LOW-LEVEL TYPE SYSTEMS FOR MODULARITY AND OBJECT-ORIENTED CONSTRUCTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Arthur Neal Glew

January 2000

LOW-LEVEL TYPE SYSTEMS FOR MODULARITY AND OBJECT-ORIENTED CONSTRUCTS

Arthur Neal Glew, Ph.D.
Cornell University 2000

Typed Assembly Language (TAL) is a formal language for an idealised machine augmented with type annotations, typing rules, and a memory allocation primitive. TAL's type system is sound; that is, well typed TAL programs do not commit run-time type errors during execution. This guarantee can be used to debug type-directed compilers and to build more general security properties in an extensible system. This dissertation presents a basic version of TAL and extensions to support the compilation of modules and object-oriented languages. First, it describes a modular version of TAL that consists of typed object files, linking operations, and link compatibility conditions. Together these features provide for type-sound separate compilation and substantially extend previous work on linking. Second, it shows how to use a new formulation of self quantifiers to compile an efficient implementation of a single-inheritance class-based object-oriented language into TAL. Third, it presents a new type constructor called a *tag type*, and shows how to use them to compile downcasting and exceptions into TAL.

Biographical Sketch

Neal Glew was born in Wellington, New Zealand, in 1972. He attended Bellevue Primary School, Newlands Intermediate School, and Newlands College. During this time he was introduced to a succession of home computers: first one based on a 6809, then one based on a Z80, and finally a series of PCs. In 1990 he started a BSc in Mathematics and Computer Science at Victoria University of Wellington, which he completed in November 1992. A year later he completed a BSc(hons) in Computer Science, and received both degrees in June of 1994. During this time, Neal wrote two custom software packages and successfully sold a number of copies. Neal began his Ph.D. in August of 1994 at Cornell University. In 1996 he did a summer internship at Grammatech, and in 1997 at the Systems Research Center of Digital Equipment Corporation (now Compaq). For the fall of 1999, Neal will be an instructor at Cornell University, and beyond that he knows not what.

Acknowledgements

This dissertation describes work supported in part by NSF grants CCR-9317320 and CCR-9708915, AFOSR grant F49620-97-1-0013, and ARPA/RADC grant F30602-1-0317. This dissertation does not reflect the views of these agencies.

First and foremost I thank my advisers. Dexter Kozen has given me both the freedom to pursue my own research and the occasional advice and guidance that I have needed. I have also appreciated playing hockey, playing rugby, and enjoying a quiet beer with him. Greg Morrisett has exposed me to a wealth of programming language theory, and has taught me by example the skills of a researcher. I feel particularly indebted for his efforts on our MTAL paper to improve my writing.

I have benefited greatly from interactions with other students at Cornell. I would especially like to thank Dave Walker for many stimulating conversations, and for his and other TALC project members's many proof readings of my publications. I also thank my friends and relatives for their emotional support and entertainment, which has made my life much richer.

Finally I thank my parents, without whom I would not be here. They have always provided me with loving support, encourged my inquisitiveness, given me access to resources, and supported my career path no matter where it took me.

Table of Contents

List of Figures

Chapter 1

Introduction

The goal of this dissertation is to show that it is possible to design types systems with low-level abstractions for machine languages such that the output of a variety of compilation strategies for a variety of source constructs will type check. More specifically, it will present a typed assembly language (MooTal) and show how to compile a prototypical procedural language, modules, objects, classes, and run-type type dispatch to MooTal. Type systems are very good at checking certain safety properties and have been used effectively by programmers to catch certain errors at compile time. As described below, type systems have been used more recently to debug compilers, check safety of compilation, and provide security in extensible systems. To realise these benefits fully, we need type systems not only for source language and high-level intermediate languages but also for low-level intermediate and target languages. My group at Cornell designed an initial Typed Assembly Language (Tal), and showed how to compile a core functional language to Tal. My research extended this effort to support separate compilation and the compilation of objects, classes, and run-time type dispatch, as described further below. I will expand of these points in the next three sections and then outline the rest of the dissertation.

1.1 Type-Directed Compilation

Traditional compilers parse a program, type check it, discard the type information, and generate target code. An alternative is to retain type information through compilation, and use it to drive analysis and optimisation. Recently, there have been numerous efforts to build such compilers [TMC$^+$96, Sha97, TDMW97, BRTT93, PHH$^+$93]. These compilers translate the type information along with the code and use the type information to guide translation and optimisation, enabling implementation techniques that would be impossible or difficult otherwise [TMC$^+$96]. Additionally, the type information is used to debug the compiler. After each compilation stage, the intermediate form is type checked. If type checking fails, then there is a bug in that stage (or the type checker). While this idea does not catch all errors, in practice it is useful [MTC$^+$96].

However, in all of these compilers, there is stage where type information is discarded. For example, the TIL/ML compiler [TMC$^+$96] retains type information only until code generation. In order to realize the benefits of type-directed compilation all the way to target code, *typed target languages* such as Tal are needed. With such languages, we could debug aspects of compilation such as calling conventions, register allocation, and instruction scheduling—all details that can be tedious to get right.

1.2 Language-Based Security

A number of modern systems allow extended functionality through foreign code. For example, web pages may contain Java applets, which are code fragments intended to run inside the web browser. However, these applets are written by someone the web surfer may not trust. The web surfer may wish to run the applets with a guarantee of security, such as, "the applet cannot trash my files." The *secure code problem* is to check, instrument, and/or contain untrusted code so as to guarantee desired security properties. As well as web applets, it also arises in extensible web servers, extensible operating systems, and active networks.

TAL is one of many approaches to the secure code problem (Kozen provides an excellent overview of language-based security [Koz99] and there are many systems [WLAG93, BSP+95, LY96, Nec98, Koz98, *etc.*]). Extensible systems based on TAL would require extensions to be written in TAL. Extensions that do not type check are rejected. TAL is type safe, so the extensions are guaranteed not to commit run-time type errors. Extensible systems would use this guarantee to interpose a security monitor between extensions and critical resources. The security monitor can implement many different security policies. Since TAL is assembly code, potentially any language could be compiled to TAL by any compiler, even an aggressive optimising compiler. However, this goal will only be realised if TAL's type system is expressive enough to type check the code that compilers produce. The goal of the TAL project is to achieve this expressiveness by identifying key abstractions at the machine level and designing type mechanisms for them.

1.3 TAL

Typed Assembly Language was introduced by Morrisett et al. [MWCG98]. It is a statically-typed variant of a conventional RISC assembly language motivated by the problems of type-directed compilation and secure extensible systems. TAL has three important properties: first, TAL is type safe; second, it is possible to compile real programming languages to TAL; third, there is a connection source-level constructs and what TAL's type systems checks for.

Morrisett et al. [MWCG99] describes the original TAL, proves type safety, and shows how to translate System F [Gir71, Rey74] to TAL. The latter demonstrates (in theory) that ML-like languages could be compiled to TAL, and that System F's abstractions are related to TAL's abstractions. A major shortcoming of the original TAL was the absence of a stack abstraction. Most compilers use a run-time stack to store activation frames, and modern processor architectures have hardware support for a stack-based approach. Morrisett et al. [MCGW98a] extends the original TAL to include a stack abstraction and provides additional typing constructs that allow flexible use of the stack. Later work showed that a simple stack-based compilation strategy is compatible with the type system. Morrisett et al. also show how to specify, as formal type translations, various calling conventions. They include numerous variations: passing parameters and results on the stack or in registers, caller or callee argument reclamation, and callee-saves registers.

To further show that TAL is a reasonable compiler target, and to investigate its practicality, Morrisett et al. [MCG+99] implemented a version of TAL for Intel's 32-bit architecture (IA32) called TALX86. TALX86 includes a rich type-constructor language expressive enough for numerous source constructs needed in a realistic language. Morrisett et al. also built a particular type-directed compiler for a safe C-like language called Popcorn.

However, the TAL described so far lacks a number of important features. First, it considers only whole programs—programmers cannot divide their programs into separate TAL files and type check them in isolation, nor can they use abstract data types. Second, while procedural and functional languages can be compiled to TAL, object-oriented languages cannot. Efficient compiled object-oriented code will not type check because TAL's type system cannot express object encodings or run-time type dispatch. I will elaborate on these problems in the rest of this section.

Modules To make project management feasible, large software projects are divided into separate compilation units, and increasingly such projects are realised as a set of components rather than monolithic programs. These compilation units and components are compiled and built separate from other compilation units and components, requiring only the interfaces of the components they refer to. For TAL to be effective as a typed target language, programmers must be able to produce and type check TAL code for a separately compiled unit. The original TAL design allowed only complete programs, so I designed a module system to allow incomplete TAL fragments. A TAL module is based on conventional object files and includes code, data, and a list of imported and exported labels and their types. The module system also defines *link compatibility*: conditions sufficient to ensure that linking two fragments together produces well-formed output. The module system also provides a theory of conventional linking, extending previous work by Cardelli [Car97]. A theory of linking and link-compatibility conditions are vital to language-based security, as these systems rely on the type-safety guarantee especially the linking checks.

Object Encodings Objects are an important and popular construct. They provide data encapsulation, abstraction, and extensibility conveniently in one package. As machines do not have objects, some compiler stage must translate objects into more primitive constructs, usually functions and records. These translations are known as *object encodings*, and there are many examples in the literature. Not many of these encodings have all the desired theoretical properties, nor, as I shall argue in Chapter 5, are they adequate for implementation. The essence of the problem is typing *self* (`self` in Smalltalk, `this` in Java). None of the existing approaches are able to capture *self*'s type without adding run-time overhead. To solve this problem, I devise a new formulation of self quantifiers and use it to type *self*.

A problem related to object encodings is *class encodings*: translating classes into more primitive constructs. Some class encodings translate classes into objects; others translate classes and objects simultaneously into records and functions. Several class encodings have been proposed, but they differ from what most compilers do. I start with an efficient encoding for single-inheritance class-based languages and show how to type it using self quantifiers and F-bounded polymorphism [CCH+89]. Together these two encodings provide a basis for the compilation of object-oriented languages to TAL.

Downcasting and Other Type Dispatch I have investigated one other object oriented construct: run-time type dispatch. An example is Java's downcasting operation `(`c`)`e. Suppose the variable `glew` has static type `Person`, but actually contains an instance of `GraduateStudent`. Then the expression `(Student)glew` evaluates `glew` to an object and checks if that object is an instance of `Student` or one of its subclasses. In this case it is, so the expression's result is this object with static type `Student`. If it were not, then an exception would be thrown. A typical implementation of this construct includes in every object a pointer to the class from which it was instantiated. To perform the cast, the object's run-time class is retrieved and compared against the target class of the cast. If this comparison succeeds, then so does the cast. For the type system to change the type of the object, it must infer the change from the successful comparison of the two classes. To do this, the type system must know that the classes represent type information and must be able to link the object to its class. I devised a solution to this problem I call *type tagging*. The solution applies to other run-time type-dispatch mechanisms including ML-style exception matching, hierarchical extensible sums, and multimethod dispatch (as in Cecil [Cha97] and Dylan [SMS96]).

1.4 Overview of the Dissertation

In this dissertation I describe Typed Assembly Language and my contributions to it. I begin with a description of the original work on TAL [MWCG98, MWCG99, MCGW98a, MCGW98b], which forms the basis for my own contributions. The bulk of the dissertation contains my theoretical extensions to TAL. The first is an extension to support separate compilation and abstract types. It provides both separate type checking and a theory of linking. The second is an object closure-conversion translation and a proof of its correctness. While similar work exists for lambda calculi, the object-oriented framework is a simpler setting, and the translation formalises the well known connection between objects and closures. The third is a new object and class encoding, including the type machinery necessary at the TAL level to support the compilation of many class-based object-oriented languages. The fourth is a tagging construct and its implementation in TAL. This construct is at the core of a number of type dispatch mechanisms such as class case, class cast, ML-style and Java-style exception matching, hierarchical extensible sums, and multimethods. After my contributions I describe TALX86 [MCG$^+$99], an implementation of TAL for IA32. This implementation addresses many practical concerns that the theoretical work does not, and provides some evidence that typed target languages are practical and that type-directed compilation to them is feasible.

The dissertation describes several different typed assembly languages. Rather than formalise all of them, I present one formalised typed assembly language, which I call MOOTAL (Modular and Object Oriented TAL) and describe the relevant fragment of it in each chapter. A reference description of MOOTAL appears in Appendix A along with full technical details.

Chapter 2

Typed Assembly Language

Typed Assembly Language (TAL) was introduced by Morrisett et al. [MWCG98, MWCG99, MCGW98a, MCGW98b]. It is an idealised machine language augmented with typing rules. The goals are: First, type-correct programs do not commit run-time type errors. Second, typing is decidable. Third, it is possible to compile realistic source languages into type-correct TAL. To satisfy these goals, TAL includes typing annotations and a memory management primitive that conventional machine languages lack. This chapter will explain these annotations, the typing rules, and how the machine is formalised. Then, it will present a small low-level language and show how to compile it into TAL.

Notational Conventions There is some set of labels, registers, and type constructors variables. Labels ℓ are used to name both types and values for intermodule references, and are used as the addresses of memory. The set of registers (ranged over by r) could be a countably infinite set of virtual registers or a finite set of physical registers. Type constructor variables are ranged over by α and β. Integers are ranged over by i. Syntactic objects are considered equal up to α-equivalence. The capture-avoiding substitution of x for y in z is written $z\{x := y\}$. An unordered map that maps x_i to y_i is written $\{x_1{:}y_1, \ldots, x_n{:}y_n\}$ for type-level constructs and $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ for term-level constructs. It is a syntactic restriction that the x_i be distinct. The domain of a map X is written $\mathrm{dom}(X)$, the value of X at x is written $X(x)$, and map update is written $X\{x{:}y\}$ or $X\{x \mapsto y\}$. Syntactic objects are considered equal up to reordering of unordered maps. An ordered map is written $x_1{:}y_1, \ldots, x_n{:}y_n$; it is a syntactic restriction that the x_i be distinct. The notation $X, x{:}y$ denotes $x{:}y$ appended to X; X_1, X_2 denotes X_2 appended to X_1. A vector of objects from syntax class x is written $\vec{x}$, for example, $\vec{\alpha}$ stands for $\alpha_1, \ldots, \alpha_n$; the notation $\overrightarrow{\alpha{:}\kappa \le c}$ will be used to denote sequences like $\alpha_1{:}\kappa_1 \le c_1, \ldots, \alpha_n{:}\kappa_n \le c_n$. Generally typing judgements have the forms $C \vdash x$, $C \vdash x : X$, or $C \vdash x_1 R x_2$ to mean that in context C, x is well formed, x has "type" X, or x_1 is R related to x_2 (R might be equality, subtyping, compatibility, or disjointness). Typing rules have the form

$$(\mathrm{n})\ \frac{J_1 \quad \cdots \quad J_n}{J}\ (P)$$

where n is the rule name, P is a side condition, J is the conclusion judgement, and J_1 through J_n are the hypothesis judgements. The meaning of the rule is that if J_1 through J_n are derivable and P holds then J is derivable. Sometimes the hypotheses will be indexed as in $C \vdash x_i : X_i$ where there are x_1 through x_n and X_1 through X_n somewhere else in the typing rule. I will not write out explicitly what i ranges over; it should be clear. In particular, in:

$$\frac{C \vdash x_i \leq y_i \quad C \vdash x_j}{C \vdash x_1, \ldots, x_m \leq y_1, \ldots, y_n} \ (m \geq n)$$

i ranges over $1..n$ and j ranges over $1..m$.

2.1 The TAL Machine

A typical modern machine consists of a processor with a register file and an arithmetic and logic unit, a memory system, and an I/O system. It operates by fetching and then executing instructions from memory at addresses given by the program counter. Instruction execution might involve fetching values from registers and memory, performing an arithmetic operation, and storing the result back either to a register or to memory. The TAL machine is similar in that it includes a memory system, a register file, and a current sequence of instructions to execute. One step of the TAL machine involves executing the first instruction of the current instruction sequence. Unlike a real machine, the TAL machine does not have an I/O system. Like the lambda calculus, it just computes a result and halts.

TAL's memory system is divided into two parts, the heap and the stack. The *heap* stores the program's static code, static data, and dynamically allocated memory. The *stack* stores function activation records (which store parameters, return addresses, and local variables for invocations of functions). The stack is actually part of the register file: a special stack-pointer register stores the stack. Thus, a TAL program state is formalised as a triple (VH, R, I) where VH is the heap, R is the register file, and I is the current instruction sequence. The metavariable P ranges over arbitrary TAL program states.

The heap is divided into blocks; intuitively, you can think of these blocks as those dynamically allocated by a memory-management primitive. The blocks are addressed using labels and, to help the type checker, must be annotated with their types. Thus heaps, ranged over by the metavariable VH, have the form $\{\ell_1 \mapsto h_1{:}c_1, \ldots, \ell_n \mapsto h_n{:}c_n\}$ where the ℓ_i are the addresses of the blocks, the h_i are the contents of the blocks called heap values, and the c_i are the types of the blocks.[1] A *heap value* consists of a type-variable abstraction part and the heap value proper. The *type-variable abstraction part* has the form $\Lambda[t_1, \ldots, t_n]$ where the t_i are type-variable declarations and abstracts over type variables that may appear free in the type annotations in the heap value proper. *Type-variable declarations* are ranged over by the metavariable t and, for now, have the form $\alpha{:}\kappa$ where α is a type variable, and κ is its kind.[2] *Heap values proper* come in two forms: code and data. Code has the form $\mathsf{code}\ I$ where I is a sequence of instructions, and data has the form $\langle w_1, \ldots, w_n \rangle$ where w_i are word values.

The *register file* maps TAL's registers to word values. Additionally, a register file maps the stack pointer sp to the stack. Thus, a register file has the form $\{\mathsf{sp} \mapsto S, r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$ where S is a stack, r_i are registers, and w_i are word values. A stack is either empty, written se, or consists of a word value w pushed onto another stack S, written $w :: S$.

The word values include integers, labels, stack pointers, written $\mathsf{sptr}(i)$ where i is an offset in words from the bottom of the stack, junk, nonsense, and coercions. Because the type system teats heaps and stacks differently, TAL has two different word values represented uninitialised data. *Junk*, written $?_c$, represents uninitialised heap data, and is annotated with c, the type

[1] TAL's type level includes types and more general type constructors. The usual metavariable for type constructors is c rather than τ.

[2] A kind is a "type" for a type.

Kinds	κ	$::=$	$\mathsf{T} \mid \mathsf{M} \mid \mathsf{S}$
Variances	ϕ	$::=$	$+ \mid - \mid \circ \mid 0$
Type Variable Declaration	t	$::=$	$\alpha{:}\kappa$
Types	c	$::=$	$\alpha \mid \forall t.c \mid \mathsf{int} \mid \mathsf{ns} \mid *c \mid \mathsf{sptr}(c) \mid$ $\mathsf{code}\ \Gamma \mid \langle c_1^{\phi_1}, \ldots, c_n^{\phi_n} \rangle \mid$ $\mathsf{se} \mid c_1 :: c_2 \mid c_1 \circ c_2$
Register File Types	Γ	$::=$	$\{\mathsf{sp}{:}c, r_1{:}c_1, \ldots, r_n{:}c_n\}$
Type Variable Contexts	Δ	$::=$	$\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n$
Heap Types	Ψ	$::=$	$\{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\}$
Coercions	δ	$::=$	$[c]$
Small Values	v	$::=$	$i \mid \ell \mid r \mid ?_c \mid \mathsf{ns} \mid \mathsf{sptr}(i) \mid \delta(v)$
Word Values	w	$::=$	$i \mid \ell \mid ?_c \mid \mathsf{ns} \mid \mathsf{sptr}(i) \mid \delta(v)$
Instructions	ι	$::=$	$aop\ r_d, v_1, v_2 \mid bop\ r, v \mid$ $\mathsf{malloc}\ r, \langle c_1, \ldots, c_n \rangle \mid \mathsf{mov}\ r, v \mid$ $\mathsf{mov}\ r_d, [r_s + i] \mid \mathsf{mov}\ [r_d + i], r_s \mid$ $\mathsf{mov}\ \mathsf{sp}, \mathsf{sp} + i \mid \mathsf{mov}\ r, \mathsf{sp} \mid \mathsf{mov}\ \mathsf{sp}, r \mid$ $\mathsf{mov}\ r_d, [\mathsf{sp} + i] \mid \mathsf{mov}\ [\mathsf{sp} + i], r_s$
Instruction Sequences	I	$::=$	$\iota; I \mid \mathsf{halt}[c] \mid \mathsf{jmp}\ v$
Heap Values	$\hat{h}$	$::=$	$\mathsf{code}\ I \mid \langle w_1, \ldots, w_n \rangle$
	h	$::=$	$\Lambda[t_1, \ldots, t_n]\hat{h}$
Stacks	S	$::=$	$\mathsf{se} \mid w :: S$
Register Files	R	$::=$	$\{\mathsf{sp} \mapsto S, r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
Heaps	VH	$::=$	$\{\ell_1 \mapsto h_1{:}c_1, \ldots, \ell_n \mapsto h_n{:}c_n\}$
Executables	E	$::=$	(VH, ℓ)
Program States	P	$::=$	(VH, R, I)

Figure 2.1: TAL Syntax

of the word value that may eventually initialise the that part of the heap. *Nonsense*, written ns, represents uninitialised stack data, and it does not require an annotation because any word value may replace it. Finally, *coercions* are one of the type annotations. They have the form $\delta(w)$ where δ is the coercion itself, and w is the word value being coerced. The meaning of a coercion is w except the type is changed by δ from w's type to something else. For now there is just one coercion, type application, written $[c]$, which instantiates a polymorphic type.

An *executable* for the TAL machine, ranged over by the metavariable E, has the form (VH, ℓ) where VH the initial heap of the executable and ℓ is the entry label. The initial program state for an executable (VH, ℓ) has VH as its heap, a register file with an empty stack $\{\mathsf{sp} \mapsto \mathsf{se}\}$, and an instruction sequence that jumps to ℓ. To summarise the machine so far, the full syntax for TAL appears in Figure 2.1. Next I will discuss the instructions and execution of the machine.

2.2 Instructions and Execution

The TAL instruction set includes conventional assembly operations such as arithmetic, conditional and unconditional branches, loads, and stores. Additionally, for allocating a new heap block, it has one special instruction not found on real machines. The execution of one machine instruction is formalised as a reduction relation between program states, written $P_1 \mapsto P_2$; the transitive reflexive closure of this relation is written $P_1 \mapsto^* P_2$, and $P \not\mapsto$ denotes that P cannot make a transition.

An *instruction sequence* is either a terminal instruction $\mathsf{halt}[[]c]$ or $\mathsf{jmp}\ v$, or a sequence $\iota; I$ consisting of an instruction ι followed by the rest of the instructions I. The terminal instruction $\mathsf{halt}[c]$ halts the machine where c is the type of the result, which by convention is in register $\mathsf{r1}$. The terminal instruction $\mathsf{jmp}\ v$ is an unconditional jumps to v, an operand. *Operands*, ranged over by metavariable v, include literals and registers; formally, v is either a register or one of the word-value forms.

The *instructions*, ranged over by metavariable ι, are: The arithmetic instruction $aop\, r, v_1, v_2$ performs the operation aop on the operands v_1 and v_2 and stores the result into r. The conditional branch instruction $bop\ r, v$ tests condition bop of r, and if the condition holds, it jumps to v. The allocation instruction $\mathsf{malloc}\ r, \langle c_1, \ldots, c_n\rangle$ creates a new heap block of size n and stores the block's address into r. The fields of the block may eventually hold values of types c_i. The move instruction $\mathsf{mov}\, r, v$ stores v into r. The load instruction $\mathsf{mov}\, r_d[r_s+i]$ loads the ith field of the heap block addressed by r_s into r_d. The store instruction $\mathsf{mov}\ [r_d + i], r_s$ stores r_s into the ith field of the heap block addressed by r_d. The instruction $\mathsf{mov}\ \mathsf{sp}, \mathsf{sp} + i$ allocated or deallocates space on the stack where i is the number of words to remove from the stack (add if i is negative).[3] The instruction $\mathsf{mov}\ r, \mathsf{sp}$ moves the stack pointer into r. The instruction $\mathsf{mov}\ \mathsf{sp}, r$ moves r into the stack pointer. The stack load instruction $\mathsf{mov}\ r_d, [\mathsf{sp}+i]$ loads the word ith from the top of the stack into r_d. The stack store instruction $\mathsf{mov}\ [\mathsf{sp}+i], r_s$ stores r_s into the word ith from the top of the stack.[4]

Operands evaluate to word values. Given a register file R and an operand v, $\hat{R}(v)$ is the "value" of v. It maps literals to themselves and registers to their values in R:

$$\hat{R}(v) = \begin{cases} v & v = i, \ell, ?_c, \mathsf{ns}, \mathsf{sptr}(i) \\ R(r) & v = r \\ \delta(\hat{R}(v')) & v = \delta(v') \end{cases}$$

As an example of the definition of the reduction relation, consider an arithmetic operation. If the arithmetic operation aop corresponds to the binary operation $||aop||$ on integers, then the reduction rule is:

$$(VH, R, aop\ \ r, v_1, v_2; I) \mapsto (VH, R\{r \mapsto \hat{R}(v_1)||aop||\hat{R}(v_2)\}, I)$$

Where $R\{r \mapsto w\}$ is map update (it maps r to w and other registers as R does).

As another example, consider an unconditional branch. Intuitively, the operand v should contain a pointer to a heap block containing an instruction sequence. However, the heap block can be polymorphic, in which case the operand will include a number of type instantiation coercions. The final machine state's instruction sequence will be the instruction sequence pointed to by v instantiated by the coercions in v:

$$\begin{array}{l}(VH, R, \mathsf{jmp}\ \ v) \mapsto (VH, R, I\{\alpha_1, \ldots, \alpha_n := c_1, \ldots, c_n\}) \\ \text{where } \hat{R}(v) = [c_n](\cdots[c_1](\ell)\cdots);\ VH(\ell) = \Lambda[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n]\mathsf{code}\ \ I\end{array}$$

[3] As with most machines, TAL's stack grows downward from the top of memory.

[4] STAL [MCGW98a, MCGW98b] includes instructions to load and store from the stack via pointers in general registers. These instructions can be used to implement displays and other stack-allocated data. Because these features are unimportant for this dissertation, I omitted these instructions for simplicity.

The notation $x\{y := z\}$ stands for the capture-avoiding substitution of z for y in x. Note that $[c_n](\cdots[c_1](\ell)\cdots)$ is a word value that is ℓ with n type application coercions applied to it, the types c_i instantiate the polymorphic code addressed by ℓ.

As a final example, consider allocation. A new label is chosen, and the heap is updated to map the new label to a tuple of junk values. The destination register is mapped to the new label:

$$(VH, R, \mathsf{malloc}\ r, \langle c_1, \ldots, c_n \rangle; I) \mapsto (VH\{\ell \mapsto \langle ?_{c_1}, \ldots, ?_{c_n} \rangle\}, R\{r \mapsto \ell\}, I)$$
$$\text{where } \ell \notin \mathrm{dom}(VH)$$

Full details of the reduction relation appear in the appendix (or the original TAL and STAL papers [MWCG98, MWCG99, MCGW98a, MCGW98b]).

2.3 Types and Typing Rules

All of TAL's values are given types, including heap values, stacks, and word values. TAL's types are classified by kinds, and TAL has a three-tiered system like F_ω [Gir71, Gir72].[5] The kinds κ are T for word types, M for heap value types, and S for stack types.

There are many judgements in TAL's type system. The judgements used in this chapter are summarised in the following table:

Judgement	Meaning
$\Delta \vdash_{\mathsf{tc}} c : \kappa$	c has kind κ
$\Delta \vdash c_1 = c_2 : \kappa$	c_1 and c_2 are equal
$\vdash_{\mathsf{VHT}} \Psi$	Heap type Ψ is well formed
$\Psi; \Delta; \Gamma \vdash v : c$	Word value/operand v has type c
$\Psi; \Delta \vdash_{\hat{\mathsf{h}}} \hat{h} : c$	Heap value proper $\hat{h}$ has type c
$\Psi \vdash_{\mathsf{h}} h : c$	Heap value h has type c
$\Psi; \Delta_1; \Gamma_1 \vdash_{\mathsf{i}} \iota : \Delta_2; \Gamma_2$	Instruction typing (see below)
$\Psi; \Delta; \Gamma \vdash_{\mathsf{I}} I$	Instruction sequence I is well formed
$\Psi_1 \vdash_{\mathsf{VH}} VH : \Psi_2$	VH has heap type Ψ_2
$\vdash^{c}_{\mathsf{E}} E$	Executable E is well formed
$\vdash_{\mathsf{P}} P$	Program state P is well formed

Where Δ is a type variable context, Ψ a heap type, and Γ a register file type. A *type variable context*, ranged over by metavariable Δ, is a list of pairs of type variables and their kinds. A *heap type*, ranged over by metavariable Ψ, is a finite mapping from labels to their types. A *register file type*, ranged over by metavariable Γ, has the form $\{\mathsf{sp}{:}c, r_1{:}c_1, \ldots, r_n{:}c_n\}$. It states that the stack should have type c and that registers r_i should have type c_i.

There are two judgements for types: one for the kind of a type constructor (kinding) and one for equality of two type constructors. The kinding judgement $\Delta \vdash_{\mathsf{tc}} c : \kappa$ asserts that type c has kind κ in context Δ. For example, TAL type variables and polymorphic types have the following kinding rules:

$$\frac{}{\Delta \vdash_{\mathsf{tc}} \alpha : \kappa}\ (\Delta(\alpha) = \kappa) \qquad \frac{\Delta, t \vdash_{\mathsf{tc}} c : \kappa}{\Delta \vdash_{\mathsf{tc}} \forall t.c : \kappa}$$

[5] MOOTAL, the final language of this dissertation, will include the kind and type constructor language of F_ω.

Equality judgements have the form $\Delta \vdash c_1 = c_2 : \kappa$ and assert that c_1 and c_2 are equal types of kind κ in context Δ. The rules are mostly congruence rules that can be derived from the kinding rules and appear in the appendix. However, there are some computational rules for the stack types (see below).

The TAL types for word values are the integer type int, the nonsense type ns, the heap-pointer types $*c$, and the stack-pointer types $\mathsf{sptr}(c)$. For pointer types, c is the type of the heap value or stack that is pointed to.

The most novel types in TAL are those for heap values. Code sequences are given the type $\mathsf{code}\ \Gamma$ where Γ is a register file type. The code sequence expects the register file to have type Γ before it is executed, and Γ can be thought of as a code precondition. Tuples are given the type $\langle c_1^{\phi_1}, \ldots, c_n^{\phi_n} \rangle$ where the c_i are word value types, and the ϕ_i are variances. *Variances* specify what operations may be performed on the fields. Read-only fields ($+$) may only be loaded, write-only fields ($-$) may only be stored, read-write fields ($\circ$) may be loaded and stored, and uninitialised fields (0) may only be stored. An uninitialised field becomes a read-write field after a store to it. In this way, TAL conservatively tracks initialisation.

Finally, TAL has stack types: se describes an empty stack, $c_1 :: c_2$ describes a word value of type c_1 pushed onto a stack of type c_2, and $c_1 \circ c_2$ describes a stack of type c_1 on top of a stack of type c_2. Stack types are the only kind with interesting equality rules; these rules formalise that $\circ$ is an append operation:

$$\frac{\Delta \vdash_{\mathsf{tc}} c : \mathsf{S}}{\Delta \vdash \mathsf{se} \circ c = c : \mathsf{S}} \qquad \frac{\Delta \vdash_{\mathsf{tc}} (c_1 :: c_2) \circ c_3 : \mathsf{S}}{\Delta \vdash (c_1 :: c_2) \circ c_3 = c_1 :: (c_2 \circ c_3) : \mathsf{S}}$$

Next consider the typing rules for term level constructs. Heaps are given heap types by the judgement $\Psi_1 \vdash_{\mathsf{VH}} VH : \Psi_2$ where Ψ_1 describes the types of the labels that VH may refer to. If VH defines ℓ_i to be h_i and h_i has type c_i in context Ψ_1 then VH will have type $\{\ell_i{:}c_i\}$:

$$\frac{\Psi \vdash_{\mathsf{h}} h_i : c_i}{\Psi \vdash_{\mathsf{VH}} \{\ell_1 \mapsto h_1{:}c_1, \ldots, \ell_n \mapsto h_n{:}c_n\} : \{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\}}$$

The rule for executables and program states will check $\vdash_{\mathsf{VHT}} \Psi$ and $\Psi \vdash_{\mathsf{VH}} VH : \Psi$ where VH is the heap of the executable or program state. In effect this rule checks the heap in the context of its own heap type. A heap value proper is given a memory type is a straightforward way:

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathsf{I}} I}{\Psi; \Delta \vdash_{\hat{\mathsf{h}}} \mathsf{code}\ I : \mathsf{code}\ \Gamma} \qquad \frac{\Psi; \Delta \vdash w_i : c_i^{\phi_i}}{\Psi; \Delta \vdash_{\hat{\mathsf{h}}} \langle w_1, \ldots, w_n \rangle : \langle c_1^{\phi_1}, \ldots, c_n^{\phi_n} \rangle}$$

$$\frac{\Psi; \Delta \vdash w : c}{\Psi; \Delta \vdash w : c^{\phi}} \qquad \frac{\Delta \vdash c_1 = c_2 : \mathsf{T}}{\Psi; \Delta \vdash ?_{c_1} : c_2^0}$$

If a heap value proper $\hat{h}$ has type c under type definitions t_1 through t_n, then the heap value $\Lambda[t_1, \ldots, t_n]\hat{h}$ has the polymorphic heap-pointer type $\forall t_1. \ldots \forall t_n.{*c}$.

$$\frac{\Psi; t_1, \ldots, t_n \vdash_{\hat{\mathsf{h}}} \hat{h} : c}{\Psi \vdash_{\mathsf{h}} \Lambda[t_1, \ldots, t_n]\hat{h} : \forall t_1. \ldots \forall t_n.{*c}}$$

Register files are given register file types in a straightforward manner. Similarly, stacks are given stack types. The rules for word values and operands are also straightforward. All these rules are in the appendix.

Instruction sequences are typed by the judgement $\Psi;\Delta;\Gamma \vdash_{\mathsf{I}} I$ which asserts that I is well formed if the heap has type Ψ and the register file has type Γ. Instructions are typed by the judgement $\Psi;\Delta_1;\Gamma_1 \vdash_{\mathsf{i}} \iota : \Delta_2;\Gamma_2$ which asserts that in the context $\Psi;\Delta_1;\Gamma_1$ the instruction is well formed and produces a new context $\Psi;\Delta_2;\Gamma_2$. I show a few instruction typing rules to give a flavour for them. All the TAL typing rules appear in the appendix (or the original TAL and STAL papers [MWCG98, MWCG99, MCGW98a, MCGW98b]).

A simple example is arithmetic:

$$\frac{\Psi;\Delta;\Gamma \vdash v_1 : \mathsf{int} \quad \Psi;\Delta;\Gamma \vdash v_2 : \mathsf{int}}{\Psi;\Delta;\Gamma \vdash_{\mathsf{i}} \mathit{aop}\ \ r, v_1, v_2 : \Delta;\Gamma\{r{:}\mathsf{int}\}}$$

Control flow requires the destination to be a heap pointer to code with a precondition at least as weak as the current register file type:

$$\frac{\Psi;\Delta;\Gamma \vdash v : {*}\mathsf{code}\ \Gamma' \quad \Delta \vdash_{\mathsf{RT}} \Gamma \leq \Gamma'}{\Psi;\Delta;\Gamma \vdash_{\mathsf{I}} \mathsf{jmp}\ \ v}$$

Register file subtyping $\Delta \vdash_{\mathsf{RT}} \Gamma \leq \Gamma'$ requires Γ to define all the registers that Γ' does and with the same type. Later this will be subsumed by full subtyping.

Memory allocation returns a new tuple of uninitialised values:

$$\frac{\Delta \vdash_{\mathsf{tc}} c_i : \mathsf{T}}{\Psi;\Delta;\Gamma \vdash_{\mathsf{i}} \mathsf{malloc}\ \ r, \langle c_1, \ldots, c_n \rangle : \Delta;\Gamma\{r{:}\langle c_1^0, \ldots, c_n^0 \rangle\}}$$

The store instruction has two rules: one requires the field to be write only or read write, the other requires the field to be uninitialised and updates the field to be read write ($c = \langle c_0^{\phi_0}, \ldots, c_{i-1}^{\phi_{i-1}}, c_i^{\circ}, c_{i+1}^{\phi_{i+1}}, \ldots, c_n^{\phi_n} \rangle$).

$$\frac{\Psi;\Delta;\Gamma \vdash r_d : \langle c_0^{\phi_0}, \ldots, c_n^{\phi_n} \rangle \quad \Psi;\Delta;\Gamma \vdash r_s : c_i}{\Psi;\Delta;\Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ [r_d + i], r_s : \Delta;\Gamma} \ (0 \leq i \leq n; \phi_i \in \{-, \circ\})$$

$$\frac{\Psi;\Delta;\Gamma \vdash r_d : \langle c_0^{\phi_0}, \ldots, c_n^{\phi_n} \rangle \quad \Psi;\Delta;\Gamma \vdash r_s : c_i}{\Psi;\Delta;\Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ [r_d + i], r_s : \Delta;\Gamma\{r{:}c\}} \ (0 \leq i \leq n; \phi_i = 0)$$

Stack adjustment is governed by two two rules: one for allocating, one for freeing.

$$\frac{}{\Psi;\Delta;\Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ \mathsf{sp}, \mathsf{sp} + i : \Delta;\Gamma\{\mathsf{sp}{:}\underbrace{\mathsf{ns} :: \cdots :: \mathsf{ns}}_{i} :: c\}} \ (\Gamma(\mathsf{sp}) = c; i \leq 0)$$

$$\frac{\Delta \vdash c = c_1 :: \cdots :: c_i :: c' : \mathsf{S}}{\Psi;\Delta;\Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ \mathsf{sp}, \mathsf{sp} + i : \Delta;\Gamma\{\mathsf{sp}{:}c'\}} \ (\Gamma(\mathsf{sp}) = c; i > 0)$$

Moving a stack pointer from a general register to the stack pointer register illustrates a subtlety of TAL's typing rules. When the stack pointer was moved into the register, it had some stack type. However, the stack now might be in a different state, and the type rules have to prevent, amongst other things, an out-of-range stack pointer from being moved into sp. TAL does this by requiring a validity check on stack types before they are used: the stack type must be a tail of the stack pointer register's type (see [MCGW98a] for further details).

$$\frac{\Psi;\Delta;\Gamma \vdash r : \mathsf{sptr}(c_2) \quad \Delta \vdash c = c_1 \circ c_2 : \mathsf{S}}{\Psi;\Delta;\Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ \mathsf{sp}, r : \Delta;\Gamma\{\mathsf{sp}{:}c_2\}} \ (\Gamma(\mathsf{sp}) = c)$$

Finally, an executable is well formed, judgement $\vdash^{c}_{\mathsf{E}} E$, when its heap has some heap type Ψ, and Ψ gives an appropriate type c for the entry label (*i.e.*, $*\mathsf{code}\ \{\mathsf{sp}{:}\mathsf{se}\}$). A program state is well formed, judgement $\vdash_{\mathsf{P}} P$, when its heap has some heap type Ψ, the register file has some register file type Γ in context Ψ, and the instruction sequence is well formed in context $\Psi; \epsilon; \Gamma$ (ϵ denotes an empty sequence).

TAL's type system is sound (proven by Morrisett et al. [MWCG99] and in the appendix):

Theorem 2.1 *If* $\vdash_{\mathsf{P}} P$ *and* $P \mapsto^{*} P' \not\mapsto$ *then* P' *has the form* $(VH, R, \mathsf{halt}[c])$.

TAL's type soundness guarantees that certain things will not happen during execution. For example, the code will never jump to something that is not code; the code will never load or store to an offset that is out of bounds for a heap block or the stack.

2.4 Compiler

To partially support the claim that TAL's type system should be expressive enough to type check the output of typical compilers, this section presents a compiler for a core procedural language IIL (Imperative Intermediate Language) and shows how to compile it to TAL. IIL contains the key features of a procedural language, and other base types, control-flow constructs, and data structures can easily be incorporated. IIL also has the necessary constructs to serve as an intermediate language in a compiler for first-class functions, objects, classes, and type-dispatch constructs. Later chapters will use IIL with some extensions to show that these constructs can be compiled to an extended TAL. The syntax of IIL is:

Type Definitions	t	$::=$	α
Types	τ, σ	$::=$	$\alpha \mid \mathsf{int} \mid \mathsf{exn} \mid (\vec{\tau}) \to \tau \mid \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I} \mid \forall t.\tau$
Variances	ϕ	$::=$	$+ \mid - \mid \circ$
Expressions	e, b	$::=$	$x \mid x \leftarrow e \mid i \mid e_1\, p\, e_2 \mid \mathsf{if}\ r\ e\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid$
			$\mathsf{fix}\ f[\vec{t}](\overrightarrow{x{:}\tau}){:}\tau.b \mid e(\vec{e}) \mid$
			$\langle \overrightarrow{\ell = e} \rangle \mid \Lambda[\vec{t}]\langle \overrightarrow{\ell = v} \rangle \mid e.\ell \mid e_1.\ell \leftarrow e_2 \mid$
			$e[\tau] \mid \mathsf{let}\ \overrightarrow{x = e}\ \mathsf{in}\ e_2 \mid \mathsf{raise}(e) \mid \mathsf{try}\ e_1\ \mathsf{with}\ x.e_2$
Executables	E	$::=$	e

The notation $\vec{X}$ denotes a vector of objects drawn from the syntax category X. For $\tau_1, \ldots, \tau_n$, I write $\vec{\tau}$; for $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, I write $\overrightarrow{x{:}\tau}$; for $x_1 = e_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ x_n = e_n$, I write $\overrightarrow{x = e}$.

The type exn is for exception packets. For now, there are no introduction forms for exception packets, nor do I specify a translation for this type. I discuss exception packets in Chapter 6. The type $(\tau_1, \ldots, \tau_n) \to \tau$ describes functions that take n parameters of types τ_1 through τ_n and return a result of type τ. They are introduced by functions, which have the form $\mathsf{fix}\ f[\vec{t}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n){:}\tau.b$ where f is a variable that refers to the whole function, $\vec{t}$ are the function's type parameters, x_i are the funtion's value parameters of type τ_i, τ is the return type, and b is the body of the function. It is a syntactic restriction that f may not appear on the left-hand side of an assignment ($x \leftarrow e$). The type parameters are given by *type definitions*, ranged over by metavariable t, which, for now, just list a type variable, ranged over by metavariable α. Chapter 5 will add another form of type definition that allows type variables to have recursive subtyping bounds. Functions must be closed, that is, they cannot have any free type or value variables. Tuple types $\langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in I}$ list the field names ℓ_i, types τ_i, and variances ϕ_i. IIL's variances are simpler than TAL's and include only read only $(+)$, write only $(-)$, and read write ($\circ$). Ordinary tuples have the form $\langle\overrightarrow{\ell = e}\rangle$ where ℓ_i are the field names, and e_i their initial values. In addition, there are polymorphic tuples $\Lambda[\vec{t}]\langle\overrightarrow{\ell = v}\rangle$. These tuples abstract over type parameters $\vec{t}$, must consist of values for their field initialisers, must have read-only fields (polymorphism interacts badly with mutability [Wri95]), and must be closed (they will be compiled to static data). Although restricted, these polymorphic tuples will be used in a class and object encoding in Chapter 5.

The expression $x \leftarrow e$ evaluates e to a value v, assigns v to x, and results in v. Arithmetic operations have the form $e_1\ p\ e_2$, and the meaning of p, written $||p||$, is a binary operator on integers. For convenience, assume that there is a mapping from a p to an aop. Arithmetic conditions $\mathsf{if}\ r\ e\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ evaluate a condition r on the integer valued expression e, executing b_1 if the condition is true, and b_2 otherwise. Again the meaning of r, written $||r||$, is a unary predicate on integers, and there is a mapping from an r to a bop. Exceptions are raised with $\mathsf{raise}(e)$, and $\mathsf{try}\, e_1\ \mathsf{with}\ x.e_2$ handles the exceptions raised in e_1 by binding x to the exception packet and executing e_2.

The informal description given above is formalised as an operational semantics in Figure 2.2 and typing rules in Figure 2.3. The state of an IIL computation consists of a *store* that maps mutable variables to their current values, and a *heap* that maps locations, ranged over by metavariable L, to mutable tuples. An *evaluation context*, ranged over by metavariable E, selects the next subterm to evaluate in a left to right evaluation order. It contains exactly one hole, written $\{\}$, and the substitution of e for the hole in E is written $E\{e\}$. Given a heap H and value v, the auxilary function $\hat{H}(v)$ performs any type applications that might be in v returning an instantiated value. The empty sequence is written ϵ; $\mathrm{ftv}(\tau)$ is the free type variables of τ. I use the abbreviations $\forall[]\tau = \tau$ and $\forall[\alpha, \vec{\alpha}]\tau = \forall\alpha.\forall[\vec{\alpha}]\tau$. There are two typing judgements $\Delta \vdash^{\mathrm{IIL}} \tau$ (τ is a well-formed type) and $\Delta; \Gamma \vdash^{\mathrm{IIL}} e : \tau$ (e has type τ) where Δ is a type variable context and Γ is a value variable context. Type variable contexts are just a list of valid type variables. Value variable contexts are lists of pairs of variables and types.

