intermediate language realizes this by explicitly copying or deallocating data structures as needed. For instance, building $\langle x, x\rangle : (\mathsf{int} \otimes \mathsf{int}) \otimes (\mathsf{int} \otimes \mathsf{int})$ from $x : \mathsf{int} \otimes \mathsf{int}$ involves doing a deep copy of x and then constructing the desired pair from the resulting *separate* copies x_1 and x_2. Unrestricted types such as integers and functions can be copied and freed without restriction.

In the absence of abstract types (type variables), we can copy (or free) a data structure by crawling over it in a type-directed fashion. However, this no longer works when we need to copy a value of abstract type. Although our source language does not have abstract types, it does have closures and our translation introduces abstract types to give a uniform translation for closure environments. Therefore, we augment each closure with two additional methods, one for copying the environment, and one for deallocating the environment.

The syntax for the linear intermediate language is given by:

$$\begin{array}{rcl}
\tau & ::= & \mathsf{unit} \mid \mathsf{int} \mid \sigma_1 \Rightarrow \sigma_2 \\
\sigma & ::= & \alpha \mid \tau \mid \sigma_1 \otimes \sigma_2 \mid \exists\alpha.\sigma \\
e & ::= & \langle\rangle \mid i \mid x \mid \lambda x{:}\sigma.e \mid e_1\, e_2 \mid \langle e_1, e_2\rangle \\
 & \mid & \mathtt{pack}[\sigma, e]\ \mathtt{as}\ \sigma' \mid \mathtt{let}\ [\alpha, x]\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \\
 & \mid & \mathtt{let}\ \langle x_1, x_2\rangle\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \mid \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \\
v & ::= & \langle\rangle \mid i \mid \lambda x{:}\sigma.e \mid \langle v_1, v_2\rangle \mid \mathtt{pack}[\sigma, v]\ \mathtt{as}\ \exists\alpha.\sigma'
\end{array}$$

We abbreviate $\mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2$ by $e_1; e_2$ when x is not free in e_2.

Types are split into unrestricted types unit, int, and function types, which are not associated with tracked memory resources, versus linear types α, $\sigma \otimes \sigma'$, $\exists\alpha.\sigma$ which may be. These in turn correspond to restricted or linear types at the LTAL level. The function type $\sigma_1 \Rightarrow \sigma_2$ deserves explanation. It is neither linear implication $\multimap$ nor intuitionistic implication $\to$. Instead, it denotes a function which must use its argument linearly, but is itself reusable. Thus, in linear logic terms, $\sigma_1 \Rightarrow \sigma_2 \cong !(\sigma_1 \multimap \sigma_2)$. Therefore, $\Rightarrow$ functions must be (linearly) *closed*; that is, they may only refer to other closed functions and their linear argument.[2]

The operational semantics is similar to that for the source language, but includes rules for `let` forms:

$$\frac{e_1 \Downarrow v' \quad e_2[v'/x] \Downarrow v}{\mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow \langle v_1, v_2\rangle \quad e_2[v_1/x_1][v_2/x_2] \Downarrow v}{\mathtt{let}\ \langle x_1, x_2\rangle\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow \mathtt{pack}[\sigma, v']\ \mathtt{as}\ \exists\alpha.\sigma' \qquad e_2[\sigma/\alpha][v'/x] \Downarrow v}{\mathtt{let}\ [\alpha, x]\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \Downarrow v}$$

The type system for the intermediate language ensures that linear values are used exactly once. Judgments for terms are of the form $\Delta; \Gamma \vdash e : \sigma$ where Δ is the set of type variables in scope, and Γ is the set of variables in scope together with their types. Contexts are again order-sensitive.

[2] Actually, it would also be safe to permit closed functions to refer to global data of unrestricted type, but in the current system this seems to be of limited utility.

$$\Delta; \bullet \vdash \langle\rangle : \mathsf{unit} \qquad \Delta; \bullet \vdash i : \mathsf{int}$$

$$\frac{FTV(\sigma) \subseteq \Delta}{\Delta; \bullet, x{:}\sigma \vdash x : \sigma} \qquad \frac{\Delta; \Gamma \vdash x : \sigma \qquad FTV(\tau) \subseteq \Delta}{\Delta; \Gamma, y{:}\tau \vdash x : \sigma}$$

$$\frac{\bullet; \Psi, x{:}\sigma_1 \vdash e : \sigma_2}{\bullet; \Psi \vdash \lambda x{:}\sigma_1.e : \sigma_1 \Rightarrow \sigma_2}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : \sigma_1 \Rightarrow \sigma_2 \qquad \Delta; \Gamma_2 \vdash e_2 : \sigma_1}{\Delta; \Gamma_1 \bowtie \Gamma_2 \vdash e_1\, e_2 : \sigma_2}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : \sigma_1 \qquad \Delta; \Gamma_2 \vdash e_2 : \sigma_2}{\Delta; \Gamma_1 \bowtie \Gamma_2 \vdash \langle e_1, e_2\rangle : \sigma_1 \otimes \sigma_2}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : \sigma_1 \otimes \sigma_2 \qquad \Delta; \Gamma_2, x_1{:}\sigma_1, x_2{:}\sigma_2 \vdash e_2 : \sigma}{\Delta; \Gamma_1 \bowtie \Gamma_2 \vdash \mathtt{let}\ \langle x_1, x_2\rangle\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : \sigma}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma[\sigma_1/\alpha]}{\Delta; \Gamma \vdash \mathtt{pack}[\sigma_1, e]\ \mathtt{as}\ \exists\alpha.\sigma : \exists\alpha.\sigma}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : \exists\alpha.\sigma_1 \quad \Delta, \alpha; \Gamma_2, x{:}\sigma_1 \vdash e_2 : \sigma \quad \alpha \notin FTV(\sigma)}{\Delta; \Gamma_1 \bowtie \Gamma_2 \vdash \mathtt{let}\ [\alpha, x]\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : \sigma}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : \sigma_1 \qquad \Delta; \Gamma_2, x{:}\sigma_1 \vdash e_2 : \sigma}{\Delta; \Gamma_1 \bowtie \Gamma_2 \vdash \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : \sigma}$$

Figure 1: Type system for LIL

We use $\Gamma_1 \bowtie \Gamma_2$ to mean for all $x \in Dom(\Gamma_1)$ either (a) $x \notin Dom(\Gamma_2)$ or (b) if $x \in Dom(\Gamma_2)$ then there exists an τ such that $\Gamma_1(x) = \Gamma_2(x) = \tau$. In other words, the "$\bowtie$" operation allows duplication of variables that have unrestricted type. Weakening for unrestricted assumptions is made explicit through an additional rule. We write Ψ for a context that contains only function-typed variables.

Figure 1 gives the typing rules for LIL. The rule for code types ensures that the code is closed (*i.e.*, only depends upon its parameter and other functions in scope). The other rules are variants of typical linear type rules.

The type translation used by the source-to-LIL translation is shown in Figure 2. A source level closure of type $\tau_1 \to \tau_2$ is translated to a target-level term of consisting of four components:

1. an environment whose type is held abstract (α),
2. an "apply" method which takes the argument and environment and produces the result ($\mathcal{T}_1[\![\tau_1]\!] \otimes \alpha \Rightarrow \mathcal{T}_1[\![\tau_2]\!]$),
3. a "copy" method which consumes the environment and produces two copies ($\alpha \Rightarrow \alpha \otimes \alpha$), and
4. a "free" method which consumes the environment and returns linear unit ($\alpha \Rightarrow \mathsf{unit}$).

We lift the type translation to type assignments as follows:

$$\mathcal{T}_1[\![\Gamma]\!] = \mathcal{T}_1[\![|\Gamma|]\!]$$

where the auxiliary definition $|\Gamma|$ is given by:

$$\begin{array}{rcl} |\bullet| & = & \mathsf{unit} \\ |\Gamma, x{:}\tau| & = & \tau \times |\Gamma| \end{array}$$

The term translation is given in Figure 3.

The translation assumes that a distinguished variable `env` refers to a data structure that holds the values of the free variables needed to evaluate the expression. We forbid reordering of the source type environment Γ during translation. The resulting translated expression produces a value of type $\mathcal{T}_1[\![\tau]\!]$ together with a copy of the environment. The critical invariant is

$$
\begin{array}{rcl}
\mathcal{T}_1[\![\mathsf{int}]\!] & = & \mathsf{int} \\
\mathcal{T}_1[\![\tau_1 \times \tau_2]\!] & = & \mathcal{T}_1[\![\tau_1]\!] \otimes \mathcal{T}_1[\![\tau_2]\!] \\
app(\sigma_1, \sigma_2, \sigma) & = & \sigma_1 \otimes \sigma \Rightarrow \sigma_2 \\
copy(\sigma) & = & \sigma \Rightarrow \sigma \otimes \sigma \\
free(\sigma) & = & \sigma \Rightarrow \mathsf{unit} \\
\mathcal{T}_1[\![\tau_1 \to \tau_2]\!] & = & \exists\alpha.(\alpha \otimes (app(\mathcal{T}_1[\![\tau_1]\!], \mathcal{T}_1[\![\tau_2]\!], \alpha) \otimes \\
 & & \quad (copy(\alpha) \otimes free(\alpha))))
\end{array}
$$

Figure 2: $\lambda^{\to\times}$ to LIL type translation

if $\sigma_\Gamma = \mathcal{T}_1[\![\Gamma]\!]$, $e' = \mathcal{E}_1[\![\Gamma \vdash e : \tau]\!]$, and $\sigma = \mathcal{T}_1[\![\tau]\!]$, then $\mathtt{env}{:}\sigma_\Gamma \vdash e' : \sigma \otimes \sigma_\Gamma$.

The term translation depends on the auxiliary meta-level functions Copy and Free, shown in Figure 4, which are defined by induction on the structure of source-language types. These functions make the implicit reuse and discarding of context components in the source language explicit as copying and deletion in the linear language.

3.2 Correctness

The proofs of type and semantic correctness of the translation to LIL draw on the proofs of similar results for Minamide et al.'s typed closure conversion translations [15].

Lemma 3.1. *If $\Gamma \vdash e : \tau$ is derivable, then $\mathcal{E}_1[\![\Gamma \vdash e : \tau]\!]$ exists and $\mathtt{env}{:}\mathcal{T}_1[\![\Gamma]\!] \vdash \mathcal{E}_1[\![\Gamma \vdash e : \tau]\!] : \mathcal{T}_1[\![\tau]\!] \otimes \mathcal{T}_1[\![\Gamma]\!]$.*

Proof. Straightforward induction on the derivation. □

To prove semantic equivalence, we define several type-indexed simulation relations between source terms and substitutions on one hand, and target terms on the other.

Definition 3.2. *Define ground simulation relations $\approx_\tau$ relating closed source and target values and $\sim_\tau$ relating closed source and target expressions as follows:*

- $e \sim_\tau e'$ *iff* $e \Downarrow v$ *and* $e' \Downarrow v'$ *and* $v \approx_\tau v'$
- $i \approx_{\mathsf{int}} i$
- $\langle v_1, v_2\rangle \approx_{\tau_1\times\tau_2} \langle v_1', v_2'\rangle$ *iff* $v_i \approx_{\tau_i} v_i'$
- $v \approx_{\tau_1\to\tau_2} v'$ *iff*
  - *for all* $v_1 \approx_{\tau_1} v_2$, *we have* $v\ v_1 \sim_{\tau_2} \mathsf{App}(v', v_2)$
  - $\mathsf{Copy}[\![\tau_1 \to \tau_2]\!](v') \Downarrow \langle v', v'\rangle$
  - $\mathsf{Free}[\![\tau_1 \to \tau_2]\!](v') \Downarrow \langle\rangle$

Because our translation makes the environment explicit, we need an unusual simulation relation $\gamma \Vdash e \sim_{\Gamma,\tau} e'$ between source substitutions and expressions γ, e and target expressions e'.

Definition 3.3. *Let γ be a substitution mapping source variables to source terms. We write $\vdash \gamma : \Gamma$ to indicate that $Dom(\gamma) = Dom(\Gamma)$ and $\vdash \gamma(x) : \Gamma(x)$ for each $x \in Dom(\Gamma)$. We write $\hat{\gamma}(e)$ for the result of applying a substitution to an expression. We extend ground value equivalence to substitutions satisfying $\vdash \gamma : \Gamma$ as follows:*

- $\bullet \approx_\bullet \langle\rangle$
- $\gamma, \{x \mapsto v\} \approx_{\Gamma,x:\tau} \langle v', \mathtt{env}\rangle$ *if* $\gamma \approx_\Gamma \mathtt{env}$ *and* $v \approx_\tau v'$.

$$\mathcal{E}_1[\![\Gamma \vdash i : \mathsf{int}]\!] = \langle i, \mathtt{env}\rangle$$

$$\begin{array}{l}\mathcal{E}_1[\![\Gamma, x{:}\tau \vdash x : \tau]\!] = \\ \quad \mathtt{let}\ \langle x, \mathtt{env}\rangle\ \mathtt{=}\ \mathtt{env}\ \mathtt{in} \\ \quad \mathtt{let}\ \langle x_1, x_2\rangle\ \mathtt{=}\ \mathsf{Copy}[\![\tau]\!](x)\ \mathtt{in} \\ \quad \langle x_1, \langle x_2, \mathtt{env}\rangle\rangle\end{array}$$

$$\begin{array}{l}\mathcal{E}_1[\![\Gamma, y{:}\tau' \vdash x : \tau]\!] = \\ \quad \mathtt{let}\ \langle y, \mathtt{env}\rangle\ \mathtt{=}\ \mathtt{env}\ \mathtt{in} \\ \quad \mathtt{let}\ \langle x, \mathtt{env}\rangle\ \mathtt{=}\ \mathcal{E}_1[\![\Gamma \vdash x : \tau]\!]\ \mathtt{in} \\ \quad \langle x, \langle y, \mathtt{env}\rangle\rangle\end{array}$$

$$\begin{array}{l}\mathcal{E}_1[\![\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2]\!] = \\ \quad \mathtt{let}\ \langle x_1, \mathtt{env}\rangle\ \mathtt{=}\ \mathcal{E}_1[\![\Gamma \vdash e_1 : \tau_1]\!]\ \mathtt{in} \\ \quad \mathtt{let}\ \langle x_2, \mathtt{env}\rangle\ \mathtt{=}\ \mathcal{E}_1[\![\Gamma \vdash e_2 : \tau_2]\!]\ \mathtt{in} \\ \quad \langle\langle x_1, x_2\rangle, \mathtt{env}\rangle\end{array}$$

$$\begin{array}{l}\mathcal{E}_1[\![\Gamma \vdash \pi_1\, e : \tau_1]\!] = \\ \quad \mathtt{let}\ \langle p, \mathtt{env}\rangle\ \mathtt{=}\ \mathcal{E}_1[\![\Gamma \vdash e : \tau_1 \times \tau_2]\!]\ \mathtt{in} \\ \quad \mathtt{let}\ \langle x_1, x_2\rangle\ \mathtt{=}\ p\ \mathtt{in} \\ \quad \mathsf{Free}[\![\tau_2]\!](x_2); \\ \quad \langle x_1, \mathtt{env}\rangle\end{array}$$

$$\begin{array}{l}\mathcal{E}_1[\![\Gamma \vdash \pi_2\, e : \tau_2]\!] = \\ \quad \mathtt{let}\ \langle p, \mathtt{env}\rangle\ \mathtt{=}\ \mathcal{E}_1[\![\Gamma \vdash e : \tau_1 \times \tau_2]\!]\ \mathtt{in} \\ \quad \mathtt{let}\ \langle x_1, x_2\rangle\ \mathtt{=}\ p\ \mathtt{in} \\ \quad \mathsf{Free}[\![\tau_1]\!](x_1); \\ \quad \langle x_2, \mathtt{env}\rangle\end{array}$$

$$\begin{array}{l}\mathcal{E}_1[\![\Gamma \vdash e_1\, e_2 : \tau_2]\!] = \\ \quad \mathtt{let}\ \langle c, \mathtt{env}\rangle\ \mathtt{=}\ \mathcal{E}_1[\![\Gamma \vdash e : \tau_1 \to \tau_2]\!]\ \mathtt{in} \\ \quad \mathtt{let}\ \langle x, \mathtt{env}\rangle\ \mathtt{=}\ \mathcal{E}_1[\![\Gamma \vdash e_2 : \tau_1]\!]\ \mathtt{in} \\ \quad \langle \mathsf{App}(c, x), \mathtt{env}\rangle\end{array}$$

$$\begin{array}{l}\mathcal{E}_1[\![\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2]\!] = \\ \quad \mathtt{let}\ \langle \mathtt{cenv}, \mathtt{env}\rangle\ \mathtt{=}\ \mathsf{Copy}[\![|\Gamma|]\!](\mathtt{env})\ \mathtt{in} \\ \quad \mathtt{let}\ \mathtt{copy}\ \mathtt{=}\ \lambda\mathtt{env}.\mathsf{Copy}[\![|\Gamma|]\!](\mathtt{env})\ \mathtt{in} \\ \quad \mathtt{let}\ \mathtt{free}\ \mathtt{=}\ \lambda\mathtt{env}.\mathsf{Free}[\![|\Gamma|]\!](\mathtt{env})\ \mathtt{in} \\ \quad \mathtt{let}\ \mathtt{app}\ \mathtt{=}\ (\lambda\mathtt{env}. \\ \qquad\qquad \mathtt{let}\ \langle r, \mathtt{env}\rangle\ \mathtt{=}\ \mathcal{E}_1[\![\Gamma, x{:}\tau_1 \vdash e : \tau_2]\!]\ \mathtt{in} \\ \qquad\qquad \mathsf{Free}[\![|\Gamma, x{:}\tau_1|]\!](\mathtt{env}); \\ \qquad\qquad r)\ \mathtt{in} \\ \quad \mathtt{let}\ d\ \mathtt{=}\ \langle \mathtt{cenv}, \langle \mathtt{app}, \langle \mathtt{copy}, \mathtt{free}\rangle\rangle\rangle\ \mathtt{in} \\ \quad \mathtt{let}\ c\ \mathtt{=}\ \mathtt{pack}[\mathcal{T}_1[\![\Gamma]\!], d]\ \mathtt{as}\ \mathcal{T}_1[\![\tau_1 \to \tau_2]\!]\ \mathtt{in} \\ \quad \langle c, \mathtt{env}\rangle\end{array}$$

$$\begin{array}{l}\mathsf{App}(e_1, e_2) = \\ \quad \mathtt{let}\ [\alpha, \langle \mathtt{cenv}, \langle \mathtt{app}, \langle \mathtt{copy}, \mathtt{free}\rangle\rangle\rangle]\ \mathtt{=}\ e_1\ \mathtt{in} \\ \quad \mathtt{app}\langle e_2, \mathtt{cenv}\rangle\end{array}$$