There are many ways to translate IIL into type-correct TAL code, as TAL's type system captures the key features of assembly language. To illustrate this expressiveness, the rest of this section presents a simple translation. The basic idea is to use stack-based compilation: The current expression is translated into code that computes the value of the expression, placing the result into register r1. Intermediate results are saved by pushing them onto the stack. The calling convention places the arguments on the stack, the first argument first, and places the return address on the stack after the arguments. The callee frees the return address and arguments from the stack before returning. To deal with exceptions, a register re points to an exception frame on the stack. This frame has the address of code to jump to when an exception is raised. Before jumping to this code, everything above and including the exception frame is discarded from the stack, and and the exception packet is placed in register r1. Register re is the only callee-save register.

I formalise these ideas as a type-directed translation from IIL to TAL. First, the type translation and some type macros used throughout the translation are:

$$
\begin{array}{lcl}
\mathrm{ht}(\tau_b) & = & (*\mathsf{code}\ \{\mathsf{r1}{:}[\![\mathsf{exn}]\!]_{\mathsf{t}}, \mathsf{sp}{:}\tau_b\}) :: \tau_b \\
\mathrm{sc}(\tau_a, \tau_b) & = & *\mathsf{code}\ \{\mathsf{sp}{:}\tau_a \circ \mathrm{ht}(\tau_b), \mathsf{re}{:}\mathsf{sptr}(\mathrm{ht}(\tau_b))\} \\
\mathrm{ec}(\tau_a, \tau_b, \tau) & = & *\mathsf{code}\ \{\mathsf{sp}{:}\tau_a \circ \mathrm{ht}(\tau_b), \mathsf{re}{:}\mathsf{sptr}(\mathrm{ht}(\tau_b)), \mathsf{r1}{:}\tau\} \\
\\
[\![\alpha]\!]_{\mathsf{t}} & = & \alpha \\
[\![\mathsf{int}]\!]_{\mathsf{t}} & = & \mathsf{int} \\
[\![\mathsf{exn}]\!]_{\mathsf{t}} & & \text{(See Chapter 6)}
\end{array}
$$

$$
\begin{array}{llcl}
\text{Values} & v & ::= & i \mid \mathsf{fix}\ f[\vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b \mid L \mid v[\tau] \\
\text{Contexts} & E & ::= & \{\} \mid x \leftarrow E \mid E\ p\ e \mid v\ p\ E \mid \mathsf{if}\ r\ E\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \\
 & & & E(\vec{e}) \mid v(\vec{v}, E, \vec{e}) \mid \langle \overrightarrow{\ell = v}, \ell = E, \overrightarrow{\ell' = e} \rangle \mid E.\ell \mid \\
 & & & E.\ell \leftarrow e \mid v.\ell \leftarrow E \mid E[\tau] \mid \mathsf{raise}(E) \mid \mathsf{try}\ E\ \mathsf{with}\ x.e \mid \\
 & & & \mathsf{let}\ \overrightarrow{x = v}\ \mathsf{and}\ x = E\ \mathsf{and}\ \overrightarrow{x' = e}\ \mathsf{in}\ b \\
\text{Frames} & F & ::= & E \text{ without } \mathsf{try}\ E\ \mathsf{with}\ x.e \\
\text{Stores} & S & ::= & \overrightarrow{x = v} \\
\text{Heap Values} & h & ::= & \Lambda[\vec{t}]\langle \overrightarrow{\ell = v} \rangle \\
\text{Heaps} & H & ::= & \overrightarrow{L = h} \\
\text{Program States} & P & ::= & \mathsf{letrec}\ H, S\ \mathsf{in}\ e
\end{array}
$$

$$\mathsf{letrec}\ H, S\ \mathsf{in}\ E\{\iota\} \mapsto \mathsf{letrec}\ H', S'\ \mathsf{in}\ E\{e\}$$

ι	e	S'	H'	Side Conditions
h	L	S	$H\{L = h\}$	$L \notin \mathrm{dom}(H)$
x	$S(x)$	S	H	
$x \leftarrow v$	v	$S\{x = v\}$	H	
$i_1\ p\ i_2$	$i_1 \lVert p \rVert i_2$	S	H	
$\mathsf{if}\ r\ i\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_1	S	H	$\lVert r \rVert i$
$\mathsf{if}\ r\ i\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_2	S	H	not $\lVert r \rVert i$
$v(v_1, \ldots, v_n)$	$b\{f := v\}$	$S\{\vec{x} = \vec{v}\}$	H	$x_i \notin \mathrm{dom}(S); \hat{H}(v) =$ $\mathsf{fix}\ f[](x_i{:}\tau_i)_{1 \le i \le n}{:}\tau.b$
$v.\ell_k$	v_k	S	H	$\hat{H}(v) = \langle \ell_i = v_i \rangle_{i \in I}$ $k \in I$
$L.\ell_k \leftarrow v$	v	S	$H\{L = h'\}$	$H(L) = \langle \ell_i = v_i \rangle_{i \in I}$ $k \in I$ $h' = \langle \ell_i = v'_i \rangle_{i \in I}$ $v'_i = \begin{cases} v_i & i \neq k \\ v & i = k \end{cases}$
$\mathsf{let}\ \overrightarrow{x = v}\ \mathsf{in}\ b$	b	$S\{\vec{x} = \vec{v}\}$	H	$x_i \notin \mathrm{dom}(S)$
$\mathsf{try}\ v\ \mathsf{with}\ x.b$	v	S	H	
$\mathsf{try}\ F\{r\}\ \mathsf{with}\ x.b$	b	$S\{x = v\}$	H	$x \notin \mathrm{dom}(S); r = \mathsf{raise}(v)$

$$
\hat{H}(v) = \begin{cases}
H(L) & v = L \\
(\mathsf{fix}\ f[\vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b)\{\alpha := \sigma\} & v = v'[\sigma]; \hat{H}(v') = \mathsf{fix}\ f[\alpha, \vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b \\
(\Lambda[\vec{\alpha}]\langle \overrightarrow{\ell = v} \rangle)\{\alpha := \sigma\} & v = v'[\sigma]; \hat{H}(v') = \Lambda[\alpha, \vec{\alpha}]\langle \overrightarrow{\ell = v} \rangle \\
v & \text{otherwise}
\end{cases}
$$

Figure 2.2: IIL Operational Semantics

$$\frac{}{\Delta \vdash^{\textsc{Iil}} \tau}\ (\mathrm{ftv}(\tau) \subseteq \Delta)$$

$$\frac{}{\Delta; \Gamma \vdash^{\textsc{Iil}} x : \tau}\ (\Gamma(x) = \tau) \qquad \frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e : \tau}{\Delta; \Gamma \vdash^{\textsc{Iil}} x \leftarrow e : \tau}\ (\Gamma(x) = \tau)$$

$$\frac{}{\Delta; \Gamma \vdash^{\textsc{Iil}} i : \mathsf{int}} \qquad \frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e_1 : \mathsf{int} \quad \Delta; \Gamma \vdash^{\textsc{Iil}} e_1 : \mathsf{int}}{\Delta; \Gamma \vdash^{\textsc{Iil}} e_1\, p\, e_2 : \mathsf{int}}$$

$$\frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e : \mathsf{int} \quad \Delta; \Gamma \vdash^{\textsc{Iil}} b_1 : \tau \quad \Delta; \Gamma \vdash^{\textsc{Iil}} b_2 : \tau}{\Delta; \Gamma \vdash^{\textsc{Iil}} \mathsf{if}\ r\ e\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} : \tau}$$

$$\frac{\epsilon \vdash^{\textsc{Iil}} \sigma \quad \vec{\alpha}; f{:}\sigma, \overrightarrow{x{:}\tau} \vdash^{\textsc{Iil}} b : \tau}{\Delta; \Gamma \vdash^{\textsc{Iil}} \mathsf{fix}\ \ f[\vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b : \sigma}\ (\sigma = \forall[\vec{\alpha}](\vec{\tau}) \to \tau)$$

$$\frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e : (\tau_1, \ldots, \tau_n) \to \tau \quad \Delta; \Gamma \vdash^{\textsc{Iil}} e_i : \tau_i}{\Delta; \Gamma \vdash^{\textsc{Iil}} e(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e_i : \tau_i}{\Delta; \Gamma \vdash^{\textsc{Iil}} \langle \ell_i = e_i \rangle_{i \in I} : \langle \ell_i = \tau_i^{\circ} \rangle_{i \in I}}$$

$$\frac{\vec{\alpha}; \epsilon \vdash^{\textsc{Iil}} v_i : \tau_i}{\Delta; \Gamma \vdash^{\textsc{Iil}} \Lambda[\vec{\alpha}]\langle \ell_i = v_i \rangle_{i \in I} : \forall[\vec{\alpha}]\langle \ell_i = \tau_i^{+} \rangle_{i \in I}}$$

$$\frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e : \langle \ell_i = \tau_i^{\phi_i} \rangle_{i \in I}}{\Delta; \Gamma \vdash^{\textsc{Iil}} e.\ell_k : \tau_k}\ (k \in I; \phi_k \in \{+, \circ\})$$

$$\frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e_1 : \langle \ell_i = \tau_i^{\phi_i} \rangle_{i \in I} \quad \Delta; \Gamma \vdash^{\textsc{Iil}} e_2 : \tau_k}{\Delta; \Gamma \vdash^{\textsc{Iil}} e_1.\ell_k \leftarrow e_2 : \tau_k}\ (k \in I; \phi_k \in \{-, \circ\})$$

$$\frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e : \forall \alpha.\tau \quad \Delta \vdash^{\textsc{Iil}} \sigma}{\Delta; \Gamma \vdash^{\textsc{Iil}} e[\sigma] : \tau\{\alpha := \sigma\}}$$

$$\frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e_i : \tau_i \quad \Delta; \Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash^{\textsc{Iil}} e : \tau}{\Delta; \Gamma \vdash^{\textsc{Iil}} \mathsf{let}\ \ x_1 = e_1\ \ \mathsf{and}\ \ \cdots\ \ \mathsf{and}\ \ x_n = e_n\ \ \mathsf{in}\ \ e : \tau}$$

$$\frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e : \mathsf{exn}}{\Delta; \Gamma \vdash^{\textsc{Iil}} \mathsf{raise}(e) : \tau} \qquad \frac{\Delta; \Gamma \vdash^{\textsc{Iil}} e_1 : \tau \quad \Delta; \Gamma, x{:}\mathsf{exn} \vdash^{\textsc{Iil}} e_2 : \tau}{\Delta; \Gamma \vdash^{\textsc{Iil}} \mathsf{try}\ e_1\ \mathsf{with}\ \ x.e_2 : \tau}$$

Figure 2.3: Iil Typing Rules

$$
\begin{array}{lcl}
[\![(\tau_1,\ldots,\tau_n)\rightarrow\tau]\!]_{\mathsf{t}} & = & \forall[\rho_1{:}\mathsf{S},\rho_2{:}\mathsf{S}]\mathrm{sc}(\mathrm{ec}(\rho_1,\rho_2,[\![\tau]\!]_{\mathsf{t}}) :: [\![\tau_n]\!]_{\mathsf{t}} :: \cdots :: [\![\tau_1]\!]_{\mathsf{t}} :: \rho_1,\rho_2) \\
[\![\langle \ell_i{:}\tau_i^{\phi_i}\rangle_{i\in 1,\ldots,n}]\!]_{\mathsf{t}} & = & *\langle [\![\tau_1]\!]_{\mathsf{t}}^{\phi_1},\ldots,[\![\tau_n]\!]_{\mathsf{t}}^{\phi_n}\rangle \\
[\![\forall t.\tau]\!]_{\mathsf{t}} & = & \forall[\![t]\!]_{\mathsf{td}}.[\![\tau]\!]_{\mathsf{t}} \\
[\![\alpha]\!]_{\mathsf{td}} & = & \alpha{:}\mathsf{T}
\end{array}
$$

The type $\mathrm{ht}(\tau_b)$ is the type of a stack with an exception frame at the top where τ_b is the type of the rest of the stack. The type $\mathrm{sc}(\tau_a,\tau_b)$ describes command continuations where τ_a and τ_b are the types of the stack above and below respectively the current exception frame. Expression continuations have type $\mathrm{ec}(\tau_a,\tau_b,\tau)$ where τ is the type of value expected by the continuation. The translation of an IIL type τ is $[\![\tau]\!]_{\mathsf{t}}$; its definition is straightforward except for function types. A function abstracts two type variables ρ_1 and ρ_2 that represent the caller's stack between the arguments and the exception frame, and below the exception frame, respectively. A function is a command continuation that expects a return address on top of its arguments on top of the caller's stack. The return address is an expression continuation that expects the function's return value.

Compiling expressions requires generating new heap blocks to hold the code for functions nested within the expression and to hold the data of polymorphic tuples nested within the expression. Therefore, the translation of an expression will be a function that takes and returns a compilation state.[6] A *compilation state*, ranged over by metavariable CS, is a triple $(\{\ell_1,\ldots,\ell_n\}, VH, J)$ consisting of a heap VH containing the heap blocks generated so far, a set of labels ℓ_i that address blocks that have been or will be generated, and a labelled code sequence J to preceed the expression currently being compiled. A *labelled code sequence*, ranged over by metavariable J, is either $\ell \mapsto \mathsf{code}\ [\vec{t}]c$ indicating the start of a code sequence, or $J\iota$ which is a labelled code sequence J following by an instruction ι.

To make the statement of the translation look natural, in particular, like a sequence of instructions, I will use some notation that hides all of the manipulation of compilation states. The funciton *id* just maps a computation state to itself. There are a number of composition operations all denoted by ";". They are defined as follows:

$$
\begin{array}{lcl}
(f;g)(L,VH,J) & = & g(f(L,VH,J)) \\
(f;\iota)(L_1,VH_1,J_1) & = & (L_2,VH_2,J_2\iota) \\
 & & \text{where } (L_2,VH_2,J_2) = f(L_1,VH_1,J_1) \\
(f;I;J)(L_1,VH_1,J_1) & = & (L_2,VH_2\cup\{\mathit{extract}(J_2,I)\},J) \\
 & & \text{where } (L_2,VH_2,J_2) = f(L_1,VH_1,J_1) \\
\mathit{extract}(\ell\mapsto\mathsf{code}\ [\vec{t}]c, I) & = & \ell\mapsto\Lambda[\vec{t}]\mathsf{code}\ I : \forall[\vec{t}]c \\
\mathit{extract}(J\iota, I) & = & \mathit{extract}(J,\iota;I)
\end{array}
$$

Where f and g are functions from computation states to computation states, ι is an instruction, I is an instruction sequence, and J is a labelled code sequence. The function $\mathit{extract}(\cdot,\cdot)$ takes a labelled code sequence and an instruction sequence to follow it and constructs a heap value and label.

The translation of conditions must generate fresh labels to label the code for the else branch and the merge point; similarly, other control flow constructs require fresh labels. The following function achieves this where f is a function that takes the fresh label and returns a function from computation states to computation states.

$$\mathit{new}(f)(L,VH,J) = f(\ell)(L\cup\{\ell\},VH,J) \text{ where } \ell\notin L$$

[6] The translation is based on a monadic-style translation described by Morrisett et al. in a draft journal verson of their technical report [MCGW98b].

The translation of functions and polymorphic tuples need to generate a new heap value in the middle of an instruction sequence. The following function achieves this where the argument is the heap value to be add to the current heap.

$$\begin{array}{lcl} \mathit{lift}(J; f; I)(L_1, VH_1, J_1) & = & (L_2, VH_2 \cup \{\mathit{extract}(J_2, I)\}, J_1) \\ & & \text{where } (L_2, VH_2, J_2) = f(L_1, VH_1, J) \\ \mathit{lift}(\ell \mapsto h{:}c)(L, VH, J) & = & (L, VH \cup \{\ell \mapsto h{:}c\}, J) \end{array}$$

A final detail of the translation concerns where the source variables are placed. Source variables that are bound to functions by the fix form are placed in the heap, and the translation should map them to a label. All other sources are placed on the stack. Therefore, a *variable location*, ranged over by metavariable loc, is either a label or an integer that is the offset in words from the location of the return address (positive for parameters and negative for local variables). A *variable map*, ranged over by metavariable vm, is a finite map from IIL variables to variable locations. The translation takes a variable map as an argument so that it knows where the variables that are in scope are located.

The translation of an IIL value is a TAL word value, but like expressions, the translation may need to generate new heap blocks for function bodies. To accomplish this, the translation of values, written $[\![v]\!]_{\mathsf{v}}(k)$, takes a static continuation k, which is a function that given a TAL word value returns a function (from computation states to computations states) that generates the translation of the context in which v appears. The translation $[\![v]\!]_{\mathsf{v}}(k)$ returns a function that generates the value and the context in which it appears. The definition of $[\![\cdot]\!]_{\mathsf{v}}$ appears in Figure 2.4. The translation of an expression $[\![v]\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h)$ is a function from computation states to computation states, where t_i are TAL type variable declarations, vm a variable map, τ_a and τ_b the types of the stack above and below respectively the current exception frame, and h the height of the stack (number of words on top of the return address). It is given in Figures 2.5–2.7. It is a type-directed translation, but rather than include the IIL typing rules, I use τ for the type of e, τ' for the type of e', and τ_i for the type of e_i. Instantiation of a value by TAL type-variable declarations is used frequently in the translation:

$$\mathrm{inst}(v, \alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n) = [\alpha_n](\cdots[\alpha_1](v)\cdots)$$

Finally, the translation of an executable is:

$$\begin{array}{l} [\![e]\!]_{\mathsf{E}} = (VH, \ell_{main}) \\ \text{where } \;(\_, VH, \_) = \\ \qquad (\mathit{lift}(\\ \qquad\quad \ell_{main} \mapsto \mathsf{code}\ []{*}\mathsf{code}\ \{\mathsf{sp{:}se}\}; \\ \qquad\quad \mathsf{push}\ \ell_{uncaught}; \mathsf{mov\ re, sp}; \\ \qquad\quad [\![e]\!]_{\mathsf{e}}(\epsilon, \epsilon, \mathsf{se}, \mathsf{se}, 0); \\ \qquad\quad \mathsf{halt}[[\![\tau]\!]_{\mathsf{t}}] \\ \qquad); \\ \qquad \mathit{lift}(\\ \qquad\quad \ell_{uncaught} \mapsto \Lambda[]\mathsf{code\ halt}[[\![\mathsf{exn}]\!]_{\mathsf{t}}] : {*}\mathsf{code}\ \{\mathsf{r1}{:}[\![\mathsf{exn}]\!]_{\mathsf{t}}, \mathsf{sp{:}se}\} \\ \qquad))(\{\ell_{main}, \ell_{uncaught}\}, \{\}, \ell_{dummy} \mapsto []{*}\mathsf{code}\ \{\}) \end{array}$$

An example of the translation appears in Figure 2.8. It uses some peephole optimisations to keep the example short.

$$
\begin{array}{lllll}
[\![i]\!]_{\mathsf{v}}(k) & = & k(i) & & \\
[\![\mathsf{fix}\ f[\vec{t}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n){:}\tau'.b]\!]_{\mathsf{v}}(k) & = & & & \\
\quad new(\lambda\ell_f. & \text{where} & \vec{t'} & = & \overrightarrow{[\![t]\!]_{\mathsf{t}}}, \rho_1{:}\mathsf{S}, \rho_2{:}\mathsf{S} \\
\quad\quad lift(& & vm' & = & f \mapsto \ell_f, x_i \mapsto i \\
\quad\quad\quad \ell_f \mapsto \mathsf{code}\ [\vec{t'}]\mathrm{sc}(\tau'_a,\tau'_b); & & \tau'_a & = & \mathrm{ec}(\rho_1,\rho_2,[\![\tau']\!]_{\mathsf{t}}) :: \tau_{args} \\
\quad\quad\quad [\![b]\!]_{\mathsf{e}}(\vec{t'}, vm', \tau'_a, \tau'_b, 0); & & \tau'_b & = & \rho_2 \\
\quad\quad\quad \mathsf{pop\ r2}; & & \tau_{args} & = & [\![\tau_n]\!]_{\mathsf{t}} :: \cdots :: [\![\tau_1]\!]_{\mathsf{t}} :: \rho_1 \\
\quad\quad\quad \mathsf{mov\ sp, sp} + n; & & & & \\
\quad\quad\quad \mathsf{jmp\ r2} & & & & \\
\quad\quad); & & & & \\
\quad\quad k(\ell_f) & & & & \\
\quad) & & & & \\
[\![v[\tau]]\!]_{\mathsf{v}}(k) & = & [\![v]\!]_{\mathsf{v}}(\lambda w.k(w[[\![\tau]\!]_{\mathsf{t}}])) & &
\end{array}
$$

Figure 2.4: IIL to TAL Compiler for Values

IIL contains the core of a procedure language. A real language would have more base types, sum or union types, arrays, and more elaborate control structures. Most of these can be incorporated into the TAL framework. I describe them and the issues that arise in the context of an implementation of TAL done at Cornell in Chapter 7. A real language might also have a module system and a separate compilation property. The issues that arise with incorporating modules into TAL are described in Chapter 3. Functional languages and object-oriented languages have, in addition to IIL, first-class functions, closure conver, objects, and classes. These will be discussed in Chapters 4 and 5.

x		
	$id; \mathsf{mov\ \ r1}, \ell$	$vm(x) = \ell$
x		
	$id; \mathsf{mov\ \ r1}, [\mathsf{sp} + h + i]$	$vm(x) = i$
$x \leftarrow e'$		
	$[\![e']\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_n, h);$ $\mathsf{mov\ \ }[\mathsf{sp} + h + i], \mathsf{r1}$	$vm(x) = i$
v		
	$[\![v]\!]_{\mathsf{v}}(\lambda w.\mathsf{mov\ \ r1}, w)$	
$e_1\ p\ e_2$		
	$[\![e_1]\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h);$ $\mathsf{push\ \ r1};$ $[\![e_2]\!]_{\mathsf{e}}(\vec{t}, vm, \mathsf{int} :: \tau_a, \tau_b, h + 1);$ $\mathsf{pop\ \ r2};\ aop\ \ \mathsf{r1}, \mathsf{r2}, \mathsf{r1}$	
$\mathsf{if}\ r\ e'\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$		
$new(\lambda\ell_{else}.new(\lambda\ell_{end}.$		
	$[\![e']\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h);$ $bop\ \ \mathsf{r1}, \mathrm{inst}(\ell_{else}, \vec{t});$ $[\![b_1]\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h);$ $\mathsf{jmp\ \ }\mathrm{inst}(\ell_{end}, \vec{t});$	
$\ell_{else} \mapsto$	$\mathsf{code\ \ }[\vec{t}]\mathrm{sc}(\tau_a, \tau_b);$ $[\![b_2]\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h);$ $\mathsf{jmp\ \ }\mathrm{inst}(\ell_{end}, \vec{t});$	
$\ell_{end} \mapsto$	$\mathsf{code\ \ }[\vec{t}]\mathrm{ec}(\tau_a, \tau_b, [\![\tau]\!]_{\mathsf{t}})$	
$))$		
$e'(e_1, \ldots, e_n)$		
$new(\lambda\ell_{ret}.$		
	$[\![e_1]\!]_{\mathsf{e}}(\vec{t}, \tau_a, \tau_b, h); \mathsf{push\ \ r1};$	;; Compute arguments
	$\cdots$	$\tau_a^i = [\![\tau_i]\!]_{\mathsf{t}} :: \cdots :: [\![\tau_1]\!]_{\mathsf{t}} :: \tau_a$
	$[\![e_n]\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a^{n-1}, \tau_b, h + n - 1);$ $\mathsf{push\ \ r1};$	
	$[\![e']\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a^n, \tau_b, h + n);$	;; Compute function
	$\mathsf{push\ \ }\mathrm{inst}(\ell_{ret}, \vec{t}); \mathsf{jmp\ \ }[\tau_b]([\tau_a](\mathsf{r1}));$	;; Do call
$\ell_{ret} \mapsto$	$\mathsf{code\ \ }[\vec{t}]\mathrm{ec}(\tau_a, \tau_b, [\![\tau]\!]_{\mathsf{t}})$	
$)$		

Figure 2.5: Iil to Tal Compiler for Expressions 1

$\langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle$	
$[\![e_1]\!]_{\mathsf{e}}(\vec{t}, \tau_a, \tau_b, h);$	;; Compute fields
$\mathsf{push\ r1};$	$\tau_a^i = [\![\tau_i]\!]_{\mathsf{t}} :: \cdots :: [\![\tau_1]\!]_{\mathsf{t}} :: \tau_a$
$\ldots$	
$[\![e_n]\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a^{n-1}, \tau_b, h+n-1);$	
$\mathsf{push\ r1};$	
$\mathsf{malloc\ r1}, \langle [\![\tau_1]\!]_{\mathsf{t}}, \ldots, [\![\tau_n]\!]_{\mathsf{t}} \rangle;$	;; Allocate record
$\mathsf{pop\ r2}; \mathsf{mov\ } [\mathsf{r1} + n - 1], \mathsf{r2};$	;; Initalise fields
$\ldots$	
$\mathsf{pop\ r2}; \mathsf{mov\ } [\mathsf{r1} + 0], \mathsf{r2}$	
$\Lambda[\vec{t}]\langle \ell_1 = v_1, \ldots, \ell_n = v_n \rangle$	
$new(\lambda \ell.$	
$[\![v_1]\!]_{\mathsf{v}}(\lambda w_1.$	
$\ldots$	
$[\![v_n]\!]_{\mathsf{v}}(\lambda w_n.$	
$lift(\ell \mapsto \Lambda[\overrightarrow{[\![t]\!]_{\mathsf{td}}}]\langle w_1, \ldots, w_n \rangle{:}[\![\tau]\!]_{\mathsf{t}});$	
$\mathsf{mov\ r1}, \ell$	
$)\cdots)$	
$)$	
$e'.\ell_k$	
$[\![e']\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h);$	$\tau' = \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in 0,\ldots,n}$
$\mathsf{mov\ r1}, [\mathsf{r1} + k]$	
$e_1.\ell_k \leftarrow e_2$	
$[\![e_1]\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h);$	$\tau_1 = \langle \ell_i{:}\tau_i^{\phi_i} \rangle_{i \in 0,\ldots,n}$
$\mathsf{push\ r1};$	
$[\![e_2]\!]_{\mathsf{e}}(\vec{t}, vm, [\![\tau_1]\!]_{\mathsf{t}} :: \tau_a, \tau_b, h+1);$	
$\mathsf{pop\ r2};$	
$\mathsf{mov\ } [\mathsf{r2} + k], \mathsf{r1}$	

Figure 2.6: IIL to TAL Compiler for Expressions 2

$e'[\sigma]$		
	$[\![e']\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h)$;	
	mov r1, r1[$[\![\sigma]\!]_{\mathsf{t}}$]	
let $x_1 = e_1$ and $\cdots$ and $x_n = e_n$ in e'		
	$[\![e_1]\!]_{\mathsf{e}}(\vec{t}, \tau_a, \tau_b, h)$;	;; Compute lets
	push r1;	$\tau_a^i = [\![\tau_i]\!]_{\mathsf{t}} :: \cdots :: [\![\tau_1]\!]_{\mathsf{t}} :: \tau_a$
	$\cdots$	$vm' = vm, x_i \mapsto -h - i$
	$[\![e_n]\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a^{n-1}, \tau_b, h + n - 1)$;	
	push r1;	
	$[\![e']\!]_{\mathsf{e}}(\vec{t}, vm', \tau_a^n, \tau_b, h + n)$;	;; Compute body
	mov sp, sp $+ n$	
raise(e')		
	$[\![e']\!]_{\mathsf{e}}(\vec{t}, vm, \tau_a, \tau_b, h)$;	
	mov sp, re;	
	pop r2; jmp r2	
try e_1 with $x.e_2$		
$new(\lambda\ell_{with}.new(\lambda\ell_{end}.$		
	push re;	;; Save exception frame
	push $\mathrm{inst}(\ell_{handle}, \vec{t})$;	;; Install handler
	mov re, sp;	
	$[\![e_1]\!]_{\mathsf{e}}(\vec{t}, vm, \mathsf{se}, \tau_b', h + 2)$;	;; Compute body $\tau_b' = (\mathsf{sptr}(\mathrm{ht}(\tau_b)) :: \tau_a) \circ \mathrm{ht}(\tau_b)$
	mov sp, sp $+ 1$;	;; Uninstall handler
	pop re;	;; Restore exception frame
	jmp $\mathrm{inst}(\ell_{end}, \vec{t})$;	
$\ell_{with} \mapsto$	code $[\vec{t}]$∗code {r1 : exn, sp : τ_b'};	
	pop re;	;; Restore exception frame
	push r1;	;; Bind x
	$[\![e_2]\!]_{\mathsf{e}}(\vec{t}, vm', \tau_a', \tau_b, h + 1)$;	;; Compute handler $vm' = vm, x \mapsto -h - 1$ $\tau_a' = [\![\mathsf{exn}]\!]_{\mathsf{t}} :: \tau_a$
	mov sp, sp $+ 1$;	
	jmp $\mathrm{inst}(\ell_{end}, \vec{t})$;	
$\ell_{end} \mapsto$	code $[\vec{t}]\mathrm{ec}(\tau_a, \tau_b, [\![\tau]\!]_{\mathsf{t}})$	
))		

Figure 2.7: IIL to TAL Compiler for Expressions 3

The IIL source is:

$$(\mathsf{fix}\ f(n:\mathsf{int}):\mathsf{int}.\mathsf{if}\ 0\leq\ n\ \mathsf{then}\ 1\ \mathsf{else}\ n*f(n-1)\ \mathsf{fi})\ 6$$

The TAL executable is (VH, ℓ_{main}) where $\tau_a = \mathrm{ec}(\rho_1, \rho_2, \mathsf{int}) :: \mathsf{int} :: \rho_1$ and VH is:

$$
\begin{array}{ll}
\ell_{main} \mapsto & \Lambda[]\mathsf{code} \\
& \mathsf{push}\ \ell_{uncaught}; \mathsf{mov}\ \mathsf{re}, \mathsf{sp}; \\
& \mathsf{push}\ 6; \\
& \mathsf{push}\ \ell_{halt}; \\
& \mathsf{jmp}\ \ell_f[\mathsf{se}, \mathsf{se}] \\
& : *\mathsf{code}\ \{\mathsf{sp} : \mathsf{se}\} \\
\ell_{halt} \mapsto & \Lambda[]\mathsf{code} \\
& \mathsf{halt}[\mathsf{int}] \\
& :*\mathsf{code}\ \{\mathsf{sp} : \mathsf{se} \circ \mathrm{ht}(\mathsf{se}), \mathsf{re} : \mathsf{sptr}(\mathrm{ht}(\mathsf{se})), \mathsf{r1} : \mathsf{int}\} \\
\ell_{uncaught} \mapsto & \Lambda[]\mathsf{code} \\
& \mathsf{halt}[[\![\mathsf{exn}]\!]_{\mathsf{t}}] \\
& :*\mathsf{code}\ \{\mathsf{r1} : [\![\mathsf{exn}]\!]_{\mathsf{t}}, \mathsf{sp} : \mathsf{se}\} \\
\ell_f \mapsto & \Lambda[\rho_1 : \mathsf{S}, \rho_2 : \mathsf{S}]\mathsf{code} \\
& \mathsf{mov}\ \mathsf{r1}, [\mathsf{sp}+1]; \\
& \mathsf{bg}\ \mathsf{r1}, \ell_{false}[\rho_1, \rho_2]; \\
& \mathsf{mov}\ \mathsf{r1}, 1; \\
& \mathsf{jmp}\ \ell_{end}[\rho_1, \rho_2] \\
& : \forall[\rho_1 : \mathsf{S}, \rho_2 : \mathsf{S}]*\mathsf{code}\ \{\mathsf{sp} : \tau_a \circ \mathrm{ht}(\rho_2), \mathsf{re} : \mathsf{sptr}(\mathrm{ht}(\rho_2))\} \\
\ell_{false} \mapsto & \Lambda[\rho_1 : \mathsf{S}, \rho_2 : \mathsf{S}]\mathsf{code} \\
& \mathsf{mov}\ \mathsf{r1}, [\mathsf{sp}+1]; \mathsf{push}\ \mathsf{r1}; \\
& \mathsf{sub}\ \mathsf{r1}, \mathsf{r1}, 1; \mathsf{push}\ \mathsf{r1}; \\
& \mathsf{push}\ \ell_{ret}[\rho_1, \rho_2]; \mathsf{jmp}\ \ell_f[\mathsf{int} :: \tau_a, \rho_2] \\
& : \forall[\rho_1 : \mathsf{S}, \rho_2 : \mathsf{S}]*\mathsf{code}\ \{\mathsf{sp} : \tau_a \circ \mathrm{ht}(\rho_2), \mathsf{re} : \mathsf{sptr}(\mathrm{ht}(\rho_2))\} \\
\ell_{ret} \mapsto & \Lambda[\rho_1 : \mathsf{S}, \rho_2 : \mathsf{S}]\mathsf{code} \\
& \mathsf{pop}\ \mathsf{r2}; \\
& \mathsf{mul}\ \mathsf{r1}, \mathsf{r2}, \mathsf{r1}; \\
& \mathsf{jmp}\ \ell_{end}[\rho_1, \rho_2] \\
& : \forall[\rho_1{:}\mathsf{S}, \rho_2{:}\mathsf{S}]*\mathsf{code}\ \{\mathsf{sp} : (\mathsf{int} :: \tau_a) \circ \mathrm{ht}(\rho_2), \mathsf{re} : \mathsf{sptr}(\mathrm{ht}(\rho_2)), \mathsf{r1} : \mathsf{int}\} \\
\ell_{end} \mapsto & \Lambda[\rho_1 : \mathsf{S}, \rho_2 : \mathsf{S}]\mathsf{code} \\
& \mathsf{pop}\ \mathsf{r2}; \\
& \mathsf{mov}\ \mathsf{sp}, \mathsf{sp}+1; \mathsf{jmp}\ \mathsf{r2} \\
& : \forall[\rho_1 : \mathsf{S}, \rho_2 : \mathsf{S}]*\mathsf{code}\ \{\mathsf{sp} : \tau_a \circ \mathrm{ht}(\rho_2), \mathsf{re} : \mathsf{sptr}(\mathrm{ht}(\rho_2)), \mathsf{r1} : \mathsf{int}\}
\end{array}
$$

Figure 2.8: Factorial Example

Chapter 3

Modular Typed Assembly Language

A critical problem with TAL is that it formalises only complete programs—there is no notion of a compilation unit, nor of separate type checking. However, to keep soft development management, nontrivial software is divided into a number of compilation units that are compiled separately into object files. To use TAL in such a separate compilation run requires that object files be checkable in isolation with just the interfaces for the other object files. TAL lacks such a separate type-checking property. To support separate compilation, this chapter presents one one to extend TAL to include object files as modules, such that separate type checking is possible. The design of this module system led me into the area of linking; I discuss this next.

Linking separately compiled program units is an important task that is typically omitted from language definitions. In large part, this omission is due to low-level architecture and compiler dependencies that seem outside the realm of language design. However, language-based security, mentioned in the introduction, is based upon the strong safety guarantees that language definitions provide. Language-based security systems use linking and loading as a fundamental part of their operation, so it is critical to define precisely the consistency checks a linker must perform. As an example, web applets are written in Java Virtual Machine byte code (JVML [LY96]). As the JVM supports dynamic linking and loading of applets, a critical component of the JVM definition is the description of well-formed compilation units and link compatibility. Unfortunately, this component is vaguely specified and has been a source of well publicised security holes [DFWB97].

Recently, Cardelli [Car97] proposed a calculus of compilation units for the simply-typed lambda calculus and presented a set of rules for determining link compatibility. Cardelli's work specified high-level abstractions for modules and interfaces and provided a set of inference rules for determining that program fragments, when compiled under certain typing assumptions, met a set of consistency requirements necessary to ensure that the resulting linked program was well-formed and hence would not go wrong when evaluated.

It seemed natural to extend TAL with the ideas of Cardelli to give a detailed treatment of type-safe linking. However, though Cardelli's calculus is an elegant formulation of some of the high-level issues involved in linking, it abstracts important low-level details such as binding and α-variance of labels; it also omits certain critical features, notably support for cyclic inter-object file references, user-defined type abstraction, and dynamic linking. The goal of this chapter is to build upon Cardelli's work and provide a suitable treatment of these issues. In particular, I extend core TAL with a language of typed object files and formalise the concepts of linking and link compatibility. The goals of the design are to model important properties of conventional object files and linkers (*e.g.*, Unix's `ld` or Win32's `link`), and to provide a module structure that supports separate type-checking of object files and separate compilation of high-level language features such as the abstract types, signatures, structures, and functors of SML.

My design for typed object files borrows heavily from the previous work on modules for high-level languages and hence there are only a few important innovations. However, I believe this to be a virtue as it demonstrates that the programming language community has identified most of the critical issues for *any* module language, and it allows us to concentrate on those issues specific to object files.

I proceed as follows: In Section 3.1, I present the abstractions of conventional *untyped* object files and linkers (*e.g.*, Unix's `ld`) and discuss the issues of link compatibility in this simplified setting. In Section 3.2, I introduce a simple module language $\textsc{Mtal}_0$, which, in the spirit of Cardelli, provides support for separate compilation, separate type-checking, and a stronger notion of link compatibility. I extend $\textsc{Mtal}_0$ in Section 3.3 with support for abstract types in the style of Clu or Modula-2, higher-order type constructors in the style of Objective Caml, and translucent types in the style of Harper and Lillibridge [HL94] and Leroy [Ler94]. The resulting language, MTAL (pronounced metal), is sufficiently expressive that we can compile ML-style modules, including functors, to the target language. Finally, in Section 3.4 I discuss extending my model to include dynamic linking and dynamic loading.

3.1 Untyped Object Files and Linkers

I begin with a model of typical untyped object files and the process of linking. For the duration of this section imagine an untyped variant of TAL, where there are no typing annotations, types, or typing rules.

3.1.1 Object Files

Abstractly, an object file consists of three components:

1. *A heap VH.*

2. *An import set $\mathcal{I}$*: a set of labels not defined in the heap of the object file, but possibly referenced by terms in the heap.

3. *An export set $\mathcal{E}$*: a subset of the labels defined in the heap.

This description of object files could be used with heaps mapping labels to the terms, typed or untyped, of any language, but in order to provide specific examples, this section uses an untyped assembly language.

I use $[\mathcal{I} \Rightarrow VH : \mathcal{E}]$ to denote an object file and give the well-formedness conditions with the following inference rule, where $\mathrm{fl}(e)$ denotes the set of free labels occurring in a term e:

$$\frac{\mathcal{E} \subseteq \mathrm{dom}(VH) \quad \mathcal{I} \cap \mathrm{dom}(VH) = \emptyset \quad \forall \ell \in \mathrm{dom}(VH) : \mathrm{fl}(VH(\ell)) \subseteq \mathrm{dom}(VH) \cup \mathcal{I}}{\vdash_{\mathsf{O}} [\mathcal{I} \Rightarrow VH : \mathcal{E}]}$$

The set of labels that are defined in the heap but not in the export set are said to be *local labels*, as the scope of these labels is the object file only. Following standard convention for fixed-scope identifiers, we consider object files to be equivalent up to a systematic renaming (α-conversion) of local labels. The justification for this implicit α-conversion is that real object files represent local labels as relative offsets from the base address of the object file. This base address is adjusted during the linking and/or loading process to place object files in different address ranges and hence the local labels are implicitly adjusted. In contrast, exported labels do *not* α-vary so that the linker can resolve cross references among object files.

3.1.2 Linking Untyped Object Files

Linking is the process of taking two (or more) object files and combining their heaps, import sets, and export sets in a suitable fashion to produce a new object file. However, even if the input object files are well-formed, the output may not be. This motivates the notion of *link compatibility.* To object files O_1 and O_2 are *link compatible*, written $\vdash O_1 \overset{\mathsf{lc}}{\leftrightarrow} O_2$, when the following rule holds:[1]

$$\frac{\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset}{\vdash [\mathcal{I}_1 \Rightarrow VH_1 : \mathcal{E}_1] \overset{\mathsf{lc}}{\leftrightarrow} [\mathcal{I}_2 \Rightarrow VH_2 : \mathcal{E}_2]}$$

When two object files are link compatible, we may link them to produce a new object file as follows:

$$\frac{\begin{array}{c} \vdash_{\mathsf{O}} [\mathcal{I}_1 \Rightarrow VH_1 : \mathcal{E}_1] \\ \vdash_{\mathsf{O}} [\mathcal{I}_2 \Rightarrow VH_2 : \mathcal{E}_2] \\ \vdash [\mathcal{I}_1 \Rightarrow VH_1 : \mathcal{E}_1] \overset{\mathsf{lc}}{\leftrightarrow} [\mathcal{I}_2 \Rightarrow VH_2 : \mathcal{E}_2] \\ \mathrm{dom}(VH_1) \cap \mathrm{dom}(VH_2) = \emptyset \\ (\mathrm{dom}(VH_1) - \mathcal{E}_1) \cap \mathcal{I}_2 = \emptyset \\ (\mathrm{dom}(VH_2) - \mathcal{E}_2) \cap \mathcal{I}_1 = \emptyset \end{array}}{\begin{array}{l} \vdash [\mathcal{I}_1 \Rightarrow VH_1 : \mathcal{E}_1] \;\mathsf{link}\; [\mathcal{I}_2 \Rightarrow VH_2 : \mathcal{E}_2] \leadsto \\ \quad [(\mathcal{I}_1 \cup \mathcal{I}_2) \setminus (\mathcal{E}_1 \cup \mathcal{E}_2) \Rightarrow (VH_1 \cup VH_2) : (\mathcal{E}_1 \cup \mathcal{E}_2)] \end{array}}$$

Because of the bottom three conditions, in applying the link rule, α-variants of the object files must be chosen such that their local labels are disjoint. It follows from the definitions that if $\vdash O_1 \;\mathsf{link}\; O_2 \leadsto O$ then $\vdash_{\mathsf{O}} O$ (*i.e.*, the resulting object file is well-formed).

3.1.3 Static Executables

The final operation the linker performs is to produce an executable. An executable, ranged over by metavariable E, is just a closed heap paired with an entry label defined in that heap, written (VH, ℓ). Thus, an executable is well-formed according to the following rule:

$$\frac{\ell \in \mathrm{dom}(VH) \quad \forall \ell' \in \mathrm{dom}(VH) : \mathrm{fl}(VH(\ell')) \subseteq \mathrm{dom}(VH)}{\vdash_{\mathsf{E}} (VH, \ell)}$$

Given a well-formed object file and a distinguished label from its export set, we may produce a well-formed executable, written $\vdash O, \ell \overset{\mathsf{prg}}{\leadsto} E$, only when the import set of the object file is empty:

$$\frac{\vdash_{\mathsf{O}} [\mathcal{I} \Rightarrow VH : \mathcal{E}] \quad \mathcal{I} = \emptyset \quad \ell \in \mathcal{E}}{\vdash [\mathcal{I} \Rightarrow VH : \mathcal{E}], \ell \overset{\mathsf{prg}}{\leadsto} (VH, \ell)}$$

To load and run an executable, the operating system creates a new process with the heap as its initial memory image and jumps to the entry label passing in some parameters.[2] Hence,

[1] In this section on untyped object files and linking, many side conditions will be written above the line because otherwise the rules would look very awkward. In the next section, side conditions will are written on the right side of the line.

[2] On Unix system the parameters are the command line arguments and the environment. On GUI systems like Win32, the parameters are GUI handles for the application and/or the main window and the environment.

ignoring the parameters, an executable is mapped to an initial program state, written $\vdash E \overset{\mathsf{exec}}{\leadsto} P$, by taking the heap of the executable, an empty register file, and a single-instruction sequence that jumps to the code bound to the entry label of the executable:

$$\frac{\vdash_{\mathsf{E}} (VH, \ell)}{\vdash (VH, \ell) \overset{\mathsf{exec}}{\leadsto} (VH, \{\mathsf{sp} \mapsto \mathsf{se}\}, \mathsf{jmp}\ \ell)}$$

3.2 MTAL$_0$

The goal of this chapter is to formalise typed object files combining the development in Section 3.1 with Cardelli's high level, typed linking ideas [Car97]. As a step towards this goal, this section defines MTAL$_0$, a simple, typed object file language; the next section will extend MTAL$_0$ to full MTAL by adding abstract types (and abstract type constructors). While the module language is independent of the core language, MTAL$_0$ is based on TAL for concreteness. In the following sections, I briefly review the benefits of type safety, and then build a notion of typed object files on top of TAL.

3.2.1 Type Safety

The main motivation for static typing is the property that a well typed program never performs an illegal operation, such as jumping to data instead of code. To achieve this goal for MTAL$_0$, I must design linking and execution to be type preserving. Then, a set of type-correct and link-compatible object files will link to form a type correct executable. Loading this type correct executable will produce a type correct initial program state. By TAL's type safety, the execution resulting from this type correct initial state will not perform illegal operations. Thus, to prove that MTAL$_0$ is type safe, I must prove that linking, executable formation, and initial state formation are type preserving.

An extensible system writer may desire other guarantees from MTAL$_0$. For example, if an extension is checked with a fixed import interface, the MTAL$_0$ type system guarantees that the extension can access only the labels mentioned in that interface. The extensible system can use this fact to ensure that a security monitor interposed between the extension and the underlying system is not circumvented. An overview of the necessary guarantees and security properties of extensible systems is beyond the scope of this dissertation, but Leroy and Rouaix [LR98] provide a discussion of some of these issues.

3.2.2 Object Files and Interfaces

MTAL$_0$'s object files extend untyped object files with types. A MTAL$_0$ object file is a triple $[\Psi_I \Rightarrow VH : \Psi_E]$ where VH is a TAL heap, Ψ_I is an import interface, and Ψ_E is an export interface. An interface is a heap typing, that is, $\{\ell_1; c_1, \ldots, \ell_n; c_n\}$. Figure 3.1 shows an example MTAL$_0$ program consisting of two object files, `fact.tal` and `main.tal`. The intention is that an integer n is passed in register `r1` to the entry label `main`. The main object file calls the other object file's `fact` label which computes and returns the factorial of its argument. The main object file then halts with $n!$ in register `r1`. The keywords `import` and `export` are used to show the import and export interfaces respectively.

In addition to the checks made in untyped object files, the well formedness condition for `fact.tal`:

$$
\begin{array}{l}
\text{export } \texttt{fact}: \tau_{fact} \\
\texttt{fact} \mapsto \Lambda[\rho{:}\mathsf{S}]\mathsf{code} \\
\quad \begin{array}{ll}
\mathsf{mov} & \mathsf{r2}, \mathsf{r1} \\
\mathsf{mov} & \mathsf{r1}, 1 \\
\mathsf{jmp} & \texttt{loop}[\rho]
\end{array} \\
\quad : \tau_{fact} \\
\texttt{loop} \mapsto \Lambda[\rho{:}\mathsf{S}]\mathsf{code} \\
\quad \begin{array}{ll}
\mathsf{ble} & \mathsf{r1}, \mathsf{ra} \\
\mathsf{mul} & \mathsf{r2}, \mathsf{r1}, \mathsf{r2} \\
\mathsf{sub} & \mathsf{r1}, \mathsf{r1}, 1 \\
\mathsf{jmp} & \texttt{loop}[\rho]
\end{array} \\
\quad : \forall\rho{:}\mathsf{S}.{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathsf{int}, \mathsf{r2}{:}\mathsf{int}, \mathsf{ra}{:}\tau_{ret}\}
\end{array}
$$

`main.tal`:

$$
\begin{array}{l}
\text{import } \texttt{fact}: \tau_{fact} \\
\text{export } \texttt{main}: {*}\mathsf{code}\ \{\mathsf{sp}{:}\mathsf{se}, \mathsf{r1}{:}\mathsf{int}\} \\
\texttt{main} \mapsto \Lambda[]\mathsf{code} \\
\quad \begin{array}{ll}
\mathsf{mov} & \mathsf{ra}, \texttt{ret1} \\
\mathsf{jmp} & \texttt{fact}[\mathsf{se}]
\end{array} \\
\quad : {*}\mathsf{code}\ \{\mathsf{sp}{:}\mathsf{se}, \mathsf{r1}{:}\mathsf{int}\} \\
\texttt{ret1} \mapsto \Lambda[]\mathsf{code} \\
\quad \mathsf{halt}[\mathsf{int}] \\
\quad : {*}\mathsf{code}\ \{\mathsf{sp}{:}\mathsf{se}, \mathsf{r1}{:}\mathsf{int}\}
\end{array}
$$

Where $\tau_{fact} = \forall\rho{:}\mathsf{S}.{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathsf{int}, \mathsf{ra}{:}\tau_{ret}\}$ and $\tau_{ret} = {*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathsf{int}\}$.

Figure 3.1: Modular Factorial

MTAL_0 object files requires type checking:

$$\frac{\vdash_{\mathsf{VHT}} \Psi_I \quad \vdash_{\mathsf{VHT}} \Psi_A \leq \Psi_E \quad \Psi_I \cup \Psi_A \vdash_{\mathsf{VH}} VH : \Psi_A \quad \mathrm{dom}(\Psi_I) \cap \mathrm{dom}(\Psi_A) = \emptyset}{\vdash_{\mathsf{O}} [\Psi_I \Rightarrow VH : \Psi_E]}$$

The heap has an actual type Ψ_A and is checked in the context $\Psi_I \cup \Psi_A$ as it may refer to imported labels or to itself. The heap must define labels different from the imports, that is, Ψ_A and Ψ_I must have disjoint domains. The heap must provide the exported labels at the types specified, $\vdash_{\mathsf{VHT}} \Psi_A \leq \Psi_E$.[3]

A typed object file can be checked in isolation. While it contains type information about labels in other object files, it does not contain any term level information about those labels. Put another way, MTAL_0 has a separate-type-checking property and thus MTAL_0 supports separate compilation in the following fashion: If a source-level module can be type checked using only source-level interfaces for other modules, then it can be compiled to a typed object file without needing the implementations of the other modules.

3.2.3 Linking

Crucial to typed link compatibility is *interface compatibility*, $\vdash_{\mathsf{VHT}} \Psi_1 \sim \Psi_2$. In particular, if two interfaces mention the same label then they must give it compatible types as stated in the following rule:[4]

$$\frac{\forall \ell \in \mathrm{dom}(\Psi_1) \cap \mathrm{dom}(\Psi_2) : \Psi_1(\ell) = \Psi_2(\ell)}{\vdash_{\mathsf{VHT}} \Psi_1 \sim \Psi_2}$$

Given interface compatibility, link compatibility is easily defined:

$$\frac{\vdash_{\mathsf{VHT}} \Psi_{I1} \sim \Psi_{I2} \quad \vdash_{\mathsf{VHT}} \Psi_{I1} \sim \Psi_{E2} \quad \vdash_{\mathsf{VHT}} \Psi_{I2} \sim \Psi_{E1} \quad \mathrm{dom}(\Psi_{E1}) \cap \mathrm{dom}(\Psi_{E2}) = \emptyset}{\vdash [\Psi_{I1} \Rightarrow VH_1 : \Psi_{E1}] \overset{\mathsf{lc}}{\leftrightarrow} [\Psi_{I2} \Rightarrow VH_2 : \Psi_{E2}]}$$

[3] $\vdash_{\mathsf{VHT}} \Psi_A \leq \Psi_E$ means that Ψ_A is a subinterface of Ψ_E, formally, $\forall \ell \in \mathrm{dom}(\Psi_E) : \Psi_A(\ell) = \Psi_E(\ell)$. When subtyping is added, subinterface will mean $\forall \ell \in \mathrm{dom}(\Psi_E) : \Psi_A(\ell) \leq \Psi_E(\ell)$.

[4] In MTAL_0, compatible types are equal types. When subtyping is added, compatibility of types could be a weaker condition, see the appendix for details.

The two object files have compatible imports and exports, and the exports must (as before) be disjoint. The link operation is defined in the same way as the untyped link operation but uses typed object files and typed judgements. Again, if $\vdash O_1 \; \mathsf{link} \; O_2 \rightsquigarrow O$ then $\vdash_{\mathsf{O}} O$. This theorem is much stronger than in the untyped case as it asserts that no type errors are introduced by a linking operation.

$\text{M}\textsc{tal}_0$ has a separate link-checking property. That is, link compatibility is defined entirely in terms of the imported and exported interfaces of the two modules and is independent of the modules's heaps. A type-safe linker will load each object file and type check it separately, then perform the linking, doing checks that involve only the interface information; the code need not be rechecked.

3.2.4 Executables and Execution

An executable is a closed TAL heap and a label. The heap must be well formed and the label must have an appropriate type:

$$\frac{\vdash_{\mathsf{VHT}} \Psi \quad \Psi \vdash_{\mathsf{VH}} VH : \Psi}{\vdash_{\mathsf{E}}^{c_e} (VH, \ell)} \; (\Psi(\ell) = c_e)$$

where c_e is the type the entry convention gives the entry label. The factorial example's intended entry convention has $c_e = *\mathsf{code}\ \{\mathsf{sp{:}se}, \mathsf{r1{:}int}\}$. The entry convention is an important low-level detail of how programs get executed, which we can formally specify as a $\text{M}\textsc{tal}_0$ type.

We can check when an object file is complete, written $\vdash O, \ell : c_e \; \mathsf{complete}$, as follows:

$$\frac{\vdash_{\mathsf{O}} [\epsilon \Rightarrow VH : \Psi_E]}{\vdash [\epsilon \Rightarrow VH : \Psi_E], \ell : c_e \; \mathsf{complete}} \; (\Psi_E(\ell) = c_e)$$

However, programmers and language designers want to reason about when a collection of object files together forms a complete program. That is, they want a set of checks to ensure that when those object files are linked the result will be a complete program according to the judgement above. Informally, each object file's imports must be contained within the exports of the other object files and the entry label must be exported by one of the object files with type c_e. Glew and Morrisett [GM99a] formalised these checks.

The production of an executable and the process of execution is the same as in the untyped language. However, the consistency checks are sound: the formation of an executable implies that the executable is well formed, and the formation of an initial state implies that the initial state is well formed. Theorem 2.1 states that well formed intial states do not commit run-time type errors during execution.

3.3 MTAL

$\text{M}\textsc{tal}_0$ is a typed low-level language with a formalised notion of link compatibility. It extends the work of Cardelli making important low level concerns explicit, and it very closely models the operation of real linkers. However, $\text{M}\textsc{tal}_0$ does not address other shortcomings of Cardelli's language. I will progressively add constructs to $\text{M}\textsc{tal}_0$ in the following sections to obtain a full language MTAL. MTAL is a subset of MOOTAL, the complete typed assembly language of this dissertation. A complete description (of both), including syntax, operational semantics, and typing rules, appears in the appendix. The syntax changes from TAL to MTAL are summarised in Figure 3.2.

Kinds	κ	$::=$	$\cdots \mid \kappa_1 \to \kappa_2$
Type Constructors	c	$::=$	$\cdots \mid \ell \mid \lambda\alpha{:}\kappa.c \mid c_1\ c_2$
Type Heap Types	Φ	$::=$	$\{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}$
Interfaces	Int	$::=$	(Φ, Ψ)
Coercions	δ	$::=$	$\cdots \mid \mathsf{roll}^c \mid \mathsf{unroll}$
Type Constructor Heaps	CH	$::=$	$\{\ell_1 \mapsto c_1{:}\kappa_1, \ldots, \ell_n \mapsto c_n{:}\kappa_n\}$
Object Files	O	$::=$	$[Int_I \Rightarrow CH, VH : Int_E]$
Executables	E	$::=$	(CH, VH, ℓ)
Program States	P	$::=$	(CH, VH, R, I)

Figure 3.2: Syntax Changes from TAL to MTAL

3.3.1 Abstract Types

MTAL_0 provides many type safety guarantees but does not provide type abstraction guarantees.[5] Consider a security monitor for file access that exports an operation `open` that takes a string and returns a file handle. Suppose further that the file handle pairs the extension's access rights with an operating system file handle, each represented as an integer. In a system without type abstraction, the implementation must expose the representation of the file handle giving `open` the type **string** $\to \langle \mathsf{int}, \mathsf{int} \rangle$. Because clients see this type, not a type like **string** $\to$ `file`, they can ignore the abstraction and use integer operations to modify the access rights directly. Following high-level module designs which address this issue, I add to MTAL the ability to declare abstract types in interfaces and use them in the types given to labels.

A MTAL *interface*, ranged over by metavariable Int, is a pair (Φ, Ψ) consisting of a type part Φ and a value part Ψ. The type part, also called a *type heap typing*, is a finite map from *type labels* to kinds. Object files are quadruples $[Int_I \Rightarrow CH, VH : Int_E]$ consisting of an import and export interface, but there are now two heaps: one for types and one for values. *Type heaps* are finite mappings from labels to types and their kinds. Program states are also extended to include a type heap.

The file example is shown in Figure 3.3. It exports an abstract type `file` which is used in the types of the values that it exports. The concrete type of `file` is a pair of integers, and the example sketches the relevant details of the implementation of the operations.

Definitions of typed object files, link compatibility, linking, executable formation, and execution similar to that in Sections 3.1 and 3.2 can be repeated for MTAL; I mention just the highlights.

Just as value heaps can contain cyclic references, type heaps can contain cyclic references also, introducing the possibility of recursive types. Following standard type theory, in a type heap CH a type label ℓ is isomorphic to $CH(\ell)$; for example, `file` is isomorphic to $*\langle \mathsf{int}^+, \mathsf{int}^+ \rangle$. There are two ways to reflect this isomorphism in the type system. The first way implicitly treats ℓ and $CH(\ell)$ as equal types. This makes a decision procedure for type equality considerably more complex [AC93]. I choose the second way and introduce explicit roll and unroll operations that witness the isomorphism $\ell \cong CH(\ell)$. For example, roll coerces $*\langle \mathsf{int}^+, \mathsf{int}^+ \rangle$ to `file`; unroll does the opposite.

[5] Technically the original TAL has existential and polymorphic types which can be used to implement type abstractions. However, because of the "local" scope of the quantifiers, this is too cumbersome in practice.

export type $\texttt{file}$: T;
export val open : $\forall\rho$:S.$*$code $\{$sp:ρ, r1:string, ra:$*$code $\{$sp:ρ, r1:$\texttt{file}\}\}$
export val $\texttt{readline}$: $\forall\rho$:S.$*$code $\{$sp:ρ, r1:$\texttt{file}$, ra:$*$code $\{$sp:ρ, r1:string$\}\}$
...
; A file is access rights plus O/S handle
; Access rights: bit 0 read, bit 1 write, ...
$\texttt{file} \mapsto *\langle \text{int}^+, \text{int}^+ \rangle$
open $\mapsto \Lambda[\rho$:S]code
    ; Call O/S open on r1 putting result in r2.
    ; Determine access rights and store in r3 preserving r2.
    malloc   r1, $\langle$int, int$\rangle$
    mov   [r1 + 0], r3
    mov   [r1 + 1], r2
        ; Coerce r1 from $*\langle$int, int$\rangle$ to $\texttt{file}$
    mov   r1, $\text{roll}^{\texttt{file}}$(r1)
    jmp   ra
: $\forall\rho$:S.$*$code $\{$sp:ρ, r1:string, ra:$*$code $\{$sp:ρ, r1:$\texttt{file}\}\}$
$\texttt{readline} \mapsto \Lambda[\rho$:S]code
        ; Coerce r1 from $\texttt{file}$ to $\langle \text{int}^+, \text{int}^+ \rangle$
    mov   r1, unroll(r1)
        ; Check read allowed
    mov   r2, [r1 + 0]
    band   r2, r2, 1
    bz   r2, $\texttt{error}$
        ; Read allowed place O/S handle in r1
    mov   r1, [r1 + 1]
    ; Call O/S read line on r1 putting result in r1
    jmp   ra
: $\forall\rho$:S.$*$code $\{$sp:ρ, r1:$\texttt{file}$, ra:$*$code $\{$sp:ρ, r1:string$\}\}$
...

; Client
import type $\texttt{file}$: T
; Since file is abstract the client cannot coerce $\texttt{file}$ to $\langle$int, int$\rangle$ or vice versa.

Figure 3.3: File Example

The value heap of an object file is checked using its type heap. For example the code that implements `open` and `readline` in the file example is checked using the type heap $\{\texttt{file} \mapsto *\langle \mathsf{int}^+, \mathsf{int}^+\rangle{:}\mathsf{T}\}$. Furthermore, roll^{ℓ} can be used only if the type heap defines ℓ, and similarly for unroll. So the code for `open` can perform a $\mathsf{roll}^{\texttt{file}}$ operation, and the code for `readline` can perform an unroll operation on a value of type `file`. A client of the file module, however, will import `file` as an abstract type of kind T. Since the client's type heap is disjoint from its import interface (a type checking requirement), the client's code will be checked without a definition for `file`. Thus, it will not be able to perform an $\mathsf{roll}^{\texttt{file}}$ operation nor an unroll operation on a value of type `file`. Consequently, the roll and unroll operations are used not only to mediate recursive types, but also to enforce encapsulation of abstract types like `file`.

In this respect, my treatment of label types is similar to the "generative" datatypes of ML. Unlike ML, however, my abstract type labels have global scope. This simplifies link consistency and provides a means to split mutually-recursive type definitions across compilation units as with Units [FF98] and Mixin Modules [DS98]. The price paid, however, is that programmers or compilers must ensure that two compilation units that are to be linked together do not define the same type label.

The implementation (described in Chapter 7) includes two extensions omitted from MTAL. In our implementation of interfaces, a type label may be declared abstract, given a definition, or given a bound. When given a definition, a type label is like the translucent types described by Harper and Lillibridge [HL94] and by Leroy [Ler94]. The definition is included along with the type heap of an object file during the type checking of the object file's value heap. When given a bound, a type label is like a partially abstract type. The typing rules allow a bounded type label to be unrolled to its bound but do not allow a roll operation on that type label. This approach is based upon standard type theory on singleton kinds[6] and power kinds [Car88b] respectively. However, as I only support globally-scoped type labels, the setting is greatly simplified because I do not need both internal and external names for types as Harper and Lillibridge describe. Again, the price paid is that programmers or compilers must manage the flat name space.

In summary, MTAL chooses to treat type labels as globally scoped identifiers. This simplifies the treatment of separately-compiled recursive types, generative abstract types, and translucent types but at the price of a flat name space. Since traditional linkers only provide a flat name space for value labels, I felt that the symmetry at the type-level, together with the simplification of these language features, justified the cost.

3.3.2 Abstract Type Constructors

Good modular programming requires more than just abstract types. For example, there is a large class of container abstractions whose types are parameterised by the types of the objects they contain. For instance, a stack datatype exports an abstract type constructor taking one argument (the type of the elements to be placed in the stack). To handle such constructors, MTAL's types are extended to a type constructor language and its kinds are extended to include functions, resulting in a three tiered system very similar to F_ω [Gir71, Gir72].

Figure 3.4 shows how the stack abstraction might look as a MTAL interface. It declares an abstract type constructor `stack$t` which takes the element type and returns the type of the stacks. Each of the operations is polymorphic over the element type α and the stack arguments and results have type `stack$t` α (the application of the stack type constructor to α). An

[6] Robert Harper, personal communication, July 1998.

Interface:

$\mathsf{type}\ \mathtt{stack\$t} : \mathsf{T} \to \mathsf{T}$
$\mathsf{val}\ \mathtt{stack\$empty} :$
$\quad \forall\alpha{:}\mathsf{T}.\forall\rho{:}\mathsf{S}.{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{ra}{:}{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathtt{stack\$t}\ \alpha\}\}$
$\mathsf{val}\ \mathtt{stack\$isempty} :$
$\quad \forall\alpha{:}\mathsf{T}.\forall\rho{:}\mathsf{S}.{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathtt{stack\$t}\ \alpha, \mathsf{ra}{:}{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathsf{bool}\}\}$
$\mathsf{val}\ \mathtt{stack\$push} :$
$\quad \forall\alpha{:}\mathsf{T}.\forall\rho{:}\mathsf{S}.{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\alpha, \mathsf{r2}{:}\mathtt{stack\$t}\ \alpha, \mathsf{ra}{:}{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathtt{stack\$t}\ \alpha\}\}$
$\mathsf{val}\ \mathtt{stack\$pop} :$
$\quad \forall\alpha{:}\mathsf{T}.\forall\rho{:}\mathsf{S}.{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathtt{stack\$t}\ \alpha, \mathsf{ra}{:}{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathtt{stack\$t}\ \alpha\}\}$
$\mathsf{val}\ \mathtt{stack\$top} :$
$\quad \forall\alpha{:}\mathsf{T}.\forall\rho{:}\mathsf{S}.{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\mathtt{stack\$t}\ \alpha, \mathsf{ra}{:}{*}\mathsf{code}\ \{\mathsf{sp}{:}\rho, \mathsf{r1}{:}\alpha\}\}$

Figure 3.4: Stack Example

implementation of this interface will have to give a concrete type for `stack$t`, for example:

$$\mathtt{stack\$t} \ \mapsto\ \lambda\beta{:}\mathsf{T}\,.\,1 + \langle\beta^+, (\mathtt{stack\$t}\ \beta)^+\rangle$$

To deal with this higher-order recursive type, the roll and unroll coercions must be able to operate "under" type application and type projection. Details are in the appendix.

A final note: ML-style module systems include *functors*, which are functions from modules to modules. Harper et al. [HMM90] showed how to compile functions into a type part, which is a function from the types of the source module to the types of the destination module, and a value part, which is a function polymorphic in the source module types that types the values of the source module to the values of the destination module. Their scheme requires the type system of F_ω. Because MTAL includes the type system of F_ω, some functor systems can be compiled to MTAL. Extending these results to include type sharing requires singleton kinds [SH00].

3.4 Dynamic Linking

Modern operating systems and languages provide dynamic linking and dynamic loading. *Dynamic linking* allows the linker to produce "executables" that contain references to labels that will be resolved at the time the operating system loads the executable into a process's address space. Each executable contains a set of names for dynamically linked libraries, and for each name a set of labels it imports from that library. When the executable is loaded, the operating system searches for appropriate libraries and links them with the executable to form the initial process image. In our model, dynamically linked executables can be represented by normal object files. Indeed, the only difference between the dynamic and static linking in the model is that the final steps of linking and the formation of the "real" executable are delayed until load-time.

Dynamic loading involves linking object files or libraries into the process image during execution. A program might contain references to labels in these dynamically loaded object files. It must ensure that it loads an appropriate object file before using these references. However, it can delay loading until right before use, and if it does not use the references, it need not load the object file. With dynamic loading there is also the possibility of unloading, that is, removing a linked object file from the process image during execution, making references to that object file unusable.

Incorporating dynamic loading into my model is an area of future work. I will briefly discuss some issues that arise. Dynamic loading introduces new failure modes and many interface choices. For example, we could make it the responsibility of an executable to explicitly load definitions for labels before they may be dereferenced. Failure can then be isolated to points where dynamic loading explicitly occurs. Alternatively, as in Java, we could support implicit loading upon reference to an undefined label. Failure in this model can potentially occur at any label dereference.

An important technical issue with dynamic loading is that we must extend our evaluation relation to support execution on program states with unresolved labels. Type or kind information for those labels must be maintained at run-time in order to ensure consistency when dynamic loading is performed. This begs the question of exactly how much type and interface information must be retained and whether it is under program control or operating system control. The presence of this information enables further possibilities, particularly introspection or reflection: the ability of a program to query what labels are defined and at what types.

Recently, Crary et al. [CHW99] have proposed a model of safe dynamic linking. Their work chooses a particular simple dynamic linking primitive and shows how to build a more extensive dynamic linking service upon the simple primitive.

3.5 Related Work

This chapter is taken from Glew and Morrisett [GM99a]. The work it describes is closely related to Cardelli's work on linking [Car97] and builds on the type theory of high-level modules including work by Leroy [Ler94] and by Harper and Lillibridge [HL94]. More recently, Flatt and Felleisen [FF98] have proposed a new advanced module system. Their system includes a first class notion of modules called *units*. Units can import and export named types and values. The named types and values of one unit can be connected to the named types and values of other units. Programmers can abstract over Units, and linking is a first class primitive. MTAL is similar but describes what operating systems provide at the low level, whereas Flatt and Felleisen concentrate on source level module systems. Dean [Dea97] has investigated the dynamic linking and loading aspect of Java; his work focuses on the class loader and how its operation interacts with static typing. His work is a very abstract description of this interaction and does not describe actual linking and link compatibility.

My work is also related to the security of extensible systems. I formalise the checks necessary for linking, but do not address orthogonal security concerns. For example, extensible systems must authenticate principals and determine which interfaces extensions from those principals may link against. Other systems, such as the SPIN project [SFPB96], have addressed these concerns and their ideas could be combined with MTAL's.

Finally, I alluded to the importance of abstract types to building secure extensible systems. The designer of security monitors uses certain guarantees of abstract types. One guarantee is that the client cannot manufacture values of the abstract type, but must call the implementation. Another guarantee is that the client cannot manipulate values of the abstract type except by calling operations in the provided interface. The proof of type safety for MTAL does not directly guarantee these properties of abstract types. They must be proven as an additional result. Recently, Zdancewic et al. [ZGM99] proposed a new syntactic proof technique and showed how to use it to prove these kinds of properties. I believe that their ideas could be adapted to MTAL and used to show suitable properties of abstract types useful for secure extensible systems.

Chapter 4

Object-Oriented Languages

The previous chapter discussed how to extend TAL to deal with separate compilation. The next few chapters will discuss extending TAL to support the compilation of object-oriented constructs. The main problem is that the typical implementations used by compilers produce output that will not type check in TAL's type system. Other schemes exist that do produce output that type checks, but these schemes have higher overheads in that they require extra fields, extra projections, or additional function calls. TAL aims to support many compilation strategies including, in particular, the typical strategies that compilers use. The remaining technical chapters will identify new typing constructs that enable TAL to type check the output of these strategies. Before getting into detail, I review basic object-oriented concepts to make terminology precise and to delineate the landscape I will and will not address.

The central concept of object-oriented languages is a construct called an *object* that combines data and code. An object consists of a number of fields containing data and a number of methods each of which has associated code. When a method of an object is invoked, the associated code is run, and it has access to the object itself through a self variable (Smalltalk's `self` and Java's `this`). Different objects can respond to methods with different code, thus method invocation is often termed *dynamic dispatch*, as it dynamically selects which code to run. Abadi and Cardelli [AC96] formalise a number of variants of a pure object calculus with method invocation, field selection, and field update operations. In their calculi, objects are created by *object constructors*, which list the fields and methods and provide initial values for the fields and code for the methods. They present a number of variants with first, second, and higher order type systems, applicative and imperative semantics, *etc.* Abadi and Cardelli's calculi are pure-object calculi in that they do not have classes. Most object-oriented languages have classes or an equivalent construct such as delegation or prototypes. As I had time only to investigate classes and not delegation or prototypes, this dissertation will focus exclusively on class-based object-oriented languages.

In a class-based language, objects are created by instantiating classes. Classes, like object constructors, specify the fields and methods of their instances and provide initial values for the fields and code for the methods. Unlike object constructors, they also allow for inheritance. A class may optionally extend another class[1], called its *superclass*. Such a class inherits all the fields of its superclass and will respond to all the methods listed in the superclass and in the class itself. The methods fall into three categories: those declared only in the superclass are said to be *inherited*, those declared in both the superclass and the class itself are said to be *overridden*—the instances will respond to the method with the class's code—and those

[1]In multiple inheritance languages, a class may extend several other classes.

```
class Window {
  field extent : Rectangle;
  method handleEvent(Event):bool {···};
  method contains(Point):bool {···};
}
class ContainerWindow extends Window {
  field children : array(Window);
  method handleEvent(Event):bool {···};
  method addChild(Window) {···};
}
```

Figure 4.1: Example Class Hierarchy

declared only in the class itself are said to be new methods. Thus, programmers can code basic functionality in superclasses and code more refined functionality in subclasses, sharing the basic code amongst the extensions.

A small example of classes appears in Figure 4.1. This example is a fragment of toolkit for graphical user interfaces. The actual method bodies are elided. The class `Window` represents windows on the user's screen. It consists of a field `extent` that stores a rectangle for the part of the screen the window occupies, a method `handleEvent` for handling user input such as key presses and mouse movements, and a method `contains` that determines if a screen coordinate is within the window's boundaries. The class `ContainerWindow` extends `Window` and represents composite windows. It inherits the `extent` field and `contains` method, but it overrides the `handleEvent` method (perhaps to distribute the event to the child windows), and it adds a field and a new method.

The primary role of classes is to provide a template for the construction of objects; I will call this the *template* role. Classes also play a number of other roles:

- Classes often name a type. The class `Window` has a corresponding type name `Window` that is an object type for all the instances of `Window` and its subclasses.
- Closely related to the previous role, classes provide for run-time type dispatch or run-time class dispatch. For example, Java has the operation `(ContainerWindow)`e that checks that e is actually an instance of the class `ContainerWindow` or one of its subclasses.
- Classes provide constructors. Objects can only be created by invoking one of these constructors so a class can ensure that an object gets properly initialised, can establish object invariants, and can maintain class invariants.
- A class has an associated object, called a class object. This object usually contains *class fields* that are shared by all of instances of a class.

Which roles a class plays and their particular details vary from language to language. But to support the compilation of object-oriented languages to TAL, TAL must provide typing constructs that allow natural compilation strategies for all of these roles. My dissertation research has concentrated on two of these roles: the template role and run-time class dispatch. Furthermore, I consider only single-inheritance classes and right-extension[2] object types. This

[2]Right-extension subtyping means that fields are ordered and that a type with more fields on the right end is a subtype of type with less fields.

chapter defines a core language for object templates and objects, the next chapter translates this language into an extension of IIL, and the following chapter discusses class dispatch. The next section presents the core language. Objects in the language might contain references to variables bound in outer-nested objects. Since these free variables lead to free variables in the translated functions and IIL requires all functions to be closed, the compiler employs *closure conversion* to make all objects closed before they are translated to IIL. Closure conversion is discussed in the second section of this chapter.

4.1 Object Template Language

In this section I define an object template language, O. This language captures the template role of classes and the basic operations on objects available in class-based object-oriented languages. The two important abstractions are *object* and *object templates*. An object template captures the role of a class as a template for creating objects. A class's template is built starting from its superclass's template and by adding fields and by adding or overriding methods. The syntax for O is:

Types	τ, σ	$::=$	$\alpha \mid \mathsf{objt}\ r$
Template Types	T	$::=$	$\mathsf{tempt}\ r$
Rows	r	$::=$	$[m_i{:}s_i;\, f_j{:}\sigma_j^{\phi_j}]_{i \in I, j \in J}$
Variances	ϕ	$::=$	$+ \mid - \mid \circ$
Signatures	s	$::=$	$[\vec{\alpha}](\vec{\tau}) \to \tau$
Expressions	e, b	$::=$	$x \mid \mathsf{let}\ t = te\ \mathsf{in}\ e \mid t[f_j = e_j]_{j \in J} \mid$
			$e.m[\vec{\tau}](\vec{e}) \mid e.f \mid e_1.f \leftarrow e_2$
Template Expressions	te	$::=$	$t \mid \mathsf{et} \mid te{+}f{:}\sigma^{\phi} \mid te.f{\leftarrow}\sigma^{\phi} \mid te \leftarrow\!\!+[m_i{=}M_i]_{i \in I}$
Method Definitions	M	$::=$	$x[\vec{\alpha}](\overrightarrow{x{:}\tau}).b{:}\tau$

Rows are used in object types to describe objects and in template types to describe the objects the templates will create. The row $[m_i{:}s_i;\, f_j{:}\sigma_j^{\phi_j}]_{i \in I, j \in J}$ describes objects with methods m_i that have signatures s_i, and fields f_j that have types σ_j and variances ϕ_j (O's variances are the same as IIL's). The indices i and j range over index sequences I and J, and I will write $I \leq J$ to mean J is a prefix of I.[3] The signature $[\alpha_1, \ldots, \alpha_m](\tau_1, \ldots, \tau_n) \to \tau$ specifies methods that take m type parameters α_1 through α_m, that take n value parameters of types τ_1 through τ_n, and that return a result of type τ.

Object templates provide a pattern for the creation of an object. For this dissertation, they contain the methods the object will have and a list of the fields the object will have, but not the initial values of the fields. An object template for objects described by row r has type $\mathsf{tempt}\ r$. Object templates are constructed by starting with the empty template et, which has no methods or fields, and by adding fields and methods. A field f of type τ and variance ϕ is added to an object template e with $e + f : \tau^{\phi}$. A field f of an object template e can be changed to have type τ and variance ϕ with $e.f \leftarrow \tau^{\phi}$; the new type and variance pair must be a subtype of the old one.[4] As methods can be mutually recursive, several methods can be added in one operation. Methods m_i with definitions M_i are added to an object template e with $e \leftarrow\!\!+[m_i = M_i]_{i \in I}$. If these methods already exist in the template, they are replaced by the new definitions. A method definition $x[\vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b$ takes type parameters $\vec{\alpha}$, takes value

[3]It is unfortunate that prefix is often written the other way around, because I want $\leq$ to consistently mean subtype.

[4]Abadi and Cardelli [AC96] provide a description of subtyping for pairs of types and variances.

parameters $\vec{x}$ of types $\vec{\tau}$, and returns a result of type τ by executing body b where x is the self variable (bound to the object itself).

As an example of the use of template operations, the templates for the example class hierarchy could be built with the following code:

$$
\begin{array}{l}
\mathsf{let}\ \mathtt{Window} = \\
\quad \mathsf{et} + \mathtt{extent} : \mathtt{Rectangle}^{\circ} \leftarrow\!\!+ [\mathtt{handleEvent} = \cdots, \mathtt{contains} = \cdots]\ \mathsf{in} \\
\mathsf{let}\ \mathtt{ContainerWindow} = \\
\quad \mathtt{Window} + \mathtt{children} : \mathsf{array}(\mathtt{Window})^{\circ} \leftarrow\!\!+ \\
\qquad [\mathtt{handleEvent} = \cdots, \mathtt{addChild} = \cdots]\ \mathsf{in}
\end{array}
$$

Objects described by row r have type $\mathsf{objt}\ r$, and are created by instantiating a template e with the operation $e[f_j = e_j]_{j \in J}$ where e_j is the initial value of field f_j. Each field listed in e must be given an initial value, and no other field can be given a value. Objects are manipulated by method invocation $e.m$, field selection $e.f$, and field update $e_1.f \leftarrow e_2$. The following example creates an instance of `Window`, creates an instance of `ContainerWindow`, and adds the former as a child to the latter.

$$
\begin{array}{l}
\mathsf{let}\ w1 = \mathtt{Window}[\mathtt{extent} = r1]\ \mathsf{in} \\
\mathsf{let}\ w2 = \mathtt{ContainerWindow}[\mathtt{extent} = \mathtt{r2}, \mathtt{children} = \mathsf{array}()]\ \mathsf{in} \\
w2.\mathtt{addChild}[](w1)
\end{array}
$$

The operational semantics of the template language appears in Figure 4.2. It is a deterministic, left to right, call by value, context based, reduction semantics. As with IIL, a heap is used to store the current values of mutable things, in O's case the objects. A heap maps locations to objects consisting of a list of methods and associated method bodies and a list of fields and their associated values. Also similar to IIL, the operational semantics uses evaluation contexts E to determine the next thing to evaluate in a left to right evaluation order. As examples of the reduction rules consider the the fourth and eight reduction rules. The fourth rule says that to evaluate the template expression $te + f{:}\sigma^{\phi}$, te is evaluated to a template value $\mathsf{temp}[m_i{:}M_i; f_j{:}\sigma_j^{\phi_j}]_{i \in I, j \in J}$ consisting of methods m_i with bodies M_i and fields f_j of types σ_j and variances ϕ_j. The result is a new template value $\mathsf{temp}[m_i{:}M_i; f_j{:}\sigma_j^{\phi_j}, f{:}\sigma^{\phi}]_{i \in I, j \in J}$ with the same methods, method bodies, and fields as before, but with an extra field f on the right with type σ and ϕ. Additionally, f (the new field) must not be one of the existing fields, as reflected in the side condition $f \notin f_{j \in J}$. The eight rule says that to evaluate the method invocation expression $e.m_k[\vec{\tau}](\vec{v})$, the expression e is evaluated to a location L. The current heap H must make L to an object $\mathsf{obj}[m_i{=}M_i; f_j{=}v_j]_{i \in I, j \in J}$ and the method body corresponding to m_k, M_k, must have the form $x[\vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b$. The result of the expression the result of substituting the actual type arguments $\vec{\tau}$ for the type parameters $\vec{\alpha}$, the actual arguments $\vec{v}$ for the value parameters $\vec{x}$, and the object itself (*i.e.*, L) for the self variable x into the code b of the method body, that is, the expression $b\{\vec{\alpha}, x, \vec{x} := \vec{\tau}, L, \vec{v}\}$.

The typing rules for the template language appear in Figures 4.3 and 4.4. To simplify the presentation of the typing rules, there are a number of syntactic constraints. First, the method names in a row must be distinct, and similarly for fields names, method names in $e \leftarrow\!\!+ [m_i = M_i]$, and field names in $e[f_j = e_j]$. Second, the type variables in Δ must be distinct, and similarly for variables in Γ. In type rules with contexts of the form Δ, α, it is implicit that $\alpha \notin \Delta$.

At the type level there are judgements for well formedness of types $\Delta \vdash^{\mathrm{O}} \tau$, rows $\Delta \vdash^{\mathrm{O}} r$, and signatures $\Delta \vdash^{\mathrm{O}} s$, and judgements for subtyping $\Delta \vdash^{\mathrm{O}} \tau_1 \leq \tau_2$, subrows $\Delta \vdash^{\mathrm{O}} r_1 \leq r_2$, subsignatures $\Delta \vdash^{\mathrm{O}} s_1 \leq s_2$, and varianced subtyping $\Delta \vdash^{\mathrm{O}} \tau_1^{\phi_1} \leq \tau_2^{\phi_2}$. Template types have

Values	v	$::=$	L
Template Values	tv	$::=$	$\mathsf{temp}[m_i = M_i; f_j : \sigma_j^{\phi_j}]_{i\in I, j\in J}$
Contexts	E	$::=$	$\{\} \mid \mathsf{let}\ t = TE\ \mathsf{in}\ e \mid \mathsf{let}\ t = tv\ \mathsf{in}\ E \mid$ $t[\overrightarrow{f_j = v_j}, f = E, \overrightarrow{f'_j = e}] \mid E.m[\vec{\tau}](\vec{e}) \mid$ $v.m[\vec{\tau}](\vec{v}, E, \vec{e}) \mid E.f \mid E.f \leftarrow e \mid v.f \leftarrow E$
	TE	$::=$	$\{\} \mid TE + f{:}\sigma^{\phi} \mid TE.f \leftarrow \sigma^{\phi} \mid TE \mathrel{\leftarrow\!\!+} [m_i = M_i]_{i\in I}$
Heap Values	h	$::=$	$\mathsf{obj}[m_i = M_i; f_j = v_j]_{i\in I, j\in J}$
Heaps	H	$::=$	$\overrightarrow{L = h}$
Program States	P	$::=$	$\mathsf{letrec}\ H\ \mathsf{in}\ e$

$$\mathsf{letrec}\ H\ \mathsf{in}\ E\{\iota\} \mapsto \mathsf{letrec}\ H'\ \mathsf{in}\ E\{e\}$$

Where $tv = \mathsf{temp}[m_i = M_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J}$, $h = \mathsf{obj}[m_i = M_i; f_j = v_j]_{i\in I, j\in J}$, and:

ι	e	H'	Side Conditions
$\mathsf{let}\ t = tv\ \mathsf{in}\ v$	v	H	
t	tv	H	$E(t) = tv$
et	$\mathsf{temp}[;]$	H	
$tv + f{:}\sigma^{\phi}$	tv'	H	$f \notin f_{j\in J}; tv' =$ $\mathsf{temp}[m_i = M_i; f_j{:}\sigma_j^{\phi_j}, f{:}\sigma^{\phi}]_{i\in I, j\in J}$
$tv \leftarrow f_k{:}\sigma^{\phi}$	tv'	H	$tv' = \mathsf{temp}[m_i = M_i; f_j{:}{\sigma'}_j^{\phi'_j}]_{i\in I, j\in J}$ $k \in J; {\sigma'}_j^{\phi'_j} = \begin{cases} \sigma_j^{\phi_j} & j \neq k \\ \sigma^{\phi} & j = k \end{cases}$
$tv \mathrel{\leftarrow\!\!+} [m_i{=}M'_i]_{i\in K}$	tv'	H	$tv' = \mathsf{temp}[m_i{=}M''_i; f_j{:}{\sigma'}_j^{\phi'_j}]_{i\in I', j\in J}$ $I' = (I, K - I)$ $M''_i = \begin{cases} M_i & i \notin K \\ M'_i & i \in K \end{cases}$
$t[f_j = v_j]$	x	$H\{x = h\}$	$x \notin \mathrm{dom}(H); E(t) = tv$
$L.m_k[\vec{\tau}](\vec{v})$	$b\{\rho\}$	H	$H(L) = h; k \in I$ $M_k = x[\vec{\alpha}](\overrightarrow{x{:}\tau}){:}\tau.b$ $\rho = (\vec{\alpha}, x, \vec{x} := \vec{\tau}, L, \vec{v})$
$L.f_k$	v_k	H	$H(L) = h; k \in J$
$L.f_k \leftarrow v$	v	$H\{L = h'\}$	$H(L) = h; k \in J$ $h' = \mathsf{obj}[m_i = M_i; f_j = v'_j]_{i\in I, j\in J}$ $v'_j = \begin{cases} v_j & j \neq k \\ v & j = k \end{cases}$

Figure 4.2: O Operational Semantics

$$\boxed{\Delta \vdash^{\mathrm{O}} \tau \quad \Delta \vdash^{\mathrm{O}} r \quad \Delta \vdash^{\mathrm{O}} \tau_1 \leq \tau_2 \quad \Delta \vdash^{\mathrm{O}} r_1 \leq r_2 \quad \Delta \vdash^{\mathrm{O}} s_1 \leq s_2}$$

$$\frac{}{\Delta \vdash^{\mathrm{O}} \tau}\ (\mathrm{ftv}(\tau) \subseteq \Delta) \qquad \frac{}{\Delta \vdash^{\mathrm{O}} r}\ (\mathrm{ftv}(r) \subseteq \Delta)$$

$$\frac{}{\Delta \vdash^{\mathrm{O}} \alpha \leq \alpha}\ \alpha \in \Delta \qquad \frac{\Delta \vdash^{\mathrm{O}} r_1 \leq r_2}{\Delta \vdash^{\mathrm{O}} \mathsf{objt}\ r_1 \leq \mathsf{objt}\ r_2}$$

$$\frac{\Delta \vdash^{\mathrm{O}} r_1 \quad \Delta \vdash^{\mathrm{O}} s_i^1 \leq s_i^2 \quad \Delta \vdash^{\mathrm{O}} \sigma_j^{1\phi_j^1} \leq \sigma_j^{2\phi_j^2}}{\Delta \vdash^{\mathrm{O}} r_1 \leq r_2} \left(\begin{array}{c} r_k = [m_i{:}s_i^k; f_j{:}\sigma_j^{k\phi_j^k}]_{i \in I^k, j \in J^k} \\ I^1 \leq I^2 \\ J^1 \leq J^2 \end{array} \right)$$

$$\frac{\Delta, \vec{\alpha} \vdash^{\mathrm{O}} \sigma_i \leq \tau_i \quad \Delta, \vec{\alpha} \vdash^{\mathrm{O}} \tau \leq \sigma}{\Delta \vdash^{\mathrm{O}} [\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n) \to \tau \leq [\vec{\alpha}](x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n) \to \sigma}$$

Figure 4.3: O Typing Rules for Types

trivial subtyping, and object types have right-extension breadth subtyping, covariant depth subtyping for methods,[5] and depth subtyping for fields given by their variances.[6] Methods are covariant in their results and contravariant in their arguments.

At the term level there are judgements for typing expressions $\Delta; \Gamma; \Theta \vdash^{\mathrm{O}} e : \tau$, template expressions $\Delta; \Gamma; \Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te : T$, and method definitions $\Delta; \Gamma; \Theta \vdash^{\mathrm{O}}_{\mathsf{M}} M : \tau \rhd s$. The empty template has the empty template type. Field extension requires the template not to have the field and adds the field at the right. Field update requires the template to have the field, the new type and variance pair to be a subtype of the old pair, and updates the field. The rule for method extension and update is the most complicated. The old template has type $\mathsf{tempt}\ r$, and the new template has type $\mathsf{tempt}\ r'$ where r' reflects the modified methods's new signatures and the new methods. The new signatures for modified methods must be subsignatures of the old signatures. The new object type $\mathsf{objt}\ r'$ is used as the type for self in checking the new and modified method definitions.

Template instantiation requires a template of type $\mathsf{tempt}\ r$ and produces an object of type $\mathsf{objt}\ r$. It checks that each field is given an initial value of the appropriate type. Method invocation requires an object with the requested method, checks that the type arguments are well formed, checks that the arguments have the appropriate type, and produces a result according to the method's return type with the type arguments substituted for the type parameters. Field selection requires an object with the requested field and that the field be readable. Field update requires an object with the requested field and that the field be writable, and results in an object of the original type.

The typing rules are sound with respect to the operational semantics. The proof uses standard techniques—I have proven a similar language sound [Gle99a].

The language, although simple, can encode a number of higher-order constructs, such as the object constructors of Abadi and Cardelli and higher-order functions. These encodings are presented below as syntactic sugar for the language (variables that appear only on the right hand side are fresh):

$$\mathsf{obj}[m_i = M_i; f_j = e_j{:}\sigma_j^{\phi_j}]_{i \in I, j \in J} = \mathsf{let}\ t = \mathsf{et} +_{j \in J} f_j{:}\sigma_j^{\phi_j} \leftarrow\!\!+ [m_i = M_i]_{i \in I}\ \mathsf{in}\ t[f_j = e_j]_{j \in J}$$

---

[5] Covariant depth subtyping means that if $\tau_1 \leq \tau_2$ then $\mathsf{obj}[m{:}\tau_1;] \leq \mathsf{obj}[m{:}\tau_2]$.