Figure 3: $\lambda^{\to\times}$ to LIL translation

$$
\begin{array}{l}
\mathsf{Copy}[\![\tau]\!] : \mathcal{T}_1[\![\tau]\!] \Rightarrow \mathcal{T}_1[\![\tau]\!] \otimes \mathcal{T}_1[\![\tau]\!] \\
\mathsf{Copy}[\![\mathsf{int}]\!](x) = \langle x, x\rangle \\
\mathsf{Copy}[\![\tau_1 \times \tau_2]\!](p) = \\
\quad \texttt{let } \langle x, y\rangle \texttt{ = } p \texttt{ in} \\
\quad \texttt{let } \langle x_1, x_2\rangle \texttt{ = } \mathsf{Copy}[\![\tau_1]\!](x) \texttt{ in} \\
\quad \texttt{let } \langle y_1, y_2\rangle \texttt{ = } \mathsf{Copy}[\![\tau_2]\!](y) \texttt{ in} \\
\quad \langle\langle x_1, y_1\rangle, \langle x_2, y_2\rangle\rangle \\
\mathsf{Copy}[\![\tau_1 \to \tau_2]\!](c) = \\
\quad \texttt{let } [\alpha, d] \texttt{ = } c \texttt{ in} \\
\quad \texttt{let } \langle \texttt{env}, \langle \texttt{app}, \langle \texttt{copy}, \texttt{free}\rangle\rangle\rangle \texttt{ = } d \texttt{ in} \\
\quad \texttt{let } \langle e_1, e_2\rangle \texttt{ = copy(env) in} \\
\quad \texttt{let } d_1 \texttt{ = } \langle e_1, \langle \texttt{app}, \langle \texttt{copy}, \texttt{free}\rangle\rangle\rangle \texttt{ in} \\
\quad \texttt{let } d_2 \texttt{ = } \langle e_2, \langle \texttt{app}, \langle \texttt{copy}, \texttt{free}\rangle\rangle\rangle \texttt{ in} \\
\quad \texttt{let } c_1 \texttt{ = pack}[\alpha, d_1] \texttt{ in} \\
\quad \texttt{let } c_2 \texttt{ = pack}[\alpha, d_2] \texttt{ in } \langle c_1, c_2\rangle \\
\\
\mathsf{Free}[\![\tau]\!] : \mathcal{T}_1[\![\tau]\!] \Rightarrow \mathsf{unit} \\
\mathsf{Free}[\![\mathsf{int}]\!](x) = \langle\rangle \\
\mathsf{Free}[\![\tau_1 \times \tau_2]\!](p) = \\
\quad \texttt{let } \langle x, y\rangle \texttt{ = } p \texttt{ in} \\
\quad \mathsf{Free}[\![\tau_1]\!](x)\,;\, \mathsf{Free}[\![\tau_2]\!](y) \\
\mathsf{Free}[\![\tau_1 \to \tau_2]\!](c) = \\
\quad \texttt{let } [\alpha, d] \texttt{ = } c \texttt{ in} \\
\quad \texttt{let } \langle \texttt{env}, \langle \texttt{app}, \langle \texttt{copy}, \texttt{free}\rangle\rangle\rangle \texttt{ = } d \texttt{ in free(env)}
\end{array}
$$

Figure 4: Definitions of $\mathsf{Copy}[\![\cdot]\!]$ and $\mathsf{Free}[\![\cdot]\!]$

Finally, we define contextual simulation $\cdot \Vdash \cdot \approx_{\Gamma;\tau} \cdot$ *and* $\cdot \Vdash \cdot \sim_{\Gamma;\tau} \cdot$ *as follows:*

- $\gamma \Vdash v \approx_{\Gamma;\tau} \langle v', \texttt{env}\rangle$ *iff* $\gamma \approx_\Gamma \texttt{env}$ *and* $v \approx_\tau v'$.
- $\gamma \Vdash e \sim_{\Gamma;\tau} e'$ *iff* $\hat{\gamma}(e) \Downarrow v$, $e' \Downarrow v'$, *and* $\gamma \Vdash v \approx_{\Gamma;\tau} v'$.

We first need to show that the Copy and Free macros work as advertised for the translations of source terms. It suffices to check that they work properly on closed values:

Lemma 3.4. *Assume* $v \sim_\tau v'$. *Then* $\mathsf{Copy}[\![\tau]\!](v') \Downarrow \langle v', v'\rangle$ *and* $\mathsf{Free}[\![\tau]\!](v) \Downarrow \langle\rangle$.

The proof is straightforward by induction on the definition of τ. The real insight here is that the invariants needed to prove the case for function types need to be built into $\approx_{\tau_1 \to \tau_2}$. Now we can prove simulation:

Theorem 3.5. *Suppose* $\Gamma \vdash e : \tau$, $\vdash \gamma : \Gamma$, *and* $\gamma \approx_\Gamma \texttt{env}$ *are derivable and* $e' = \mathcal{E}_1[\![\Gamma \vdash e : \tau]\!]$. *Then* $\gamma \Vdash e \sim_{\Gamma;\tau} e'$.

Proof. By induction on the derivation $\Gamma \vdash e : \tau$. The interesting cases are those for variables, applications, and abstraction. □

3.3 Generating Linear TAL Code

We will describe a simple translation to assembly language that performs minimal register allocation and other optimizations, though to some extent these are possible with Linear TAL just as with ordinary assembly language. We assume there are at least six registers r_s (stack register), r_a (answer register), r_t, r_u (temporary registers), r_f (freelist register), and r_r (return address register). Our

type translation is as follows:

$$\begin{array}{rcl}
\mathcal{T}_2[\![\mathsf{unit}]\!] & = & \mathsf{word} \\
\mathcal{T}_2[\![\mathsf{int}]\!] & = & \mathsf{int} \\
\mathcal{T}_2[\![\sigma_1 \otimes \sigma_2]\!] & = & @\langle \mathcal{T}_2[\![\sigma_1]\!] \otimes \mathcal{T}_2[\![\sigma_2]\!] \rangle \\
\mathcal{T}_2[\![\exists \alpha.\sigma]\!] & = & \exists \alpha.\mathcal{T}_2[\![\sigma]\!] \\
\mathcal{T}_2[\![\alpha]\!] & = & \alpha \\
\mathsf{flist} & = & \mu\alpha.?\langle \mathsf{word} \otimes \alpha \rangle \\
CC(\omega_s, \omega_a, \omega_u, \tau) & = & \mathtt{code}\{r_s : \omega_s, r_a : \omega_a, r_u : \omega_u, \\
 & & \quad r_t : \mathsf{word}, r_f : \mathsf{flist}, r_r : \tau\} \\
\mathcal{T}_2[\![\sigma_1 \Rightarrow \sigma_2]\!] & = & \forall \alpha, \beta. CC(@\langle \mathcal{T}_2[\![\sigma_1]\!] \otimes \alpha \rangle, \mathsf{word}, \beta, \\
 & & \quad CC(\alpha, \mathcal{T}_2[\![\sigma_2]\!], \beta, \mathsf{word}))
\end{array}$$

Integers are represented as words, pairs as pointers to memory blocks, and existentials and type variables as their corresponding forms in LTAL. The translation of functions expresses a simple calling convention. In a procedure call,

- the argument is passed on the stack, which also consists of unspecified additional contents α
- the result register r_a and "caller-save" temporary register r_t must not point to anything important
- the "callee-save" temporary r_u may refer to any type β
- the freelist register r_f must point to a freelist
- and the return register must contain an appropriate return address, which requires:
  - the function's argument must have been popped off the stack and disposed of, leaving the stack remainder α
  - the result must be in the return register r_a
  - r_r and r_t must not refer to anything important
  - r_f must still point to a freelist
  - and r_u must contain its original contents β.

The translation of a restricted or unrestricted LIL type is a restricted or unrestricted LTAL type, respectively. As before, the translation of a context is an iterated tuple:

$$\begin{array}{rcl}
\mathcal{T}_2[\![\Gamma, x : \sigma]\!] & = & @\langle \mathcal{T}_2[\![\sigma]\!] \otimes \mathcal{T}_2[\![\Gamma]\!] \rangle \\
\mathcal{T}_2[\![\bullet]\!] & = & \mathsf{word}
\end{array}$$

We introduce several macros in Figure 5 that greatly simplify the translation. The `alloc` and `free` macros assume only that $r_f \neq r$ and points to a freelist; the other macros assume that r_f, r_t, r, s are all distinct and r_t does not point to anything important. To be completely precise, the label in the `alloc` macro needs a type that depends heavily on context; we omit that level of detail in favor of clarity. Also, if we can tell from the type of r_f that there is at least one cell at the head of the list, then the branch operation in `alloc` can be removed.