[6] If $\tau_1 \leq \tau_2$ then $\mathsf{obj}[; f{:}\tau_1^+] \leq \mathsf{obj}[; f{:}\tau_2^+]$, $\mathsf{obj}[; f{:}\tau_2^-] \leq \mathsf{obj}[; f{:}\tau_1^-]$, and so on. See the rules for precise details.

$$\boxed{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e:\tau \quad \Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te:T \quad \Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{M}} M:\tau \triangleright s}$$

$$\Theta ::= t_1{:}T_1, \ldots, t_n{:}T_n$$

$$\frac{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e_1 : \tau_2 \quad \Delta \vdash^{\mathrm{O}} \tau_1 \leq \tau_2}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e_1 : \tau_2} \qquad \frac{}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} x : \tau}\ (\Gamma(x) = \tau)$$

$$\frac{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te : T \quad \Delta;\Gamma;\Theta, t{:}T \vdash^{\mathrm{O}} e : \tau}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} \mathsf{let}\ t = te\ \mathsf{in}\ e : \tau}$$

$$\frac{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} t : \mathsf{tempt}\ r \quad \Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e_j : \sigma_j}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} t[f_j = e_j]_{j\in J} : \mathsf{objt}\ r}\ (r = [m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J})$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e : \mathsf{objt}[m_i{:}s_i, m{:}[\alpha_1,\ldots,\alpha_m](\tau_1,\ldots,\tau_n) \to \tau;]_{i\in I} \\ \Delta \vdash^{\mathrm{O}} \sigma_i \quad \Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e_i : \tau_i\{\vec{\alpha} := \vec{\sigma}\}\end{array}}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e.m[\sigma_1,\ldots,\sigma_m](e_1,\ldots,e_n) : \tau\{\vec{\alpha} := \vec{\sigma}\}}$$

$$\frac{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e : \mathsf{objt}[; f_j{:}\sigma_j^{\phi_j}, f{:}\sigma^{\phi}]_{j\in J}}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e.f : \sigma}\ (\phi \leq +)$$

$$\frac{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e_1 : \mathsf{objt}[; f_j{:}\sigma_j^{\phi_j}, f{:}\sigma^{\phi}]_{j\in J} \quad \Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e_2 : \sigma}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}} e_1.f \leftarrow e_2 : \sigma}\ (\phi \leq -)$$

$$\frac{}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} t : T}\ (\Theta(t) = T) \qquad \frac{}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} \mathsf{et} : \mathsf{tempt}[;]}$$

$$\frac{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te : \mathsf{tempt}\ [m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J} \quad \Delta \vdash^{\mathrm{O}} \sigma}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te + f{:}\sigma^{\phi} : \mathsf{tempt}\ [m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}, f{:}\sigma^{\phi}]_{i\in I, j\in J}}\ (f \notin f_{j\in J})$$

$$\frac{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te : \mathsf{tempt}\ [m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J} \quad \Delta \vdash^{\mathrm{O}} \sigma}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te \leftarrow f_k{:}\sigma^{\phi} : \mathsf{tempt}\ [m_i{:}s_i; f_j{:}\sigma_j'^{\phi_j'}]_{i\in I, j\in J}}\ \left(k{\in}J; \sigma_j'^{\phi_j'} {=} \left\{\begin{array}{ll} \sigma_j^{\phi_j} & j \neq k \\ \sigma^{\phi} & j = k \end{array}\right.\right)$$

$$\frac{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te : \mathsf{tempt}\ [m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J} \quad \Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{M}} M_i : r' \triangleright s_i' \quad \Delta \vdash^{\mathrm{O}} s_i' \leq s_i}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{te}} te \leftarrow\!\!+[m_i = M_i]_{i\in K} : \mathsf{tempt}\ [m_i{:}s_i''; f_j{:}\sigma_j^{\phi_j}]_{i\in(I,K-I), j\in J}}$$

where $s_i'' = s_i$ for $i \in I - K$, and $s_k'' = s_k'$ for $i \in K$.

$$\frac{\Delta, \vec{\alpha}; \Gamma, x{:}\mathsf{objt}\ r, \overline{x{:}\vec{\tau}} \vdash^{\mathrm{O}} b : \tau}{\Delta;\Gamma;\Theta \vdash^{\mathrm{O}}_{\mathsf{M}} x[\vec{\alpha}](\overline{x{:}\vec{\tau}}){:}\tau.b : r \triangleright [\vec{\alpha}](\vec{\tau}) \to \tau}$$

Figure 4.4: O Typing Rules for Expressions

$$\begin{array}{lcl} \textsf{fix } f[\vec{\alpha}](\overline{x{:}\vec{\tau}}){:}\tau.b & = & \textsf{obj}[\textsf{apply} = f[\vec{\alpha}](\overline{x{:}\vec{\tau}}){:}\tau.b;\,] \\ \lambda x{:}\tau_1.b{:}\tau_2 & = & \textsf{fix } f[\,](x{:}\tau_1){:}\tau_2.b \\ e[\vec{\tau}](\vec{e}) & = & e.\textsf{apply}[\vec{\tau}](\vec{e}) \end{array}$$

4.2 Closure Conversion

O allows objects to have free variables that are bound by outer-nested objects. For example, consider the code:

$$\textsf{obj}[\textsf{apply} = s1[\,]().\textsf{obj}[\textsf{apply} = s2[\,]().s1.\textsf{apply}[\,]():\tau;\,]:\tau;\,]$$

Where $\tau = \textsf{objt}[\textsf{apply}{:}\textsf{obj}[;\,];\,]$. In the inner object the method body of apply has a free variable $s1$ that is bound by the outer object's apply method. The inner object could exist after the outer object's apply method has terminated, yet to execute the inner object's apply method the value of the variable $s1$ is needed. Somehow the compiler must transmit the value of $s1$ from the time the outer apply method executes to the time the inner apply method executes. To do this, compilers employ a translation called closure conversion, which converts code into closed code and auxiliary data structures. This process is particularly important in the implementation of functional programming languages, because functions with free variables are commonly used in this style. The process is also important given the recent addition of inner classes to Java.

In previous work [Gle99b, Gle99c], I present a direct object closure conversion, prove it correct, and relate closure to single method objects. I will now adapt this translation to the template language, thus also providing partial evidence that TAL supports the compilation of functional programming languages. Unfortunately, because of right extension subtyping, adapting my translation to the full template language is not possible without significant modifications. Therefore I will impose a restriction on the input to closure conversion: if a template has free variables in its methods, then field extension is no longer allowed. This restriction does not exclude either first-class functions or the equivalent of Java's inner classes, as both of these are final templates (in the sense of Java's final classes).

The idea behind object closure conversion is simple: as objects contain both code and data, closure conversion just adds extra fields to the objects to store the values of the free variables of the object's methods. For example, the object above is closure converted to:

$$\textsf{obj}[\textsf{apply} = s1[\,]().\textsf{obj}[\textsf{apply} = s2[\,]().s2.f.\textsf{apply}[\,]():\tau;\, f = s1]:\tau;\,]$$

The free variable $s1$ is now stored in an extra field f, and the body of the inner apply method refers to it by the field selection $s2.f$. The typing translation is the identity. While different objects of the same type will, in general, have different sets of free variables with different types, the extra fields can be hidden by subsumption. Also, method invocation, the analogue of function application, does not need translation. Functional closure conversion requires a type translation and does need to translate applications. For objects these issues arise in encoding objects into records and functions, the subject of the next chapter.

The closure-conversion translation is formalised in Figure 4.5. It uses the function $\mathrm{fv}(M)$ that returns the free variables of M annotated with their types. Determining the types of the free variables requires knowing the type context Γ used to type check M, so the translation is type directed. The translation uses a parameter ϕ to remember for each template variable in scope the extra fields that were added to the template and the variables that should be used to initialise them. This information is used at instantiation points to add additional field initialisers for the fields that store free variables. A template expression translates into a pair

$$\phi ::= t_1 \mapsto \overrightarrow{f_1 = x_1}, \ldots, t_n \mapsto \overrightarrow{f_n = x_n}$$

$$\begin{array}{lcl}
[\![x]\!]_{\mathsf{e}}(\phi) & = & x \\
[\![\mathsf{let}\ t = te\ \mathsf{in}\ e]\!]_{\mathsf{e}}(\phi) & = & \mathsf{let}\ t = te'\ \mathsf{in}\ [\![e]\!]_{\mathsf{e}}(\phi\{t \mapsto \overrightarrow{f = x}\}) \\
 & & \text{where } (te'; \overrightarrow{f = x}) = [\![te]\!]_{\mathsf{te}}(\phi) \\
[\![t[f_j = e_j]_{j \in J}]\!]_{\mathsf{e}}(\phi) & = & t[f_j = [\![e_j]\!]_{\mathsf{e}}(\phi), \phi(t)] \\
\llbracket e.m[\vec{\tau}](e_1, \ldots, e_n)\rrbracket_{\mathsf{e}}(\phi) & = & [\![e]\!]_{\mathsf{e}}(\phi).m[\vec{\tau}]([\![e_1]\!]_{\mathsf{e}}(\phi), \ldots, [\![e_n]\!]_{\mathsf{e}}(\phi)) \\
[\![e.f]\!]_{\mathsf{e}}(\phi) & = & [\![e]\!]_{\mathsf{e}}(\phi).f \\
[\![e_1.f \leftarrow e_2]\!]_{\mathsf{e}}(\phi) & = & [\![e_1]\!]_{\mathsf{e}}(\phi).f \leftarrow [\![e_2]\!]_{\mathsf{e}}(\phi)
\end{array}$$

$$\begin{array}{lcl}
[\![t]\!]_{\mathsf{te}}(\phi) & = & (t; \phi(t)) \\
[\![\mathsf{et}]\!]_{\mathsf{te}}(\phi) & = & (\mathsf{et}; \epsilon) \\
[\![te + f{:}\sigma^{\phi}]\!]_{\mathsf{te}}(\phi) & = & (te' + f{:}\sigma^{\phi}; \epsilon) \\
 & & \text{where } (te'; \epsilon) = [\![te]\!]_{\mathsf{te}}(\phi) \\
[\![te.f \leftarrow \sigma^{\phi}]\!]_{\mathsf{te}}(\phi) & = & (te'.f \leftarrow \sigma^{\phi}; \overrightarrow{f = x}) \\
 & & \text{where } (te'; \overrightarrow{f = x}) = [\![te]\!]_{\mathsf{te}}(\phi) \\
[\![te \leftarrow\!\!+ [m_i = M_i]_{i \in I}]\!]_{\mathsf{te}}(\phi) & = & (te' +_{1 \le i \le n} g_i{:}\sigma_i^{+} \leftarrow\!\!+ [m_i = M_i']_{i \in I}; \overrightarrow{f = x}, \overrightarrow{g = y})
\end{array}$$

$$\begin{array}{llcl}
\text{where} & (te'; \overrightarrow{f = x}) & = & [\![te]\!]_{\mathsf{te}}(\phi) \\
 & \cup_{i \in I} \mathrm{fv}(M_i) & = & y_1{:}\sigma_1, \ldots, y_n{:}\sigma_n \\
 & M_i & = & x_i[\vec{\alpha_i}](\overrightarrow{x_i{:}\tau_i}){:}\tau_i.b_i \\
 & M_i' & = & x_i[\vec{\alpha_i}](\overrightarrow{x_i{:}\tau_i}){:}\tau_i.[\![b_i]\!]_{\mathsf{e}}(\phi)\{\vec{y} := \overrightarrow{x_i.g}\} \\
 & g_1, \ldots, g_n \text{ are fresh} & &
\end{array}$$

Figure 4.5: Closure-Conversion Translation

which consists of the new template expression and the list of extra fields and their variables. The restriction of the input to the translation is reflected in the rule for field extension: it requires the template being extended to have no extra fields.

The translation produces closed code, preserves typing, and preserves meaning. I prove these properties for a similar pure object language [Gle99c].

Chapter 5

Object and Class Encoding

An important part of compiling object-oriented languages to target code is translating objects into more primitive constructs. For reasons explained later, such translations will not type check in TAL or will not be efficient. Efficient translations that type check require adding additional typing constructs to TAL. This chapter will devise these additional typing constructs and formulate a typed translation from O to TAL extended with these constructs. The essence of the problem is apparent in the translation of objects to records and functions. Since IIL has records and functions, and since Section 2.4 presented an IIL to TAL compiler, this chapter concentrates on translating O to IIL and on devising additional typing constructs for IIL. These additions are what MOOTAL needs to support object-oriented languages.

Over the last fifteen years, much work has formulated translations from languages with object primitives to variants of the typed lambda calculus; these translations are called *object encodings*. However, the motivations of these encodings were theoretical: they precisely specify the meaning of object constructs, and they are used to compare the expressiveness of various object constructs versus various lambda calculus constructs. Notably absent from these encodings is any search for an efficient one.

Another area of work has been on *class encodings*: how to encode class constructs into pure object constructs or a class-based object-calculus directly into a lambda calculus. Again, the concern is with precisely specifying the meaning of class constructs and whether classes provide any additional expressiveness. These encodings also do not consider efficiency.

There is a well known efficient method for implementing objects called the self-application semantics [Kam88]. In this semantics, an object is a data structure that contain functions for each of the methods the object responds to. Method invocation involves selecting the appropriate function and passing the object as an extra argument (hence self application). This semantics is easy to express as an untyped object encoding, but only recently have typed versions been formulated (see later in the chapter). Compilers for popular class-based object-oriented languages, such as Java, typically use a variant of the self-application semantics. For each class, a "vtable" is constructed that contains an entry for each method the class's instances respond to. This dissertation will call these "vtables" *method tables*. An object is a record with an entry for its method table and an entry for each field. Method invocation involves selecting the appropriate function from the method table and passing the object as an extra argument. Thus is requires two record projections and a function application.

This chapter presents a new class and object encoding into an extension of IIL. The encoding mirrors what compilers typically for Java, is efficient, and preserves typing, subtyping, and operational semantics. The key idea is to type self application with a self quantifier and devise the right formulation of self quantifiers. Abadi and Cardelli [AC96] present and discuss a self quantifier, but their formulation is unable to type the self-application semantics. I present a different set of rules for the self quantifier that is able to type the self-application semantics. I believe this lends further insight into the area of object encodings, as it lends a natural interpretation: the self variable is typed by self types, self types are modelled by a self quantifier, and several object encodings can be seen as interpretations of the self quantifier.

Before motivating and presenting the new encoding I first elaborate on previous work on object and class encodings.

5.1 Object Encodings

An *object encoding* is a translation from a language with a primitive notion of objects to one without, typically a language that includes records and functions as primitives. An adequate object encoding must preserve the meaning of programs. For typed translations it must also preserve both typing and subtyping. An object encoding should also be efficient and fully abstract. Another dimension for evaluating object encodings is the set of features that can be encoded. Bruce et al. [BCP97] provide an excellent comparison of most of the known object encodings.

The first typed object encoding was proposed by Cardelli [Car88a]. He encoded an object as a record that can recursively refer to itself (often called a recursive record interpretation). At the type level, he encoded an object type as the fix point of a record type whose elements are the methods's types. The encoding preserves meaning, typing, and subtyping, but it cannot encode method update. The recursive types interpretation was pursued by Reddy [Red88, KR94], Cook [Coo89, CHC90], the Hopkins Object Group [ESTZ95], and others.

Pierce and Turner [PT94] proposed a simple object and class encoding that requires only existentials and not recursive types. They split objects into a private *state* component and a public *method suite*. The functions that encode methods are passed the state but not the method suite. Furthermore, if a method's return type is the self type, then the function returns only the state component, and the method invocation sites must pair the returned state with the original method suite to construct the returned object. Their encoding is the only encoding with a nonuniform translation of method invocation. Their encoding preserves meaning, typing, and subtyping, but it cannot encode method update. The lack of method update did not concern them as they were considering class-based languages. Finally, methods can be made private and immutable fields can be made public, but mutable fields cannot be made public, as they are not passed to the methods's functions.

Bruce et al. [Bru94, BSvG95] designed a functional and an imperative class-based object-oriented language, and the denotational semantics for this language can be seen both as an object and class encoding. Like Pierce and Turner, they were concerned with class-based languages. Their encoding is very similar to Pierce and Turner's, but methods whose result type is the self type return the whole object, not just the state component. Thus, they need to wrap Pierce and Turner's translation of an object type with an extra fix point. Bruce et al. also argue for the use of matching rather than subtyping, which has many advantages but leads to a different object and class model than Cardelli or Pierce and Turner's.

Rémy [Rém94, RV97] uses a variant of Pierce and Turner's encoding with row variables. Row variables are used to specify polymorphism over the type of self and enable a natural extension of ML to include objects and classes without sacrificing type inference. However, this system does not include subsumption, and an object must be explicitly coerced from a subtype to a supertype.

Finally in 1996 Abadi, Cardelli, and Viswanathan [ACV96] discovered an adequate typed object encoding for objects with method invocation and method update. Their encoding uses bounded existentials and recursive types to effectively encode a self type. However, the technique they chose requires an additional projection and an additional field that are needed purely for typing purposes.

Abadi et al.'s encoding is also not fully abstract. In particular, the translation of method update allows the target language to distinguish objects that were indistinguishable in the source language. Viswanathan [Vis98] fixed this problem, but only by introducing considerably more computation.

Recently, Hickey, Crary and League et al. have proposed typed encodings of the self-application semantics. Hickey [Hic99] shows how to type the self-application semantics with the Nuprl type theory. He uses an intersection type to make methods polymorphic over the type of self. However, the Nuprl type theory is undecidable, so it is not clear how to use this encoding in a type-directed compiler. Crary [Cra99] shows how to use an unbounded existential and binary intersection types to type the self-application semantics. League et al. [LST99] show how to type the self-application semantics using existentially-quantified row variables and recursive types. They also show how to deal with classes, as described below. Both Crary and League et al.'s ideas could be seen as encodings of the self quantifier proposed here. Abadi, Cardelli, and Viswanathan's encoding could also been seen as an encoding of a self quantifier. But rather than reflecting the self quantifier proposed here, it reflects the self quantifier described by Abadi and Cardelli [AC96].

5.2 Class Encodings

Abadi and Cardelli [AC96] show informally how to encode classes into their pure object calculi. In their encoding, a class becomes an object with *premethods* for each of the instance's methods and a new method for instantiating the class. The new method copies the premethods into a newly created object. Subclasses copy the premethods of the superclass that they inherit and provide their own premethods for overridden and additional methods. F-bounded polymorphism is used to type the premethods. This encoding shows that classes add no expressiveness to a pure object calculus, but taken at face value is not as efficient as the method table approach used by most compilers, as it requires indirections or the unnecessary creation of closures.

Fisher and Mitchell along with other authors [Fis96, Mit90, FHM94, FM95a, FM95b, FM96, BF98, FM98] have pursued a line of research into encoding classes as extensible objects. The object calculi they consider have a method extension operation for adding a new method to an already existing object. This construct does not appear in the object calculi usually considered for object encodings. Method extension interacts poorly with breadth subtyping and so extensible object calculi need to have complicated type systems for tracking the absence of methods. Often a distinction is made between prototype objects, which are extensible but do not have breadth subtyping, and proper objects, which are not extensible but do have breadth subtyping. Like Abadi and Cardelli's class encoding, these encodings show that classes do not add expressiveness. They also provide a good basis for the design and definition of languages. However, also like Abadi and Cardelli's encoding, they are not efficient if taken at face value. In particular, class instantiation involves creating an empty object, then adding all its methods to the object.

Pierce and Turner's class encoding [PT94], unlike the above, encodes classes directly into records and functions and not into objects. Essentially it encodes a class as a function f that returns a record of functions for the public methods of the class. These latter functions take the private state component and return the result of method they correspond to. However, to dispatch to other public methods, these functions use f's argument, which is also a record of functions for the public methods of the class. Instantiating a class involves taking the fix point of f. However, subclasses may have more fields than superclasses, so f is parameterised by functions to convert between the final representation and the one the current class defines. Unfortunately these conversion functions persist beyond class instantiation time and in general are evaluated every time a method is invoked, making this encoding particularly inefficient.

Bruce et al.'s class encoding [Bru94, BSvG95] essentially encodes a class as a pair of the initial values of the private fields of the class and a function for the public methods. Additionally the pair is polymorphic in the final object type and the type of the private fields. The function for the public methods takes the final object and returns a record of the results of each method. Similar to Pierce and Turner, class instantiation requires taking the fix point of the function for the public methods to produce a function from the private state to the method suite, and then packaging this with the initial private state. Taken at face value, this encoding is also not efficient.

Recently, League et al. [LST99] showed how to encode a subset of the Java class model into a variant of F_ω [Gir71, Gir72]. A class is encoded as a method table (they call it a dictionary), a function to initialise the class's private fields, and a function to instantiate the class. Using a combination of row polymorphism and existential types, they are able to encode class private fields and their work can probably be extended to handle most of Java's protection modifiers. They also claim without proof that the encoding is fully abstract. Their paper focuses on Java and is somewhat complicated by some of Java's features. This dissertation attempts to be more generic and to flesh out the key ideas.

Reppy and Reicke [RR96a] show how to encode classes as modules in the SML module system extended with objects in the core language [RR96b]. Their encoding is essentially the same as Abadi and Cardelli's, but with some twists for handling protection. Vouillon [Vou98] shows how to combine the classes and modules of Objective ML [RV97] into a single construct. Essentially their modules have all the features of classes and objects that are necessary, so in a sense there is no encoding.

My class encoding uses the method-table technique described in the introduction to this chapter, encoding templates and objects into a language of records and functions. It does not require the indirection of Abadi and Cardelli nor the representation conversion functions of Pierce and Turner. It is similar to League et al. but is simpler and more generic, making the key ideas more apparent.

5.3 My Encoding

The purpose of this chapter is to present an encoding of the template language into an extension of Iil and then compile that extension into an extension of Tal. The encoding presented is efficient in that it uses the self-application semantics and method-table techniques described earlier and used by most compilers. Before getting into formal details, this section will discuss the intuition behind the encoding, which will motivate the constructs that are added to Iil and Tal. The following sections will present these extensions and the formal translation.

The main purpose of this section is to spell out the self-application semantics and method-table techniques and to show how to type the result. The main problem is assigning a type to self, so I will first devise types for (translations of) objects and method tables assuming the type for self is known, and then show how to get the type for self. An object will be translated into a record of type $[\![r]\!]_{\mathsf{rf}}(\tau)$[1] where τ is the (yet to be determined) type of self and r is the row describing the object. Part of the object is a method table shared with other instances of the template from which the object was created. A method table has type $[\![r]\!]_{\mathsf{rm}}(\tau)$ where τ is the type of self and r describes the object. Both $[\![\cdot]\!]_{\mathsf{rm}}(\cdot)$ and $[\![\cdot]\!]_{\mathsf{rf}}(\cdot)$ are used to build the translated types for objects and templates.

Under the self-application semantics, a method is compiled to a function taking an extra argument, and during method invocation the object itself is passed as the extra argument. For example, the method `handleEvent` of class `Window`, which has signature $[](\mathsf{event}) \rightarrow \mathsf{bool}$, is compiled to a function, named say `Window::handleEvent`, of the form:

$$\lambda(x{:}\alpha, y{:}\mathtt{Event}).b$$

where x is the extra self parameter, b is the body of the method, and α is, for now, a type variable that stands for the type of self. This function has type $(\alpha, \mathtt{Event}) \rightarrow \mathsf{bool}$.

In a class-based language, all instances of a class have the same methods. In order to save space, objects share a structure with other instances of the class, the method table. A method table is a record with one field for each method the object responds to. For example, the `Window` class has a method table, named say `Window::mtb`:

$$\langle \mathtt{handleEvent} = \mathtt{Window::handleEvent}, \mathtt{contains} = \mathtt{Window::contains} \rangle$$

and `ContainerWindow` would have a method table:

$$\begin{array}{l}\langle \mathtt{handleEvent} = \mathtt{ContainerWindow::handleEvent}, \\ \quad \mathtt{contains} = \mathtt{Window::contains}, \\ \quad \mathtt{addChild} = \mathtt{ContainerWindow::addChild} \rangle\end{array}$$

Using the suggested typing of `Window::handleEvent`, `Window`'s method table has type:

$$[\![\mathtt{Window}]\!]_{\mathsf{rm}}(\alpha) = \langle \mathtt{handleEvent}{:}(\alpha, \mathtt{Event}) \rightarrow \mathsf{bool}, \mathtt{contains}{:}(\alpha, \mathtt{Point}) \rightarrow \mathsf{bool} \rangle$$

The question is what to do with α. Abadi and Cardelli [AC96] observe that the methods in these method tables are polymorphic in the final object type, and so can be given an F-bounded polymorphic type (F bounds were introduced by Canning et al. [CCH$^+$89]). Using the $[\![r]\!]_{\mathsf{rf}}(\alpha)$ type, `Window`'s method table gets type:

$$\begin{array}{l}\langle \mathtt{handleEvent} : \forall \alpha \leq [\![\mathtt{Window}]\!]_{\mathsf{rf}}(\alpha).(\alpha, \mathtt{Event}) \rightarrow \mathsf{bool}, \\ \quad \mathtt{contains} : \forall \alpha \leq [\![\mathtt{Window}]\!]_{\mathsf{rf}}(\alpha).(\alpha, \mathtt{Point}) \rightarrow \mathsf{bool} \rangle\end{array}$$

I will use this idea with one twist. Instead of making the methods polymorphic, I will make the method table polymorphic. Thus `Window`'s method table has type:

$$\begin{array}{rl} & \forall \alpha \leq [\![\mathtt{Window}]\!]_{\mathsf{rf}}(\alpha).\langle \mathtt{handleEvent} : (\alpha, \mathtt{Event}) \rightarrow \mathsf{bool}, \mathtt{contains} : (\alpha, \mathtt{Point}) \rightarrow \mathsf{bool} \rangle \\ = & \forall \alpha \leq [\![\mathtt{Window}]\!]_{\mathsf{rf}}(\alpha).[\![\mathtt{Window}]\!]_{\mathsf{rm}}(\alpha)\end{array}$$

This means that a method table can be installed into an object simply by instantiating it at an appropriate type. In general the translation of a template type, written $[\![\mathsf{temp}\ r]\!]_{\mathsf{T}}$, is

[1] The subscripts rf and rm indicate full object type and the method table type, respectively.

$\forall\alpha \le [\![r]\!]_{\mathsf{rf}}(\alpha).[\![r]\!]_{\mathsf{rm}}(\alpha)$ and $[\![r]\!]_{\mathsf{rm}}(\alpha)$ is a record type with one entry for each method in r which is a function taking an α and the parameters of that method to the result of that method:

$$[\![[m_i{:}[\vec{\alpha}_i](\vec{\tau_i}) \to \tau_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in J}]\!]_{\mathsf{rm}}(\alpha) = \langle m_i{:}\forall[\vec{\alpha}_i](\alpha, \overrightarrow{[\![\tau_i]\!]_{\mathsf{t}}}) \to [\![\tau_i]\!]_{\mathsf{t}}\rangle_{i\in I}$$

For single-inheritance languages, the method tables can be ordered such that a subclass's method table is a subtype, under right-extension subtyping, of its superclass's method table. Ordered records with right-extension subtyping have a natural and efficient implementation, so most compilers for single-inheritance class-based languages use this technique.

An object is a record with an entry for its class's method table and an entry for each of its fields. For example, an instance of `Window` would have the form:

$$\langle \mathsf{mtb} = \mathtt{Window::}\mathsf{mtb}, \mathtt{extent} = r_1\rangle$$

and an instance of `ContainerWindow` would have the form

$$\langle \mathsf{mtb} = \mathtt{ContainerWindow::}\mathsf{mtb}, \mathtt{extent} = r_2, \mathtt{children} = a\rangle$$

where r_i is some rectangle and a some array of `Windows`.

The type of an instance of `Window` has the form

$$[\![\mathtt{Window}]\!]_{\mathsf{rf}}(\alpha) = \langle \mathsf{mtb} : [\![\mathtt{Window}]\!]_{\mathsf{rm}}(\alpha), \mathtt{extent} : \mathtt{Rectangle}\rangle \tag{5.1}$$

Again, the problem is what to do with α. Naively, this is the object type itself, so a recursive type should be used. Unfortunately this does not work. Instances of `ContainerWindow`, which also have type `Window`, have a self type equal to:

$$\langle \mathsf{mtb} : [\![\mathtt{ContainerWindow}]\!]_{\mathsf{rm}}(\alpha), \mathtt{extent} : \mathtt{Rectangle}, \mathtt{children} : \mathsf{array}(\mathtt{Window})\rangle$$

This type is a strict subtype of the type in 5.1. We need a way to make α refer to the actual type of the object. My solution is to use a self quantifier. The type $\mathsf{self}\ \alpha.\tau$ contains values v of type τ where α is the actual (or principle) type of v. Thus `Window`'s instances have type:

$$\mathsf{self}\ \alpha.\langle \mathsf{mtb} : [\![\mathtt{Window}]\!]_{\mathsf{rm}}(\alpha), \mathtt{extent} : \mathtt{Rectangle}\rangle$$

In general $[\![\mathsf{obj}\ r]\!]_{\mathsf{t}} = \mathsf{self}\ \alpha.[\![r]\!]_{\mathsf{rf}}(\alpha)$ and $[\![r]\!]_{\mathsf{rf}}(\alpha)$ is a record type with an entry for a method table and an entry for each field of r:

$$[\![r]\!]_{\mathsf{rf}}(\alpha) = \langle \mathsf{mtb}{:}[\![r]\!]_{\mathsf{rm}}(\alpha), f_j{:}[\![\sigma_j]\!]_{\mathsf{t}}^{\phi_j}\rangle_{j\in J} \quad \text{where} \quad r = [m_i{:}\alpha.\tau_i; f_j{:}\sigma_j^{\phi_j}]_{i\in I, j\in I}$$

Again, for single-inheritance languages, the fields can be ordered such that a subclass's fields are a right-extension subtype of its superclass's fields.

The only remaining problem is how to formalise self quantifiers. Abadi and Cardelli [AC96] provide a formulation of self quantifiers, but their formulation leads to the need for "recoup" fields and the inefficiencies of an extra field and extra projection. We need a new formulation of self quantifiers that avoids the problems of recoup fields. I use $\varsigma\alpha.\tau$ to refer to their self types, and $\mathsf{self}\ \alpha.\tau$ to refer to my formulation.

Abadi and Cardelli's formulation involves two operations, one to introduce self quantifiers, and one to eliminate them. The introduction form (they call "wrap") is $\mathsf{pack}\ e, \sigma\ \mathsf{as}\ \varsigma\alpha.\tau$, and it produces an expression e packaged up with its actual self type σ. The typing rule is:

$$\frac{\Delta \vdash^{\mathrm{IIL}} \sigma \le \tau\{\alpha := \sigma\} \quad \Delta; B; \Gamma \vdash^{\mathrm{IIL}} e : \sigma}{\Delta; B; \Gamma \vdash^{\mathrm{IIL}} \mathsf{pack}\ e, \sigma\ \mathsf{as}\ \varsigma\alpha.\tau : \varsigma\alpha.\tau}$$

$$\frac{}{\Delta; B \vdash^{\text{IIL}} \alpha \leq \alpha} (\alpha \in \Delta) \qquad \frac{}{\Delta; B \vdash^{\text{IIL}} \mathsf{int} \leq \mathsf{int}} \qquad \frac{}{\Delta; B \vdash^{\text{IIL}} \mathsf{exn} \leq \mathsf{exn}}$$

$$\frac{\Delta; B \vdash^{\text{IIL}} \tau_{2i} \leq \tau_{1i} \quad \Delta; B \vdash^{\text{IIL}} \tau_1 \leq \tau_2}{\Delta; B \vdash^{\text{IIL}} (\tau_{11}, \ldots, \tau_{1n}) \to \tau_1 \leq (\tau_{21}, \ldots, \tau_{2n}) \to \tau_2}$$

$$\frac{\Delta \vdash^{\text{IIL}} \langle \ell_i{:}\tau_{1i}^{\phi_{1i}} \rangle_{i \in I_1} \quad \Delta \vdash^{\text{IIL}} \tau_{1i}^{\phi_{1i}} \leq \tau_{2i}^{\phi_{2i}}}{\Delta; B \vdash^{\text{IIL}} \langle \ell_i{:}\tau_{1i}^{\phi_{1i}} \rangle_{i \in I_1} \leq \langle \ell_i{:}\tau_{2i}^{\phi_{2i}} \rangle_{i \in I_2}} (I_1 \leq I_2)$$

$$\frac{\Delta, \alpha; B \vdash^{\text{IIL}} \tau_1 \leq \tau_2}{\Delta; B \vdash^{\text{IIL}} \forall \alpha.\tau_1 \leq \forall \alpha.\tau_2} \qquad \frac{\Delta, \alpha; B \vdash^{\text{IIL}} \tau_{21} \leq \tau_{11} \quad \Delta, \alpha; B \vdash^{\text{IIL}} \tau_{12} \leq \tau_{22}}{\Delta; B \vdash^{\text{IIL}} \forall \alpha \leq \tau_{11}.\tau_{12} \leq \forall \alpha \leq \tau_{21}.\tau_{22}}$$

$$\frac{}{\Delta; B \vdash^{\text{IIL}} \mathsf{rec}\ \alpha.\tau \leq \mathsf{rec}\ \alpha.\tau} \qquad \frac{\Delta, \alpha; B \vdash^{\text{IIL}} \tau_1 \leq \tau_2}{\Delta; B \vdash^{\text{IIL}} \mathsf{self}\ \alpha.\tau_1 \leq \mathsf{self}\ \alpha.\tau_2}$$

$$\frac{\Delta; B \vdash^{\text{IIL}} \tau_1 \leq \tau_2}{\Delta; B \vdash^{\text{IIL}} \tau_1^{\phi} \leq \tau_2^{+}} (\phi \leq +) \qquad \frac{\Delta; B \vdash^{\text{IIL}} \tau_2 \leq \tau_1}{\Delta; B \vdash^{\text{IIL}} \tau_1^{\phi} \leq \tau_2^{-}} (\phi \leq -) \qquad \frac{}{\Delta; B \vdash^{\text{IIL}} \tau^{\circ} \leq \tau^{\circ}}$$

Figure 5.1: IIL Subtyping Rules

For σ to actually be e's self type, e must have type σ. In addition e also must have type τ with α replaced by e's self type, that is, e must have type $\tau\{\alpha := \sigma\}$. The latter is achieved by requiring that $\sigma \leq \tau\{\alpha := \sigma\}$. The elimination form (they call "use as") is $\mathsf{unpack}\ \alpha, x = e_1 \mathsf{in} e_2$. Intuitively, the expression e_1 is a value packaged with its self type, and unpack unpacks the value into x and the self type into α, then executes e_2. Abadi and Cardelli's typing rule is:

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e_1 : \varsigma\alpha.\tau_1 \quad \Delta, \alpha; B, \alpha \leq \varsigma\alpha.\tau_1; \Gamma, x : \tau_1 \vdash^{\text{IIL}} e_2 : \tau_2 \quad \Delta \vdash^{\text{IIL}} \tau_2}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Recall that if e_1 evaluates to the packaged value $\mathsf{pack}\ v, \sigma\ \mathsf{as}\ \varsigma\alpha.\tau_1$ then v has type σ, but the above rule gives x type τ_1 a supertype of σ. Thus, to get a value of type σ (which α is bound to), τ_1 must have a field of type α called a recoup field. However, this is unnecessary since x really has type σ, and the above rule could be modified to reflect this by giving x type α. This observation leads to my rule for self-type elimination:

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e_1 : \mathsf{self}\ \alpha.\tau_1 \quad \Delta, \alpha; B, \alpha \leq \tau_1; \Gamma, x : \alpha \vdash^{\text{IIL}} e_2 : \tau_2 \quad \Delta \vdash^{\text{IIL}} \tau_2}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Note, however, that the bound $\alpha \leq \tau_1$ has α on both the left and the right sides; that is, it is a recursive or F bound. Since, the system uses F bounds for typing method tables, using them in the unpack rule adds no additional complexity. In fact, this brings a nice symmetry to the system, as F bounds are used in the typing of premethods and F bounds are used in the typing of method invocation.

5.4 Encoding Target

According to the previous section, to encode O into IIL, we need to add an F-bounded form of quantification, recursive types, and self quantifiers to IIL. In addition IIL must have subtyping and subsumption to reflect the subtyping and subsumption of O. The subtyping rules for

$$\begin{array}{llll} \text{Values} & v & ::= & \cdots \mid \mathsf{roll}^{\tau}(v) \mid \mathsf{pack}^{\tau}(v) \\ \text{Contexts} & E & ::= & \cdots \mid \mathsf{roll}^{\tau}(E) \mid \mathsf{unroll}(E) \mid \mathsf{pack}^{\tau}(E) \mid \mathsf{unpack}\ \alpha, x = E\ \mathsf{in}\ b \end{array}$$

ι	e	H'	S'	Side Conditions
$\mathsf{unroll}(\mathsf{roll}^{\tau}(v))$	v	H	S	
$\mathsf{unpack}\ \alpha, x = v\ \mathsf{in}\ b$	$b\{\alpha := \tau'\}$	H	$S\{x = \mathsf{roll}^{\tau'}(v')\}$	$v = \mathsf{pack}^{\tau}(\mathsf{roll}^{\tau'}(v'))$

Figure 5.2: Extended IIL Operational Semantics

IIL appear in Figure 5.1 and are standard except as noted below. The syntax for the other extensions is:

$$\begin{array}{lll} t & ::= & \cdots \mid \alpha \leq \tau \\ \tau & ::= & \cdots \mid \mathsf{rec}\ \alpha.\tau \mid \mathsf{self}\ \alpha.\tau \\ e & ::= & \cdots \mid \mathsf{roll}^{\tau}(e) \mid \mathsf{unroll}(e) \mid \mathsf{pack}^{\tau}(e) \mid \mathsf{unpack}\ \alpha, x = e_1\ \mathsf{in}\ e_2 \end{array}$$

A type definition of the form $\alpha \leq \tau$ is called an F bound, as α is bound in τ. These F bounds and the recursive types, like the type names of MTAL, are mediated by explicit roll ($\mathsf{roll}^{\tau}(e)$) and unroll ($\mathsf{unroll}(e)$) coercions. Thus, α is not a subtype of τ but unrolls to τ. Similarly, $\mathsf{rec}\ \alpha.\tau$ unrolls to $\tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}$, and the latter rolls to the former.

The type $\mathsf{self}\ \alpha.\tau$ contains values packaged with their self type by the $\mathsf{pack}^{\mathsf{self}\ \alpha.\tau}$ coercion. The value before packing has type τ where α is replaced by the self type of the value. In general these self types are recursive, so the typing rule for pack allows only recursive self types, and requires the unrolled form to be a subtype of τ with α replaced by the rolled form. Consequently, a value of type $\mathsf{self}\ \alpha.\tau$ has the form $\mathsf{pack}^{\mathsf{self}\ \alpha.\tau}(\mathsf{roll}^{\sigma}(v))$ where σ is the self type. The operational rule for unpack uses this fact to substitute the self type for α in e_2. Otherwise, the self quantifiers, pack coercion, and unpack expression follow the intuition and rules of the previous section.

The extensions to the operational semantics for IIL appear in Figure 5.2. The additional typing rules, including subsumption, appear in Figure 5.3.

5.5 Translation

The translation, which is type directed, appears in Figures 5.4 and 5.5. As with many other translations in this dissertation, rather than present it as a function of typing derivations, it is presented as a function of syntax with $: \tau$ annotations to indicate the necessary typing information. The actual typing derivation used to translate a term affects only the typing information of the translated term, and the term structure is solely determined by the term structure of the source expression. Further, the proofs of type preservation and operational correctness show that no matter which typing derivation is used, the translated term has the translated type and simulates the behaviour of the source term, thus providing a coherence argument (see [Gle99a] for details).

The translation uses the ideas developed in Section 5.3. For a row r there are two important target types: $[\![r]\!]_{\mathsf{rm}}(\tau)$ for method tables and $[\![r]\!]_{\mathsf{rf}}(\tau)$ for the objects. The record type of a method table is $[\![r]\!]_{\mathsf{rm}}(\tau)$ where τ is the type of self. The record type of an object is $[\![r]\!]_{\mathsf{rf}}(\tau)$ where τ is the type of self. As discussed, a template type is polymorphic over self, so is

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e : \tau_1 \quad \Delta; B \vdash^{\text{IIL}} \tau_1 \leq \tau_2}{\Delta; B; \Gamma \vdash^{\text{IIL}} e : \tau_2}$$

τ	$\mathrm{unroll}_B(\tau)$	$\mathrm{roll}(\tau)$
α	$B(\alpha)$	no
$\mathsf{rec}\ \alpha.\sigma$	$\sigma\{\alpha := \tau\}$	yes

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e : \tau_2}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{roll}^{\tau_1}(e) : \tau_1}\ (\mathrm{unroll}_B(\tau_1) = \tau_2; \mathrm{roll}(\tau))$$

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e : \tau_1}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{unroll}(e) : \tau_2}\ (\mathrm{unroll}_B(\tau_1) = \tau_2)$$

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e : \tau_1 \quad \Delta; B \vdash^{\text{IIL}} \tau_2 \leq \tau\{\alpha := \tau_1\}}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{pack}^{\mathsf{self}\ \alpha.\tau}(e) : \mathsf{self}\ \alpha.\tau}\ (\mathrm{unroll}_B(\tau_1) = \tau_2)$$

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e_1 : \mathsf{self}\ \alpha.\tau_1 \quad \Delta, \alpha; B, \alpha \leq \tau_1; \Gamma, x{:}\alpha \vdash^{\text{IIL}} e_2 : \tau_2 \quad \Delta \vdash^{\text{IIL}} \tau_2}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{unpack}\ \ \alpha, x = e_1\ \ \mathsf{in}\ \ e_2 : \tau_2}$$

Figure 5.3: Extended IIL Typing Rules

$$\begin{array}{lcl}
[\![\alpha]\!]_{\mathsf{t}} & = & \alpha \\
[\![\mathsf{obj}\ r]\!]_{\mathsf{t}} & = & \mathsf{self}\ \alpha.[\![r]\!]_{\mathsf{rf}}(\alpha) \\
[\![\mathsf{temp}\ r]\!]_{\mathsf{T}} & = & \forall \alpha \leq [\![r]\!]_{\mathsf{rf}}(\alpha).[\![r]\!]_{\mathsf{rm}}(\alpha) \\
[\![[m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]]\!]_{\mathsf{rm}}(\tau) & = & \langle m_i{:}[\![s_i]\!]_{\mathsf{s}}(\tau)^+\rangle_{i\in I} \\
[\![[m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]]\!]_{\mathsf{rf}}(\tau) & = & \langle \mathsf{mtb}{:}[\![[m_i{:}s_i; f_j{:}\sigma_j^{\phi_j}]]\!]_{\mathsf{rm}}(\tau)^+, f_j{:}[\![\sigma_j]\!]_{\mathsf{t}}^{\phi_j}\rangle_{j\in J} \\
\llbracket [\alpha_1, \ldots, \alpha_m](\tau_1, \ldots, \tau_m) \to \tau \rrbracket_{\mathsf{s}}(\sigma) & = & \forall \alpha_1. \ldots \forall \alpha_m.(\sigma, [\![\tau_1]\!]_{\mathsf{t}}, \ldots, [\![\tau_n]\!]_{\mathsf{t}}) \to [\![\tau]\!]_{\mathsf{t}}
\end{array}$$

Figure 5.4: Object and Class Encoding, Types

$$
\begin{array}{lcl}
mt & ::= & \langle\overrightarrow{\ell = v}\rangle \\
\varphi & ::= & t_1 \mapsto (x_1, mt_1), \ldots, t_n \mapsto (x_n, mt_n)
\end{array}
$$

$$
\begin{array}{lcl}
[\![x]\!]_{\mathsf{e}}(\varphi) & = & x \\
[\![\mathsf{let}\ t = te{:}\mathsf{temp}\ r\ \mathsf{in}\ e]\!]_{\mathsf{e}}(\varphi) & = & \mathsf{let}\ x = (\Lambda[\vec{\alpha}, \alpha \leq [\![r]\!]_{\mathsf{rf}}(\alpha)]mt)[\vec{\alpha}]\ \mathsf{in} \\
 & & [\![e]\!]_{\mathsf{e}}\varphi\{t \mapsto (x, mt)\} \\
 & & \text{where}\ \ x, \alpha \text{ fresh} \\
 & & \qquad\qquad [\![te]\!]_{\mathsf{te}}(\varphi, \alpha) \ = \ mt \\
 & & \qquad\qquad \vec{\alpha} \ = \ \mathrm{ftv}(mt) - \{\alpha\} \\
[\![t{:}\mathsf{temp}\ r[f_j = e_j]_{j\in J}]\!]_{\mathsf{e}}(\varphi) & = & \mathsf{pack}^{[\![\mathsf{obj}\ r]\!]_{\mathsf{t}}}(\mathsf{roll}^{\mathsf{rec}\ \alpha.[\![r]\!]_{\mathsf{rf}}(\alpha)}(e)) \\
 & & \text{where}\ \ e \ = \ \langle \mathsf{mtb} = mt, f_j = [\![e_j]\!]_{\mathsf{e}}(\varphi)\rangle_{j\in J} \\
 & & \qquad\qquad mt \ = \ x[\mathsf{rec}\ \alpha.[\![r]\!]_{\mathsf{rf}}(\alpha)] \\
 & & \qquad\qquad \varphi(t) \ = \ (x, \_) \\
[\![e.m[\tau_1, \ldots, \tau_p] & = & \mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathsf{e}}(\varphi)\ \mathsf{in} \\
\quad (e_1, \ldots, e_n)]\!]_{\mathsf{e}}(\varphi) & & \mathsf{unroll}(x).\mathsf{mtb}.m[[\![\tau_1]\!]_{\mathsf{t}}]\cdots[[\![\tau_p]\!]_{\mathsf{t}}] \\
 & & \quad (x, [\![e_1]\!]_{\mathsf{e}}(\varphi), \ldots, [\![e_n]\!]_{\mathsf{e}}(\varphi)) \\
[\![e.f]\!]_{\mathsf{e}}(\varphi) & = & \mathsf{unpack}\ \alpha, x = [\![e]\!]_{\mathsf{e}}(\varphi)\ \mathsf{in}\ \mathsf{unroll}(x).f \\
[\![e_1.f \leftarrow e_2]\!]_{\mathsf{e}}(\varphi) & = & \mathsf{unpack}\ \alpha, x = [\![e_1]\!]_{\mathsf{e}}(\varphi)\ \mathsf{in} \\
 & & \mathsf{unroll}(x).f \leftarrow [\![e_2]\!]_{\mathsf{e}}(\varphi)
\end{array}
$$

$$
\begin{array}{lcl}
[\![t]\!]_{\mathsf{te}}(\varphi, \tau) & = & mt \text{ where } \varphi(t) = (x, mt) \\
[\![\mathsf{et}]\!]_{\mathsf{te}}(\varphi, \tau) & = & \langle\rangle \\
[\![te + f{:}\sigma^{\phi}]\!]_{\mathsf{te}}(\varphi, \tau) & = & [\![te]\!]_{\mathsf{te}}(\varphi, \tau) \\
[\![te.f \leftarrow \sigma^{\phi}]\!]_{\mathsf{te}}(\varphi, \tau) & = & [\![te]\!]_{\mathsf{te}}(\varphi, \tau) \\
[\![te \leftrightarrow [m_i = M_i]_{i\in K} & = & \langle m_i = v_i''\rangle_{i\in(I, K-I)} \\
\ {:}\mathsf{temp}\ r]\!]_{\mathsf{te}}(\varphi, \tau) & & \text{where}\ \ [\![te]\!]_{\mathsf{te}}(\varphi, \tau) \ = \ \langle m_i = v_i\rangle_{i\in I} \\
 & & \qquad\qquad v_i'' \ = \ \begin{cases} v_i & i \in I - K \\ v_i' & i \in K \end{cases} \\
 & & \qquad\qquad [\![M_i]\!]_{\mathsf{m}}(\varphi, \tau, r) \ = \ v_i' \\
[\![M]\!]_{\mathsf{m}}(\varphi, \sigma, r) & = & (\mathsf{fix}\ \_[\vec{\beta}, \alpha \leq [\![r]\!]_{\mathsf{rf}}(\alpha), \vec{\alpha}] \\
 & & \quad (x'{:}\alpha, x_1{:}[\![\tau_1]\!]_{\mathsf{t}}, \ldots, x_n{:}[\![\tau_n]\!]_{\mathsf{t}}). \\
 & & \ \mathsf{let}\ x = \mathsf{pack}^{[\![\mathsf{obj}\ r]\!]_{\mathsf{t}}}(x')\ \mathsf{in}\ [\![b]\!]_{\mathsf{e}}(\varphi))[\vec{\beta}, \sigma] \\
 & & \text{where}\ \ M \ = \ x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n){:}\tau.b \\
 & & \qquad\qquad \vec{\beta} \ = \ \mathrm{ftv}(M)
\end{array}
$$

Figure 5.5: Object and Class Encoding, Terms

translated to $\forall\alpha \leq [\![r]\!]_{\mathsf{rf}}(\alpha).[\![r]\!]_{\mathsf{rm}}(\alpha, \circ)$, and an object type uses a self quantifier, so is translated to $\mathsf{self}\ \alpha.[\![r]\!]_{\mathsf{rf}}(\alpha)$.

At the term level, a template is translated into a record corresponding to its method table. The translation of let template expressions binds value variables to these records, which are polymorphic in the type of self. Thus the translation needs to remember for each template variable both the value variable bound to its translation and the target record it is translated to. An environment φ records this information. A complication arises because IIL polymorphic records and functions must be type-variable and value-variable closed. The translation assumes object closure conversion has been applied, but in addition it needs to close over the free type variables of polymorphic records and functions. The ideas of Morrisett et al. [MWCG98] are used. Briefly, v is closed by transforming it to $(\forall[\vec{\alpha}]v)[\vec{\alpha}]$ where $\vec{\alpha} = \mathrm{ftv}(v)$.[2] An expression e's translation is $[\![e]\!]_{\mathsf{e}}(\varphi)$; a template expression te's translation is $[\![\varphi, \tau]\!]_{\mathsf{te}}te$, where τ is the target type of self; a method body M's translation is $[\![M]\!]_{\mathsf{m}}(\varphi, \tau, r)$, where τ is the target type of self and r describes the objects in which M appears.

The translation is both type preserving and operationally correct. I prove correct a similar object encoding from an object language with covariant self types and no type or value parameters into a similar language without explicit roll and unroll [Gle99a].

5.6 MOOTAL and Extended Compiler

To compile the extended IIL to TAL, I first extend TAL to a language called MOOTAL. The extensions in MOOTAL mirror the extensions made to IIL, and are needed to order to still be able to type check compiled IIL code. First, MOOTAL has subtyping as follows: Code types are contravariant, much like function types are contravariant in their argument types. Tuple types have right-extension breadth subtyping, and depth subtyping given by their field variances. The heap and stack pointer types are covariant. Second, MOOTAL has recursively-bounded type definitions, recursive types, self types, and the associated coercions. Finally, MOOTAL has an unpack instruction $\mathsf{unpack}\ \alpha, r, v$, which unpacks the value v, places it in r, and binds α to the self type for the remaining instructions. These additional features of MOOTAL parallel those of IIL. The technical details, including operational semantics and typing rules, appear in the appendix, but are summarised here:

$$
\begin{array}{rcl}
t & ::= & \cdots \mid \alpha{:}\kappa \leq c \\
c & ::= & \cdots \mid \mathsf{rec}\ \alpha{:}\kappa.c \mid \mathsf{self}\ \alpha{:}\kappa.c \\
\delta & ::= & \cdots \mid \mathsf{roll}^c \mid \mathsf{unroll} \mid \mathsf{pack}^c \\
\iota & ::= & \cdots \mid \mathsf{unpack}\ \ \alpha, r, v
\end{array}
$$

The extended translation for the compiler of Section 2.4 appears in Figure 5.6.

5.7 Extensions

This dissertation considers only single-inheritance languages. A natural extension would be to consider multiple inheritance or the related notion of mixins.[3] The essential change would be the addition of operations to combine two or more templates into a new template. The exact semantics of this operation with respect to conflicts between the templates would need to reflect

[2] Recall that $\mathrm{ftv}(v)$ calculates the free type variables of v.

[3] Flatt et al. [FKF98] present an excellent mixin design, and they have references to earlier work.

$$
\begin{aligned}
[\![\alpha \leq \tau]\!]_{\mathsf{td}} &= \alpha{:}\mathsf{T} \leq [\![\tau]\!]_{\mathsf{t}} \\
[\![\mathsf{rec}\ \alpha.\tau]\!]_{\mathsf{t}} &= \mathsf{rec}\ \alpha{:}\mathsf{T}.[\![\tau]\!]_{\mathsf{t}} \\
[\![\mathsf{self}\ \alpha.\tau]\!]_{\mathsf{t}} &= \mathsf{self}\ \alpha{:}\mathsf{T}.[\![\tau]\!]_{\mathsf{t}}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{roll}^{\tau}(v)]\!]_{\mathsf{v}}(k) &= [\![v]\!]_{\mathsf{v}}(\lambda w.\mathsf{roll}^{[\![\tau]\!]_{\mathsf{t}}}(w)) \\
[\![\mathsf{pack}^{\tau}(v)]\!]_{\mathsf{v}}(k) &= [\![v]\!]_{\mathsf{v}}(\lambda w.\mathsf{pack}^{[\![\tau]\!]_{\mathsf{t}}}(w))
\end{aligned}
$$

$\mathsf{roll}^{\tau}(e)$	
$[\![e]\!]_{\mathsf{e}}(\Delta, vm, \tau_a, \tau_b, h);$	
$\mathsf{mov\ \ r1}, \mathsf{roll}^{[\![\tau]\!]_{\mathsf{t}}}(\mathsf{r1})$	
$\mathsf{unroll}(e)$	
$[\![e]\!]_{\mathsf{e}}(\Delta, vm, \tau_a, \tau_b, h);$	
$\mathsf{mov\ \ r1}, \mathsf{unroll}(\mathsf{r1})$	
$\mathsf{pack}^{\tau}(e)$	
$[\![e]\!]_{\mathsf{e}}(\Delta, vm, \tau_a, \tau_b, h);$	
$\mathsf{mov\ \ r1}, \mathsf{pack}^{[\![\tau]\!]_{\mathsf{t}}}(\mathsf{r1})$	
$\mathsf{unpack}\ \ \alpha, x = e_1\ \ \mathsf{in}\ \ e_2$	
$[\![e_1]\!]_{\mathsf{e}}(\Delta, vm, \tau_a, \tau_b, h);$	$\tau_1 = \mathsf{self}\ \alpha.\tau_1'$
$\mathsf{unpack}\ \ \alpha, \mathsf{r1}, \mathsf{r1};$	$\Delta' = \Delta, \alpha{:}\mathsf{T} \leq [\![\tau_1']\!]_{\mathsf{t}}$
$\mathsf{push\ \ r1};$	$vm' = vm, x \mapsto -h-1$
$[\![e_2]\!]_{\mathsf{e}}(\Delta', vm', \alpha :: \tau_a, \tau_b, h+1)$	

Figure 5.6: Object Extended IIL to MOOTAL Compiler

the intended semantics of multiple inheritance with respect to conflicts between superclasses. How to translate classes into templates then depends upon how to handle diamond inheritance. If D inherits from both B and C, and B and C both inherit from A, then D could have one or two "copies" of A. If the desired semantics is two copies, then each class would translate into a template that is the combination of its direct superclasses's templates modified by the declarations in the class itself. If the desired semantics is one copy, then each class would translate into both a direct and full template. The direct template would contain just the declarations of the class itself. The full template would be the combination of all the class's ancestors's direct templates. An object would be created by instantiating the appropriate class's full template.

Another property of object and class encodings not discussed in this chapter is full abstraction. In a secure extensible system, the security monitor coder would be using a high-level language, for our purposes an object-oriented language. She will think in terms of the abstractions of that language, in particular, the abstraction of an object. She will consider all the operations that the untrusted code could do to an object and ensure that none of these operations will violate the security policy. However, if the secure extensible system is checking the untrusted code at the level of IIL or TAL, then there may be operations that are possible on the object that were not possible in the high-level language. Full abstraction is the absence of such operations. Formally, if two O expressions are contextually equivalent, then their translations are are contextually equivalent in IIL. The Abadi, Cardelli, and Viswanathan encoding [ACV96] is not fully abstract. Viswanathan [Vis98] has proposed another object encoding that is fully abstract, but his encoding is very inefficient. I conjecture that the encoding presented in this chapter is fully abstract and intend to try to prove this in the future.

Chapter 6

Type Tagging

This chapter describes the final extension to TAL that provides support for the compilation of run-type type dispatch. Run-type type dispatch is important to real object-oriented languages. An example is Java's downcasting operator [GJS96]. The expression `(`c`)`e[1] tests if e's run-time class is a subclass of c, and if not throws an exception. Java also has a class case construct, but only for examining the class of an exception packet. The try statement `try` *blk* `catch (`*classname*$_1$ x_1`)` *blk*$_1$ $\cdots$ `catch (`*classname*$_n$ x_n`)` *blk*$_n$ first executes *blk*, and if *blk* throws an exception, matches that exception's run-time class against *classname*$_1$ through *classname*$_n$. If *classname*$_i$ is the first matching class, x_i is bound to the exception, and *blk*$_i$ is executed. The ability to examine run-time classes is crucial to Java's exception mechanism, and is generally useful in a number of other situations.

In typed languages, *type refinement* is a key property of downcasting: After dispatching on the run-time class of a value, that value's static type changes to reflect the new type information. For example, in Java, if `Student` is a subclass of `Person` and x is declared to have type `Person`, then in the expression `(Student)x`, while x has static type `Person` the whole expression has the refined type `Student`.

Downcasting is one example of a construct that dispatches on a value's run-time type. The literature contains work addressing a number of similar constructs. Abadi et al. [ACPP91] introduce a type `dynamic` with a *type case* construct, formalise its semantics and typing rules, and prove soundness. The intermediate language λ_i^{ML} [HM95, Mor95] treats another type-case construct, also formalising it and proving soundness. Crary, Weirich, and Morrisett [CWM98, CW99] show how to formalise λ_i^{ML}'s type case in a type-erasure interpretation rather than the type-passing interpretation used in other works. Their languages λ^R and LXvcase do not have term constructs that dispatch on types, but instead their languages have typing machinery for checking the implementation of type dispatch. Thus, in some sense they address the implementation of type dispatch, unlike the other works mentioned.

This work on type dynamic and type case does not extend to object oriented languages as it neglects two important features: subtypes and the creation of "new" (*i.e.*, generative) types. For these features, the only previous work is that of Reppy and Riecke [RR96b] and Harper and Stone [HS97]. Reppy and Riecke describe an extension of SML with objects. Included in their language are hierarchically organised object type constructors and a case mechanism for dispatching on an object's run-time class. They formalise this construct and prove it sound. They sketch an implementation, but do not formalise the implementation. Harper and Stone give an alternative semantics for SML. To model exception declarations,

[1] Java also has an `instanceof` operator that tests if a cast would succeed.

they include an extensible sum construct called tags. This chapter also has a tag construct very similar to theirs, but my tags can deal with class hierarchies, whereas their can only deal with top-level classes.

This chapter reworks the ideas of Reppy and Reicke with a more general presentation, and extends it to include a translation of the tag construct into an extended version of MooTal. As Reppy and Riecke observe, class case is very similar to exception matching and to pattern matching of hierarchical extensible sums. Thus, my construct explains class cast and class constructs, ML-style exceptions, and hierarchical extensible sums, and could be used in a typed compilation of multidispatch languages such as Cecil [Cha97].

I begin the chapter by describing in detail the programming language constructs being addressed. From these constructs I extract a core mechanism and add it to Iil, forming the source language of this chapter. Next, I informally discuss the implementation and typing issues that arise. This leads to another extension of Iil, the target language of this chapter, and a formal translation from the source language to the target language. Finally, I extend MooTal and the Iil to Tal compiler of Section 2.4.

6.1 Four Type Dispatch Constructs

Consider the following four language constructs:

Class Casting and Class Case: In Java, and in other class-based languages, objects are created by instantiating a class, and a reference to that class is stored in the object when it is created. Java has a downcasting operation $(c)e$ that evaluates e to an object and then tests to see if that object's class is in the subhierarchy under class c. If so, the cast expression evaluates to the object, but has static type c, which is generally a refinement of e's type. If not, an exception is thrown. More generally, these languages might provide a class case mechanism for testing membership in one of several classes. Java has this operation for the particular case of handling exceptions.

Exceptions: At first glance, ML-style exceptions might not seem related to downcasting, but in fact, there is a strong connection. Exception declarations are similar to classes in that they create a new exception name with an associated type. Exception packets, like objects, are created from an exception name, and that name is stored in the packet. Exception matching, then, is like downcasting: Known exception names are compared against the name in an exception packet, and successful comparisons allow access to the carried value at the type of the known exception name. Unlike classes, which are arranged hierarchically, ML-style exception names are not hierarchical. On the other hand, Java implements exception packets by using objects, and the declaration of new exception names is achieved by subclassing `throwable`.

Hierarchical Extensible Sums: ML-style exceptions are also an example of extensible sums. The exception type is like a global sum type that can be extended by user declarations. Each user-declared exception name is a new branch in the sum. A hierarchical extensible sum allows the sum branches to be arranged in a hierarchy. For example, a programmer might define a hierarchical extensible sum type for the primitives of a compiler intermediate language. She might define a constructor of this sum, `intbin`, for binary integer operators, and then subconstructors under `intbin` for the addition operation, the subtraction operation, and so on. The intermediate language's type checker could match against `intbin`, since all these operators have the same typing rule, whereas a code generator would match against the more specific constructors to determine the correct instruction to generate. Reppy and Riecke [RR96b] describe hierarchical extensible sums in connection with their class-case mechanism that generalises ML-style exceptions. Reppy and Fisher are incorporating a form of hierarchical extensible sums in the language Moby [FR99], a research vehicle for ML2000.

Multimethods: Java has single dispatch: Methods can be thought of as functions that are specialised on their first argument's class. Multimethods (*e.g.*, Cecil [Cha97]) are a generalisation of this paradigm: A multimethod is a function that is specialised on any, possibly all, of its arguments' classes. Implementing multimethods requires calling specialised code after determining which specialisation applies. The latter could be implemented by comparing the arguments' run-time classes against patterns of known classes. In a type-directed compilation framework, when one of these comparisons succeeds, the types of the arguments must be refined to match the types required by the specialised code. These comparisons are instances of the class case construct described above. Multimethods are similar to Castagna et al.'s overloaded functions [CGL95], except the latter are considered in a structural rather than named typed system.

The core mechanism in all of these examples is a *tagging* mechanism. Exception names, classes, and the constructors of an extensible sum are all examples of *tags* that are placed with or within values. Associated with these tags are types that correspond to the tagged values. The language has a *tag if/case* construct with type refinement in the successful branches. Furthermore, in the case of classes and hierarchical sums, the tags form a *tag hierarchy* and the associated types are in a subtype hierarchy parallel to the tag hierarchy. Usually, the tests of a tag case are not "is tag t_1 equal to tag t_2" but "is tag t_1 in the subhierarchy under tag t_2". I shall call "testing if a tag is under another in the tag hierarchy" a *tag check*. In the next section, I will add this core mechanism to IIL.

6.2 Translation Source

This section adds a tagging mechanism to IIL that abstracts the core operation of the type dispatch constructs described in the previous section. The desired operations are: creating hierarchies of type tags, tagging a value with a tag, and comparing the tag of a tagged value against known tags.

A new tag is created by one of two operations: $\mathsf{newtag}(\tau)$ or $\mathsf{subtag}(\tau, e)$. In both cases the new tag is for tagging values of type τ and has type $\mathsf{tag}(\tau)$. The $\mathsf{newtag}(\cdot)$ form creates a top level tag, and the $\mathsf{subtag}(\cdot, e)$ form creates a subtag of tag e. For example, the ML-style exception declaration $\mathsf{exception}\ \mathit{Failure}\ \mathsf{of\ string}$ could be coded in the tagging language as:

$$\mathsf{let}\ \ \mathit{Failure} = \mathsf{newtag}(\mathsf{string})\ \ \mathsf{in}$$

For expository purposes, examples will use constructs not in IIL, such as strings and floating point numbers. For an example of subtags, assume $\mathsf{string}[10]$ is the type of strings of length 10, and is a subtype of string. The subexception declaration $\mathsf{exception}\ \mathit{MyFailure}\ \mathsf{extends}\ \mathit{Failure}\ \mathsf{of\ string}[10]$ could be coded as:

$$\mathsf{let}\ \ \mathit{MyFailure} = \mathsf{subtag}(\mathsf{string}[10], \mathit{Failure})\ \ \mathsf{in}$$

Values are tagged with the operation $\mathsf{mktagged}(e_1, e_2)$ where e_1 is the tag and e_2 the value to be tagged. The result is a value of type tagged. For example, exception packet creation

$\mathsf{let}\ ep = \mathit{Failure}$ "unimplemented" could be coded as:

$$\mathsf{let}\ ep = \mathsf{mktagged}(\mathit{Failure}, \text{"unimplemented"})\ \mathsf{in}$$

Tagged values are compared with known tags using the operation $\mathsf{if}\ \mathsf{tag}(e_1) \leq e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ where e_1 is a tagged value and e_2 is a tag. Informally, the tag in e_1 is extracted and compared, along with all its ancestors in the tag hierarchy, to e_2. If any ancestor is equal to e_2, b_1 is executed with x bound to the value in e_1. Otherwise b_2 is executed. For example, exception matching such as:

```
match ep with
  Failure(x) -> printf "Computation failed: %s" x
| _ -> printf "Some other exception"
```

could be coded as:

```
if tag(ep) ≤ Failure then
  x.printf "Computation failed: %s" x
else
  printf "Some other exception"
fi
```

Adding these operations to IIL, the extended syntax is:

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \mathsf{tag}(\tau) \mid \mathsf{tagged} \\ e & ::= & \cdots \mid \mathsf{newtag}(\tau) \mid \mathsf{subtag}(\tau, e) \mid \mathsf{mktagged}(e_1, e_2) \mid \\ & & \mathsf{if}\ \mathsf{tag}(e_1) \leq e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \end{array}$$

The operational semantics is given in Figure 6.1. The key part of the semantics is modelling the identity of tags. Intuitively, the heap of an IIL program state stores the identities and details of all tags created in the execution so far. I add to heap values tag definitions $\mathsf{tag}(\tau, s)$, which consist of the type being tagged τ and the optional supertag s, which is either ϵ for no supertag or a variable that is the identity of the supertag.

The most interesting reduction rules are the rules for $\mathsf{if}\ \mathsf{tag}(\cdot) \leq \cdot\ \mathsf{then} \cdot \mathsf{else} \cdot \mathsf{fi}$. A tag check of x_1 against x_2 is formalised by the predicate $\mathit{tagchk}^H(x_1, x_2)$, where H is the heap containing the tag definitions, x_1 is the address of the unknown tag, and x_2 is the address of the known tag. The definition of this predicate says that either x_1 and x_2 are the same tag or x_1 has a supertag x_3 and the predicate $\mathit{tagchk}^H(x_3, x_2)$ holds. (Technically, $\mathit{tagchk}^{\cdot}(\cdot,\cdot)$ is the least predicate that is a fix point of an appropriate recursive definition.)

The source language is *erasable* in that the type τ in $\mathsf{newtag}(\tau)$, $\mathsf{subtag}(\tau, e)$, $\mathsf{tag}(\tau, \epsilon)$, and $\mathsf{tag}(\tau, v)$ is not needed at run-time.

The typing rules appear in Figure 6.2. Subtyping for tag and tagged types is trivial. The rule for subtags requires that e be a tag for type τ' and that τ be a subtype of τ'. The latter requirement ensures that types associated with tags form a subtype hierarchy in parallel to the tag hierarchy. The rule for $\mathsf{mktagged}(e_1, e_2)$ requires that e_1 be a type tag for τ and e_2 have type τ. The rule for $\mathsf{if}\ \mathsf{tag}(e_1) \leq e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ requires e_1 to have type tagged, e_2 to be a tag for some type σ, b_1 to type check in a context with x of type σ, and b_2 to type check.

The typing rules are sound with respect to the operational semantics. The proof uses the standard techniques. The only interesting case is in the type preservation of a successful tag comparison. In that case, a tag of type $\mathsf{tag}(\sigma_1)$ is compared against one of type $\mathsf{tag}(\sigma_2)$. If the comparison succeeds, the next program state has a store of the form $S\{x := v'\}$ where S has the desired type if x has type σ_2. However, v' has type σ_1, so we need to show that $\Delta \vdash^{\text{IIL}} \sigma_1 \leq \sigma_2$, which follows from this lemma:

Extended IIL syntax:

$$
\begin{array}{lcl}
v & ::= & \cdots \mid \mathsf{mktagged}(v_1, v_2) \\
E & ::= & \cdots \mid \mathsf{subtag}(\tau, E) \mid \mathsf{mktagged}(E, e) \mid \mathsf{mktagged}(v, E) \mid \\
 & & \mathsf{if}\ \ \mathsf{tag}(E) \leq e\ \ \mathsf{then}\ \ x.b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi} \mid \mathsf{if}\ \ \mathsf{tag}(v) \leq E\ \ \mathsf{then}\ \ x.b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi} \\
h & ::= & \cdots \mid \mathsf{tag}(\tau, \epsilon) \mid \mathsf{tag}(\tau, x)
\end{array}
$$

Reduction rules:

ι	e	H'	Side Conditions
$\mathsf{newtag}(\tau)$	x	$H\{x = \mathsf{tag}(\tau, \epsilon)\}$	$x \notin \mathrm{dom}(H)$
$\mathsf{subtag}(\tau, y)$	x	$H\{x = \mathsf{tag}(\tau, y)\}$	$x \notin \mathrm{dom}(H)$
$\mathsf{if}\ \ \mathsf{tag}(v) \leq x_2\ \ \mathsf{then}\ \ x.b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi}$	b_1	H	$x \notin \mathrm{dom}(S)$
			$S' = S\{x = v'\}$
			$v = \mathsf{mktagged}(x_1, v')$
			$\mathit{tagchk}^H(x_1, x_2)$
$\mathsf{if}\ \ \mathsf{tag}(v) \leq x_2\ \ \mathsf{then}\ \ x.b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi}$	b_2	H	$v = \mathsf{mktagged}(x_1, v')$
			not $\mathit{tagchk}^H(x_1, x_2)$

Tag checking:

$$\mathit{tagchk}^H(x_1, x_2) \stackrel{\text{def}}{=} (x_1 = x_2) \vee (H(x) = \mathsf{tag}(\tau, x_1') \wedge \mathit{tagchk}^H(x_1', x_2))$$

Figure 6.1: Tagging Source Operational Semantics

$$\frac{}{\Delta \vdash^{\text{IIL}} \mathsf{tag}(\tau) \leq \mathsf{tag}(\tau)} \qquad \frac{}{\Delta \vdash^{\text{IIL}} \mathsf{tagged} \leq \mathsf{tagged}}$$

$$\frac{\Delta \vdash^{\text{IIL}} \tau}{\Delta; \Gamma \vdash^{\text{IIL}} \mathsf{newtag}(\tau) : \mathsf{tag}(\tau)} \qquad \frac{\Delta; \Gamma \vdash^{\text{IIL}} e : \mathsf{tag}(\tau') \quad \Delta \vdash^{\text{IIL}} \tau \leq \tau'}{\Delta; \Gamma \vdash^{\text{IIL}} \mathsf{subtag}(\tau, e) : \mathsf{tag}(\tau)}$$

$$\frac{\Delta; \Gamma \vdash^{\text{IIL}} e_1 : \mathsf{tag}(\tau) \quad \Delta; \Gamma \vdash^{\text{IIL}} e_2 : \tau}{\Delta; \Gamma \vdash^{\text{IIL}} \mathsf{mktagged}(e_1, e_2) : \mathsf{tagged}}$$

$$\frac{\Delta; \Gamma \vdash^{\text{IIL}} e_1 : \mathsf{tagged} \quad \Delta; \Gamma \vdash^{\text{IIL}} e_2 : \mathsf{tag}(\sigma) \quad \Delta; \Gamma, x : \sigma \vdash^{\text{IIL}} b_1 : \tau \quad \Delta; \Gamma \vdash^{\text{IIL}} b_2 : \tau}{\Delta; \Gamma \vdash^{\text{IIL}} \mathsf{if}\ \ \mathsf{tag}(e_1) \leq e_2\ \ \mathsf{then}\ \ x.b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi} : \tau}$$

Figure 6.2: Tagging Source Typing Rules

Lemma 6.1 *If*

$$\begin{gathered} tagchk^{H}(x_1, x_2), \\ \vdash^{\text{IIL}} H : \Gamma, \\ \epsilon; \Gamma \vdash^{\text{IIL}} x_1 : \mathsf{tag}(\sigma_1),\ \textit{and} \\ \epsilon; \Gamma \vdash^{\text{IIL}} x_2 : \mathsf{tag}(\sigma_2) \end{gathered}$$

then $\Delta \vdash^{\text{IIL}} \sigma_1 \leq \sigma_2$.

6.3 Implementation

Real machines do not have the primitives $\mathsf{newtag}(\tau)$, $\mathsf{subtag}(\tau, e)$, $\mathsf{mktagged}(e_1, e_2)$, and $\mathsf{if}\ \mathsf{tag}(e_1) \leq e_2\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$. Compiler writers must select data structures to represent the tags and algorithms to implement tag checks. The goal of this section is to formalise a typed translation that eliminates these primitives. For now, think of the target of this translation as IIL with physical pointer equality and some typing machinery that I will develop in this section. This typing machinery is general enough to type strategies for implementing the tag primitives other than the one presented here [Gle99d].

Consider first how a compiler would translate the examples in the previous section, ignoring types. To create the new tag *Failure*, the compiler would dynamically allocate a new block of memory. Throughout its lifetime, the address of the block is different from the address of any other dynamically allocated memory block, so the compiler can use this address as a unique identifier for the tag. The compiler needs to record the position of *Failure* in the tag hierarchy, so it stores a null pointer into the newly allocated block to indicate that *Failure* is at the top level of the hierarchy. I will use ML's none constructor to represent the null pointer, and ML's $\mathsf{some}(v)$ to represent a non-null pointer to v. Similarly, to create *MyFailure* the compiler would allocate a new block of memory and store $\mathsf{some}(\mathit{Failure})$ in it.

To create the tagged value *ep*, the compiler would create a pair consisting of *Failure* and the literal string. So the first three examples might become:

$$\begin{array}{l} \mathsf{let}\ \mathit{Failure} = \langle \mathsf{none} \rangle\ \mathsf{in} \\ \mathsf{let}\ \mathit{MyFailure} = \langle \mathsf{some}(\mathit{Failure}) \rangle\ \mathsf{in} \\ \mathsf{let}\ \mathit{ep} = \langle \mathit{Failure}, \text{“unimplemented”} \rangle\ \mathsf{in} \end{array}$$

To translate the last example, the compiler would extract the tag in *ep*, which is the address of some dynamic memory block, and compare it against *Failure*. If they are equal, x would be bound to the second component of *ep* and b_1 executed. Otherwise, the supertag would be extracted and the process repeated until there is no supertag, in which case b_2 would be executed. The translated code might be:

$$\begin{array}{l} \mathsf{let}\ z = \mathit{ep}.1\ \mathsf{in} \\ \mathit{loop}1 : \mathsf{if}\ z = \mathit{Failure}\ \mathsf{then}\ \mathsf{let}\ x = \mathit{ep}.2\ \mathsf{in}\ b_1 \\ \qquad\quad \mathsf{else}\ \mathsf{match}\ z.1\ \mathsf{with}\ \mathsf{none} \rightarrow b_2 \\ \qquad\qquad\quad \mid \mathsf{some}(z') \rightarrow (z \leftarrow z'; \mathsf{goto}\ \mathit{loop}1) \end{array}$$

Now consider designing a type system to annotate the code above. The key difficulty is giving x the correct type. In general, the type of a tagged value like *ep* is unknown, yet if the comparison $z = \mathit{Failure}$ succeeds, the type of $\mathit{ep}.2$ is string, and this fact is needed to give x type string. What makes this difficult is that z and *Failure* are values unrelated to $\mathit{ep}.2$. In order to make this connection clear, the target type system needs to do two things: First, it needs to generate type equalities from physical pointer comparisons; second, it needs to link z and *ep* together, so that type information generated by the comparison will change *ep*'s type.

A solution to the second problem is to use type variables to link tags to values being tagged. For example, *ep*'s type could be $\langle tag(\alpha), \alpha\rangle$ for a yet to be determined type constructor $tag(\cdot)$ and some type variable α. Then comparing z, which has type $tag(\alpha)$, to *Failure*, which has type $tag(\mathsf{string})$, will cause the type checker to change α to string in the successful branch, thus changing $ep.2$'s type to string also.

A solution to the first problem lies in the following observation. The compiler is using the address of the memory block allocated for *Failure* as a name for the type string. It never uses the same address as a name for two different types, so if two addresses are equal, the types they name must be equal. To reflect this behaviour, the target type system must track which addresses are names for types and which types they name. The compiler gives the target language type $\mathsf{tag}(\tau_{ds}, \tau_t)$ to these pointers. These types resemble the source language type $\mathsf{tag}(\tau_t)$, as both are for tags for type τ_t. However, whereas tags were primitives in the source language, they are explicit datastructures in the target language, and τ_{ds} reflects the type of these datastructures. In particular, a value v is in the type $\mathsf{tag}(\tau_{ds}, \tau_t)$ if v also has type τ_{ds} and the programmer declared v to be a tag for type τ_t.

Now consider the datastructures used, and the type τ_{ds} for the examples. *Failure* is a linked list of *Failure*'s ancestors, and each pointer in this linked list is being used as a name for the type string. Linked lists have type $\mathsf{rec}\ \beta.\langle\beta?\rangle$ (where $\tau?$ is an option type). So *Failure* has type $tag(\mathsf{string})$, where:

$$tag(\tau) = \mathsf{rec}\ \beta.\mathsf{tag}(\langle\beta?\rangle, \tau)$$

The tagged value *ep* is a pair of such a tag and a string, except that string is abstracted over, thus *ep* has type $\exists\alpha.\langle tag(\alpha), \alpha\rangle$. To get the initial value of z, *ep* is unpacked, introducing α into the type context and giving z the type $tag(\alpha)$. If $z = Failure$ succeeds, then the type that z tags and the type that *Failure* tags must be the same, that is, α is string. The target type system will use this in type checking $\mathsf{let}\ x = ep.2\ \mathsf{in}\ b_1$. Since $ep.2$ has type α and α is string, x will get type the correct type.

Two complications arise with this basic scheme. The first is ensuring that a pointer is used to name only one type. If the same pointer is used to name two different types, run-time type errors could occur. To see this, assume that there is an operation $\mathsf{mktag}(e, \tau)$ that declares that value e is a tag for τ, and consider the following malicious code:

$$\begin{array}{l}
\mathsf{let}\ x_1 = \langle\mathsf{none}\rangle\ \mathsf{in} \\
\mathsf{let}\ x_2 = \mathsf{mktag}(x_1, \mathsf{string})\ \mathsf{in} \\
\mathsf{let}\ x_3 = \mathsf{mktag}(x_1, \mathsf{float})\ \mathsf{in} \\
\mathsf{let}\ y = \langle x_2, \text{“hello”}\rangle\ \mathsf{in}
\end{array}$$

The variables x_1, x_2, and x_3 are all bound to the same pointer, which points to a tuple with a single element none. However, the type system types x_2 as a tag for strings and x_3 as a tag for floats. The code uses x_2 to created a tagged “hello” value, which is bound to y. Now consider the following innocent code:

$$\begin{array}{l}
\mathsf{fun}\ foo[\alpha](z : \langle\mathsf{rec}\ \beta.\mathsf{tag}(\alpha, \langle\beta?\rangle), \alpha\rangle) = \\
\quad \mathsf{if}\ z.1 = x_3\ \mathsf{then}\ sin(z.2)\ \mathsf{else}\ 1.0\ \mathsf{fi}
\end{array}$$

The body of *foo* compares $z.1$, a tag for α, to x_3, a tag for float. In the then branch, $z.2$ is refined to type float and the sine computation type checks. However, suppose *foo* was applied to string and y. Since $y.1$ is x_2, which equals x_3, the *then* branch is executed. But $z.2$ is a string and the sine computation fails. The target type system must ensure that x_1 can be declared a tag for at most one type.

One way to ensure a value is declared a tag for at most one type is to use a linear type system. If v is of linear type τ^1, then v can be "used" only once. Then it is sufficient for $\mathsf{mktag}(e, \tau)$ to require $e : \sigma^1$ for some σ. However, this requires all the machinery of linear type systems in the target language. A simpler solution, pursued in this dissertation, is to allow $\mathsf{mktag}(e, \tau)$ only at points where new heap values are created. For example, $\langle\overrightarrow{\ell = e}\rangle$ creates a new heap value; the target operation $\mathsf{mktag}(\langle\overrightarrow{\ell = e}\rangle, \tau)$ does the same thing but gives the result type $\mathsf{tag}(\langle\overrightarrow{\ell{:}\tau}\rangle, \tau)$ where $\vec{e} : \vec{\tau}$.

The other complication concerns the interaction between subtyping and tag types. In particular, if $\mathsf{tag}(\ldots, \tau_1) \leq \mathsf{tag}(\ldots, \tau_2)$, then what is the relationship between τ_1 and τ_2? (The first position is covariant, *i.e.*, $\tau_1 \leq \tau_2$ implies $\mathsf{tag}(\tau_1, \tau) \leq \mathsf{tag}(\tau_2, \tau)$.) As we shall see, different and conflicting relationships are required by the process of creating subtags and the process of extracting from tagged values. The solution is to use variances to state the relationship that holds.

First some terminology. The value $\mathsf{mktag}(e, \tau)$ is said to have been *created* as a tag for τ. For example, *Failure* was created as a tag for string.

One requirement on the relationship arises from the creation of subtags. For example, using the scheme above, the translated code for the subtag *MyFailure* is:

$$\mathsf{let}\ \ \mathit{MyFailure} = \mathsf{mktag}(\langle\mathsf{some}(\mathit{Failure})\rangle, \mathsf{string}[10])\ \ \mathsf{in}$$

This code type checks as follows: *MyFailure* should have the type $\mathit{tag}(\mathsf{string}[10])$, and the right hand side has this type if *Failure* has type $\mathit{tag}(\mathsf{string}[10])$. In fact, *Failure* has type $\mathit{tag}(\mathsf{string})$, so we require that $\mathit{tag}(\mathsf{string}) \leq \mathit{tag}(\mathsf{string}[10])$, that is:

$$\mathsf{rec}\ \beta.\mathsf{tag}(\langle\beta?\rangle, \mathsf{string}) \leq \mathsf{rec}\ \beta.\mathsf{tag}(\langle\beta?\rangle, \mathsf{string}[10])$$

This would hold if $\mathsf{tag}(\ldots, \mathsf{string}) \leq \mathsf{tag}(\ldots, \mathsf{string}[10])$, in other words, if the subtyping rule were contravariant: if $\tau_1 \leq \tau_2$ then $\mathsf{tag}(\ldots, \tau_2) \leq \mathsf{tag}(\ldots, \tau_1)$.

However, now consider tagged value destruction and the code from above:

$$\begin{array}{l} z : \mathsf{rec}\ \beta.\mathsf{tag}(\alpha, \langle\beta?\rangle), \qquad ep : \langle\mathsf{rec}\ \beta.\mathsf{tag}(\alpha, \langle\beta?\rangle), \alpha\rangle \\ \mathsf{if}\ \ z = \mathit{Failure}\ \ \mathsf{then}\ \ \mathsf{let}\ \ x = ep.2\ \ \mathsf{in}\ \ b_1\ \ \mathsf{else}\ \ \ldots\ \ \mathsf{fi} \end{array}$$

Under the contravariant rule, *Failure* is a tag for type $\mathsf{string}[10]$, so the type system could type check $\mathsf{let}\ \ x = ep.2\ \ \mathsf{in}\ \ b_1$ under the assumption that α is $\mathsf{string}[10]$. Under this incorrect assumption, x has type $\mathsf{string}[10]$, but $ep.2$, which evaluates to "unimplemented", is actually a thirteen character string. Thus for destruction, subtyping must not be contravariant.

I use a variances to track the subtyping rules used. A tag type has the form $\mathsf{tag}(\tau_{ds}, \tau_t^{\phi})$, and a value is in this type if it has type τ_{ds} and was created to tag type σ. Furthermore, the variance states the relationship between τ_t and σ. For covariance, τ_t is a supertype of σ, for contravariance, τ_t is a subtype of σ, and for invariance, τ_t is σ. Using this new form, we can revise the type for z to $\mathsf{tag}(\ldots, \alpha^-)$, and the type for *Failure* to $\mathsf{tag}(\ldots, \mathsf{string}^+)$. If z equals *Failure*, we know that the types these tags were created for are equal. If σ is this type, we further know that $\alpha \leq \sigma$, since z has the contravariant tag type, and that $\sigma \leq \mathsf{string}$, as *Failure* has the covariant tag type. So $\alpha \leq \mathsf{string}$ and it safe to assume $x : \mathsf{string}$ in b_1.

The key is the relationship between the static tag type and the run-time tag type. In creating tagged values and subtags, we want the static tag type to be a subtype of the run-time

Extended syntax:

$$
\begin{array}{rcl}
v & ::= & \cdots \mid \mathsf{none}^{\tau} \mid \mathsf{some}(v) \\
E & ::= & \cdots \mid \mathsf{mktag}(\langle \overrightarrow{\ell = v}, \ell = E, \overrightarrow{\ell' = e} \rangle, \tau) \mid \mathsf{if}\ E = e\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \\
 & & \mathsf{if}\ v = E\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \mathsf{some}(E) \mid \mathsf{if?}\ E\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \\
h & ::= & \cdots \mid \mathsf{mktag}(\langle \overrightarrow{\ell = v} \rangle, \tau)
\end{array}
$$

Reduction rules:

ι	e	H'	S'	Side Conditions
$\mathsf{if}\ L = L\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_1	H	S	
$\mathsf{if}\ L_1 = L_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_2	H	S	$L_1 \neq L_2$
$\mathsf{if?}\ \mathsf{none}^{\tau}\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_2	H	S	
$\mathsf{if?}\ \mathsf{some}(v)\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$	b_1	H	$S\{x = v\}$	$x \notin \mathrm{dom}(S)$

Figure 6.3: Tagging Target Operational Semantics

tag type, and for extracting from tagged values, we want the run-time tag type to be a subtype of the static type, so that the type system conservatively refines types. The variance mechanism tracks and ensures the correct relationships.

Using these ideas, I present the target language in the next section and a translation in the following section.

6.4 Translation Target

The target of the translation is IIL, but instead of the extensions in Section 6.2, it has the following extensions:

$$
\begin{array}{rcl}
\tau & ::= & \cdots \mid \mathsf{tag}(\tau_1, \tau_2^{\phi}) \mid \tau? \\
e & ::= & \cdots \mid \mathsf{mktag}(\langle \ell_i = e_i \rangle_{i \in I}, \tau) \mid \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \mid \\
 & & \mathsf{none}^{\tau} \mid \mathsf{some}(e) \mid \mathsf{if?}\ e\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}
\end{array}
$$

The operation $\langle \overrightarrow{\ell = e} \rangle$ has been replaced by $\mathsf{mktag}(\langle \overrightarrow{\ell = e} \rangle, \tau)$, which creates a new tuple in the heap that can be used as a tag for the type τ; it has type $\mathsf{tag}(\langle \overrightarrow{\ell{:}\tau} \rangle, \tau^{\circ})$ where $\vec{e} : \vec{\tau}$. The type $\mathsf{tag}(\tau_1, \tau_2^{\phi})$ contains values of type τ_1 that are used as tags for the type τ_2. The value in this type may have been created as a tag for a subtype of τ_2 if ϕ is $+$, a supertype of τ_2 if ϕ is $-$, but only τ_2 if ϕ is $\circ$. Two values that are used to tag types can be compared for physical pointer equality using the operation $\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$. This operation is asymmetric as it is intended to compare a tag for an unknown type e_1 with a tag for a known type e_2. If the two values are equal b_1, is executed and e_2's tag type is used to refine e_1's; otherwise b_2 is executed. An option type $\tau?$ is either the value none^{τ} or the value $\mathsf{some}(v)$ for some $v{:}\tau$; the operation $\mathsf{if?}\ e_1\ \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$ can be used to discriminate between the two.

The operational semantics appears in Figure 6.3. The heap value $\langle \overrightarrow{\ell = v} \rangle$ has been replaced by the heap value $\mathsf{mktag}(\langle \overrightarrow{\ell = v} \rangle, \tau)$. Like in the source language, the heap is used to remember identities, particularly the identities of the tuples created. Two tuples, or pointers, are equal if they have the same address, that is, if they are the same variable. This leads to the rules for the if construct.

$$\frac{\Delta; B \vdash^{\text{IIL}} \tau_{11} \leq \tau_{21} \quad \Delta; B \vdash^{\text{IIL}} \tau_{12}^{\phi_1} \leq \tau_{22}^{\phi_2}}{\Delta; B \vdash^{\text{IIL}} \mathsf{tag}(\tau_{11}, \tau_{12}^{\phi_1}) \leq \mathsf{tag}(\tau_{21}, \tau_{22}^{\phi_2})} \qquad \frac{\Delta; B \vdash^{\text{IIL}} \tau_1 \leq \tau_2}{\Delta; B \vdash^{\text{IIL}} \tau_1? \leq \tau_2?}$$

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e_i : \tau_i \quad \Delta \vdash^{\text{IIL}} \tau}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{mktag}(\langle \ell_i = e_i \rangle_{i \in I}, \tau) : \mathsf{tag}(\langle \ell_i {:} \tau_i^\circ \rangle_{i \in I}, \tau^\circ)}$$

$$(\text{t1}) \ \frac{\begin{array}{c} \epsilon \vdash^{\text{IIL}} \sigma \\ \Delta; B; \Gamma \vdash^{\text{IIL}} e_1 : \mathsf{tag}(\tau_1, \alpha^-) \\ \Delta; B; \Gamma \vdash^{\text{IIL}} e_2 : \mathsf{tag}(\tau_2, \sigma^+) \\ \Delta; B\{\alpha \leq \sigma'\}; \Gamma \vdash^{\text{IIL}} b_1 : \tau \\ \Delta; B; \Gamma \vdash^{\text{IIL}} b_2 : \tau \end{array}}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{if}\ \ e_1 = e_2\ \ \mathsf{then}\ \ b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi} : \tau} \ (\text{unroll}_\epsilon(\sigma) = \sigma')$$

$$(\text{t2}) \ \frac{\begin{array}{c} \Delta; B; \Gamma \vdash^{\text{IIL}} e_1 : \mathsf{tag}(\tau_1, \sigma_1^-) \qquad \epsilon \vdash^{\text{IIL}} \sigma_1 \\ \Delta; B; \Gamma \vdash^{\text{IIL}} e_2 : \mathsf{tag}(\tau_2, \sigma_2^+) \qquad \epsilon \vdash^{\text{IIL}} \sigma_2 \\ (\epsilon; \epsilon \vdash^{\text{IIL}} \sigma_1 \leq \sigma_2 \quad \Rightarrow \quad \Delta; B; \Gamma \vdash^{\text{IIL}} b_1 : \tau) \\ \Delta; B; \Gamma \vdash^{\text{IIL}} b_2 : \tau \end{array}}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{if}\ \ e_1 = e_2\ \ \mathsf{then}\ \ b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi} : \tau}$$

$$\frac{\Delta \vdash^{\text{IIL}} \tau}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{none}^\tau : \tau?} \qquad \frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e : \tau}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{some}(e) : \tau?}$$

$$\frac{\Delta; B; \Gamma \vdash^{\text{IIL}} e_1 : \sigma? \quad \Delta; B; \Gamma, x : \sigma \vdash^{\text{IIL}} b_1 : \tau \quad \Delta; B; \Gamma \vdash^{\text{IIL}} b_2 : \tau}{\Delta; B; \Gamma \vdash^{\text{IIL}} \mathsf{if?}\ \ e_1\ \ \mathsf{then}\ \ x.b_1\ \ \mathsf{else}\ \ b_2\ \ \mathsf{fi} : \tau}$$

Figure 6.4: Tagging Target Typing Rules

The target language has the type-erasure interpretation. In particular, the operation $\mathsf{mktag}(e, \tau)$ is operationally equivalent to e, and the annotation $\mathsf{mktag}(\cdot, \tau)$ on heap values is not needed at run time.