We assume that all functions have been lifted to the top level; thus, a program is a sequence of let-bindings of functions to variables f followed by a the program body. All function applications are of function variables to data. We introduce auxiliary judgments $\Psi \vdash e \uparrow \sigma$ (for toplevel expressions) and $\Psi; \Delta; \Gamma \vdash e \downarrow \sigma$ (for function and program bodies), where Ψ is a context binding top-level function names to their types. The rules for $\uparrow$ judgments are:

$$\frac{\Psi; \bullet; \bullet, x : \sigma_1 \vdash e \downarrow \sigma_2 \qquad \Psi, f : \sigma_1 \Rightarrow \sigma_2 \vdash e' \uparrow \sigma'}{\Psi \vdash \mathtt{let}\ f\ \mathtt{=}\ \lambda x : \sigma_1.e\ \mathtt{in}\ e' \uparrow \sigma'}$$

```
alloc r   =  bnz r_f, L;
             halt;
          L: mov r, r_f;
             ld r_f, r[1]

free r    =  st r[1], r_f;
             mov r_f, r

push r, s =  alloc r_t;
             st r_t[0], s;
             st r_t[1], r;
             mov r, r_t

pop r, s  =  mov r_t, s;
             ld r, r_t[0];
             ld s, r_t[1];
             free r_t;

drop r    =  ld r_t, r[0];
             free r_t;
             mov r, r_t;

rot r, s  =  ld r_t, s[1];
             st s[1], r;
             mov r, s;
             mov s, r_t;
```

Figure 5: LTAL Macros

$$\frac{\Psi; \bullet; \Gamma \vdash e \downarrow \sigma}{\Psi \vdash e \uparrow \sigma}$$

The rules for $\downarrow$ judgments are the same as the ordinary typing rules except the $\Rightarrow$-introduction rule is omitted. This guarantees that all application heads are function variable names bound in Ψ or Γ.

Figures 6 and 7 show the code generation translation from LIL to LTAL. We fix a specific label L_S as the entry point for the entire program. We also generate a canonical label L_f for each LIL `let`-bound function name f, and we generate a fresh local "return address" label L_R for each translated application. We have taken the liberty of presenting a simple, but not completely accurate translation: for terms like $\langle e_1, e_2 \rangle$, we re-use the same context Γ in translating each branch rather than split the Γ up into the parts Γ_1, Γ_2 used to typecheck the two subexpressions e_1, e_2. This is because we want the shape of the environment to stay the same; that is, we don't want to have to split the stack up whenever the linear environment splits. To be completely formal about this we would have to have a type system with judgments like $\Delta; \Gamma \vdash e : \tau \mid \Gamma'$, where Γ' has the same shape (domain) as Γ but some of its linear types may have been "flattened" (replaced with unit). This is essentially how our implementation works.

The translation of variable accesses is slightly tricky. We treat the context as an ordered list, implemented as nested memory cells. In our linear setting, it is not possible to traverse this list without modifying it. Thus, to look up a variable x in the environment Γ, x, Γ', we must traverse the initial segment corresponding to Γ', saving it in an auxiliary list, then copy x, then restore the environment. This is accomplished with the aid of the macro `rot` r, s, which moves the head cons cell of a nonempty list starting at s onto another, possibly empty list starting at r. This is an extremely inefficient way to accomplish a simple task: if we ignore the linearity restriction, it is easy to do lookups in approximately $n = |\Gamma|$ instructions on average rather than $8n$. And since lookups are very common, this turns out to be an important source of inefficiency in LTAL.

3.4 Correctness

We have presented a highly schematic description of the translation from LIL to LTAL. To make the type-correctness of the translation precise, we need to be more explicit about the form of the translation. We view the results of $\downarrow$ translation as a triple (I, C, Ψ) of instruction sequence, code section, and code type context (over the same set of labels). The result of $\uparrow$ translation is a pair (C, Ψ).

Definition 3.6. *A code section C' and signature Ψ' are well-formed with respect to Ψ if for any code section C disjoint from C' satisfying at least $\vdash C : \Psi$, we have $\vdash C \uplus C' : \Psi \uplus \Psi'$. We write*

$\mathcal{E}_2[\![\Psi \vdash \texttt{let}\ f\ \texttt{=}\ \lambda x{:}\sigma_1.e\ \texttt{in}\ e' \uparrow \sigma]\!] =$
$L_f:\quad \mathcal{T}_2[\![\sigma_1 \Rightarrow \sigma_2]\!]$
$\quad\quad \mathcal{E}_2[\![\Psi; \bullet, x : \sigma_1 \vdash e \downarrow \sigma_2]\!];$
$\quad\quad \texttt{drop}\ r_s;$
$\quad\quad \texttt{jmp}\ r_r;$
$\quad\quad \mathcal{E}_2[\![\Psi, f : \sigma_1 \Rightarrow \sigma_2 \vdash e' : \sigma']\!]$

$\mathcal{E}_2[\![\Psi \vdash e \uparrow \sigma]\!] =$
$L_S:\quad \forall\alpha\beta.CC(\alpha, \mathsf{word}, \beta, CC(\alpha, \mathcal{T}_2[\![\sigma]\!], \beta, \mathsf{word}))$
$\quad\quad \mathcal{E}_2[\![\Psi; \bullet; \bullet \vdash e \downarrow \sigma]\!];$
$\quad\quad \texttt{jmp}\ r_r$

$\mathcal{E}_2[\![\Psi; \Delta; \Gamma \vdash i \downarrow \mathsf{int}]\!] = \texttt{mov}\ r_a, i$

$\mathcal{E}_2[\![\Psi; \Delta; \Gamma \vdash \langle\rangle \downarrow \mathsf{unit}]\!] = \texttt{mov}\ r_a, 0$

$\mathcal{E}_2[\![\Psi; \Delta; \Gamma_1 \vdash e1; e2 \downarrow \sigma]\!] =$
$\quad\quad \mathcal{E}_2[\![\Psi; \Delta; \Gamma \vdash e_1 : \mathsf{unit}]\!]$
$\quad\quad \mathcal{E}_2[\![\Psi; \Delta; \Gamma \vdash e_2 : \sigma]\!]$

$\mathcal{E}_2[\![\Psi, f : \sigma_1 \Rightarrow \sigma_2; \Delta; \Gamma \vdash f \downarrow \sigma_1 \Rightarrow \sigma_2]\!] = \texttt{mov}\ r_a, L_f;$

$\mathcal{E}_2[\![\Psi; \Delta; \Gamma, x : \sigma \vdash x \downarrow \sigma]\!] = \texttt{ld}\ r_a, r_s[0]$

$\mathcal{E}_2[\![\Psi; \Delta; \Gamma, y : \sigma \vdash x \downarrow \sigma']\!] =$
$\quad\quad \texttt{rot}\ r_u, r_s$
$\quad\quad \mathcal{E}_2[\![\Psi; \Delta; \Gamma \vdash x \downarrow \sigma']\!];$
$\quad\quad \texttt{rot}\ r_s, r_u$

$\mathcal{E}_2[\![\Psi; \Delta; \Gamma \vdash f\ e \downarrow \sigma_2]\!] =$
$\quad\quad \mathcal{E}_2[\![\Psi; \Delta; \Gamma \vdash e \downarrow \sigma_1]\!];$
$\quad\quad \texttt{push}\ r_u, r_a;$
$\quad\quad \mathcal{E}_2[\![\Psi; \Delta; \Gamma \vdash f \downarrow \sigma_1 \Rightarrow \sigma_2]\!];$
$\quad\quad \texttt{push}\ r_s, r_r;$
$\quad\quad \texttt{rot}\ r_s, r_u;$
$\quad\quad \texttt{mov}\ r_r, L_R;$
$\quad\quad \texttt{jmp}\ r_a$
$L_R:\quad CC(\mathcal{T}_2[\![\Gamma]\!], \mathcal{T}_2[\![\sigma_2]\!], \omega, \mathsf{word})$
$\quad\quad \texttt{pop}\ r_r, r_s;$