The typing rules appear in Figure 6.4. The old typing rules for tuple projection and update are revised to look for tagged tuple types. The two rules for tag comparison deserve mention. Rule (t1) is for comparing an unknown tag against a known one. This is the rule used to type the translation, which always unpacks an existentially quantified package, extracts from it a tag for the quantified type, and compares it to a known tag. The rule requires the unknown tag e_1 to be a tag for a supertype of some type variable α. The known tag must be for a monomorphic type (Java and ML-style exceptions have this restriction). Thus, the rule requires e_2 to be a tag for a subtype of a closed type. However, rule (t1) is not closed under type substitution. In particular, if a closed type is substituted for α, then the expression compares two tags for known types, and the rule no longer applies. Thus to prove type substitution and type soundness, the rule (t2) is used to type this case. It requires both e_1 and e_2 to be tags for known closed types σ_1 and σ_2 respectively. If $\epsilon; \epsilon \vdash^{\text{IIL}} \sigma_1 \leq \sigma_2$ does not hold then it is impossible for e_1 to be equal to e_2, therefore b_1 is only type checked when this condition holds. In fact, b_1 will probably not type check when this condition does not hold, as it may use values of type σ_1 where values of type σ_2 are expected.

The static semantics is sound with respect to the operational semantics. I have proven a similar language sound [Gle99d]. Standard techniques are used in the proof and the only

$$
\begin{array}{lcl}
\mathit{tag}(\phi,\tau) & = & \mathsf{tag}(\langle \mathit{tag}'(\tau)\rangle, \tau^{\phi}) \\
\mathit{tag}'(\tau) & = & \mathsf{rec}\ \alpha.\mathsf{tag}(\langle\alpha\rangle, \tau^{-})? \\
\mathit{tagged}(\tau) & = & \mathsf{rec}\ \alpha.\langle \mathit{tag}(-,\alpha), \tau\rangle \\
[\![\mathsf{exn}]\!]_{\mathsf{t}} & = & [\![\mathsf{tagged}]\!]_{\mathsf{t}} \\
[\![\langle \ell_i{:}\tau_i^{\phi_i}\rangle_{i\in I}]\!]_{\mathsf{t}} & = & \mathsf{tag}(\langle \ell_i{:}[\![\tau_i]\!]_{\mathsf{t}}^{\phi_i}\rangle_{i\in I}, \mathsf{int}^{\circ}) \\
[\![\mathsf{tag}(\tau)]\!]_{\mathsf{t}} & = & \mathit{tag}(\circ, \mathit{tagged}([\![\tau]\!]_{\mathsf{t}})) \\
[\![\mathsf{tagged}]\!]_{\mathsf{t}} & = & \mathsf{self}\ \alpha.\langle \mathit{tag}(-,\alpha)\rangle \\
 & & \\
[\![\mathsf{newtag}(\tau)]\!]_{\mathsf{e}} & = & \mathsf{mktag}(\langle \mathsf{roll}^{\mathit{tag}'(\tau')}(\mathsf{none}^{\mathit{tag}(-,\tau')})\rangle, \tau') \\
 & & \text{where } \tau' = \mathit{tagged}([\![\tau]\!]_{\mathsf{t}}) \\
[\![\mathsf{subtag}(\tau,e)]\!]_{\mathsf{e}} & = & \mathsf{mktag}(\langle \mathsf{roll}^{\mathit{tag}'(\mathit{tagged}([\![\tau]\!]_{\mathsf{t}}))}(\mathsf{some}([\![e]\!]_{\mathsf{e}}))\rangle, \mathit{tagged}([\![\tau]\!]_{\mathsf{t}})) \\
[\![\mathsf{mktagged}(e_1,e_2{:}\tau)]\!]_{\mathsf{e}} & = & \mathsf{pack}^{[\![\mathsf{tagged}]\!]_{\mathsf{t}}}(\mathsf{roll}^{\mathit{tagged}([\![\tau]\!]_{\mathsf{t}})}(\langle [\![e_1]\!]_{\mathsf{e}}, [\![e_2]\!]_{\mathsf{e}}\rangle)) \\
\mathit{tagchk}(\alpha,\sigma) & = & \mathsf{fix}\ y_1(y_2{:}\alpha, y_2 : \mathit{tag}(-,\tau), y_3{:}\mathit{tag}(+,\sigma)) : \sigma?. \\
 & & \quad \mathsf{if}\ y_3 = y_4\ \mathsf{then}\ \mathsf{some}(\mathsf{unroll}(\mathsf{unroll}(y_2)).2)\ \mathsf{else} \\
 & & \quad \mathsf{if?}\ \mathsf{unroll}(y_3.1)\ \mathsf{then}\ y_5.y_1(y_2,y_5,y_4) \\
 & & \quad \mathsf{else}\ \mathsf{none}^{\sigma}\ \mathsf{fi}\ \mathsf{fi} \\
[\![\mathsf{if}\ \mathsf{tag}(e_1) \le e_2 : \mathsf{tag}(\sigma) & = & \mathsf{unpack}\ \alpha, x_1 = [\![e_1]\!]_{\mathsf{e}}\ \mathsf{in}\ \mathsf{let}\ x_2 = [\![e_2]\!]_{\mathsf{e}}\ \mathsf{in} \\
\quad \mathsf{then}\ x.b_1\ \mathsf{else}\ b_2\ \mathsf{fi}]\!]_{\mathsf{e}} & & \mathsf{if?}\ \mathit{tagchk}(\alpha, [\![\sigma]\!]_{\mathsf{t}})(x_1, \mathsf{unroll}(x_1).1, x_2)\ \mathsf{then} \\
 & & x.[\![b_1]\!]_{\mathsf{e}}\ \mathsf{else}\ [\![b_2]\!]_{\mathsf{e}}\ \mathsf{fi}
\end{array}
$$

Figure 6.5: Tagging Translation

difficulty is with the tag comparison operation. In showing type preservation for a successful tag comparison, I use the fact that $\epsilon;\epsilon;\Gamma \vdash^{\text{IIL}} x : \mathsf{tag}(-)\sigma_1, \tau_1$ and $\epsilon;\epsilon;\Gamma \vdash^{\text{IIL}} x : \mathsf{tag}(+)\sigma_2, \tau_2$ implies $\epsilon;\epsilon \vdash^{\text{IIL}} \sigma_1 \le \sigma_2$. Then by rule (t2) the *then* branch must type check. The other difference is the type substitution lemma mentioned earlier.

6.5 Translation

The translation from the tagging language to the target language is given in Figure 6.5. It is based on the ideas sketched earlier. However, as IIL does not have existentials, I use a self quantifier to abstract over the type in a tagged value. This means that the tags actually describe the tagged value, not the value itself, leading to recursive types.

The key to the type translation is the translation of tag types. A tag for the target type τ is a tuple with a tag option, suggesting the type $\mathsf{rec}\ \alpha.\mathsf{tag}(\langle\alpha?\rangle, \tau^{-})$. The contravariant form is used because the ancestor tags might tag a supertype of τ. However, the tag might be used for comparisons, where a covariant form is needed. Therefore, I unroll the type once, change the outermost tag type to be both contravariant and covariant (*i.e.*, invariant), and shift some type constructors, giving $\mathit{tag}(\circ,\tau)$ where $\mathit{tag}(\phi,\tau) = \mathsf{tag}(\langle \mathsf{rec}\ \alpha.\mathsf{tag}(\langle\alpha\rangle, \tau^{-})?\rangle, \tau^{\phi})$.

A tagged value in the target language will have type $\mathsf{rec}\ \alpha.\langle \mathit{tag}(-,\alpha), [\![\tau]\!]_{\mathsf{t}}\rangle$, where τ is the type of the value being tagged; this type is abbreviated $\mathit{tagged}([\![\tau]\!]_{\mathsf{t}})$. A self quantifier is used to abstract over τ, giving $\mathsf{self}\ \alpha.\langle \mathit{tag}(-,\alpha)\rangle$. Given this, we can see that a source tag type $\mathsf{tag}(\tau)$ is translated into $\mathit{tag}(\circ, \mathit{tagged}([\![\tau]\!]_{\mathsf{t}}))$.

The operations $\mathsf{newtag}(\tau)$, $\mathsf{subtag}(\tau, e)$, and $\mathsf{mktagged}(e,)$ are translated as I described earlier modulo all the typing annotations needed for recursive types, option types, and self quantifiers.

The $tagchk^H(x, y)$ predicate is reified as a recursive function $tagchk(\alpha, \sigma)$ that searches the superchain and returns a σ option, where σ is the known type. The translation of the tag comparison operation unpacks the tagged value, evaluates the known tag, uses the reified tag check predicate to do the comparison, and then executes the appropriate translated branch.

Technically, the translation is type directed, as it needs type information in two places. Thus the translation may not be defined for all source terms, but it is easy to show that it is defined for all typeable source terms. Furthermore, because the tag type is invariant, it is easy to show that there is only one type possible in the places where type information is required, so the translation is coherent. Rather than presenting the translation as a function of typing derivations, I have indicated the type information with a $: \tau$ notation on the source terms.

Unfortunately the rules for self quantifiers are not expressive enough for this translation. In particular, the translation is valid only with following rule for subtag:

$$\frac{\Delta; \Gamma \vdash^{\text{IIL}} e : \mathsf{tag}(\tau)}{\Delta; \Gamma \vdash^{\text{IIL}} \mathsf{subtag}(\tau, e) : \mathsf{tag}(\tau)}$$

More powerful rules for self quantifiers could eliminate this restriction, I leave the details to future work. Other than this, the translation is both type preserving and operationally correct. A similar translation that uses existentials instead of self quantifiers is presented and proven correct elsewhere [Gle99e, Gle99d].

6.6 Extended MooTal and Compiler

To compile the extended Iil into MooTal, MooTal needs to be extended with the type machinery of the previous section and a physical pointer equality instruction. The option type is fairly standard, and Chapter 7 describes how our implementation of Tal deals with it, so I do not add it to MooTal. Instead I concentrate on the novel aspects of Iil. MooTal's heap pointer type $*c_1$ is extended to include a tag type $*\mathsf{tag}(c_2^\phi)c_1$. Heap values are extended to include a tag type, and the malloc instruction is extended to include the tag type for the new heap value. Finally, a new instruction $\mathsf{tagcmp}\ v_1, v_2, v_b$ is added, which compares values v_1 and v_2 that must be heap pointers. If they are equal, it jumps to the code pointed to by v_b, otherwise execution continues with the following instruction. These changes are summarised below:

$$\begin{array}{rcl} c & ::= & \cdots \mid *\mathsf{tag}(c_1^\phi)c_2 \\ \iota & ::= & \cdots \mid \mathsf{malloc}\ r, \mathsf{tag}(c)\langle c_1, \ldots, c_n\rangle \mid \mathsf{tagcmp}\ v_1, v_2, v_b \\ h & ::= & \mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\hat{h} \end{array}$$

The typing rules and operational semantics parallel those of Iil, and details appear in the appendix.

The Iil to Tal compiler of Section 2.4 is extended to include the new constructs as follows. The type translation replaces the rule for tuples with:

$$[\![\mathsf{tag}(\langle \ell_i = \tau_i^{\phi_i}\rangle_{i\in 1,\ldots,n}, \tau^\phi)]\!]_{\mathsf{t}} \quad = \quad *\mathsf{tag}([\![\tau]\!]_{\mathsf{t}}^\phi)\langle [\![\tau_1]\!]_{\mathsf{t}}^{\phi_1}, \ldots, [\![\tau_n]\!]_{\mathsf{t}}^{\phi_n}\rangle$$

The extension of the expression translation appears in Figure 6.6. For the translation of if, α is the unknown tag type of e_1 and $\sigma = \text{unroll}_\epsilon(\sigma')$, where σ' is the known tag type of e_2.

$\mathsf{tag}(\langle \ell_i = e_j \rangle_{i \in 1,\ldots,n}, \tau)$

$\quad [\![e_1]\!]_{\mathsf{e}}(\Delta, \tau_a, \tau_b, h);$;; Compute fields

$\quad \mathsf{push\ r1};$ $\qquad \tau_a^i = [\![\tau_i]\!]_{\mathsf{t}} :: \cdots :: [\![\tau_1]\!]_{\mathsf{t}} :: \tau_a$

$\quad \ldots$

$\quad [\![e_n]\!]_{\mathsf{e}}(\Delta, vm, \tau_a^{n-1}, \tau_b, h+n-1);$

$\quad \mathsf{push\ r1};$

$\quad \mathsf{malloc\ r1}, \mathsf{tag}([\![\tau]\!]_{\mathsf{t}})\langle [\![\tau_1]\!]_{\mathsf{t}}, \ldots, [\![\tau_n]\!]_{\mathsf{t}} \rangle$;; Allocate record

$\quad \mathsf{pop\ r2}; \mathsf{mov\ } [\mathsf{r1} + n - 1], \mathsf{r2};$;; Initialise fields

$\quad \ldots$

$\quad \mathsf{pop\ r2}; \mathsf{mov\ } [\mathsf{r1} + 0], \mathsf{r2}$

$\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi}$

$\quad new(\lambda \ell_{true}.new(\lambda \ell_{end}.$ $\qquad \Delta' = \Delta, \alpha \leq [\![\sigma]\!]_{\mathsf{t}}$

$\quad [\![e_1]\!]_{\mathsf{e}}(\Delta, vm, \tau_a, \tau_b, h);$

$\quad \mathsf{push\ r1};$

$\quad [\![e_2]\!]_{\mathsf{e}}(\Delta, vm, [\![\tau_1]\!]_{\mathsf{t}} :: \tau_a, \tau_b, h+1);$

$\quad \mathsf{pop\ r2};$

$\quad \mathsf{tagcmp\ r2}, \mathsf{r1}, \mathrm{inst}(\ell_{true}, \Delta');$

$\quad [\![b_2]\!]_{\mathsf{e}}(\Delta, vm, \tau_a, \tau_b, h);$

$\quad \mathsf{jmp}\ \mathrm{inst}(\ell_{end}, \Delta);$

$\ell_{true} \mapsto \mathsf{code}\ [\Delta']\mathrm{sc}(\tau_a, \tau_b)$

$\quad [\![b_1]\!]_{\mathsf{e}}(\Delta, vm, \tau_a, \tau_b, h);$

$\quad \mathsf{jmp}\ \mathrm{inst}(\ell_{end}, \Delta');$

$\ell_{end} \mapsto \mathsf{code}\ [\Delta]\mathrm{ec}(\tau_a, \tau_b, [\![\tau]\!]_{\mathsf{t}})$

$\quad))$

Figure 6.6: Tagging Extended Iil to MooTal Compiler

Chapter 7

TAL Implementation

So far I have presented a theoretical language MooTal and shown how to compile a substantial class of procedural, functional, and object-oriented languages to it. However, the question remains as to whether such a language could be used in practice and how effective the typing annotations and type checker might be in a real system. To address these concerns, a group at Cornell implemented a variant of MooTal for Intel's 32-bit architecture (IA32, implemented on the 80386 through Pentium III processors) [Int97] called Talx86. The implementation consists of a tool `talc` for checking Talx86 object files and programs and some experimental compilers that target Talx86.

Our tool `talc` reads Talx86's object files in a text format, type checks them, checks link compatibility of object files, and checks program completeness for a collection of object files. It also includes an assembler that produces typed binary object files in the native COFF or ELF format (actually the type part is stored in a separate `.to` file). We include a run-time system consisting of some operating-system glue code, the Boehm-Demers-Weiser conservative garbage collector [BW88], and some array creation primitives. `talc` can link the assembled object files with the run-time system to produce a Win32 or Linux executable.

Our experimental compilers include the `popcorn` compiler, a Scheme-like language compiler, and the `solc` compiler. The `popcorn` compiler compiles a safe C-like language that includes structs, discriminated unions, checked variable-sized arrays, parametric polymorphism, and exceptions. The Scheme and `popcorn` compilers are implemented both in Objective Caml and `popcorn`. The `solc` compiler compiles a small object-oriented language; it demonstrates the effectiveness of MooTal's object support. We are currently adding object-oriented features to `popcorn` also.

For the remainder of the chapter, I describe in more detail Talx86 and its features for supporting the compilation of realistic languages.

Base Types To support characters, shorts, and longs, Talx86 includes four base types: $\mathsf{B1}$, $\mathsf{B2}$, $\mathsf{B4}$, and $\mathsf{B8}$. These correspond to 8, 16, 32, and 64 bit uninterpreted numbers. To deal with these different sizes, Talx86's kinds are more refined than Tal's and include subkinds of T and M for different size types. The kinds $\mathsf{T1}$, $\mathsf{T2}$, $\mathsf{T4}$, and $\mathsf{T8}$ are for 8, 16, 32, and 64 bit words types and are subkinds of T. The kind $\mathsf{Tm}\,i$ is for memory types of i bytes and is a subkind of M (written Tm in Talx86). Stack types could also be differentiated on size, but we have not found a need for this. Loads and stores in Talx86 use byte offsets, and to determine which field is being referred to requires knowing the sizes of the preceding fields. The type checker determines these by examining the kinds of the types. However, some types, such as a type variable of kind T, do not have a known size. The type checker rejects loads and stores where the field cannot be determined due to types of unknown size. Type variables with more specific kinds, such as $\mathsf{T4}$, do have known sizes, and do not interfere with projection. This aspect of TALX86 forces compilers to deal with the problems of polymorphism in the presence of longs (*e.g.*, by boxing longs when they are arguments to polymorphic functions).

Singletons To support both discriminated unions and arrays, TALX86 includes a singleton type $\mathsf{S}(c)$ where c is a type constructor of a new kind Sint, the kind of integers. The type constructor language also includes integers i and expressions on them, which all have this kind. A value is of type $\mathsf{S}(c)$ when it is a 32-bit integer equal to c. This aspect of TALX86 has recently been extended to allow reasoning about the relationships between various integer quantities, particularly between the indexes and sizes of an array. The mechanism is general enough that the type checker can verify for array index and subscript operations that the index is within the bounds of the array. The specifics are not relevant to the rest of this dissertation, so I will not discuss them further. Our ideas are based on those of Xi and Pfenning [XP98].

Memory Types The memory types in TALX86 are much more elaborate than in MOOTAL and allow for the compilation of the three major data-structuring techniques: structures, discriminated unions, and arrays. The design is general and can express many unboxed nested data structures. MOOTAL's tuple type is split into two type constructors in TALX86: $*\langle c_1, \ldots, c_n\rangle$ and c^ϕ. The former is called a product type and describes memory that is the concatenation of memory of types c_1 through c_n. The latter is a field type and describes memory that contains a word value of type c with ϕ specifying the allowable operations. In addition there are the types $+\langle c_1, \ldots, c_n\rangle$, $\mathsf{array}(c_1, c_2)$, and the heap pointer type $*[i_1, \ldots, i_n]c$. The first is a discriminated union type, that is, one of c_i; the second is an array type where c_1 is the size and c_2 the element type; the third is like the heap pointer type $*c$, except that the value can also be one of the integers i_1 through i_n. To illustrate how these work, consider the ML datatype:

```
type l = Var of string | Abs of string*l | App of l*l | Abort
```

This might be translated into the TALX86 type:

$$l = \mathsf{rec}\ l{:}\mathsf{T4}.*[0]+\langle *\langle \mathsf{S}(0)^+, \mathsf{string}^+\rangle, *\langle \mathsf{S}(1)^+, \mathsf{string}^+, l^+\rangle, *\langle \mathsf{S}(2)^+, l^+, l^+\rangle\rangle$$

This type is read as "a value that is equal to 0 or a pointer to a heap block of one of the types $*\langle \mathsf{S}(0)^+, \mathsf{string}^+\rangle$, $*\langle \mathsf{S}(1)^+, \mathsf{string}^+, l^+\rangle$, or $*\langle \mathsf{S}(2)^+, l^+, l^+\rangle$." The last of the variants is read as "a block of memory that consists of a read-only word value that equals 2, followed by a read-only word value of type l, followed by a read-only word value of type l." Strings are not built into TALX86, but could be defined as $\mathsf{string} = \exists n{:}\mathsf{Sint}.*[]*\langle \mathsf{S}(n)^+, \mathsf{array}(n, \mathsf{B1}^+)\rangle$. This should be read as "there exists an integer n such that the value is a pointer to a heap block that has a read-only word value that equals n followed by n read-only word values each of which is an 8-bit number."

All of these types are introduced by coercion from a singleton (*e.g.*, $\mathsf{S}(0)$ into l) or an equivalent product type (*e.g.*$*[]*\langle \mathsf{S}(1)^+, \mathsf{string}^+, l^+\rangle$ into l). The sum types are eliminated by type refinement after appropriate values are compared against known integers. For example, if register eax has type l and is compared against 6 followed by a jump on less than, then the type checker will assume that eax has type $*[0]$ in the branch and that eax has type $*[]\cdots$ in the fall through. This refinement requires the mild assumption that all pointers are at least some small integer (currently 4096). Similarly, if eax has type l without the 0 and $[\mathsf{eax}+0]$ is compared against 1 followed by a jump on less than, then the type checker will assume eax has type $*[]{+}\langle *\langle \mathsf{S}(0)^+, \mathsf{string}^+\rangle\rangle$ in the branch and type $*[]{+}\langle *\langle \mathsf{S}(1)^+, \mathsf{string}^+, l^+\rangle, *\langle \mathsf{S}(2)^+, l^+, l^+\rangle\rangle$ in the fall through. The former can be coerced back into a product type, and the second and third fields can then be extracted. In an earlier version of TALX86, this type refinement was achieved through special instructions, but recently we have added "named value types" to TALX86 that allow us to conservatively track aliasing and the types of specific values such that refinement of all aliases is possible. Again, I will not go into the specifics of this mechanism any further.

The array type is eliminated by indexed moves, that is, moves of the form $\mathsf{mov}\, r_d, [r_a + r_i + j]$ or $\mathsf{mov}\ [r_a + r_i + j], r_s$ where r_a points to a heap block containing an array, j is the offset of the array within the block, and r_i contains the scaled index of the desired array element. The current TALX86 checks that the array has type $\mathsf{array}(c_n, c)$, that the index register has type $\mathsf{S}(c_i)$, and that $0 \le c_i < \mathit{sizeof}(c) \times c_n$.

Instructions A large portion of the IA32 instruction set is supported in TALX86. We support all of the arithmetic, bitwise, shift, and rotate instructions; condition and unconditional jumps; call; return; various moves including load, store, and array subscript and update; stack manipulation instructions; exchange; and various sign and zero extension instructions.

We do not support any privileged instructions, and assume a flat model.[1] We do not support a number of instructions that have to do with backwards compatibility with the 8086 through 80286 processors. Most of these could be easily added with the exception of the string instructions. We do not support the concurrency operations like test-and-set and compare-and-swap, but these can be added easily when we want to support multithreaded execution. We do not currently support floating point or MMX instructions, but these are easy to add and we intend to do so. Finally, software interrupts and CPUID are not supported. The latter is easy to add were it useful, but the former are more challenging.

Interfaces A TALX86 interface can declare a type name to be abstract, bounded above by c, or equal to c. This allows the programmer to express abstract, partially abstract, and concrete named types.

Experience Our experience with TALX86 and the various compilers, while limited, has taught us that building a practical typed assembly language is possible and that the size of type annotations and type checking time are within reasonable limits. Further work remains, but Grossman and Morrisett describe initial results [GM99b].

[1]The flat model assumption means that the segment registers cannot be referred to and are all assumed to point to a single 4Gb virtual address space that all code, data, and stacks reside within. We also do not allow any far jumps or calls.

Chapter 8

Future Work

This dissertation has presented Typed Assembly Language—a RISC-like assembly language augmented with typing annotations, typing rules, and a memory allocation primitive. The language was designed to be sound, and a proof of MooTal's type soundness appears in the appendix. Soundness means that during execution, a MooTal program will not commit a type error, which implies many basic security properties such as memory safety and control flow safety. This guarantee can be used in type-directed compilation and in secure extensible systems. A type-directed compiler could target MooTal, and if its output passes the type checker, certain correctness properties follow. Thus, running the type checker after compilation provides a means to check for a certain class of errors in the compiler. A secure extensible system can use MooTal as the language for extensions. A extension that passes the type checker will not violate the basic security properties above, and in addition will not violate certain abstractions, such as the abstractness of type names. These basic properties can be used by a security monitor to provide more elaborate security properties.

Most of the dissertation was spent describing the constructs of MooTal and presenting a formal compiler to demonstrate how to use these constructs in the compilation of various common programming language features. I have shown how to compile products, functions, objects, single inheritance classes, exceptions, run-time class dispatch, and the basic types and control flow constructs of procedural, functional, and object oriented languages. I have hinted at how our implementation deals with discriminated unions, arrays, and different sized values. Taken as a whole, this shows the applicability of MooTal and Talx86 to a wide range of realistic language features.

The soundness and targetability of MooTal are two important parts of demonstrating that typed assembly language can address the problems of type directed compilation and secure extensible systems. What remains are pragmatic questions: how large are the type annotations and how efficient is type checking? I conclude with some other future directions for research.

Dynamic Linking The linking model of Mtal models static linking only. A form of dynamic linking, in which the complete linking of a program is delayed until the loading of a program into an initial process image, can also be modelled. However, another form of dynamic linking, often called dynamic loading, in which an object file is linked into a process image during program execution, cannot be modelled. There are a number of issues that arise with dynamic loading that need to be addressed in order to build a theory of dynamic linking. One is the failure model for dynamic loading. Should unresolved labels cause failure only at dynamic loading points, or could any instruction that dereferences a label fail? Another issue is whether the program can directly refer to labels in a dynamically loaded module, or only indirectly through a value returned by the dynamic loading operation. A final issue is whether modules can be unloaded, and if so, whether they can be reloaded later.

Other Class-Based Models The object encoding that I presented handles single inheritance classes only. These types of classes have a particularly efficient implementation but suffer from the "fragile base class" problem—a change to a superclass requires recompiling all the subclasses. Other class models exist that are more expressive and do not suffer from the fragile base class problem. It would be worth investigating if the template language and object encoding presented in Chapter 5 could be extended to express these class models. Mixins are closely related to class-based object-oriented languages, except that they do not specify the superclass but only the superclass's interface. Perhaps they too could be incorporated into a template framework and an extension of my object encoding. Other features of classes such as abstract methods, final methods, and final classes should also be investigated.

Prototypes and Delegation Prototypes and delegation are an alternative to classes and mixins. In languages based on these ideas, there are only objects, and objects can inherit or delegate their definition and behaviour to other objects. Fisher, Mitchell, and others [Fis96, Mit90, FHM94, FM95a, FM95b, FM96, BF98, FM98] have done considerable work in providing foundations for these kinds of object-oriented languages. They have also shown how to encode class-based languages into their languages. It would be interesting to compare the two approaches.

Full Abstraction As mentioned in the introduction, there must be a formal connection between abstractions in the source language and their translation into TALX86, such that the well formedness of TALX86 implies that the source abstractions are respected. Such a result can be shown by proving a full abstraction property—that two indistinguishable source terms translate into indistinguishable TALX86 terms. I believe that my object encoding has the full abstraction property, but I have not constructed a proof.

Related to this, the techniques of Zdancewic et al. [ZGM99] can be used to prove that various expected properties of abstract types actually do hold in MTAL. This would involve treating each MTAL object file as a different *coloured code* (see Zdancewic et al.'s paper for further details), modifying linking to reflect the correct combination of coloured code, reflecting the coloured reduction rules into TAL's operational semantics, and repeating their proofs.

Pragmatic Issues To fully assess the effectiveness of TAL, it needs to be measured on a number of large benchmarks. This requires the construction of a complete compiler for a realistic language that does at least some optimisation, and the measurement of the `talc` tool on some large benchmarks compiled into TALX86. Such work would measure the overhead of the type annotations (*i.e.*, the percentage increase in size of the typed object file versus its underlying untyped version), the time to perform type checking in comparison to other operations such as transferring object files across the network and loading executables into a process image, and the relative performance of the code versus untyped aggressively optimising compilers and higher level approaches such as the Java virtual machine. Ideally, such research would also investigate a number of different language styles and optimisation techniques. A number of members of the TAL project at Cornell are currently addressing these concerns.

Appendix A

MooTal

This Appendix gives a complete formal description of MooTal, my version of typed assembly language. MooTal is a typed assembly language with support for separate compilation and the compilation of modules, objects, classes, and run-time type dispatch. Its instruction set is simliar to a typical RISC architecture—the instruction set is uniform and has only simple addressing modes. This appendix will describe the syntax of MooTal, formalise its semantics as a reduction relation between machine states, formliase its typing rules, and proof that the typing rules are sound with respect to the operational semantics. It begins with a description of basic concepts and notation, then describes the module language and core language, and finishes with the soundness proof. The syntax is summarised in Figure A.1.

A.1 Notational Conventions

There is some set of labels, registers, and type constructors variables. Labels ℓ are used to name both types and values for intermodule references, and are used as the addresses of memory. The set of registers (ranged over by r) could be a countably infinite set of virtual registers or a finite set of physical registers; it does not matter which, but MooTal requires that r1 is a register (this register holds the result value when a MooTal program halts). Type constructor variables are ranged over by α and β. Integers are ranged over by i. Syntactic objects are considered equal up to α-equivalence. The capture-avoiding substitution of x for y in z is written $z\{x := y\}$. An unordered map that maps x_i to y_i is written $\{x_1{:}y_1, \ldots, x_n{:}y_n\}$ for type level constructs and $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ for term level constructs. It is a syntactic restriction that the x_i be distinct. The domain of a map X is written $\mathrm{dom}(X)$, the value of X at x is written $X(x)$, and map update is written $X\{x{:}y\}$ or $X\{x \mapsto y\}$. Syntactic objects are considered equal up to reordering of unordered maps. An ordered map is written $x_1{:}y_1, \ldots, x_n{:}y_n$; it is a syntactic restriction that the x_i be distinct. The notation $X, x{:}y$ denotes $x{:}y$ appended to X; X_1, X_2 denotes X_2 appended to X_1. A vector of objects from syntax class x is written $\vec{x}$, for example, $\vec{\alpha}$ stands for $\alpha_1, \ldots, \alpha_n$; the notation $\overrightarrow{\alpha{:}\kappa \leq c}$ will be used to denote sequences like $\alpha_1{:}\kappa_1 \leq c_1, \ldots, \alpha_n{:}\kappa_n \leq c_n$. Generally typing judgements have the forms $C \vdash x$, $C \vdash x : X$, or $C \vdash x_1 R x_2$ to mean that in context C, x is well formed, x has "type" X, or x_1 is R related to x_2 (R might be equality, subtyping, compatibility, or disjointness). Typing rules have the form

$$(\mathrm{n})\ \frac{J_1 \quad \cdots J_n}{J}\ (P)$$

where n is the rule name, P is a side condition, J is the conclusion judgement, and J_1 through J_n are the hypothesis judgements. The meaning of the rule is that if J_1 through J_n are derivable

Kinds	κ	$::=$	$\mathsf{T} \mid \mathsf{M} \mid \mathsf{S} \mid \kappa_1 \to \kappa_2$
Variances	ϕ	$::=$	$+ \mid - \mid \circ \mid 0$
Type Variable Declaration	t	$::=$	$\alpha{:}\kappa \mid \alpha{:}\kappa \leq c$
Type Constructors	c	$::=$	$\alpha \mid \ell \mid \lambda\alpha{:}\kappa.c \mid c_1\ c_2 \mid$
			$\forall t.c_2 \mid \mathsf{rec}\ \alpha : \kappa.c \mid \mathsf{self}\ \alpha : \kappa.c$
			$\mathsf{int} \mid \mathsf{ns} \mid *\mathsf{tag}(c_1^{\phi})c_2 \mid \mathsf{sptr}(c) \mid$
			$\mathsf{code}\ \Gamma \mid \langle c_1^{\phi_1}, \ldots, c_n^{\phi_n} \rangle \mid$
			$\mathsf{se} \mid c_1 :: c_2 \mid c_1 \circ c_2$
Register File Types	Γ	$::=$	$\{\mathsf{sp}{:}c, r_1{:}c_1, \ldots, r_n{:}c_n\}$
Type Variable Bounds	B	$::=$	$\alpha_1 \leq c_1, \ldots, \alpha_n \leq c_n$
Type Variable Contexts	Δ	$::=$	$\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n$
Type Heap Types	Φ	$::=$	$\{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}$
Value Heap Types	Ψ	$::=$	$\{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\}$
Interfaces	Int	$::=$	(Φ, Ψ)
Coercions	δ	$::=$	$[c] \mid \mathsf{roll}^c \mid \mathsf{unroll} \mid \mathsf{pack}^c$
Small Values	v	$::=$	$i \mid \ell \mid r \mid ?_c \mid \mathsf{ns} \mid \mathsf{sptr}(i) \mid \delta(v)$
Word Values	w	$::=$	v without r and $\mathsf{unroll}(\mathsf{roll}^c(v))$
Instructions	ι	$::=$	$\mathit{aop}\ r_d, v_1, v_2 \mid \mathit{bop}\ r, v \mid$
			$\mathsf{malloc}\ r, \mathsf{tag}(c)\langle c_1, \ldots, c_n \rangle \mid \mathsf{mov}\ r, v \mid$
			$\mathsf{mov}\ r_d, [r_s + i] \mid \mathsf{mov}\ [r_d + i], r_s \mid$
			$\mathsf{mov}\ \mathsf{sp}, \mathsf{sp} + i \mid \mathsf{mov}\ r, \mathsf{sp} \mid \mathsf{mov}\ \mathsf{sp}, r \mid$
			$\mathsf{mov}\ r_d, [\mathsf{sp} + i] \mid \mathsf{mov}\ [\mathsf{sp} + i], r_s \mid$
			$\mathsf{tagcmp}\ v_1, v_2, v_b \mid \mathsf{unpack}\ \alpha, r, v$
Instruction Sequences	I	$::=$	$\iota; I \mid \mathsf{halt}[c] \mid \mathsf{jmp}\ v$
Heap Values	$\hat{h}$	$::=$	$\mathsf{code}\ I \mid \langle w_1, \ldots, w_n \rangle$
Heap Values	h	$::=$	$\mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\hat{h}$
Stacks	S	$::=$	$\mathsf{se} \mid w :: S$
Register Files	R	$::=$	$\{\mathsf{sp} \mapsto S, r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
Type Constructor Heaps	CH	$::=$	$\{\ell_1 \mapsto c_1{:}\kappa_1, \ldots, \ell_n \mapsto c_n{:}\kappa_n\}$
Value Heaps	VH	$::=$	$\{\ell_1 \mapsto h_1{:}c_1, \ldots, \ell_n \mapsto h_n{:}c_n\}$
Object Files	O	$::=$	$[\mathit{Int}_I \Rightarrow \mathit{CH}, \mathit{VH} : \mathit{Int}_E]$
Executables	E	$::=$	$(\mathit{CH}, \mathit{VH}, \ell)$
Program States	P	$::=$	$(\mathit{CH}, \mathit{VH}, R, I)$

Figure A.1: MooTal Syntax

and P holds then J is derivable. Sometimes the hypotheses will be indexed as in $C \vdash x_i : X_i$ where there are x_1 through x_n and X_1 through X_n somewhere else in the typing rule. I will not write out explicitly what i ranges over; it should be clear. In particular, in:

$$\frac{C \vdash x_i \leq y_i \quad C \vdash x_j}{C \vdash x_1, \ldots, x_m \leq y_1, \ldots, y_n} \ (m \geq n)$$

i ranges over $1..n$ and j ranges over $1..m$.

A.2 Module Language

The module language of MooTal consists of object files with associated interfaces. Object files can be linked together to construct progressively more complete object files. Complete object files can be formed into executables, which can then be executed. The module language is mostly independent of the core language, provided the core language has certain syntax categories and certain typing judgements for them. First, the core language has *type constructors* classified into *kinds*. Second, the core language has a syntax class *heap value* for term level constructs that can be referred to across modules. Third, there are typing judgements for *kinding* a type constructor, type equality at a kind, subtyping at a kind, and typing a heap value. Fourth, the module system assumes the core language has a simple kind structure without kind equality or subkinding. More elaborate kind structures are easily added.

A.2.1 Object Files

The basic units of MooTal's module system are object files and their associated interfaces. An interface *Int* has the form (Φ, Ψ) consisting of two parts: the type part Φ and the value part Ψ. The type part, also called a type heap type, has the form $\{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}$ mapping labels, which should be thought of as names for types, to kinds. The value part, also called a value heap type, has the form $\{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\}$ mapping labels, which should be thought of as names for terms, to types. An object file O has the form $[\mathit{Int}_I \Rightarrow \mathit{CH}, \mathit{VH} : \mathit{Int}_E]$ consisting of four parts: an interface Int_I specifying the types and values imported, a type constructor heap CH that defines types, a value heap VH that defines heap values, and an interface Int_E specifying which of the defined types and heap values are exported. A type constructor heap CH has the form $\{\ell_1 \mapsto c_1{:}\kappa_1, \ldots, \ell_n \mapsto c_n{:}\kappa_n\}$ mapping type names to type constructors and their kinds. A value heap VH has the form $\{\ell_1 \mapsto h_1{:}c_1, \ldots, \ell_n \mapsto h_n{:}c_n\}$ mapping labels to heap values and their types.

The labels (type names and value names) that appear in the heaps CH and VH of an object file but not in the export interface int_E are called *internal labels*, and the labels that appear in the heaps and the export interface are called *external labels*. Internal labels are like variables and are considered to α-vary, that is, an object file is α-equivalent to itself with the internal labels consistently renamed. External labels are like names and do not α-vary.

The important judgements for interfaces are: well formedness, disjointness, and subinterface. An interface is well formed, written $\vdash_{\mathsf{Int}} \mathit{Int}$, when all the kinds and types in it are well formed. Type well formedness is expressed by the judgement $\Phi; \Delta \vdash_{\mathsf{tc}} c : \kappa$ where Φ specifies the allowable type names and their kinds and Δ is a core language typing context. For our purposes, the only requirement on Δ is that it contain the empty context ϵ.

$$\text{(wf-Int)}\ \frac{\Phi \vdash_{\mathsf{VHT}} \Psi}{\vdash_{\mathsf{Int}} (\Phi, \Psi)} \qquad \text{(wf-VHT)}\ \frac{\Phi; \epsilon \vdash_{\mathsf{tc}} c_i : \kappa_i}{\Phi \vdash_{\mathsf{VHT}} \{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\}}$$

Two interfaces are disjoint $\vdash_{\mathsf{Int}} Int_1 \mid Int_2$ when they define different type names and different value names.

$$\text{(dj-Int)}\ \frac{\vdash_{\mathsf{CHT}} \Phi_1 \mid \Phi_2 \quad \vdash_{\mathsf{VHT}} \Psi_1 \mid \Psi_2}{\vdash_{\mathsf{Int}} (\Phi_1, \Psi_1) \mid (\Phi_2, \Psi_2)}$$

$$\text{(dj-CHT)}\ \frac{}{\vdash_{\mathsf{CHT}} \Phi_1 \mid \Phi_2}\ (\mathrm{dom}(\Phi_1) \cap \mathrm{dom}(\Phi_2) = \emptyset)$$

$$\text{(dj-VHT)}\ \frac{}{\vdash_{\mathsf{CHT}} \Psi_1 \mid \Psi_2}\ (\mathrm{dom}(\Psi_1) \cap \mathrm{dom}(\Psi_2) = \emptyset)$$

Subinterfaces provide everything that a superinterface does, that is, they include all the type names and value names contained in the superinterface and at at least as specific kinds and types. Subtyping is given by the judgement $\Phi; \Delta; B \vdash c_1 \leq c_2 : \kappa$ where B is a core language bound set. Again, B must include the empty context ϵ.

$$\text{(sub-Int)}\ \frac{\vdash_{\mathsf{CHT}} \Phi_1 \leq \Phi_2 \quad \Phi_1 \vdash_{\mathsf{VHT}} \Psi_1 \leq \Psi_2 \quad \Phi_2 \vdash_{\mathsf{VHT}} \Psi_2}{\vdash_{\mathsf{Int}} (\Phi_1, \Psi_1) \leq (\Phi_2, \Psi_2)}$$

$$\text{(sub-CHT)}\ \frac{}{\vdash_{\mathsf{CHT}} \{\ell_1{:}\kappa_1, \ldots, \ell_m{:}\kappa_m\} \leq \{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}}\ (m \geq n)$$

$$\text{(sub-VHT)}\ \frac{\Phi; \epsilon \vdash_{\mathsf{tc}} c_i : \kappa_i \quad \Phi; \epsilon; \epsilon \vdash c_i \leq c_i' : \kappa_i}{\Phi \vdash_{\mathsf{VHT}} \{\ell_1{:}c_1, \ldots, \ell_m{:}c_m\} \leq \{\ell_1{:}c_1', \ldots, \ell_n{:}c_n'\}}\ (m \geq n)$$

An object file is well formed, written $\vdash_{\mathsf{O}} O$, when its imports and definitions are disjoint, its definitions are well formed, and its definitions implement its exports. To formalise this, we view the definitions as having a type, the actual interface Int_A, and require that Int_I and Int_A be disjoint and Int_A to be a subinterface of Int_E. The type definitions may refer to other type names, either from the imports or in the object file itself. Thus, the type heap is checked using both the imported and actual type heap type. Similarly for the value heap. The value heap is also checked using the type definitions, as a heap value may convert between a type name and its definition and vice versa.

$$\text{(objfile)}\ \frac{\begin{array}{c}\vdash_{\mathsf{Int}} (\Phi_I, \Psi_I) \quad \vdash_{\mathsf{Int}} (\Phi_I, \Psi_I) \mid (\Phi_A, \Psi_A) \quad \vdash_{\mathsf{Int}} (\Phi_A, \Psi_A) \leq Int_E \\ \Phi_I \cup \Phi_A \vdash_{\mathsf{CH}} CH : \Phi_A \quad \Phi_I \cup \Phi_A; CH; \Psi_I \cup \Psi_A \vdash_{\mathsf{VH}} VH : \Psi_A\end{array}}{\vdash_{\mathsf{O}} [(\Phi_I, \Psi_I) \Rightarrow CH, VH : Int_E]}$$

A type heap CH has type Φ when each type has the corresponding kind. A value heap VH has type Ψ when each heap value has the corresponding type. Heap value typing is given by $\Phi; CH; \Psi \vdash_{\mathsf{h}} h : c$.

$$\text{(CH)}\ \frac{\Phi; \epsilon \vdash_{\mathsf{tc}} c_i : \kappa_i}{\Phi \vdash_{\mathsf{CH}} \{\ell_1 \mapsto c_1{:}\kappa_1, \ldots, \ell_n \mapsto c_n{:}\kappa_n\} : \{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}}$$

$$\text{(VH)}\ \frac{\Phi; CH; \Psi \vdash_{\mathsf{h}} h_i : c_i}{\Phi; CH; \Psi \vdash_{\mathsf{VH}} \{\ell_1 \mapsto h_1{:}c_1, \ldots, \ell_n \mapsto h_n{:}c_n\} : \{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\}}$$

A.2.2 Linking

Linking is the process of combining two object files together. In MooTal linking is specified as a type directed translation $\vdash O_1\ \mathsf{link}\ O_2 \rightsquigarrow O$. The type heaps, value heaps, and export interfaces are combined, and the final import interface is the union of the imports minus what the other object file provided. So if $O_i = [Int_{Ii} \Rightarrow CH_i, VH_i : Int_{Ei}]$ then:

$$\text{(link)}\ \frac{\vdash_{\mathsf{O}} O_1 \quad \vdash_{\mathsf{O}} O_2 \quad \vdash O_1 \overset{\mathsf{lc}}{\leftrightarrow} O_2}{\vdash O_1\ \mathsf{link}\ O_2 \leadsto [Int_I \Rightarrow CH_1 \cup CH_2, VH_1 \cup VH_2 : Int_{E1} \cup Int_{E2}]}\ (*)$$

where $Int_I = (Int_{I1} \cup Int_{I2}) - (Int_{E1} \cup Int_{E2})$ and $(*)$ is $\mathrm{dom}(CH_1) \cap \mathrm{dom}(CH_2) = \emptyset = \mathrm{dom}(VH_1) \cap \mathrm{dom}(VH_2)$. Linking checks whether the linking operation is valid and requires that the two object files be well formed and link compatible, written $\vdash O_1 \overset{\mathsf{lc}}{\leftrightarrow} O_2$. Condition $(*)$ ensures that combining the two heaps results in a valid heap. By α-varying the internal labels, condition $(*)$ can always be satisfied if $\vdash O_1 \overset{\mathsf{lc}}{\leftrightarrow} O_2$; this α-variation corresponds to relocation in a conventional linker.