Figure 6: LIL to LTAL term translation (I)

$$
\begin{array}{l}
\mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash \langle e_1, e_2\rangle \downarrow \sigma_1 \otimes \sigma_2]\!] = \\
\quad \mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash e_1 \downarrow \sigma_1]\!]; \\
\quad \texttt{push}\ r_u, r_a; \\
\quad \mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash e_2 \downarrow \sigma_2]\!]; \\
\quad \texttt{rot}\ r_a, r_u; \\
\mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash \texttt{let}\ \langle x_1, x_2\rangle\ \texttt{=}\ e\ \texttt{in}\ e' \downarrow \sigma]\!] = \\
\quad \mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash e \downarrow \sigma_1 \otimes \sigma_2']\!]; \\
\quad \texttt{rot}\ r_s, r_a; \\
\quad \texttt{push}\ r_s, r_a; \\
\quad \mathcal{E}_2[\![\Psi;\Delta;\Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash e' \downarrow \sigma]\!]; \\
\quad \texttt{drop}\ r_s; \\
\quad \texttt{drop}\ r_s; \\
\mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash \texttt{pack}[\sigma_1, e] \downarrow \exists\alpha.\sigma]\!] = \\
\quad \mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash e \downarrow \sigma[\sigma_1/\alpha]]\!]; \\
\mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash \texttt{let}\ [\alpha, x]\ \texttt{=}\ e\ \texttt{in}\ e' \downarrow \sigma]\!] = \\
\quad \mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash e \downarrow \exists\alpha.\sigma]\!]; \\
\quad \texttt{push}\ r_s, r_a; \\
\quad \mathcal{E}_2[\![\Psi;\Delta,\alpha;\Gamma, x : \sigma \vdash e \downarrow \sigma]\!]; \\
\quad \texttt{drop}\ r_s; \\
\mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \downarrow \sigma']\!] = \\
\quad \mathcal{E}_2[\![\Psi;\Delta;\Gamma \vdash e \downarrow \sigma]\!]; \\
\quad \texttt{push}\ r_s, r_a \\
\quad \mathcal{E}_2[\![\Psi;\Delta;\Gamma, x : \sigma \vdash e' \downarrow \sigma']\!]; \\
\quad \texttt{drop}\ r_s;
\end{array}
$$

Figure 7: LIL to LTAL term translation (II)

this as $\Psi \vdash C' : \Psi'$.

Theorem 3.7. *If* $\Psi \vdash e \uparrow \sigma$ *is derivable then* $(C, \Psi') = \mathcal{E}_2[\![\Gamma \vdash e \uparrow \sigma]\!]$ *satisfies* $\mathcal{T}_2[\![\Psi]\!] \vdash C : \Psi'$.

If $\Psi; \Delta; \Gamma \vdash e \downarrow \sigma$ *is derivable then* $(I, C, \Psi') = \mathcal{E}_2[\![\Gamma \vdash e \uparrow \sigma]\!]$ *satisfies* $\mathcal{T}_2[\![\Psi]\!] \vdash C : \Psi'$, *and for any type* ω *satisfying* $\Delta \vdash \omega$,

$$\Delta; CC(\mathcal{T}_2[\![\Gamma]\!], \mathsf{word}, \omega, CC(\mathcal{T}_2[\![\Gamma]\!], \mathcal{T}_2[\![\sigma]\!], \omega, \mathsf{word})) \vdash I$$

relative to code context $\mathcal{T}_2[\![\Psi]\!] \uplus \Psi'$.

We believe that it is possible to show that the translation from IL to LTAL preserves operational behavior also (that is, that our compiler is correct). Indeed, the type-correctness argues in favor of this. However, formalizing and proving this seems hard, so we defer it to future work.

4 Formal Results

Like TAL, LTAL satisfies type soundness relative to its operational semantics: no type-safe program can get stuck. This requires checking not only that the program is well-formed, but that the heap and register file typecheck with respect to the current register context. We handle this by defining judgments $H \models R : \Gamma$, $H \models v : \omega$, and $H \models h : \sigma$ which characterize the single pointer property we need. Soundness also entails memory conservation: if a program terminates normally, then all the memory not used by the result of the computation can be reused. It is worth pointing out that the judgment $H \models R : \Gamma$ is a special case of the "heaps as possible worlds" view used in Kripke-style semantics of Bunched Implications [23]; this is an interesting connection we plan to investigate in future work. Our choice of the notation $\models$ is based on this observation and a similar choice made in HBAL [2].

In what follows, let C be some fixed code section and Ψ be its signature, that is, $\vdash C : \Psi$.

Lemma 4.1 (Progress). *If* $\vdash P$ *then there exists* P' *such that* $P \longrightarrow_C P'$.

The key to proving Type Preservation is to ensure that whenever the type of the register file changes, the register file and heap also change in a way that is compatible with the single-pointer property.

Lemma 4.2 (Conservation). *If* $\bullet; \Gamma \vdash i \mid \Delta; \Gamma'$, $H \models R : \Gamma$, *and* $(H, R, i; I) \longrightarrow_C (H', R', I)$ *then there exists a type substitutiuon* $\delta : \Delta$ *such that* $H' \models R' : \delta(\Gamma')$.

Lemma 4.3 (Preservation). *If* $\vdash P$ *and* $P \longrightarrow_C P'$ *then* $\vdash P'$.

Theorem 4.4 (Soundness). *If* $\vdash P$ *and* $P \longrightarrow_C^* P'$ *then* P' *is not stuck.*

It is easy to verify that memory cells are never created or destroyed. Programs that terminate successfully (*i.e.* by calling their exit continuation) are guaranteed to leave memory in a consistent state; that is, all memory not used by the answer is restored to the freelist. Conversely, execution never becomes stuck as a result of a memory management error such as accessing "freed" memory through a dangling pointer. Naturally, the usual kind of type errors are prohibited also.

Of course, programs can `halt` at any time, leaving the heap in a mess, so this result is a little unsatisfying. However, our translation only uses `halt` to terminate the program when it runs out of memory. We consider how to deal with situations like running out of memory more gracefully via exception handling in the next section.

5 Extensions

Our source language $\lambda^{\rightarrow\times}$ doesn't deal with any really interesting features of functional languages: datatypes with pattern matching, recursive functions, polymorphism, control operators, laziness, or references. Dealing with recursion is straightforward, perhaps surprisingly in a linear setting.

$\mathcal{E}_2[\![\Gamma \vdash \texttt{fix}\ f(x{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2]\!] =$

```
  let mkc =
      (λp.
          let copy = λenv.Copy[[|Γ|]](env) in
          let free = λenv.Free[[|Γ|]](env) in
          let ⟨app, env⟩ = p in
          let ⟨cenv, env⟩ = Copy[[|Γ|]](env) in
          let d = ⟨cenv, ⟨app, ⟨copy, free⟩⟩⟩ in
          let c = pack[T2[[Γ]], d] as T2[[τ1 → τ2]] in
          ⟨c, env⟩) in
  let app =
      (fix app'(env) =>
          let ⟨x, env⟩ = env in
          let env = mkc(⟨app', env⟩) in
          let env = ⟨x, env⟩ in
          let ⟨r, env⟩ = E2[[Γ' ⊢ e : τ2]] in
          Free[[|Γ'|]](env);
          r) in
  mkc(⟨app, env⟩)
```

where $\Gamma' = \Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1$

Figure 8: Translation of recursive functions

Datatypes and polymorphism are also fairly easy to deal with, if complicated in implementation. In this section we describe these simple extensions. References and laziness, which rely crucially on sharing, seem difficult or impossible to accommodate in LTAL as-is. We discuss options for supporting them as well. LTAL's memory model is very primitive, supporting essentially only "cons cells," whereas real languages employ a wide variety of memory models, and we discuss possible extensions to LTAL to support these models.

5.1 Easy-to-handle features

5.1.1 Recursion

LTAL already supports recursion, since blocks are typechecked in the context of all code labels. However, our source and intermediate languages do not. It is easy to add standard syntax for recursively-defined functions to the source and intermediate languages, with the usual typing and operational semantics rules. In LIL, we require that the function refer only to function labels, f, and x. Thus, like ordinary functions, recursive functions can refer only to other functions.

$$\frac{\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fix}\ f(x{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2}$$

$$\frac{\bullet; \Psi, f{:}\sigma_1 \Rightarrow \sigma_2, x{:}\sigma_1 \vdash e : \sigma_2}{\bullet; \Psi \vdash \texttt{fix}\ f(x{:}\sigma_1){:}\sigma_2\ \texttt{=>}\ e : \sigma_1 \Rightarrow \sigma_2}$$

Recursive functions translate to closures whose application function copies the environment and constructs a new closure before proceeding with the body of the function. The source-to-LIL translation for recursive functions is given in Figure 8. Since the activity of constructing a closure recurs at the top-level closure construction as well as inside the application function, we introduce a new function `mkc` which given an application function and an environment constructs a closure. To copy the environment during an application, we first need to split the function argument off from the rest of the environment. Extending this to mutually recursive functions is straightforward, if tedious.

$$
\begin{array}{rcl}
\mathcal{T}_2[\![\mathsf{list}[\tau]]\!] & = & \mu\alpha.?\langle \mathcal{T}_2[\![\tau]\!] \otimes \alpha \rangle \\
\mathcal{T}_2[\![\forall\alpha.\tau]\!] & = & \forall\alpha.copy(\alpha) \otimes free(\alpha) \Rightarrow \mathcal{T}_2[\![\tau]\!] \\
\mathcal{T}_2[\![\exists\alpha.\tau]\!] & = & \exists\alpha.\tau \otimes copy(\alpha) \otimes free(\alpha) \\
\mathcal{T}_2[\![\alpha]\!] & = & \alpha
\end{array}
$$

Figure 9: Type translations for extensions

One drawback to this simplistic encoding of recursive functions is that the closure is copied once per recursive call. An alternative, possibly better approach is to define $app(\tau_1, \tau_2, \tau_e)$ as $\tau_1 \otimes \tau_e \Rightarrow \tau_1 \otimes \tau_e$, and require the caller of a function to free its environment after the call. This permits recursive functions to be implemented by threading a single context through the recursive calls.

5.1.2 Lists and Datatypes

Lists over arbitrary types can easily be added both to the source and linear IL. The type translation of lists in LTAL is shown in Figure 9. Extending the term translation to deal with `cons`, `nil`, and `case` expressions is easy. To deal with arbitrary, user-defined datatypes, we could introduce singleton integer types and union types along the lines of the encoding of recursive types in Alias Types [30] or DTAL [33].

5.1.3 Polymorphism

Another important feature which we have not shown how to implement is parametric polymorphism. LTAL already supports polymorphism; however, our source and intermediate languages do not. We can easily add universally quantified types to the source and LIL in the usual way. The main problem is how to translate polymorphic source-level types and values to the LIL, because abstract types may be freely copied and forgotten in the source language but not in LIL. The approach we take is to translate polymorphic terms to functions that also abstract copy and free operations for their type arguments. The type translation is augmented as shown in Figure 9. Unfortunately, this encoding does not interact well with closure conversion. Closure conversion is integral to the translation to LIL, while translating polymorphic code introduces new functions. Although these features do not seem difficult to handle in isolation, so far a good solution to handling them simultaneously has evaded us. We currently are looking for a way to simplify the translation by splitting it into two phases: a linearization phase that adds copy and free functions and abstractions to the source language, then closure conversion.

Abstract (existential) types can also be supported. Unlike for polymorphism, the source-to-LIL translation of existentials is straightforward, since we already handle closures as a special case.

5.1.4 Exceptions and Continuations

LTAL includes a `halt` instruction for escaping from situations like running out of memory without cleaning up. We'd prefer to have some exception handling mechanism by which programs that run into unexpected situations can backtrack and attempt to recover. This kind of behavior is a necessity in any systems program which is expected to do something reasonable no matter what happens. It would also be nice to be able to handle first-class continuations (`callcc`).

Unsurprisingly, the answer is continuation-passing style conversion. However, in our linear setting, there are several complications. First, before calling an exception handler, LTAL programs have to dispose of all storage that has been allocated since the handler was installed. Second, as noted in [6], some forms of control flow, such as exceptions and coroutines, can be accomplished using linear CPS translations, whereas other forms such as first-class `callcc` cannot. Intuitively the reason is that copying continuations requires "copying the world". In light of this result, we expect the former kinds of control flow to be easier to implement in LTAL, whereas the latter would require maintaining "copy the world" and "free the world" functions throughout computation.

5.1.5 Garbage collection and reference counting

As much as LTAL tries to pretend otherwise, garbage collection is part of the functional programming world, so it is important for LTAL to at least co-exist with it peacefully. Fortunately, linear logic already provides a bridge back to intuitionistic logic that fits our purposes: the exponential modality ! ("of course"). In linear logic, $!A$ is an "always valid" proposition, thus can be copied and forgotten without restriction, and this models garbage-collected values perfectly. We can easily add a new unrestricted types $!\omega$ to LTAL and $!\sigma$ to LIL, and allow $@\langle\tau_1 \otimes \tau_2\rangle$ to be coerced to $!@\langle\tau_1 \otimes \tau_2\rangle$ for any unrestricted τ_1, τ_2. Aliases of linear data may not escape into garbage-collected memory: all the components of a heap cell we want to make unrestricted also have to be unrestricted. This prevents us from accidentally recycling a shared cell.

An alternative approach that would permit both sharing and reclamation would be to interpret ! types as reference-counted pointers. However, previous work [8] on reference-counting readings of linear logic is at a higher level of abstraction than LTAL, and it is not clear how to push the reference counting interpretation all the way down to the linear level without introducing reference counting pseudo-instructions.

5.2 Hard-to-handle features

5.2.1 References and Laziness

Although we have presented a compiler for a call-by-value language, it is not difficult to support call-by-name evaluation in LTAL by translating source expressions to suspensions using standard techniques. But a more efficient call-by-need implementation is not easy because sharing is not allowed. References are also not easy to incorporate into LTAL. We group laziness and references together because they seem roughly equivalent in terms of difficulty: thunks can be implemented in a strict language like ML using references, whereas references can be simulated in a lazy language like Haskell using monads. The real problem in dealing with both in LTAL is sharing, addressed in the next section. But first we discuss an intriguing alternative.

O'Hearn and Reynolds [20] described translations from the languages Idealized Algol (IA) and Syntactic Control of Interference (SCI) to a polymorphic call-by-name linear λ-calculus similar to our LIL. IA and SCI are higher-order imperative languages with stack allocation, call-by-name parameter passing, and integer references. Although their target language includes ! types, they are used only on function types in the translation. Thus, we believe that it is possible to compile their target language to LTAL. However, IA and SCI provide only statically-scoped integer references, ducking the problems of higher-order storage and unrestricted reference deallocation.

5.2.2 Sharing

Functional languages expect data to be sharable, so copying can be accomplished by copying pointers. LTAL specifically forbids sharing. This is quite draconian. There are several known techniques for controlling aliasing which could be added to LTAL.

The most heavyweight approach would probably be to augment LTAL with a type system based on Alias Types [25, 30], a very powerful but complex approach to dealing with aliasing. In Wadler's approach to linear typing [27], linear values may be temporarily shared using a let! operator, as long as aliases to them do not escape from the body of the expression. Walker and Watkins' linear region calculus [31] focuses on tracking regions rather than individual cells, but includes a sharing operation similar to let! that uses type-level region names to guarantee that aliases cannot escape. There are additional variations on this theme in the Vault, Clean, and Cyclone programming languages [9, 10, 22].

Recent work on using separation logics for reasoning about pointer programs [24] may offer a more compelling alternative to name-based sharing. It has been repeatedly observed that Alias Types are similar in many ways to such pointer logics. We believe that investigating this relationship may lead to more powerful and less *ad hoc* sharing mechanisms. It is interesting to note that the

aforementioned analysis of O'Hearn and Reynolds on IA and SCI using linear logic has already been superseded by interpretations employing separation/bunched logics [23].

5.3 Beyond Cons-Cells

Our model of memory is very primitive, in that it assumes that all memory is only used in blocks of two words. This makes our presentation and proofs of soundness and memory preservation relatively simple, yet many functional languages execute using this simple architecture. Nevertheless it is desirable to be able to flatten large data structures. Our approach can easily be extended to accommodate this need by allowing arbitrary n-tuples of words as memory types. The price of this flexibility would be slightly more complicated (but essentially the same) typing rules.

One serious defect in LTAL is that lookups into the environment are not constant time, but instead require traversing a linked list. Incorporating n-tuples rather than pairs would defray this cost somewhat, at the cost of less transparent memory management. Moreover, managing temporary data storage (such as the intermediate values generated during construction of a pair) requires the relatively heavyweight `alloc` approach. It is far more convenient to use a flat stack to store the environment. Then lookups and temporary stack allocation become constant time operations, and stack allocation/deallocation of multiple cells can be coalesced into a single operation. We have experimented with incorporating a simplified form of *stack types* [17] in to Linear TAL, yielding Linear Stack TAL (LSTAL). Currently, LSTAL performs no stack overflow checking, so memory use is not bounded in LSTAL, but typical performance is much better. We compare LSTAL with LTAL in the following section.

Another unrealistic aspect of LTAL is that programs expect memory to be organized into freelist prior to execution. It would be better to view all memory as a contiguous segment of words, *i.e.* a page provided by the operating system, that can be broken up into blocks of desired sizes as needed, and restored when adjacent blocks are freed. This approach is related to that taken in the orderly linear lambda calculus of [21] and the stack logic of [1]; however, neither approach addresses general dynamic memory management. The former focuses on a garbage collection architecture in which deallocation is implicit and the latter describes stack allocation and deallocation but not heap deallocation. Yu et al. [34] have shown how to build a certified malloc/free memory manager using separation logic. Their approach is based on PCC, so is very expressive but also much more complex than the type-based system we would like to develop.

6 Implementation

We have implemented a typechecker and interpreter for LTAL and a certifying compiler from our source language to LTAL based on the translations described in Section 3. The implementation[3] is written in OCaml 3.06 and consists of about 6000 lines of code. Our implementation includes recursive functions, lists, and a partial implementation of polymorphism as outlined in Sections 5. This is a toy implementation, intended as a proof of concept and to shake the bugs out of the type system, not to run real programs. Nevertheless, in the rest of this section, we describe experiments that suggest that LTAL programs are just as inefficient as we feared. We realize that this does not prove anything: anyone can write a bad compiler. However, we believe that these results shed enough light on the relevant issues to be worth presenting.