Link compatibility requires that the object files make consistent assumptions about the global name space of types and values. The exports of the object files must be disjoint, and the imports of one must be *compatible* with the imports and exports of the other. At the type level, this is straightforward: Two type heap types are compatible, written $\vdash_{\mathsf{CHT}} CH_1 \sim CH_2$, if type names they both define have the same kind.

$$\text{(cmp-CHT)}\ \frac{\forall \ell \in \mathrm{dom}(CH_1) \cap \mathrm{dom}(CH_2) : CH_1(\ell) = CH_2(\ell)}{\vdash_{\mathsf{CHT}} CH_1 \sim CH_2}$$

Value heap type compatibility is complicated by subtyping. Import/export compatibility requires the exported type to be a subtype of the imported type. For import/import compatibility, there are several choices. One choice is to require the existence of a meet of the two types. This choice raises the issues of whether enough meets exist and whether computing meets is easy. For expediency, I take the simpler but more restrictive approach of requiring the two types to be equal. Type equality is given by the judgement $\Phi; \Delta \vdash c_1 = c_2 : \kappa$.

$$\text{(cmp-CHT-EI)}\ \frac{\forall \ell \in \mathrm{dom}(CH_1) \cap \mathrm{dom}(CH_2) : \Phi; \epsilon; \epsilon \vdash VH_1(\ell) \leq VH_2(\ell) : \kappa_\ell}{\Phi \vdash_{\mathsf{VHT}} VH_1 \overset{\mathsf{EI}}{\sim} VH_2}$$

$$\text{(cmp-CHT-II)}\ \frac{\forall \ell \in \mathrm{dom}(CH_1) \cap \mathrm{dom}(CH_2) : \Phi; \epsilon; \epsilon \vdash VH_1(\ell) = VH_2(\ell) : \kappa_\ell}{\Phi \vdash_{\mathsf{VHT}} VH_1 \overset{\mathsf{II}}{\sim} VH_2}$$

Given these definitions, interface compatibility and link compatibility are easily defined:

$$\text{(cmp-Int)}\ \frac{\vdash_{\mathsf{CHT}} \Phi_1 \sim \Phi_2 \quad \Phi \vdash_{\mathsf{VHT}} \Psi_1 \overset{x}{\sim} \Psi_2}{\vdash_{\mathsf{Int}} (\Phi_1, \Psi_1) \overset{x}{\sim} (\Phi_2, \Psi_2)}$$

$$\text{(lc)}\ \frac{\vdash_{\mathsf{Int}} Int_{I1} \overset{\mathsf{II}}{\sim} Int_{I2} \quad \vdash_{\mathsf{Int}} Int_{E1} \overset{\mathsf{EI}}{\sim} Int_{I2} \quad \vdash_{\mathsf{Int}} Int_{E2} \overset{\mathsf{EI}}{\sim} Int_{I1} \quad \vdash_{\mathsf{Int}} Int_{E1} \mid Int_{E2}}{\vdash [Int_{I1} \Rightarrow CH_1, VH_1 : Int_{E1}] \overset{\mathsf{lc}}{\leftrightarrow} [Int_{I1} \Rightarrow CH_1, VH_1 : Int_{E1}]}$$

Linking always produces a well-formed object file.

Theorem A.1 (Linking Correctness) *If* $\vdash O_1\ \mathsf{link}\ O_2 \leadsto O$ *then* $\vdash_{\mathsf{O}} O$.

Proof: By inspection of the various typing and linking judgement rules and Context Strengthening (see Section A.3.8). □

A.2.3 Executables

An executable is a closed type and value heap paired with an entry label that specifies which term to execute first, $E ::= (CH, VH, \ell)$. An executable is well formed when there is an interface that describes the type and value heap, and the entry label has a type matching the operating system's convention. The latter type is called the *entry label type*, and $\vdash^{c}_{\mathsf{E}} E$ asserts that E is a well formed executable with entry label type c.

$$\text{(exe)}\ \frac{\vdash_{\mathsf{Int}} (\Phi, \Psi) \quad \Phi \vdash_{\mathsf{CH}} CH : \Phi \quad \Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi}{\vdash^{c}_{\mathsf{E}} (CH, VH, \ell)}\ (\Psi(\ell) = c)$$

Executables are formed by taking a complete object file and a distinguished label and discarding the import and export interfaces. An object file is complete when it imports nothing. If $O = [Int_I \Rightarrow CH, VH : Int_E]$ then:

$$\text{(prg)}\ \frac{\vdash_{\mathsf{O}} O \quad \vdash O, \ell : c\ \mathsf{complete}}{\vdash O, \ell : c \overset{\mathsf{prg}}{\leadsto} (CH, VH, \ell)}$$

$$\text{(comp)}\ \frac{}{\vdash [(\epsilon, \epsilon) \Rightarrow CH, VH : (\Phi_E, \Psi_E)], \ell : c\ \mathsf{complete}}\ (\Psi(\ell) = c)$$

Executable formation always produces a well formed executable.

Theorem A.2 (Executable Formation Correctness) *If* $\vdash O, \ell : c \overset{\mathsf{prg}}{\leadsto} E$ *then* $\vdash^{c}_{\mathsf{E}} E$*.*

Proof: By inspection of the various typing and linking judgement rules. □

To explain how an executable is executed, I first need to explain the operation of the abstract machine itself and the core language. I shall return to execution in Section A.3.9.

A.3 Core Language

MooTal's core language corresponds to a typed version of an idealised RISC processor. Like F_ω it is a three-tiered system. At the base are term-level constructs like program states, heap values, and small values. The term level constructs are given types, and the types are classified into kinds. MooTal's kinds are used to divide the types into those for different term constructs, but also to include more general type constructors, for example, to express generic data types. Memory in MooTal comes in three forms: the registers, the stack, and the heap. The registers, like a conventional machine, can only store small values such as 32-bit integers or pointers. Larger values must be placed on the stack or in the heap. Therefore, MooTal has three kinds of values: word values, stacks, and heap values; there are three kinds of types that correspond to these different kinds of values. The parts of MooTal specific to supporting object oriented features are separated out into their own section.

A.3.1 Kinds

Kinds classify types into those for word values, heaps values, stack values, and type functions (for defining parameterised types). The kinds are:

$$\kappa ::= \mathsf{T} \mid \mathsf{M} \mid \mathsf{S} \mid \kappa_1 \to \kappa_2$$

Where T is for word value types, M for heap value types, S for stack types, and $\kappa_1 \to \kappa_2$ for type functions taking arguments of kind κ_1 to results of kind κ_2.

A.3.2 Type Constructors

MooTal has a variety of different type constructors. Most of them will be described in conjunction with the term constructs they are naturally associated with. However, some type constructors are not directly related to terms but have more to do with typing itself. This section will describe these type constructors and typing judgements about type constructors. Type constructors c have three important judgements associated with them: type kinding $\Phi; \Delta \vdash_{\mathsf{tc}} c : \kappa$ asserting that c has kind κ, type equality $\Phi; \Delta \vdash c_1 = c_2 : \kappa$ asserting that c_1 and c_2 are equal type constructors at kind κ, and subtyping $\Phi; \Delta; B \vdash c_1 \leq c_2 : \kappa$ asserting that c_1 is a subtype of c_2 at kind κ. The context component Φ is a type heap typing; $\Delta ::= \alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n$ is a core language typing context, and lists the type constructor variables in scope and their kinds; B is a set of type constructor bounds, and is described in Section A.3.7.

To reduce the number of rules, I will present only a kinding rule, and this will implicitly define a congruence equality rule. There are also additional equality rules for β-reductions. An example of how to form the congruence from the kinding rule is:

$$\frac{\Phi; \Delta, \alpha{:}\kappa_1 \vdash_{\mathsf{tc}} c : \kappa_2}{\Phi; \Delta \vdash_{\mathsf{tc}} \lambda\alpha{:}\kappa_1.c : \kappa_1 \to \kappa_2} \quad \Rightarrow \quad \frac{\Phi; \Delta, \alpha{:}\kappa_1 \vdash c_1 = c_2 : \kappa_2}{\Phi; \Delta \vdash \lambda\alpha{:}\kappa_1.c_1 = \lambda\alpha{:}\kappa_1.c_2 : \kappa_1 \to \kappa_2}$$

The congruence rule will be named (eq-X) where (X) is the name of the kinding rule. Equality is symmetric and transitive; subtyping is reflexive and transitive.

$$(\text{eq-sym})\ \frac{\Phi; \Delta \vdash c_2 = c_1 : \kappa}{\Phi; \Delta \vdash c_1 = c_2 : \kappa}$$

$$(\text{eq-trans})\ \frac{\Phi; \Delta \vdash c_1 = c_2 : \kappa \quad \Phi; \Delta \vdash c_2 = c_3 : \kappa}{\Phi; \Delta \vdash c_1 = c_3 : \kappa}$$

$$(\text{sub-ref})\ \frac{\Phi; \Delta \vdash c_1 = c_2 : \kappa}{\Phi; \Delta; B \vdash c_1 \leq c_2 : \kappa}$$

$$(\text{sub-trans})\ \frac{\Phi; \Delta; B \vdash c_1 \leq c_2 : \kappa \quad \Phi; \Delta; B \vdash c_2 \leq c_3 : \kappa}{\Phi; \Delta; B \vdash c_1 \leq c_3 : \kappa}$$

The type constructors not described elsewhere are:

$$c ::= \alpha \mid \ell \mid \lambda\alpha{:}\kappa.c \mid c_1\ c_2 \mid \forall t.c$$

A type constructor variable has the kind given by the core language context, and is a subtype of itself.

$$(\text{tcvar})\ \frac{}{\Phi; \Delta \vdash_{\mathsf{tc}} \alpha : \kappa}\ (\Delta(\alpha) = \kappa) \quad (\text{sub-tcvar})\ \frac{}{\Phi; \Delta; B \vdash \alpha \leq \alpha : \kappa}\ (\Delta(\alpha) = \kappa)$$

A type name has the kind given by the type heap typing, and is a subtype of itself.

$$(\text{tcname})\ \frac{}{\Phi; \Delta \vdash_{\mathsf{tc}} \ell : \kappa}\ (\Phi(\ell) = \kappa) \quad (\text{sub-tcname})\ \frac{}{\Phi; \Delta; B \vdash \ell \leq \ell : \kappa}\ (\Phi(\ell) = \kappa)$$

Type functions and their applications follow standard simply typed lambda calculus rules. In addition to the congruence rule, there is the usual β rule.

$$(\text{tcabs})\ \frac{\Phi; \Delta, \alpha{:}\kappa_1 \vdash_{\mathsf{tc}} c : \kappa_2}{\Phi; \Delta \vdash_{\mathsf{tc}} \lambda\alpha{:}\kappa_1.c : \kappa_1 \to \kappa_2}$$

$$(\text{tcapp})\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}} c_1 : \kappa_2\to\kappa_1 \quad \Phi;\Delta\vdash_{\mathsf{tc}} c_2:\kappa_2}{\Phi;\Delta\vdash_{\mathsf{tc}} c_1\ c_2 : \kappa_1}$$

$$(\text{tfn}\beta)\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}} (\lambda\kappa_1{:}c_1')\ c_2 : \kappa_2}{\Phi;\Delta\vdash (\lambda\kappa_1{:}c_1')\ c_2 = c_1'\{\alpha:=c_2\} : \kappa_2}$$

A type function is a subtype of another type function when the bodies are in the same relation. Subtyping for type function application is a little restrictive, the function part may be a subtype, but the arguments are required to be equal. A less restrictive rule would require knowing that the functions were monotonic or antimonotonic, thus requiring the introduction of polarised type functions.

$$(\text{sub-tcabs})\ \frac{\Phi;\Delta,\alpha{:}\kappa_1;B\vdash c_1\le c_2:\kappa_2}{\Phi;\Delta;B\vdash \lambda\alpha{:}\kappa_1.c_1\le\lambda\alpha{:}\kappa_1.c_2:\kappa_1\to\kappa_2}$$

$$(\text{sub-tcapp})\ \frac{\Phi;\Delta;B\vdash c_{11}\le c_{21}:\kappa_2\to\kappa_1 \quad \Phi;\Delta\vdash c_{12}=c_{22}:\kappa_2}{\Phi;\Delta;B\vdash c_{11}\ c_{12}\le c_{21}\ c_{22}:\kappa_1}$$

A universally quantified type has the form $\forall t.c$ where t is a type-variable definition. There are two forms of type-variable definitions: $\alpha : \kappa$ and $\alpha : \kappa \le c$. The first form is called unbounded, α is the abstracted type variable, and κ its kind. The other form is called F-bounded and is described in Section A.3.7. With an unbounded form, the type is well formed when the body is well formed assuming the abstracted variable has the given kind. A unbounded universal type is a subtype of another unbounded universal type if the bodies are in the same relation.

$$(\text{allu})\ \frac{\Phi;\Delta,\alpha{:}\kappa_1\vdash_{\mathsf{tc}} c:\kappa_2}{\Phi;\Delta\vdash_{\mathsf{tc}}\forall\alpha{:}\kappa_1.c:\kappa_2} \qquad (\text{sub-allu})\ \frac{\Phi;\Delta,\alpha{:}\kappa_1;B\vdash c_1\le c_2:\kappa_2}{\Phi;\Delta;B\vdash\forall\alpha{:}\kappa_1.c_1\le\forall\alpha{:}\kappa_1.c_2:\kappa_2}$$

A.3.3 Program States

A MooTal program state represents the state of an idealised RISC processor and its associated memory system. This state includes the program counter and the memory, which is conceptually divided into three areas: the processors registers, the dynamic memory heap, and the stack. As it is technically smoother, the stack is modelled by a special register sp, and the program counter is modelled by the instructions it points to. Thus a program state P is a quadruple (CH, VH, R, I) consisting of a type heap, a value heap, a register file, and the current instruction sequence. A program state is well formed when its components are:

$$(\text{prog})\ \frac{\begin{array}{c}\vdash_{\mathsf{Int}} (\Phi,\Psi) \quad \Phi\vdash_{\mathsf{CH}} CH:\Phi \quad \Phi;CH;\Psi\vdash_{\mathsf{VH}} VH:\Psi \\ \Phi;CH;\Psi\vdash_{\mathsf{R}} R:\Gamma \quad \Phi;CH;\Psi;\epsilon;\epsilon;\Gamma\vdash_{\mathsf{I}} I\end{array}}{\vdash_{\mathsf{P}} (CH, VH, R, I)}$$

A register file R has the form $\{\mathsf{sp}\mapsto S, r_1\mapsto w_1,\ldots,r_n\mapsto w_n\}$ mapping registers r to word values w, and mapping a special register sp to a stack S. Register files are given register files types Γ have the form $\{\mathsf{sp}{:}c, r_1{:}c_1,\ldots,r_n{:}c_n\}$ mapping registers to word types and sp to a stack type. A register file type is well formed when its types are of the appropriate kinds:

$$(\text{wf-regfile})\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}} c:\mathsf{S} \quad \Phi;\Delta\vdash_{\mathsf{tc}} c_i:\mathsf{T}}{\Phi;\Delta\vdash_{\mathsf{RT}}\{\mathsf{sp}{:}c, r_1{:}c_1,\ldots,r_n{:}c_n\}}$$

There is a congruence equality rule formed as described in the section on basic types. A register file type is a subtype of another register file type if it defines at least the same registers with at least as specific types:

$$\text{(sub-regfile)}\ \frac{\Phi;\Delta;B\vdash c\leq c':\mathsf{S}\quad \Phi;\Delta;B\vdash c_i\leq c_i':\mathsf{T}\quad \Phi;\Delta\vdash_{\mathsf{tc}} c_i:\mathsf{T}}{\Phi;\Delta;B\vdash_{\mathsf{RT}}\{\mathsf{sp}{:}c,r_1{:}c_1,\ldots,r_m{:}c_m\}\leq\{\mathsf{sp}{:}c',r_1{:}c_1',\ldots,r_n{:}c_n'\}}\ (*)$$

Where $(*)$ is $m \geq n$. A register file has a type when its components have the corresponding types.

$$\text{(regfile)}\ \frac{\Phi;CH;\Psi\vdash_{\mathsf{S}} S:c\quad \Phi;CH;\Psi;\epsilon;\epsilon\vdash w_i:c_i}{\Phi;CH;\Psi\vdash_{\mathsf{R}}\{\mathsf{sp}\mapsto S,r_1\mapsto w_1,\ldots,r_n\mapsto w_n\}:\{\mathsf{sp}{:}c,r_1{:}c_1,\ldots,r_n{:}c_n\}}$$

Stacks S are either empty, written se, or consist of a word w pushed onto a stack S, written $w :: S$. The stack types are the empty stack type se, the type $c_1 :: c_2$ describing a word of type c_1 pushed onto a stack of type c_2, and the type $c_1 \circ c_2$ describing the concatenation of two stacks of types c_1 and c_2. The following well formedness and subtyping rules hold:

$$\text{(se)}\ \frac{}{\Phi;\Delta\vdash_{\mathsf{tc}}\mathsf{se}:\mathsf{S}}$$

$$\text{(cons)}\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}} c_1:\mathsf{T}\quad \Phi;\Delta\vdash_{\mathsf{tc}} c_2:\mathsf{S}}{\Phi;\Delta\vdash_{\mathsf{tc}} c_1::c_2:\mathsf{S}}$$

$$\text{(append)}\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}} c_1:\mathsf{S}\quad \Phi;\Delta\vdash_{\mathsf{tc}} c_2:\mathsf{S}}{\Phi;\Delta\vdash_{\mathsf{tc}} c_1\circ c_2:\mathsf{S}}$$

$$\text{(sub-se)}\ \frac{}{\Phi;\Delta;B\vdash\mathsf{se}\leq\mathsf{se}:\mathsf{S}}$$

$$\text{(sub-cons)}\ \frac{\Phi;\Delta;B\vdash c_{11}\leq c_{21}:\mathsf{T}\quad \Phi;\Delta;B\vdash c_{12}\leq c_{22}:\mathsf{S}}{\Phi;\Delta;B\vdash c_{11}::c_{12}\leq c_{21}::c_{22}:\mathsf{S}}$$

$$\text{(sub-append)}\ \frac{\Phi;\Delta;B\vdash c_{11}\leq c_{21}:\mathsf{S}\quad \Phi;\Delta;B\vdash c_{12}\leq c_{22}:\mathsf{S}}{\Phi;\Delta;B\vdash c_{11}\circ c_{12}\leq c_{21}\circ c_{22}:\mathsf{S}}$$

$$\text{(stk-se)}\ \frac{}{\Phi;CH;\Psi\vdash_{\mathsf{S}}\mathsf{se}:\mathsf{se}}$$

$$\text{(stk-cons)}\ \frac{\Phi;CH;\Psi;\epsilon;\epsilon\vdash w:c_1\quad \Phi;CH;\Psi\vdash_{\mathsf{S}} S:c_2}{\Phi;CH;\Psi\vdash_{\mathsf{S}} w::S:c_1::c_2}$$

In addition there are a number of computational equality rules for the append stack type:

$$\text{(stk}\beta1)\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}} c:\mathsf{S}}{\Phi;\Delta\vdash\mathsf{se}\circ c=c:\mathsf{S}}\qquad \text{(stk}\beta3)\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}} c:\mathsf{S}}{\Phi;\Delta\vdash c\circ\mathsf{se}=c:\mathsf{S}}$$

$$\text{(stk}\beta2)\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}}(c_1::c_2)\circ c_3:\mathsf{S}}{\Phi;\Delta\vdash(c_1::c_2)\circ c_3=c_1::(c_2\circ c_3):\mathsf{S}}$$

$$\text{(stk}\beta4)\ \frac{\Phi;\Delta\vdash_{\mathsf{tc}}(c_1\circ c_2)\circ c_3:\mathsf{S}}{\Phi;\Delta\vdash(c_1\circ c_2)\circ c_3=c_1\circ(c_2\circ c_3):\mathsf{S}}$$

The size of a stack will be important to the typing rules and proof of soundness:

$$|S| = \begin{cases} 0 & S=\mathsf{se} \\ 1+|S'| & S=w::S' \end{cases}$$

A.3.4 Heap Values

Heap values h have the form $\mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\hat{h}$ consisting of two parts: a type annotation and a heap value proper $\hat{h}$. The first part of the annotation $\mathsf{tag}(c)$ has to do with the tagging mechanism and is described in Section A.3.7. The second part abstracts type variables that the heap value proper are parameterised over. There are two forms of heap values, $\mathsf{code}\ I$ for code and $\langle w_1, \ldots, w_n\rangle$ for data. A code heap value is simply a sequence of instructions I. A data heap value is a tuple of word values. Heap values are given memory types, which also come in two forms. A code type $\mathsf{code}\ \Gamma$ specifies the types the registers must have at the start of the code's instruction sequence, and can be thought of as a code precondition. A tuple type $\langle c_1^{\phi_1}, \ldots, c_n^{\phi_n}\rangle$ specifies the types of the words in the tuples as well as their variance (see Abadi and Cardelli [AC96] for a discussion of variances). There are four variances ϕ: covariant or read only $+$, contravariant or write only $-$, invariant or read write $\circ$, and uninitialised 0.

The formation rules for heap value types are straightforward:

$$(\text{code})\ \frac{\Phi; \Delta \vdash_{\mathsf{RT}} \Gamma}{\Phi; \Delta \vdash_{\mathsf{tc}} \mathsf{code}\ \Gamma : \mathsf{M}}$$

$$(\text{tuple})\ \frac{\Phi; \Delta \vdash_{\mathsf{tc}} c_i : \mathsf{T}}{\Phi; \Delta \vdash_{\mathsf{tc}} \langle c_1^{\phi_1}, \ldots, c_n^{\phi_n}\rangle : \mathsf{M}}$$

For similar reasons to function types, the code type is contravariant:

$$(\text{sub-code})\ \frac{\Phi; \Delta; B \vdash_{\mathsf{RT}} \Gamma_2 \leq \Gamma_1}{\Phi; \Delta; B \vdash \mathsf{code}\ \Gamma_1 \leq \mathsf{code}\ \Gamma_2 : \mathsf{M}}$$

Tuple types have both right extension breadth subtyping and depth subtyping given by the usual variance rules:

$$(\text{sub-tuple})\ \frac{\Phi; \Delta; B \vdash c_i^{\phi_i} \leq c_i'^{\phi_i} : \mathsf{T} \quad \Phi; \Delta \vdash_{\mathsf{tc}} c_i : \mathsf{T}}{\Phi; \Delta; B \vdash \langle c_1^{\phi_1}, \ldots, c_m^{\phi_m}\rangle \leq \langle c_1'^{\phi_1'}, \ldots, c_n'^{\phi_n'}\rangle : \mathsf{M}}\ (m \geq n)$$

$$(\text{cov})\ \frac{\Phi; \Delta; B \vdash c_1 \leq c_2 : \kappa}{\Phi; \Delta; B \vdash c_1^{\phi} \leq c_2^{+} : \kappa}\ (\phi \leq +) \qquad (\text{con})\ \frac{\Phi; \Delta; B \vdash c_2 \leq c_1 : \kappa}{\Phi; \Delta; B \vdash c_1^{\phi} \leq c_2^{-} : \kappa}\ (\phi \leq -)$$

$$(\text{inv})\ \frac{\Phi; \Delta \vdash c_1 = c_2 : \kappa}{\Phi; \Delta; B \vdash c_1^{\circ} \leq c_2^{\circ} : \kappa} \qquad (\text{unin})\ \frac{\Phi; \Delta \vdash c_1 = c_2 : \kappa}{\Phi; \Delta; B \vdash c_1^{\phi} \leq c_2^{0} : \kappa}\ (\phi \leq 0)$$

Where:

$\leq$	$+$	$-$	$\circ$	0
$+$	T	F	F	F
$-$	F	T	F	F
$\circ$	T	T	T	T
0	F	F	F	T

Heap values proper are given heap value types in an obvious way:

$$(\text{hv-code})\ \frac{\Phi; \Delta \vdash_{\mathsf{RT}} \Gamma \quad \Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{I}} I}{\Phi; CH; \Psi; \Delta; B \vdash_{\hat{\mathsf{h}}} \mathsf{code}\ I : \mathsf{code}\ \Gamma}$$

$$(\text{hv-tuple})\ \frac{\Phi; CH; \Psi; \Delta; B \vdash w_i : c_i^{\phi_i}}{\Phi; CH; \Psi; \Delta; B \vdash_{\hat{\mathsf{h}}} \langle w_1, \ldots, w_n\rangle : \langle c_1^{\phi_1}, \ldots, c_n^{\phi_n}\rangle}$$

The rules for word values having variant types are a little subtle. First, a well typed word value may have any variant of that type. Second, to avoid the interactions between polymorphism and mutability, mutable variants must be closed. Third, a junk value $?_c$ is in the uninitialised variant.

$$\text{(hv-init)}\ \frac{\begin{array}{c}\phi \neq + \\ \Rightarrow \\ \Phi; \epsilon \vdash_{\mathsf{tc}} c : \mathsf{T} \qquad \Phi; CH; \Psi; \Delta; B \vdash w : c\end{array}}{\Phi; CH; \Psi; \Delta; B \vdash w : c^{\phi}}$$

$$\text{(hv-uninit)}\ \frac{\Phi; \Delta \vdash c_1 = c_2 : \mathsf{T}}{\Phi; CH; \Psi; \Delta; B \vdash ?_{c_1} : c_2^{0}}$$

Heap values themselves are given the types that labels bound to them have, which are word types. The tag and bound parts are explained further in Section A.3.7; the pointer type $*\mathsf{tag}(c_1^{\phi})c_2$ is explained further in the section on small values; the rest is straightforward. If $t_i = \alpha_i{:}\kappa_i(\leq c_i)^?$ then $\Delta(t_1, \ldots, t_n) = \alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n$ and $B(t_1, \ldots, t_n)$ has $\alpha_i \leq c_i$ if $t_i = \alpha_i{:}\kappa_i \leq c_i$. If $\Delta(t_1, \ldots, t_n) = \Delta$ and $B(t_1, \ldots, t_n) = B$ then:

$$\text{(hv)}\ \frac{\Phi; \epsilon \vdash_{\mathsf{tc}} c' : \kappa \quad \Phi; \Delta \vdash_{\mathsf{tc}} c_i : \kappa_i \quad \Phi; CH; \Psi; \Delta; B \vdash_{\hat{\mathsf{h}}} \hat{h} : c}{\Phi; CH; \Psi \vdash_{\mathsf{h}} \mathsf{tag}(c')\Lambda[t_1, \ldots, t_n]\hat{h} : \forall[t_1, \ldots, t_n]{*}\mathsf{tag}(c'^{\circ})c}$$

A.3.5 Small Values

To avoid tedious duplication, the word values are a subset of a broader syntax category, the small values, or operands, v. Values include integers i, value names ℓ, registers r, junk values $?_c$, the nonsense value ns, stack pointers $\mathsf{sptr}(i)$, and typing coercions $\delta(v)$. Junk values represent uninitialised heap memory and are parameterised by the type that will eventually initialise that memory (see the section on heap values and the (i-init) rule for further details). The nonsense value represents uninitialised stack memory. Stack pointers have a zero-based offset from the bottom of the stack. Typing coercions change the type of a value but have no operational effect. Word values w are those small values that do not have a register as a subterm, nor a subterm of the form $\mathsf{unroll}(\mathsf{roll}^c(v))$. The same typing rules are used for word values as for small values, but as the Γ component of the context is not needed, it will often be omitted.

Operationally, small values evaluate to word values. Given a register file R, the meaning of a small value v is given by $\hat{R}(v)$:

$$\hat{R}(v) = \begin{cases} v & v = i, \ell, ?_c, \mathsf{ns}, \mathsf{sptr}(i) \\ R(r) & v = r \\ w & v = \mathsf{unroll}(v'); \hat{R}(v') = \mathsf{roll}^c(w) \\ \delta(\hat{R}(v')) & v = \delta(v') \end{cases}$$

Small values are given word types, which include the integer type int, heap pointer types $*\mathsf{tag}(c_1^{\phi})c_2$, the nonsense type ns, and stack pointer types $\mathsf{sptr}(c)$. There is a subsumption rule (values in a subtype are also in a supertype) for small values:

$$\text{(subsume)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v : c_1 \quad \Phi; \Delta; B \vdash c_1 \leq c_2 : \mathsf{T}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v : c_2}$$

The rules for integers and nonsense are straight forward:

$$\text{(int)}\ \frac{}{\Phi; \Delta \vdash_{\mathsf{tc}} \mathsf{int} : \mathsf{T}} \qquad \text{(sub-int)}\ \frac{}{\Phi; \Delta; B \vdash \mathsf{int} \leq \mathsf{int} : \mathsf{T}}$$

$$\text{(sv-int)}\ \frac{}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash i : \mathsf{int}}$$

$$\text{(ns)}\ \frac{}{\Phi; \Delta \vdash_{\mathsf{tc}} \mathsf{ns} : \mathsf{T}} \qquad \text{(sub-ns)}\ \frac{}{\Phi; \Delta; B \vdash \mathsf{ns} \le \mathsf{ns} : \mathsf{T}}$$

$$\text{(sv-ns)}\ \frac{}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash \mathsf{ns} : \mathsf{ns}}$$

A pointer to a heap value of type c is given the type $*\mathsf{tag}(c'^{\phi})c$ (the tag part is explained in Section A.3.7). A pointer to a stack of type c is given the type $\mathsf{sptr}(c)$. Both pointer types are covariant.

$$\text{(hptr)}\ \frac{\Phi; \Delta \vdash_{\mathsf{tc}} c_1 : \kappa \quad \Phi; \Delta \vdash_{\mathsf{tc}} c_2 : \mathsf{M}}{\Phi; \Delta \vdash_{\mathsf{tc}} *\mathsf{tag}(c_1^{\phi})c_2 : \mathsf{T}}$$

$$\text{(sub-hptr)}\ \frac{\Phi; \Delta; B \vdash c_{11}^{\phi_1} \le c_{21}^{\phi_2} : \kappa \quad \Phi; \Delta; B \vdash c_{12} \le c_{22} : \mathsf{M}}{\Phi; \Delta; B \vdash *\mathsf{tag}(c_{11}^{\phi_1})c_{12} \le *\mathsf{tag}(c_{21}^{\phi_2})c_{22} : \mathsf{T}}$$

$$\text{(sptr)}\ \frac{\Phi; \Delta \vdash_{\mathsf{tc}} c : \mathsf{S}}{\Phi; \Delta \vdash_{\mathsf{tc}} \mathsf{sptr}(c) : \mathsf{T}} \qquad \text{(sub-sptr)}\ \frac{\Phi; \Delta \vdash c_1 \le c_2 : \mathsf{S}}{\Phi; \Delta \vdash \mathsf{sptr}(c_1) \le \mathsf{sptr}(c_2) : \mathsf{T}}$$

Value names have a type given by the value heap typing, which in MooTal is always a pointer type. Stack pointers $\mathsf{sptr}(i)$ have any stack pointer type $\mathsf{sptr}(c)$ where c has i elements. This might seem too weak, but the stack validity conditions in the instruction typing rules complement this rule to produce a sound system.

$$\text{(svname)}\ \frac{}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash \ell : c}\ (\Psi(\ell) = c)$$

$$\text{(sv-sptr)}\ \frac{\Phi; \Delta \vdash_{\mathsf{tc}} c : \mathsf{S}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash \mathsf{sptr}(i) : \mathsf{sptr}(c)}\ (|c| = i)$$

Registers have types given by the register file type. Note that junk values do not have types, only uninitialised variants of types.

$$\text{(reg)}\ \frac{}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash r : c}\ (\Gamma(r) = c)$$

A typing coercions δ changes a type c_1 to c_2 when $\Phi; CH; \Delta; B \vdash \delta : c_1 \Rightarrow c_2$. Given this the rule for typing coercions is clear:

$$\text{(coerce)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v : c_1 \quad \Phi; CH; \Delta; B \vdash \delta : c_1 \Rightarrow c_2}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash \delta(v) : c_2}$$

The coercions are type instantiation $[c]$, roll roll^c, unroll unroll, and pack pack^c. Type instantiations are for universally quantified types; the instantiating type must satisfy the bound. The judgement $\Phi; CH; \Delta; B \vdash c \rhd t \Rightarrow \rho$ asserts that c satisfies the bound in the type-variable definition t and that ρ is a substitution that will do the instantiation when applied to the body of the universal type. For unbounded type-variable definitions this judgement just requires c to have the correct kind (bounded type definitions are discussed in Section A.3.7).

$$\text{(inst)}\ \frac{\Phi; CH; \Delta; B \vdash c \rhd t \Rightarrow \rho}{\Phi; CH; \Delta; B \vdash [c] : \forall t.c' \Rightarrow c'\{\rho\}}$$

$$\text{(satu)}\ \frac{\Phi; \Delta \vdash_{\mathsf{tc}} c : \kappa}{\Phi; CH; \Delta; B \vdash c \rhd \alpha{:}\kappa \Rightarrow \alpha := c}$$

For convenience, an iterated judgement is also useful:

$$\frac{\Phi; CH; \Delta; B \vdash c_i \rhd t_i\{\rho_1, \ldots, \rho_{i-1}\} \Rightarrow \rho_i}{\Phi; CH; \Delta; B \vdash c_1, \ldots, c_n \rhd t_1, \ldots, t_n \Rightarrow \rho_1, \ldots, \rho_n}$$

Roll and unroll convert a recursive type to and from its definition. There are several forms of recursive types, some are equal to their definitions, others are subtypes of their definitions. To aid in stating a rule, two functions are used. The type $\mathrm{unroll}_{CH;B}(c)$ is the unrolled type for c, and $\mathrm{roll}(c)$ is true if rolling to c is permitted. A type name is one of the forms of recursive type and its definition is giving by the constructor heap. The other forms are explained in Section A.3.7.

$$\text{(roll)}\ \frac{}{\Phi; CH; \Delta; B \vdash \mathsf{roll}^{c_1} : c_2 \Rightarrow c_1}\ (\mathrm{unroll}_{CH;B}(c_1) = c_2; \mathrm{roll}(c_1))$$

$$\text{(roll-app)}\ \frac{\Phi; CH; \Delta; B \vdash \mathsf{roll}^{c_1} : c_2 \Rightarrow c_1}{\Phi; CH; \Delta; B \vdash \mathsf{roll}^{c_1\ c} : c_2\ c \Rightarrow c_1\ c}$$

$$\text{(unroll)}\ \frac{}{\Phi; CH; \Delta; B \vdash \mathsf{unroll} : c_1 \Rightarrow c_2}\ (\mathrm{unroll}_{CH;B}(c_1) = c_2)$$

$$\text{(unroll-app)}\ \frac{\Phi; CH; \Delta; B \vdash \mathsf{unroll} : c_1 \Rightarrow c_2}{\Phi; CH; \Delta; B \vdash \mathsf{unroll} : c_1\ c \Rightarrow c_2\ c}$$

$$\mathrm{unroll}_{CH;B}(\ell) = CH(\ell) \qquad \mathrm{roll}(\ell)$$

A.3.6 Instructions

MooTal instruction sequences consist of a number of basic instructions ι followed by a terminal instruction, either $\mathsf{halt}[c]$ or $\mathsf{jmp}\ v$. The execution of a single instruction is modelled by a small step reduction semantics, so if P represents the state of the MooTal machine, $P \mapsto P'$ represents the execution of one instruction, and P' is the new state of the machine. The formal definition of $\mapsto$ appears in Figure A.2 and is explained further below. The notation $\mathsf{ns}^i :: X$ denotes $\underbrace{\mathsf{ns} :: \cdots :: \mathsf{ns}}_{i} :: X$.

The basic instructions have typing rules of the form $\Phi; CH; \Psi; \Delta_1; B_1; \Gamma_1 \vdash_{\mathsf{i}} \iota : \Delta_2; B_2; \Gamma_2$, asserting that in context $\Phi; CH; \Psi; \Delta_1; B_1; \Gamma_1$ instruction ι is well formed and results in a context $\Phi; CH; \Psi; \Delta_2; B_2; \Gamma_2$ for the next instruction. Note that $\Phi; CH; \Psi$ never change—the type and value heaps are global. Instruction sequences are typed by the judgement $\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{I}} I$, which asserts that I is well formed. The instructions are:

$$\begin{array}{rcl}
I & ::= & \iota; I \mid \mathsf{halt}[c] \mid \mathsf{jmp}\ v \\
\iota & ::= & aop\ \ r_d, v_1, v_2 \mid bop\ \ r, v \mid \mathsf{malloc}\ \ r, \mathsf{tag}(c)\langle c_1, \ldots, c_n\rangle \mid \\
& & \mathsf{mov}\ \ r, v \mid \mathsf{mov}\ \ r_d, [r_s + i] \mid \mathsf{mov}\ \ [r_d + i], r_s \mid \\
& & \mathsf{mov}\ \ \mathsf{sp}, \mathsf{sp} + i \mid \mathsf{mov}\ \ r, \mathsf{sp} \mid \mathsf{mov}\ \ \mathsf{sp}, r \mid \mathsf{mov}\ \ r_d, [\mathsf{sp} + i] \mid \mathsf{mov}\ \ [\mathsf{sp} + i], r_s
\end{array}$$

The typing rule for sequencing is straight forward:

$$\text{(i-seq)}\ \frac{\Phi; CH; \Psi; \Delta_1; B_1; \Gamma_1 \vdash_{\mathsf{i}} \iota : \Delta_2; B_2; \Gamma_2 \quad \Phi; CH; \Psi; \Delta_2; B_2; \Gamma_2 \vdash_{\mathsf{I}} I}{\Phi; CH; \Psi; \Delta_1; B_1; \Gamma_1 \vdash_{\mathsf{I}} \iota; I}$$

MooTal programs interact with their environments by halting with a result value in register r1. There is no operational rule for halt because this instruction freezes the machine. Instead,

$$(CH,\, VH, R, I) \mapsto P)$$

I	P
$aop\ \ r_d, v_1, v_2; I'$	$(CH,\, VH, R\{r_d \mapsto i_1 \|\|aop\|\| i_2\}, I')$ $\hat{R}(v_1) = i_1; \hat{R}(v_2) = i_2$
$bop\ \ r, v; I'$	$(CH,\, VH, R, I')$ when not $\|\|bop\|\|i; R(r) = i$
$bop\ \ r, v; I''$	$(CH,\, VH, R, I'\{\vec{\alpha} := \vec{c}\})$ when $\|\|bop\|\|i; R(r) = i$ $\hat{R}(v) = Y(\ell);\ VH(\ell) = \mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\mathsf{code}\ \ I'$
$\mathsf{jmp}\ \ v$	$(CH,\, VH, R, I'\{\vec{\alpha} := \vec{c}\})$ $\hat{R}(v) = Y(\ell);\ VH(\ell) = \mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\mathsf{code}\ \ I'$
$\mathsf{malloc}\ \ r, \mathsf{tag}(c)$ $\quad\langle c_1, \ldots, c_n\rangle; I'$	$(CH,\, VH\{\ell \mapsto \mathsf{tag}(c)\langle ?_{c_1}, \ldots, ?_{c_n}\rangle\}, R\{r \mapsto \ell\}, I')$ $\ell \notin \mathrm{dom}(VH)$
$\mathsf{mov}\ \ r, v; I'$	$(CH,\, VH, R\{r \mapsto \hat{R}(v)\}, I')$
$\mathsf{mov}\ \ r_d, [r_s + i]; I'$	$(CH,\, VH, R\{r_d \mapsto w_i\{\vec{\alpha} := \vec{c}\}\}, I')$ $R(r_s) = Y(\ell);\ VH(\ell) = \mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\langle w_0, \ldots, w_n\rangle$ $0 \leq i \leq n$
$\mathsf{mov}\ \ [r_d + i], r_s; I'$	$(CH,\, VH\{\ell \mapsto \mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]h'\}, R, I')$ $R(r_d) = Y(\ell);\ VH(\ell) = \mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\langle w_0, \ldots, w_n\rangle$ $h' = \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_n\rangle; 0 \leq i \leq n$
$\mathsf{mov}\ \ \mathsf{sp}, \mathsf{sp} + i; I'$	$(CH,\, VH, R\{\mathsf{sp} \mapsto S\})$ $i > 0; R(\mathsf{sp}) = w_1 :: \cdots :: w_i :: S$
$\mathsf{mov}\ \ \mathsf{sp}, \mathsf{sp} - i; I'$	$(CH,\, VH, R\{\mathsf{sp} \mapsto \mathsf{ns}^i :: R(\mathsf{sp})\}); i \geq 0$
$\mathsf{mov}\ \ r, \mathsf{sp}; I'$	$(CH,\, VH, R\{r \mapsto \mathsf{sptr}(\|R(\mathsf{sp})\|)\}, I')$
$\mathsf{mov}\ \ \mathsf{sp}, r; I'$	$(CH,\, VH, R\{\mathsf{sp} \mapsto w_i :: \cdots :: w_1 :: \mathsf{se}\}, I')$ $R(r) = \mathsf{sptr}(i); R(\mathsf{sp}) = w_n :: \cdots :: w_1 :: \mathsf{se}; 0 \leq i \leq n$
$\mathsf{mov}\ \ r_d, [\mathsf{sp} + i]; I'$	$(CH,\, VH, R\{r_d \mapsto w_i\}, I')$ $R(\mathsf{sp}) = w_0 :: \cdots :: w_i :: S; 0 \leq i$
$\mathsf{mov}\ \ [\mathsf{sp} + i], r_s; I'$	$(CH,\, VH, R\{\mathsf{sp} \mapsto w_0 :: \cdots :: w_{i-1} :: R(r_s) :: S\}, I')$ $R(\mathsf{sp}) = w_0 :: \cdots :: w_i :: S; 0 \leq i$

$$\text{Where:}\qquad \begin{array}{rcl} t_i & = & \alpha_i{:}\kappa_i(\leq c_i')^? \\ Y(x) & = & [c_n](\cdots[c_1](x)\cdots) \end{array}$$

Figure A.2: MooTal Operational Semantics program states of the form $(CH, VH, R, \mathsf{halt}[c])$ are called *terminal configurations*. Other irreducible program states are called *stuck configurations*, and one of the purposes of the typing rules is to prevent stuck configurations. The typing rule requires $\mathsf{r1}$ to have type c.

$$\text{(i-halt)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash \mathsf{r1} : c}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{I}} \mathsf{halt}[c]}$$

The jump instruction unconditionally transfers control to the code pointed to by its operand. Operationally, v must be a label for code with all the type variables instantiated; the next program state will begin execution of that code's instructions appropriately instantiated. The typing rule requires v to point to a code type whose register file type matches the current one.

$$\text{(i-jmp)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v : *\mathsf{tag}(c^{\phi})\mathsf{code}\ \Gamma}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{I}} \mathsf{jmp}\ v}$$

MooTal includes a number of three-operand arithmetic instructions $aop\ r_d, v_1, v_2$, where aop stands for some set of arithmetic operators such as addition, subtraction, bitwise and, *etc.* Each arithmetic operator has a meaning $||aop||$, which is a binary operation on the integers. Operationally, r_d is replaced with $||aop||$ applied to the integers v_1 and v_2 evaluate to. The typing rule requires v_1 and v_2 to be integers and updates r_d to be an integer.

$$\text{(i-aop)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v_1 : \mathsf{int} \quad \Phi; CH; \Psi; \Delta; B; \Gamma \vdash v_2 : \mathsf{int}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} aop\ r_d, v_1, v_2 : \Delta; B; \Gamma\{r_d{:}\mathsf{int}\}}$$

There are a number of conditional branching instructions $bop\ r, v$, where bop stands for some set of arithmetic conditions such as equal to zero, greater than zero, *etc.* Each arithmetic condition has a meaning $||bop||$, which is a unary predicate on the integers. Operationally, if the integer in r satisfies the predicate $||bop||$, execution proceeds as in the jump instruction; otherwise, execution proceeds with the next instruction. The typing rule requires r to be an integer, and v to point to code whose register file type matches the current one.

$$\text{(i-bop)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash r : \mathsf{int} \quad \Phi; CH; \Psi; \Delta; B; \Gamma \vdash v : *\mathsf{tag}(c^{\phi})\mathsf{code}\ \Gamma}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} bop\ r, v : \Delta; B; \Gamma}$$

A new heap block is created with the malloc instruction. It takes a tag type c and a list of field types c_i, and creates a new heap block with that tag and with uninitialised fields. A label for this new heap block is placed in r. The tagging mechanism is explained further in Section A.3.7. The typing rule requires the types to be well formed, and updates r to point to a tuple of uninitialised fields. If $\Gamma' = \Gamma\{r{:}*\mathsf{tag}(c^{\circ})\langle c_1^0, \ldots, c_n^0\rangle\}$ then:

$$\text{(i-malloc)}\ \frac{\Phi; \Delta \vdash_{\mathsf{tc}} c : \kappa \quad \Phi; \Delta \vdash_{\mathsf{tc}} c_i : \mathsf{T}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{malloc}\ r, \mathsf{tag}(c)\langle c_1, \ldots, c_n\rangle : \Delta; B; \Gamma'}$$

The move instruction updates a register r with a value v. The typing rule requires v to be well formed, and updates r to reflect v's type.

$$\text{(i-mov)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v : c}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ r, v : \Delta; B; \Gamma\{r{:}c\}}$$

The load instruction $\mathsf{mov}\ r_d, [r_s + i]$ loads the ith field of a tuple pointed to by r_s into r_d, where i is zero based. Operationally, r_s must be a label for data with all type variables instantiated, i must be within the bounds of the tuple, and r_d is updated with the appropriate field appropriately instantiated. The typing rule requires r_d to have a tuple type, i to be within the tuple's bounds, and the field to be a readable variant. It updates r_d with the field's type.

$$\text{(i-load)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash r_s : *\mathsf{tag}(c^{\phi})\langle c_0^{\phi_0}, \ldots, c_n^{\phi_n}\rangle}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ r_d, [r_s + i] : \Delta; B; \Gamma\{r_d{:}c_i\}}\ (0 \le i \le n; \phi_i \le +)$$

The store instruction $\mathsf{mov}\ \ [r_d + i], r_s$ stores r_s into the ith field of a tuple pointed to by r_d, where i is zero based. Operationally, r_d must be a label for data with all type variables instantiated and i must be within the bounds of the tuple. The ith field of the heap block is updated with the current value of r_s. There are two typing rules for two different uses of the instruction. The first is for storing a value into a writable field; it requires r_d to have a tuple type, i to be within the tuple's bounds, the field to be a writable variant, and r_s to have the field's type. The second is for initialising a field; it has the same requirements, except the field must be uninitialised. It updates r_d to reflect a read-write field.

$$\text{(i-store)}\ \frac{\begin{array}{c}\Phi; CH; \Psi; \Delta; B; \Gamma \vdash r_d : *\mathsf{tag}(c^{\phi})\langle c_0^{\phi_0}, \ldots, c_n^{\phi_n}\rangle \\ \Phi; CH; \Psi; \Delta; B; \Gamma \vdash r_s : c_i\end{array}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ [r_d + i], r_s : \Delta; B; \Gamma}\ (0 \le i \le n; \phi_i \le -)$$

If $c' = *\mathsf{tag}(c^{\phi})\langle c_0^{\phi_0}, \ldots, c_{i-1}^{\phi_{i-1}}, c_i^{\circ}, c_{i+1}^{\phi_{i+1}}, \ldots, c_n^{\phi_n}\rangle$ then:

$$\text{(i-init)}\ \frac{\begin{array}{c}\Phi; CH; \Psi; \Delta; B; \Gamma \vdash r_d : *\mathsf{tag}(c^{\phi})\langle c_0^{\phi_0}, \ldots, c_n^{\phi_n}\rangle \\ \Phi; CH; \Psi; \Delta; B; \Gamma \vdash r_s : c_i\end{array}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ [r_d + i], r_s : \Delta; B; \Gamma\{r_d{:}c'\}}\ (0 \le i \le n; \phi_i = 0)$$

Stack space is allocated or deallocated with the instruction $\mathsf{mov}\ \mathsf{sp}, \mathsf{sp} + i$. Operationally, if $i > 0$ then the top i words of the stack are removed, and if $i < 0$ then $-i$ words of nonsense are added to the top of the stack. In the former case, the typing rule requires the stack to have i words, and it removes them; in the latter case, the typing rule simply updates the stack type.

$$\text{(i-sadd)}\ \frac{\Phi; \Delta \vdash c = c_1 :: \cdots :: c_n :: c' :}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ \mathsf{sp}, \mathsf{sp} + i : \Delta; B; \Gamma\{\mathsf{sp}{:}c'\}}\ (\Gamma(\mathsf{sp}) = c; i > 0)$$

$$\text{(i-ssub)}\ \frac{}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ \mathsf{sp}, \mathsf{sp} - i : \Delta; B; \Gamma\{\mathsf{sp}{:}\mathsf{ns}^i :: c\}}\ (\Gamma(\mathsf{sp}) = c; i \ge 0)$$

The current stack pointer can be placed into a register with the instruction $\mathsf{mov}\ \ r, \mathsf{sp}$. Operationally, r is updated with a stack pointer with an offset equal to the current stack size. The typing rule updates r to reflect the current stack type.

$$\text{(i-gsp)}\ \frac{}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ r, \mathsf{sp} : \Delta; B; \Gamma\{r{:}\mathsf{sptr}(c)\}}\ (\Gamma(\mathsf{sp}) = c)$$

The stack pointer can be updated with the instruction $\mathsf{mov}\ \ \mathsf{sp}, r$. Operationally, r must be a stack pointer whose offset is within the current stack bounds, and the stack is truncated to that offset. The typing rule requires r to have a stack pointer type, updates the current stack type to reflect this, and has a stack pointer validity condition. The stack pointer validity condition $\Phi; \Delta \vdash c = c_1 \circ c_2 : \mathsf{S}$ compliments the weak rule for stack pointer values by requiring a stack pointer to have a type c_2 that is a tail of the current stack type c.

$$\text{(i-ssp)}\ \frac{\Phi; \Delta \vdash c = c_1 \circ c_2 : \mathsf{S} \quad \Phi; CH; \Psi; \Delta; B; \Gamma \vdash r : \mathsf{sptr}(c_2)}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ \mathsf{sp}, r : \Delta; B; \Gamma\{\mathsf{sp}{:}c_2\}}\ (\Gamma(\mathsf{sp}) = c)$$

A value on the stack can be loaded into r_d with the instruction $\mathsf{mov}\, r_d, [\mathsf{sp}{+}i]$. Operationally, the current stack must have at least $i+1$ elements, and r_d is updated with the ith element. The typing rule requires the current stack type to have at least $i+1$ elements, and updates r_d to reflect the ith type.

$$\text{(i-sload)}\ \frac{\Phi; \Delta \vdash c = c_0 :: \cdots :: c_i :: c' : \mathsf{S}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ r_d, [\mathsf{sp}+i] : \Delta; B; \Gamma\{r_d{:}c_i\}}\ (i \geq 0; \Gamma(\mathsf{sp}) = c)$$

The register r_s can be stored onto the stack with the instruction $\mathsf{mov}\ \ [\mathsf{sp}+i], r_s$. Operationally, the current stack must have at least $i+1$ elements, and the ith element is updated with the contents of r_s. The typing rule requires the current stack type to have at least $i+1$ elements, r_s to have a type, and updates the ith element to reflect this type. If $\Gamma' = \Gamma\{\mathsf{sp}{:}c_0 :: \cdots :: c_{i-1} :: c_2' :: c_1'\}$ then:

$$\text{(i-sstore)}\ \frac{\begin{array}{c}\Phi; \Delta \vdash c_1 = c_0 :: \cdots :: c_i :: c_1' : \mathsf{S} \\ \Phi; CH; \Psi; \Delta; B; \Gamma \vdash r_s : c_2'\end{array}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ \ [\mathsf{sp}+i], r_s : \Delta; B; \Gamma'}\ (i \geq 0; \Gamma(\mathsf{sp}) = c_1)$$

A.3.7 Object Support

Object support in MooTal consists of recursive bounds, recursive types, the self quantifier, and a tagging mechanism.