The implementation performs several optimization phases on LIL prior to generating the LTAL code. The optimizations include copy propagation, inlining, and dead code elimination. The implemented LIL makes copy and free operations explicit temporarily in order to perform "copy-free elimination" (that is, optimizing away copies whose results are just freed afterward). These optimizations seem crucial in reducing the inefficiency of the resulting code (and the complexity of typechecking it) to an acceptable level.

We have also implemented a stack-based variant of LTAL, called LSTAL. It includes a downward-growable stack type along with `push` and `pop` instructions. In LSTAL, the environment is stored

[3] `http://www.cs.cornell.edu/talc/releases/ltal_0_1.tgz`

	LTAL			LSTAL			
File	Cs	Hp	Cycs	Cs	Hp	Stk	Cycs
`fact`	0.9	2	10	0.4	1	1	3
`ack`	2.3	1	620	0.8	1	1	197
`fib`	2.6	2	77	0.9	1	1	22
`tabulate`	7.5	6	465	1.9	2	3	99
`map`	14	16	2,050	3.6	11	3	705
`filter`	17.8	27	5,100	4.6	21	3	1,900
`rev`	11.7	26	5,400	3.3	21	3	2,200
`sum`	9.0	13	1,700	2.4	9	3	600

Table 1: LTAL performance microbenchmarks.

flat on the stack, so environment accesses take only $O(1)$ rather than $O(|\Gamma|)$ instructions; thus, we expected LSTAL programs to be much smaller and faster than pure LTAL ones.

We have written several microbenchmarks for LTAL to gauge its impracticality. The benchmarks include simple arithmetic functions like factorial, Ackermann's function, and Fibonacci sequence, as well as list examples including constructing, reversing, filtering, mapping, and summing lists. We include code size **Cs** (thousands of instructions), total number of instructions executed **Cycs** (thousands), and heap size **Hp** (rounded up to the nearest 4096-byte page). For LSTAL, we also indicate how many pages of stack space **Stk** were needed. The list examples work on lists of size 100. The results are shown in Table 1.

The results show, first of all, that LTAL programs are remarkably space- and time-inefficient. Naively compiling higher-order functional programs down to linear closures which are copied and freed once per recursive call results in far too much "bookkeeping" (copying and freeing) relative to actual computation. Since this bookkeeping crosses abstraction boundaries, it is difficult to optimize away. Even simple functions like list traversals incur memory overheads (and concomitant slowdowns from copying) far in excess of what is really needed for simple functions like `map`, `filter` and `rev`. However, many of these problems are more the fault of the simplistic source-to-LIL translation than LTAL: it is possible to hand-code many of these functions efficiently in LTAL. For example it is not hard to hand-code a factorial function in approximately 10 instructions that requires no heap-allocated memory.

As we can see by comparing with LSTAL, environment lookups are also a big part of the problem. Storing the environment on a growable stack results in dramatic improvements in code size and running time. In LSTAL, code size typically shrinks to 20-50% of the heap-only LTAL size, and running times improve by a factor of 2-4. The improvement in memory footprint is not very pronounced. For the small examples, any improvement is below the resolution of our page metric to capture; for larger examples, typically we save only a page or two. This makes some sense given the intermediate language's exclusive reliance on pairs.

Compiling from an explicitly linear source language (such as LFPL) or performing more advanced analyses to detect existing linearity would almost certainly result in significant improvement. Preliminary experiments with compiling an LFPL-like language to LTAL indicate that exploiting existing linearity can decrease code size by a factor of 3-5 and decrease memory usage and running time by at least a factor of 10. However, for most of our examples there is no direct comparison because higher-order functions like `filter` and `map` cannot be expressed in LFPL.

7 Related Work

The idea of using linearity to implement functional languages without garbage collection has a long history. Here we focus on work that is closely related to or strongly influenced ours. Lafont's Linear Abstract Machine [12] was an early approach showing how to translate linear functional programs to the instructions of a linear abstract machine that recycled memory directly. Wadler

[27] was an early proponent of using linearity to support imperative features in purely functional languages, and Wakeling and Runciman [28] studied the practical advantages and drawbacks to linear implementation techniques. Baker [4] proposed employing linearity in Lisp to manage memory without garbage collection. Chirimar, Gunter and Riecke [8] gave a reference counting interpretation of linear logic, in which explicit copies and frees corresponded to reference counting operations. Maraist et al. [14] compared interpretations of call-by-name, -need, and -value calculi into linear logic, and Turner and Wadler[26] studied the memory-management properties of the call-by-value and call-by-need interpretations.

Hofmann's LFPL [11] is a linear first-order functional language with list and tree data structures where space usage is tracked; LFPL programs can be compiled to C programs that do not allocate any new memory. Aspinall and Compagnoni have shown how to translate LFPL to HBAL [2], a variant of TAL that also provides some space usage guarantees. However, data structures like lists and trees are dealt with using built-in instructions rather than "plain" assembly language as in LTAL. In recent work, Aspinall and Hofmann [3] have incorporated "usage annotations" to allow some sharing in LFPL.

Our approach to encoding duplicatable closures in a linear language seems (at least superficially) related to Benton's encoding of a linear term calculus in System F, in which exponentials were encoded as coinductive datatypes using existential quantification [5]. There, the objective was to prove strong normalization, rather than compile programs.

8 Summary and Conclusions

Linear TAL is a safe, low-level language that "stands on its own": it does not require any outside run-time support from a garbage collector or operating system to guarantee safe execution within a fixed memory space. Moreover, LTAL is relatively simple, yet expressive enough to compile the functional core of ML. As far as we know, no other certified code technique combines these levels of transparency, independence and expressiveness.

But this independence comes at a steep price. LTAL's linearity discipline prohibits any useful kind of sharing. This has disastrous consequences for naive compilation from high-level, non-linear functional languages, where the ability to copy is taken for granted. Furthermore, LTAL cannot accommodate language features like references or laziness that rely on sharing. We intend to generalize LTAL to support both sharing and safe low-level memory management in future work.

Acknowledgments

Matthew Fluet and Yanling Wang contributed significantly to earlier versions of this work. Dave Walker and Dan Grossman provided valuable feedback.

References

[1] Amal Ahmed and David Walker. The logical approach to stack typing. In *Proc. Int. Workshop on Types in Language Design and Implementation*, pages 74–85. ACM Press, 2003.

[2] David Aspinall and Adriana Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 2003. To appear.

[3] David Aspinall and Martin Hofmann. Another type system for in-place update. In D. Le Metayer, editor, *Proc. European Symposium on Programming*, pages 36–52. Springer-Verlag, 2002. LNCS 2305.

[4] Henry G. Baker. Lively linear Lisp — 'Look Ma, no garbage!'. *ACM SIGPLAN Notices*, 27(9):89–98, 1992.

[5] P. N. Benton. Strong normalisation for the linear term calculus. *Journal of Functional Programming*, 5(1):65–80, January 1995.

[6] Josh Berdine, Peter O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linear continuation-passing. *Higher-order and Symbolic Computation*, 15(2/3):181–208, 2002.

[7] Don Box and Chris Sells. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, 2003.

[8] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, 1996.

[9] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. Conference on Programming Language Design and Implementation*, pages 13–24. ACM Press, June 2002.

[10] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone user's manual, 2003. `http://www.cs.cornell.edu/projects/cyclone/online-manual/`.

[11] Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. In *Proc. European Symposium on Programming*, volume 1782 of *LNCS*, pages 165–179. Springer-Verlag, 2000.

[12] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[14] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *Proc. Int. Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Lousiana, 1995. Elsevier.

[15] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *Proc. Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[16] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proc. Conference on Programming Language Design and Implementation*, pages 81–91, Snowbird, Utah, June 2001. ACM Press.

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

[18] 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, 1999.

[19] George C. Necula. Proof-carrying code. In *Proc. Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[20] Peter W. O'Hearn and John C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, 2000.

[21] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *Proc. symposium on Principles of Programming Languages*, pages 172–184. ACM Press, 2003.

[22] Rinus Plasmeijer and Marco van Eekelen. Clean language report version 2.0, 2002. `http://www.cs.kun.nl/~clean/`.

[23] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: The semantics of BI. *Theoretical Computer Science*, 2003. To appear.

[24] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[25] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer-Verlag, 2000.

[26] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1–2):231–248, 1999.

[27] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Proc. IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[28] David Wakeling and Colin Runciman. Linearity and laziness. In *Functional Programming Languages and Computer Architecture*, pages 215–240, 1991.

[29] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[30] David Walker and Greg Morrisett. Alias types for recursive data structures. In R. Harper, editor, *Proc. Int. Workshop on Types in Compilation*, volume 2071 of *LNCS*, pages 177–206, Montreal, Canada, September 2001. Springer-Verlag.

[31] David Walker and Kevin Watkins. On regions and linear types. In *Proc. Int. Conference on Functional Programming*, pages 181–192, 2001.

[32] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. *ACM SIGPLAN Notices*, 36(3):166–178, 2001.

[33] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *International Conference on Functional Programming*, pages 169–180, 2001.

[34] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. European Symposium on Programming*, April 2003. To appear.

A Typing Rules

A.1 Type Well-Formedness

Formally, type contexts Δ map type variables to signs $+, -, 0$ which we use to restrict recursive type variables to positive occurrences. We write Δ^- for the context Δ with all $+$'s and $-$'s interchanged.

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

$$\Delta \vdash \mathsf{word} \qquad \Delta \vdash \mathsf{int}$$

$$\frac{\Delta, \alpha^0 \vdash \tau}{\Delta \vdash \forall \alpha.\tau} \qquad \frac{\Delta^- \vdash \omega_i}{\Delta \vdash \mathtt{code}\{r_1 : \omega_1, \ldots, r_n : \omega_n\}}$$

$\boxed{\Delta \vdash \omega}$

$$\frac{s \in \{0, +\}}{\Delta, \alpha^s \vdash \alpha} \qquad \frac{\Delta \vdash \sigma}{\Delta \vdash @\langle\sigma\rangle} \qquad \frac{\Delta \vdash \sigma}{\Delta \vdash ?\langle\sigma\rangle}$$

$$\frac{\Delta, \alpha^+ \vdash \omega \qquad \omega \neq \alpha}{\Delta \vdash \mu\alpha.\omega} \qquad \frac{\Delta, \alpha^0 \vdash \omega}{\Delta \vdash \exists \alpha.\omega}$$

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

$$\frac{\Delta \vdash \omega_1 \qquad \Delta \vdash \omega_2}{\Delta \vdash \sigma_1 \otimes \sigma_2}$$

$\boxed{\vdash \Psi}$

$$\vdash \bullet \qquad \frac{\bullet \vdash \forall \vec{\alpha}.\mathtt{code}(\Gamma) \qquad \vdash \Psi}{\vdash \Psi, f : \forall \vec{\alpha}.\mathtt{code}(\Gamma)}$$

A.2 Code Well-Formedness

The remaining rules refer implicitly to a global well-formed context Ψ_0.

$\boxed{\Delta; \Gamma \vdash op : \tau}$

$$\Delta; \Gamma, r : \tau \vdash r : \tau \qquad \Delta; \Gamma \vdash f : \Psi_0(f)$$

$$\Delta; \Gamma \vdash i : \mathsf{int} \qquad \Delta; \Gamma, r : \tau \vdash r : \mathsf{word}$$

$$\frac{\Delta \vdash \omega \qquad \Delta; \Gamma \vdash op : \forall \alpha.\tau}{\Delta; \Gamma \vdash op : \tau[\omega/\alpha]}$$

$\boxed{\Delta; \Gamma \vdash op : \omega \mid \Delta'; \Gamma'}$

$$\frac{\Delta; \Gamma \vdash op : \tau}{\Delta; \Gamma \vdash op : \tau \mid \Delta; \Gamma} \qquad \Delta; \Gamma, r : \omega \vdash r : \omega \mid \Delta; \Gamma$$

$$\frac{\alpha \notin \Delta \qquad \Delta; \Gamma \vdash op : \exists \alpha.\omega \mid \Delta'; \Gamma'}{\Delta; \Gamma \vdash op : \omega \mid \Delta', \alpha; \Gamma'}$$

$$\frac{\Delta; \Gamma \vdash op : \omega[\omega'/\alpha] \mid \Delta'; \Gamma'}{\Delta; \Gamma \vdash op : \exists \alpha.\omega \mid \Delta'; \Gamma'} \qquad \frac{\Delta; \Gamma \vdash op : \mu\alpha.\omega \mid \Delta'; \Gamma'}{\Delta; \Gamma \vdash op : \omega[\mu\alpha.\omega/\alpha] \mid \Delta'; \Gamma'}$$

$$\frac{\Delta; \Gamma \vdash op : \omega[\mu\alpha.\omega/\alpha] \mid \Delta'; \Gamma'}{\Delta; \Gamma \vdash op : \mu\alpha.\omega \mid \Delta'; \Gamma'}$$

$\boxed{\Delta; \Gamma \vdash i \mid \Delta'; \Gamma'}$

$$\frac{\Delta; \Gamma \vdash r : \tau \qquad \Delta; \Gamma \vdash r' : \mathsf{int} \qquad \Delta; \Gamma \vdash op : \mathsf{int}}{\Delta; \Gamma \vdash \mathtt{arith}\ r, r', op \mid \Delta; \Gamma, r : \mathsf{int}}$$

$$\frac{\Delta; \Gamma \vdash r : \mathsf{int} \qquad \Delta; \Gamma \vdash op : \mathtt{code}(\Gamma)}{\Delta; \Gamma \vdash \mathtt{bnz}\ r, op \mid \Delta; \Gamma}$$

$$\frac{\Delta; \Gamma \vdash r : ?\langle\sigma\rangle \mid \Delta'; \Gamma' \qquad \Delta'; \Gamma' \vdash op : \mathtt{code}(\Gamma', r : @\langle\sigma\rangle)}{\Delta; \Gamma \vdash \mathtt{bnz}\ r, op \mid \Delta''; \Gamma'}$$

$$\frac{\Delta; \Gamma \vdash r : \mathsf{word} \qquad \Delta; \Gamma \vdash r' : @\langle\omega_0 \otimes \omega_1\rangle \mid \Delta'; \Gamma'}{\Delta; \Gamma \vdash \mathtt{ld}\ r, r'[0] \mid \Delta'; \Gamma', r' : @\langle\mathsf{word} \otimes \omega_1\rangle, r : \omega_0}$$

$$\frac{\Delta; \Gamma \vdash r : \mathsf{word} \qquad \Delta; \Gamma \vdash r' : @\langle\omega_0 \otimes \omega_1\rangle \mid \Delta'; \Gamma'}{\Delta; \Gamma \vdash \mathtt{ld}\ r, r'[1] \mid \Delta'; \Gamma', r' : @\langle\omega_0 \otimes \mathsf{word}\rangle, r : \omega_1}$$

$$\frac{\Delta; \Gamma \vdash r : \mathsf{word} \qquad \Delta; \Gamma \vdash op : \omega \mid \Delta'; \Gamma'}{\Delta; \Gamma \vdash \mathtt{mov}\ r, op \mid \Delta'; \Gamma', r : \omega}$$

$$\frac{\Delta; \Gamma \vdash r : @\langle\mathsf{word} \otimes \omega_1\rangle \mid \Delta'; \Gamma' \qquad \Delta'; \Gamma' \vdash r' : \omega_0 \mid \Delta''; \Gamma''}{\Delta; \Gamma \vdash \mathtt{st}\ r[0], r' \mid \Delta''; \Gamma'', r : @\langle\omega_0 \otimes \omega_1\rangle, r' : \mathsf{word}}$$

$$\frac{\Delta; \Gamma \vdash r : @\langle\omega_0 \otimes \mathsf{word}\rangle \mid \Delta'; \Gamma' \qquad \Delta'; \Gamma' \vdash r' : \omega_1 \mid \Delta''; \Gamma''}{\Delta; \Gamma \vdash \mathtt{st}\ r[1], r' \mid \Delta''; \Gamma'', r : @\langle\omega_0 \otimes \omega_1\rangle, r' : \mathsf{word}}$$

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

$$\frac{\Delta; \Gamma \vdash op : \mathtt{code}(\Gamma)}{\Delta; \Gamma \vdash \mathtt{jmp}\ op} \qquad \Delta; \Gamma \vdash \mathtt{halt}$$

$$\frac{\Delta; \Gamma \vdash i \mid \Delta'; \Gamma' \qquad \Delta'; \Gamma' \vdash I}{\Delta; \Gamma, \vdash i; I}$$

A.3 State Well-Formedness

$\boxed{H \models R : \Gamma}$

$$\bullet \vdash R : \bullet \qquad \frac{H \models R : \Gamma \qquad H' \models R(v) : \omega}{H, H' \models R : \Gamma, r : \omega}$$

$\boxed{H \models h : \sigma}$

$$\frac{H \models v_1 : \omega_1 \qquad H' \models v_2 : \omega_2}{H, H' \models \langle v_1, v_2\rangle : \omega_1 \otimes \omega_2}$$

$\boxed{H \models v : \omega}$

$$\bullet \models v : \tau \qquad \frac{H \models h : \sigma}{H, l \mapsto h \models l : @\langle\sigma\rangle}$$

$$\bullet \models 0 : ?\langle\sigma\rangle \qquad \frac{H \models v : @\langle\sigma\rangle}{H \models v : ?\langle\sigma\rangle}$$

$$\frac{H \models v : \omega[\mu\alpha.\omega/\alpha]}{H \models v : \mu\alpha.\omega} \qquad \frac{\bullet \vdash \omega \qquad H \models v : \omega[\omega'/\alpha]}{H \models v : \exists\alpha.\omega}$$

A.4 Program Well-Formedness

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

$$\vdash \bullet : \bullet \qquad \frac{\vdash C : \Psi \qquad \vec{\alpha}; \Gamma \vdash I}{\vdash C, f \mapsto I : \Psi, f{:}\forall\vec{\alpha}.\mathtt{code}(\Gamma)}$$

$\boxed{\vdash P}$

$$\frac{\vdash C : \Psi_0 \qquad H \models R : \Gamma \qquad \bullet; \Gamma \vdash I}{\vdash (H, R, I)}$$