A type variable definition of the form $\alpha : \kappa \leq c$ is called an F-bound. As α is bound in c, it specifies a form of recursive inequality. Because MooTal interprets recursion with coercions, a type variable is not a subtype of its bound, but unrolls to it. A bound set is well formed when its variables are defined and its bounds have the appropriate kinds.

$$\text{unroll}_{CH;B}(\alpha) = B(\alpha) \qquad \text{not roll}(\alpha)$$

$$\text{(wf-B)}\ \frac{\Phi; \Delta \vdash_{\mathsf{tc}} c_i : \kappa_i}{\Phi; \Delta \vdash_{\mathsf{B}} \alpha_1 \leq c_1, \ldots, \alpha_n \leq c_n}\ (\Delta(\alpha_i) = \kappa_i)$$

A universally quantified F-bounded type is well formed when its bound and body are well formed in a context that includes the abstracted variable. The subtyping rule makes the bound contravariant and the body covariant (the undecidability results [Pie94] are avoided because a variable is not a subtype of its bound).

$$\text{(allf)}\ \frac{\Phi; \Delta, \alpha{:}\kappa_1 \vdash_{\mathsf{tc}} c_1 : \kappa_1 \quad \Phi; \Delta, \alpha{:}\kappa_1 \vdash_{\mathsf{tc}} c_2 : \kappa_2}{\Phi; \Delta \vdash_{\mathsf{tc}} \forall \alpha{:}\kappa_1 \leq c_1.c_2 : \kappa_2}$$

$$\text{(sub-allf)}\ \frac{\Phi; \Delta, \alpha{:}\kappa_1; B \vdash c_{21} \leq c_{11} : \kappa_1 \quad \Phi; \Delta, \alpha{:}\kappa_1; B, \alpha \leq c_{21} \vdash c_{12} \leq c_{22} : \kappa_2}{\Phi; \Delta; B \vdash \forall \alpha{:}\kappa_1 \leq c_{11}.c_{12} \leq \forall \alpha{:}\kappa_1 \leq c_{21}.c_{22} : \kappa_2}$$

Only recursive types can instantiate an F-bounded type, and the unrolled type must be a subtype of the bound with the rolled type substituted for the variable.

$$\text{(satf)}\ \frac{\Phi; \Delta; B \vdash c_2 \leq c_3\{\alpha := c_1\} : \kappa}{\Phi; CH; \Psi; \Delta; B \vdash c_1 \rhd \alpha{:}\kappa \leq c_3 \Rightarrow \alpha := c_1}\ (\text{unroll}_{CH;B}(c_1) = c_2)$$

Recursive types have the form $\mathsf{rec}\ \alpha{:}\kappa.c$, unroll to $c\{\alpha := \mathsf{rec}\ \alpha{:}\kappa.c\}$, and are rollable. They are well formed when the body is well formed in a context that includes the recursion variable. The usual subtyping rule for recursive types is difficult to express in MooTal, because MooTal does not have ordinary bounds (as opposed to recursive bounds). Rather than add these bounds and the associated machinery, I restrict subtyping for recursive types to the reflexive rule.

$$\text{unroll}_{CH;B}(\mathsf{rec}\ \alpha{:}\kappa.c) = c\{\alpha := \mathsf{rec}\ \alpha{:}\kappa.c\} \qquad \text{roll}(\mathsf{rec}\ \alpha{:}\kappa.c)$$

$$\text{(rec)}\ \frac{\Phi; \Delta, \alpha{:}\kappa \vdash_{\mathsf{tc}} c : \kappa}{\Phi; \Delta \vdash_{\mathsf{tc}} \mathsf{rec}\ \alpha.c : \kappa}$$

A self-quantified type has the form $\mathsf{self}\ \alpha{:}\kappa.c$. Intuitively, a value of type c' can be packed into a self type when c' is a recursive type whose unrolling is a subtype of c with α replaced by c'. A self type is well formed when its body is well formed in a context that includes the quantified variable. Self types are covariant.

$$\text{(self)}\ \frac{\Phi; \Delta, \alpha{:}\kappa \vdash_{\mathsf{tc}} c : \kappa}{\Phi; \Delta \vdash_{\mathsf{tc}} \mathsf{self}\ \alpha.c : \kappa}$$

$$\text{(sub-self)}\ \frac{\Phi; \Delta, \alpha{:}\kappa; B \vdash c_1 \le c_2 : \kappa}{\Phi; \Delta; B \vdash \mathsf{self}\ \alpha.c_1 \le \mathsf{self}\ \alpha.c_2 : \kappa}$$

Self types are introduced by the pack coercion pack^c. A recursive type can be packed into the self type $\mathsf{self}\ \alpha{:}\kappa.c$ when the unrolled type is a subtype of c with α replaced by the rolled type.

$$\text{(pack)}\ \frac{\Phi; \Delta; B \vdash c_2 \le c\{\alpha := c_1\} : \kappa}{\Phi; CH; \Psi; \Delta; B \vdash \mathsf{pack}^{\mathsf{self}\ \alpha:\kappa.c} : c_1 \Rightarrow \mathsf{self}\ \alpha{:}\kappa.c}\ (\text{unroll}_{CH;B}(c_1) = c_2)$$

Self types are eliminated by the unpack instruction $\mathsf{unpack}\ \ \alpha, r, v$ where α is bound to the unknown self type, r gets the unpacked value, and v is the value to unpack. Operationally, v must be a packed rolled value, r is updated by the rolled value, and α is replaced by the type in the rolled value. Formally $(CH, VH, R, I) \mapsto P$ where:

I	P
$\mathsf{unpack}\ \ \alpha, r, v; I'$	$(CH, VH, R\{r \mapsto \mathsf{roll}^c(w)\}, I'\{\alpha := c\})$ $\hat{R}(v) = \mathsf{pack}^{c'}(\mathsf{roll}^c(w))$

The typing rule requires v to have a self type $\mathsf{self}\ \alpha{:}\mathsf{T}.c$, and returns a context with α added with bound τ and r of type α.

$$\text{(i-unpack)}\ \frac{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v : \mathsf{self}\ \alpha{:}\mathsf{T}.c}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{unpack}\ \ \alpha, r, v : \Delta, \alpha{:}\mathsf{T}; B\{\alpha \le c\}; \Gamma\{r{:}\alpha\}}$$

MooTal has a tagging mechanism, whose purpose is to allow run-time values to represent type information and equality of type representatives to cause type refinement. Each heap block has an associated type called a tag type. The heap block and the label bound to it are said to tag that type. In $\mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\hat{h}$, c is the tag type, and in $\mathsf{malloc}\ r, \mathsf{tag}(c)\langle c_1, \ldots, c_n\rangle$, c is the tag type of the newly created heap block. In both cases, the heap block is said to have been created to tag c. The tag types are reflected in the tag component of the pointer types. Labels in the pointer type $*\mathsf{tag}(c^\phi)c'$ are tagging type c, and if c'' is the type the block was created to tag, then ϕ indicates the relationship between c and c''. If $\phi = +$ then c'' is a subtype of c; if $\phi = -$ then c'' is a supertype of c; if $\phi = \circ$ then c'' equals c. The variance 0 is unused; tag types with this variance are harmless but not generally useful. A pointer type is well formed when the tag type is a well formed type of an arbitrary kind. The subtyping rule establishes the above relationships through the usual variance subtyping rules. Note that while the tag type kinds are arbitrary, they must be the same for subtyping.

$$\frac{\Phi; \Delta \vdash_{\mathsf{tc}} c_1 : \kappa \quad \cdots}{\Phi; \Delta \vdash_{\mathsf{tc}} *\mathsf{tag}(c_1^{\phi})c_2 : \mathsf{T}}$$

$$\frac{\Phi; \Delta; B \vdash c_{11}^{\phi_1} \leq c_{21}^{\phi_2} : \kappa \quad \cdots}{\Phi; \Delta; B \vdash *\mathsf{tag}(c_{11}^{\phi_1})c_{12} \leq *\mathsf{tag}(c_{21}^{\phi_2})c_{22} : \mathsf{T}}$$

The typing rules for heap blocks and the malloc instruction check that the tag type is well formed and include it in the pointer type with a $\circ$ variance. The tag type for a heap block must be closed for the same reason that mutable field types must be closed.

$$\frac{\Phi; \epsilon \vdash_{\mathsf{tc}} c : \kappa \quad \cdots}{\Phi; CH; \Psi \vdash_{\mathsf{h}} \mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\hat{h} : \forall[t_1, \ldots, t_n]*\mathsf{tag}(c^{\circ})c'}$$

$$\frac{\Phi; \Delta \vdash_{\mathsf{tc}} c : \kappa \quad \cdots}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{malloc}\ \ r, \mathsf{tag}(c)\langle c_1, \ldots, c_n\rangle : \Delta; B; \Gamma\{r{:}*\mathsf{tag}(c^{\circ})\langle c_1^0, \ldots, c_n^0\rangle\}}$$

MooTal includes the instruction $\mathsf{tagcmp}\ \ v_1, v_2, v_b$ to compare two tags and perform type refinement. Here, v_1 and v_2 must be tags for types of the same kind. If they are equal, that is, if they are the same pointer, control transfers to the code pointed to be v_b, otherwise control continues with the next instruction. Operationally, this behaviour is easy to formalise. If $t_i = \alpha_1{:}\kappa_1(\leq c_i')^?$, $Y(x) = [c_n](\cdots[c_1](x)\cdots)$, $Y_1(x) = [c_{n_1}^1](\cdots[c_1^1](x)\cdots)$, and $Y_2(x) = [c_{n_2}^2](\cdots[c_1^2](x)\cdots)$ then $(CH, VH, R, I) \mapsto P$, where:

I	P
$\mathsf{tagcmp}\ \ v_1, v_2, v_b; I'$	(CH, VH, R, I') when $\hat{R}(v_1) = Y_1(\ell_1)$, $\hat{R}(v_1) = Y_2(\ell_2)$, and $\ell_1 \neq \ell_2$
$\mathsf{tagcmp}\ \ v_1, v_2, v_b; I''$	$(CH, VH, R, I'\{\vec{\alpha} := \vec{c}\})$ when $\hat{R}(v_1) = Y_1(\ell')$ and $\hat{R}(v_1) = Y_2(\ell')$ $\hat{R}(v_b) = Y(\ell)$; $VH(\ell) = \mathsf{tag}(c)\Lambda[t_1, \ldots, t_n]\mathsf{code}\ \ I'$

The typing rules are more complicated. The first type rule is motivated by the asymmetric use of the operation: v_1 is a tag for an unknown type for which more information is sought, and v_2 is a tag for a known type, which will provide that information. The following rule requires that v_1 be a pointer type whose tag is a type variable α and is a subtype of the type the tag was created for. Value v_2 is required to be a tag for a closed type c that is a supertype of the type the tag was created for. Therefore, if v_1 and v_2 point to the same heap block, it must be that α is a subtype of c. Because type variable bounds are recursive, $\alpha \leq c'$ is added to the context to check v_b, where $\mathrm{unroll}_{CH;\epsilon}(c) = c'$. Value v_b must point to code as in the jump instruction.

$$\text{(i-tagcmp1)}\ \frac{\begin{array}{c}\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v_1 : *\mathsf{tag}(\alpha^-)c_1 \\ \Phi; CH; \Psi; \Delta; B; \Gamma \vdash v_2 : *\mathsf{tag}(c^+)c_2 \\ \Phi; \epsilon \vdash_{\mathsf{tc}} c : \kappa \\ \Phi; CH; \Psi; \Delta; B\{\alpha \leq c'\}; \Gamma \vdash v_b : *\mathsf{tag}(c_b^{\phi})\mathsf{code}\ \ \Gamma\end{array}}{\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{i}} \mathsf{tagcmp}\ \ v_1, v_2, v_b : \Delta; B; \Gamma}\ (*)$$

Where $(*)$ is $\Delta(\alpha) = \kappa$ and $\mathrm{unroll}_{CH;\epsilon}(c) = c'$.

While the typing rules described so far are sound, it is difficult to prove this, as the above rule invalidates the type substitution lemma (Lemma A.8). The problem is that if a type is substituted for α, the above rule no longer applies; in fact, no rule applies to the instruction. So for the purpose of proving type soundness, a second rule is included to cover the substitution of a closed type for α in the above rule. This rule requires both v_1 and v_2 to tag closed types c_1 and c_2 with variances as before. In addition, either c_1 is not a subtype of c_2, in which case the two tags are not equal, v_b is not jumped to, and v_b need not type check; or v_b must type check in the same context. Let $C = \Phi; CH; \Psi; \Delta; B; \Gamma$. Then:

$$\text{(i-tagcmp2)}\ \frac{\begin{array}{c} C \vdash v_1 : *\mathsf{tag}(c_1^-)c_1' \\ C \vdash v_2 : *\mathsf{tag}(c_2^+)c_2' \\ \Phi; \epsilon \vdash_{\mathsf{tc}} c_1 : \kappa \\ \Phi; \epsilon \vdash_{\mathsf{tc}} c_2 : \kappa \end{array} \quad \begin{array}{c} \Phi; \epsilon; \epsilon \vdash c_1 \leq c_2 : \kappa \\ \Rightarrow \\ C \vdash v_b : *\mathsf{tag}(c_b^\phi)\mathsf{code}\ \Gamma \end{array}}{C \vdash_{\mathsf{i}} \mathsf{tagcmp}\ v_1, v_2, v_b : \Delta; B; \Gamma}$$

A.3.8 Type Soundness

MooTal's type system is sound. That is, a well-typed program state never gets stuck during execution. This theorem and the key lemmas in its proof are stated below and proven in this appendix.

Lemma A.3 (Derived Judgements)

- *If* $\Phi; \Delta \vdash c_1 = c_2 : \kappa$ *then* $\Phi; \Delta \vdash_{\mathsf{tc}} c_i : \kappa$.
- *If* $\Phi; \Delta \vdash_{\mathsf{B}} B$ *and* $\Phi; \Delta; B \vdash c_1 \leq c_2 : \kappa$ *then* $\Phi; \Delta \vdash_{\mathsf{tc}} c_i : \kappa$.
- *If* $\Phi; \Delta \vdash_{\mathsf{RT}} \Gamma_1 = \Gamma_2$ *then* $\Phi; \Delta \vdash_{\mathsf{RT}} \Gamma_i$.
- *If* $\Phi; \Delta \vdash_{\mathsf{B}} B$ *and* $\Phi; \Delta; B \vdash_{\mathsf{RT}} \Gamma_1 \leq \Gamma_2$ *then* $\Phi; \Delta \vdash_{\mathsf{RT}} \Gamma_i$.
- *If* $\vdash_{\mathsf{Int}} Int_1 \leq Int_2$ *then* $\vdash_{\mathsf{Int}} Int_i$.
- *If* $\Phi \vdash_{\mathsf{VHT}} \Psi_1 \leq \Psi_2$ *then* $\Phi \vdash_{\mathsf{VHT}} \Psi_i$.
- *If* $\Phi \vdash_{\mathsf{CH}} CH : \Phi'$, $\Phi \vdash_{\mathsf{VHT}} \Psi$, *and* $\Phi; CH; \Psi \vdash_{\mathsf{h}} h : c$ *then* $\Phi; \epsilon \vdash_{\mathsf{tc}} c : \mathsf{T}$.
- *If* $\Phi \vdash_{\mathsf{CH}} CH : \Phi'$, $\Phi \vdash_{\mathsf{VHT}} \Psi$, $\Phi; \Delta \vdash_{\mathsf{B}} B$, *and* $\Phi; CH; \Psi; \Delta; B \vdash_{\hat{\mathsf{h}}} \hat{h} : c$ *then* $\Phi; \Delta \vdash_{\mathsf{tc}} c : \mathsf{M}$.
- *If* $\Phi \vdash_{\mathsf{CH}} CH : \Phi'$, $\Phi \vdash_{\mathsf{VHT}} \Psi$, $\Phi; \Delta \vdash_{\mathsf{B}} B$, $\Phi; CH; \Psi; \Delta; B; \Gamma \vdash v : c$, *and* $\Phi; \Delta \vdash_{\mathsf{RT}} \Gamma$ *then* $\Phi; \epsilon \vdash_{\mathsf{tc}} c : \mathsf{T}$.
- *If* $\Phi \vdash_{\mathsf{CH}} CH : \Phi'$, $\Phi; \Delta \vdash_{\mathsf{B}} B$, $\Phi; CH; \Delta; B \vdash \delta : c_1 \Rightarrow c_2$, *and* $\Phi; \Delta \vdash_{\mathsf{tc}} c_1 : \kappa$ *then* $\Phi; \Delta \vdash_{\mathsf{tc}} c_2 : \kappa$.
- *If* $\Phi \vdash_{\mathsf{CH}} CH : \Phi'$, $\Phi \vdash_{\mathsf{VHT}} \Psi$, *and* $\Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma$ *then* $\Phi; \epsilon \vdash_{\mathsf{RT}} \Gamma$.
- *If* $\Phi \vdash_{\mathsf{CH}} CH : \Phi'$, $\Phi \vdash_{\mathsf{VHT}} \Psi$, *and* $\Phi; CH; \Psi \vdash_{\mathsf{S}} S : c$ *then* $\Phi; \epsilon \vdash_{\mathsf{tc}} c : \mathsf{S}$.

Proof: By induction on the derivations and inspection of the typing rules. □

Definition A.1 *A context* $C_1 = \Phi_1; CH_1; \Psi_1; \Delta_1; B_1; \Gamma_1$ *is stronger than a context* $C_2 = \Phi_2; CH_2; \Psi_2; \Delta_2; B_2; \Gamma_2$ *if and only if all of the following hold:*

- $\vdash_{\mathsf{CHT}} \Phi_1 \leq \Phi_2$
- $\forall \ell{:}\kappa = c_2 \in CH_2 : \ell{:}\kappa = c_1 \in CH_1 \wedge \Phi_1; \epsilon \vdash c_1 = c_2 : \kappa$
- $\Phi_1 \vdash_{\mathsf{VHT}} \Psi_1 \leq \Psi_2$
- Δ_2 *is a prefix of* Δ_1
- $\forall \alpha \leq c_2 \in B_2 : \alpha \leq c_1 \in B_1 \wedge \Phi_1; \Delta_1; \epsilon \vdash c_1 \leq c_2 : \Delta_1(\alpha)$
- $\Phi_1; \Delta_1; B_1 \vdash_{\mathsf{RT}} \Gamma_1 \leq \Gamma_2$

Lemma A.4 (Context Strengthening) *If a judgement J is derivable, then the same judgement with a stronger context is also derivable.*

Proof: By induction on the derivations and inspection of the typing rules. □

A type constructor is in *normal form* if it contains no subterms of the form $(\lambda\alpha{:}\kappa.c_1)\ c_2$, $(c_1 :: c_2) \circ c_3$, $(c_1 \circ c_2) \circ c_3$, $\mathsf{se} \circ c$, or $c \circ \mathsf{se}$. Using well known techniques (*e.g.*, [Gal90]), we can prove that every well kinded type constructor is equal to exactly one type constructor in normal form. It follows that two type constructors of the same kind in the same context are equal exactly when their normal forms are syntactically equal.

Lemma A.5 *If $\Phi; \epsilon; \epsilon \vdash c_1 \leq c_2 : \kappa$ and the normal form of c_i is c_i', then $\Phi; \epsilon; \epsilon \vdash c_1' \leq c_2' : \kappa$ by a derivation that does not use the rules (sub-ref) or (sub-trans).*

Proof: By induction on the derviation of $\Phi; \epsilon; \epsilon \vdash c_1 \leq c_2 : \kappa$. If the rule (sub-ref) is used, then the normal forms of c_1 and c_2 are syntactically equal. The desired derivation is built by observing that each kinding rule corresponds to the reflexive case of some subtyping rule. If the rule (sub-trans) is used, the result follows from the induction hypothesis and observing that all rules preserve transitivity. If some other rule is used, observe that the normal form of c_i is the same type constructor applied to the normal forms of c_i's subterms. The result then follows by the same rule and the induction hypothesis. □

Lemma A.6 (Subtyping)

- *If* $\Phi; \epsilon; \epsilon \vdash *\mathsf{tag}(c_{11}^{\phi_1})c_{12} \leq *\mathsf{tag}(c_{21}^{\phi_2})c_{22} : \mathsf{T}$ *then* $\Phi; \epsilon; \epsilon \vdash c_{11}^{\phi_1} \leq c_{21}^{\phi_2} : \kappa$ *and* $\Phi; \epsilon; \epsilon \vdash c_{12} \leq c_{22} : \mathsf{M}$.
- *If* $\Phi; \epsilon; \epsilon \vdash \mathsf{code}\ \Gamma_1 \leq \mathsf{code}\ \Gamma_2 : \mathsf{M}$ *then* $\Phi; \epsilon; \epsilon \vdash_{\mathsf{RT}} \Gamma_1 \leq \Gamma_2$.
- *If* $\Phi; \epsilon; \epsilon \vdash \langle c_1^{\phi_1}, \ldots, c_n^{\phi_n} \rangle \leq \langle {c_1'}^{\phi_1'}, \ldots, {c_n'}^{\phi_n'} \rangle : \mathsf{M}$ *then* $\Phi; \epsilon; \epsilon \vdash c_i^{\phi_i} \leq {c_i'}^{\phi_i'} : \mathsf{T}$.
- *If* $\Phi; \epsilon; \epsilon \vdash c_1^{\phi_1} \leq c_2^{\phi_2} : \kappa$ *and* $\phi_2 \leq +$ *then* $\Phi; \epsilon; \epsilon \vdash c_1 \leq c_2 : \kappa$.
- *If* $\Phi; \epsilon; \epsilon \vdash c_1^{\phi_1} \leq c_2^{\phi_2} : \kappa$ *and* $\phi_2 \leq -$ *then* $\Phi; \epsilon; \epsilon \vdash c_2 \leq c_1 : \kappa$.
- *If* $\Phi; \epsilon; \epsilon \vdash c_1^{\phi_1} \leq c_2^{0} : \kappa$ *then* $\Phi; \epsilon; \epsilon \vdash c_2 \leq c_1 : \kappa$.
- *If* $\Phi; \epsilon; \epsilon \vdash c_1^{\phi_1} \leq c_2^{\phi_2} : \kappa$ *and* $\phi_2 \neq +$ *then* $\phi_1 \neq +$.
- *If* $\Phi; \epsilon; \epsilon \vdash c_1 \leq c_2 : \kappa$ *and* $\mathrm{unroll}_{CH;\epsilon}(c_2)c_2'$ *then* $\mathrm{unroll}_{CH;\epsilon}(c_1)c_1'$ *and* $\Phi; \epsilon; \epsilon \vdash c_1' \leq c_2' : \kappa$.

Proof: By Lemma A.5 and inspection of the rules. □

A concrete stack type is a type constructor satisfying the grammar:

$$s ::= \mathsf{se} \mid c :: s \mid s_1 \circ s_2$$

Note that if $\Phi; \epsilon \vdash s = c : \mathsf{S}$ then c is also an s. For concrete stack types, the size and indices are defined as:

$$\begin{array}{rcl}
|\mathsf{se}| & = & 0 \\
|c :: s| & = & 1 + |s| \\
|s_1 \circ s_2| & = & |s_1| + |s_2| \\
\mathsf{se}[i] & & \text{Undefined} \\
(c_1 :: s_2)[i] & = & \left\{ \begin{array}{ll} c_1 & i = 1 \\ s_2[i-1] & i > 1 \end{array} \right. \\
(s_1 \circ s_2)[i] & = & \left\{ \begin{array}{ll} s_1[i] & i \leq |s_1| \\ s_2[i - |s_1|] & i > |s_1| \end{array} \right.
\end{array}$$

Lemma A.7 (Stack Equality) *$\Phi; \epsilon \vdash s_1 = s_2 : \mathsf{S}$ if and only if $|s_1| = |s_2|$ and $\forall 1 \leq i \leq |s_1|$: $\Phi; \epsilon \vdash s_1[i] = s_2[i] : \mathsf{T}$*

Proof: ($\Rightarrow$) By induction on the derivation.

($\Leftarrow$) Follows by rules (eq-sym), (eq-trans), (eq-cons), and (eq-se) from $\Phi; \epsilon \vdash s = s[1] :: \cdots :: s[|s|] :: \mathsf{se} : \mathsf{S}$. The latter is by induction on the structure of s:

case $s = \mathsf{se}$: Immediate.

case $s = c :: s'$: By the induction hypothesis $\Phi; \epsilon \vdash s' = s'[1] :: \cdots :: s'[|s'|] :: \mathsf{se} : \mathsf{S}$. By definition, the latter equals $s[2] :: \cdots :: s[|s|] :: \mathsf{se}$. The result follows by rule (eq-cons).

case $s = s_1 \circ s_2$: By the induction hypothesis $\Phi; \epsilon \vdash s_1 = s_1[1] :: \cdots :: s_1[|s_1|] :: \mathsf{se} : \mathsf{S}$ and $\Phi; \epsilon \vdash s_2 = s_2[1] :: \cdots :: s_2[|s_2|] :: \mathsf{se} : \mathsf{S}$. By the (eq-append) rule $\Phi; \epsilon \vdash s = (s_1[1] :: \cdots :: s_1[|s_1|] :: \mathsf{se}) \circ (s_2[1] :: \cdots :: s_2[|s_2|] :: \mathsf{se}) : \mathsf{S}$. By repeated use of the (stkβ2) and (eq-trans) rules, $\Phi; \epsilon \vdash s = s_1[1] :: \cdots :: s_1[|s_1|] :: (\mathsf{se} \circ (s_2[1] :: \cdots :: s_2[|s_2|] :: \mathsf{se})) : \mathsf{S}$. The result follows by the (stkβ1) rule, repeated use of the (eq-cons) rule, and the (eq-trans) rule.

□

Lemma A.8 (Type Substitution) *Let $B' = (B - \alpha)\{\alpha := c\}$. If $\Phi; \epsilon \vdash_{\mathsf{tc}} c : \kappa$ and $B(\alpha) = c'$ implies $\Phi; \epsilon; \epsilon \vdash \mathrm{unroll}_{CH;\epsilon}(c) \leq c'\{\alpha := c\} : \kappa$, then:*

- $\Phi; \alpha{:}\kappa, \Delta \vdash_{\mathsf{tc}} c' : \kappa'$ *implies* $\Phi; \Delta \vdash_{\mathsf{tc}} c'\{\alpha := c\} : \kappa'$
- $\Phi; \alpha{:}\kappa, \Delta \vdash c_1 = c_2 : \kappa'$ *implies* $\Phi; \Delta \vdash c_1\{\alpha := c\} = c_2\{\alpha := c\} : \kappa'$
- $\Phi; \alpha{:}\kappa, \Delta; B \vdash c_1 \leq c_2 : \kappa'$ *implies* $\Phi; \Delta; B' \vdash c_1\{\alpha := c\} \leq c_2\{\alpha := c\} : \kappa'$
- $\Phi; \alpha{:}\kappa, \Delta \vdash_{\mathsf{RT}} \Gamma$ *implies* $\Phi; \Delta \vdash_{\mathsf{RT}} \Gamma\{\alpha := c\}$
- $\Phi; \alpha{:}\kappa, \Delta \vdash_{\mathsf{RT}} \Gamma_1 = \Gamma_2$ *implies* $\Phi; \Delta \vdash_{\mathsf{RT}} \Gamma_1\{\alpha := c\} = \Gamma_2\{\alpha := c\}$

- $\Phi; \alpha{:}\kappa, \Delta; B \vdash_{\mathsf{RT}} \Gamma_1 \leq \Gamma_2$ *implies* $\Phi; \Delta; B' \vdash_{\mathsf{RT}} \Gamma_1\{\alpha := c\} \leq \Gamma_2\{\alpha := c\}$
- $\Phi; CH; \Psi; \alpha{:}\kappa, \Delta; B; \Gamma \vdash v : c'$ *implies* $\Phi; CH; \Psi; \Delta; B'; \Gamma\{\alpha := c\} \vdash v\{\alpha := c\} : c'\{\alpha := c\}$
- $\Phi; CH; \Psi; \alpha{:}\kappa, \Delta_1; B_1; \Gamma_1 \vdash_{\mathsf{i}} \iota : \alpha{:}\kappa, \Delta_2; B_2; \Gamma_2$ *implies* $\Phi; CH; \Psi; \Delta_1; (B_1{-}\alpha)\{\alpha := c\}; \Gamma_1\{\alpha := c\} \vdash_{\mathsf{i}} \iota\{\alpha := c\} : \Delta_2; (B_2{-}\alpha)\{\alpha := c\}; \Gamma_2\{\alpha := c\}$
- $\Phi; CH; \Psi; \alpha{:}\kappa, \Delta; B; \Gamma \vdash_{\mathsf{I}} I$ *implies* $\Phi; CH; \Psi; \Delta; B'; \Gamma\{\alpha := c\} \vdash_{\mathsf{I}} I\{\alpha := c\}$

Proof: By induction on the derivations. The only interesting case is for the rule (i-tagcmp1). In this case the result follows by (i-tagcmp2). To show the implication hypothesis of the rule, assume $\Phi; \epsilon; \epsilon \vdash c \leq c_2 :$ and $\Phi; CH; \alpha{:}\kappa; \Delta; B\{\alpha \leq c_2'\}\Gamma \vdash v_b : *\mathsf{tag}(c_b^{\phi})\mathsf{code}\ \Gamma$ where $\mathrm{unroll}_{CH;\epsilon}(c_2)c_2'$. By Subtyping $\Phi; \epsilon; \epsilon \vdash \mathrm{unroll}_{CH;\epsilon}(c) \leq c_2' : \kappa$, so by the induction hypothesis $\Phi; CH; \Delta; B'; \Gamma\{\alpha := c\} \vdash v_b\{\alpha := c\} : *\mathsf{tag}(c_b\{\alpha := c\}^{\phi})\Gamma\{\alpha := c\}$ as required. □

Inspection of the rules for coercions reveals that the only forms for c where $\Phi; CH; \Delta; B \vdash \mathsf{roll}^c : c' \Rightarrow c$ are ℓ, $\mathsf{rec}\ \alpha{:}\kappa.c''$, and $c_1\ c_2$ where c_1 also has one of these forms. Further inspection reveals that such types have only trivial subtyping rules, that is, they are subtypes only of types equal to them. Thus, between a use of the (coerce) rule for $\delta = \mathsf{roll}^c$ and a use of the (coerce) rule for $\delta = \mathsf{unroll}$, only trivial uses of the (subsume) rule can appear.

Lemma A.9 *If* $\Phi; CH; \epsilon; \epsilon \vdash \mathsf{roll}^{c'} : c_1 \Rightarrow c_2$, $\Phi; \epsilon \vdash c_2 = c_3 : \kappa$, *and* $\Phi; CH; \epsilon; \epsilon \vdash \mathsf{unroll} : c_3 \Rightarrow c_4$ *then* $\Phi; \epsilon \vdash c_1 = c_4 : \kappa$.

Proof: By induction on the derivations of $\Phi; CH; \epsilon; \epsilon \vdash \mathsf{roll}^{c'} : c_1 \Rightarrow c_2$ and $\Phi; CH; \epsilon; \epsilon \vdash \mathsf{unroll} : c_2 \Rightarrow c_3$, inspection of the rules, and the observation that $\mathrm{unroll}_{CH;\epsilon}(c)$ is a partial function of c. □

Lemma A.10 ($\hat{R}$ Typing) *If both* $\Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma$ *and* $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v : c$ *then* $\Phi; CH; \Psi; \epsilon; \epsilon \vdash \hat{R}(v) : c$.

Proof: By induction on the derivation of $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v : c$. If the last rule used was (reg), then $v = r$, $\hat{R} = R(r)$, and $c = \Gamma(r)$. By inspection of the (regfile) rule, $\Phi; CH; \Psi; \epsilon; \epsilon \vdash R(r) : \Gamma(r)$ as required. If the last rule used was (subsume), then $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v : c'$ and $\Phi; \Delta; B \vdash c' \leq c : \mathsf{T}$. By the induction hypothesis $\Phi; CH; \Psi; \epsilon; \epsilon \vdash \hat{R}(v) : c'$, and the result follows by the (subsume) rule. If the last rule used was (coerce), then $v = \delta(v')$, $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v' : c'$, and $\Phi; CH; \epsilon; \epsilon \vdash \delta : c' \Rightarrow c$. By the induction hypothesis $\Phi; CH; \Psi; \epsilon; \epsilon \vdash \hat{R}(v') : c'$. There are two subcases. If $\delta = \mathsf{unroll}$ and $\hat{R}(v') = \mathsf{roll}^{c''}(w)$ then by reasoning above, only trivial (subsume) rules can be between the (coerce) rule for $\mathsf{roll}^{c''}$ and the (coerce) rule for unroll. Therefore, it must be that $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash w : c'''$, $\Phi; CH; \epsilon; \epsilon \vdash \mathsf{roll}^{c''} : c''' \Rightarrow c''$, and $\Phi; \epsilon \vdash c'' = c' : \mathsf{T}$. By Lemma A.9, $\Phi; \epsilon \vdash c''' = c : \mathsf{T}$ and the result follows by the (sub-ref) and (subsume) rules. Otherwise, the result follows by the (coerce) rule and the fact that $\hat{R}(\delta(v')) = \delta(\hat{R}(v'))$. If any other rule was used then $\hat{R}(v) = v$ and the result is immediate. □

Lemma A.11 (Register File Update)

- *If* $\Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma$ *and* $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w : c$ *then* $\Phi; CH; \Psi \vdash R\{r \mapsto w\} : \Gamma\{r{:}c\}$.
- *If* $\Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma$ *and* $\Phi; CH; \Psi \vdash_{\mathsf{S}} S : c$ *then* $\Phi; CH; \Psi \vdash R\{\mathsf{sp} \mapsto s\} : \Gamma\{\mathsf{sp}{:}c\}$.

Proof: By inspection of the rule (regfile). □

Lemma A.12 (Heap Extension) *If* $\Phi \vdash_{\mathsf{VHT}} \Psi$*,* $\Phi; \epsilon \vdash_{\mathsf{tc}} c : \mathsf{T}$*, and* $\ell \notin \mathrm{dom}(\Psi)$ *then:*

- $\Phi \vdash_{\mathsf{VHT}} \Psi\{\ell{:}c\}$
- $\Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi$ *and* $\Phi; CH; \Psi\{\ell{:}c\} \vdash_{\mathsf{h}} h : c$ *imply* $\Phi; CH; \Psi\{\ell{:}c\} \vdash_{\mathsf{VH}} VH\{\ell \mapsto h\} : \Psi\{\ell{:}c\}$
- *If* $J(\Psi)$ *is a derivable judgement with* Ψ *as the value heap typing component of its context then* $J(\Psi\{\ell{:}c\})$ *is also derivable.*

Proof: The first item follows by inspection of the rule (wf-VHT). By inspection of the rule (wf-VHT) and (sub-VHT) it is easy to establish $\Phi \vdash_{\mathsf{VHT}} \Psi\{\ell{:}c\} \leq \Psi$. The second item is by inspection of the rule (VH) and Context Strengthening, and the third item follows immediately from Context Stregnthening. □

Lemma A.13 (Heap Update) *If* $\Phi \vdash_{\mathsf{VHT}} \Psi$ *and* $\Phi; \epsilon; \epsilon \vdash c \leq \Psi(\ell) : \mathsf{T}$ *then:*

- $\Phi \vdash_{\mathsf{VHT}} \Psi\{\ell{:}c\}$
- *If* $\Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi$ *and* $\Phi; CH; \Psi\{\ell{:}c\} \vdash_{\mathsf{h}} h : c$*, then* $\Phi; CH; \Psi\{\ell{:}c\} \vdash_{\mathsf{VH}} VH\{\ell \mapsto h\} : \Psi\{\ell{:}c\}$*.*
- *If* $J(\Psi)$ *is a derivable judgement with* Ψ *as the value heap typing component of its context then* $J(\Psi\{\ell{:}c\})$ *is also derivable.*

Proof: The first item is by inspection of the rule (wf-VHT) and Derived Judgements. By inspection of the rule (wf-VHT) and (sub-VHT) it is easy to establish $\Phi \vdash_{\mathsf{VHT}} \Psi\{\ell{:}c\} \leq \Psi$. The second item is by inspection of the rule (VH) and Context Strengthening, and the third item follows immediately from Context Strengthening. □

Lemma A.14 (Canonical Stack Forms) *If* $\Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma$ *then* $R(\mathsf{sp}) = w_1 :: \cdots :: w_n :: \mathsf{se}$*,* $\Gamma(\mathsf{sp}) = \tau_1 :: \cdots :: \tau_n :: \mathsf{se}$*, and* $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w_i : \tau_i$*.*

Proof: By induction on the derivation and inspection of the rules (regfile), (stk-nil), and (stk-cons). □

Lemma A.15 (Canonical Heap Forms) *If* $\Phi; CH; \Psi \vdash_{\mathsf{h}} h : c$ *then:*

$$\begin{array}{rrcl} & c & = & \forall[t_1, \ldots, t_n]{*}\mathsf{tag}(c_1^{\circ})c_2 \\ & h & = & \mathsf{tag}(c_1)\Lambda[t_1, \ldots, t_n]\hat{h} \\ \textit{where} & \Delta & = & \Delta(t_1, \ldots, t_n) \\ & B & = & B(t_1, \ldots, t_n) \end{array}$$

and either:

- $c_2 = \mathsf{code}\ \Gamma$*,* $\hat{h} = \mathsf{code}\ I$*, and* $\Phi; CH; \Psi; \Delta; B; \Gamma \vdash_{\mathsf{I}} I$*, or*
- $c_2 = \langle c_1^{\phi_1}, \ldots, c_n^{\phi_n} \rangle$*,* $\hat{h} = \langle w_1, \ldots, w_n \rangle$*,* $\phi_i \neq +$ *implies* $\Phi; \epsilon \vdash_{\mathsf{tc}} c_i : \mathsf{T}$*, and* $\Phi; CH; \Psi; \Delta; B \vdash w_i : c_i^{\phi_i}$

Proof: By inspection of the rules (hv), (hv-code), and (hv-tuple). □

Lemma A.16 (Canonical Forms) *If* $\Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi$ *and* $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w : c$ *then:*

1. $c = \mathsf{int}$ *implies* $w = i$
2. $c = \mathsf{ns}$ *implies* $w = \mathsf{ns}$
3. $c = \forall[t_1, \ldots, t_n]{*}\mathsf{tag}(c_1^{\phi})c_2$ *implies:*
   - $w = [c_m](\cdots[c_1](\ell)\cdots)$
   - $VH(\ell) = \mathsf{tag}(c')\Lambda[t''_1, \ldots, t''_m, t'_1, \ldots, t'_n]\hat{h}$
   - $\Psi(\ell) = \forall[t''_1, \ldots, t''_m, t''_m, t'_1, \ldots, t'_n]{*}\mathsf{tag}(c'^{\circ})c'_2$
   - $\Phi; CH; \Psi \vdash_{\mathsf{h}} VH(\ell) : \Psi(\ell)$
   - $\phi = +$ *implies* $\Phi; \epsilon; \epsilon \vdash c' \leq c_1 : \kappa$
   - $\phi = -$ *implies* $\Phi; \epsilon; \epsilon \vdash c_1 \leq c' : \kappa$
   - $\Phi; CH; \epsilon; \epsilon \vdash c_1, \ldots, c_m \rhd t''_1, \ldots, t''_m \Rightarrow \rho$
   - $\Phi; \epsilon; \epsilon \vdash (\forall[t'_1, \ldots, t'_n]{*}\mathsf{tag}(c'^{\circ})c'_2)\{\rho\} \leq c : \mathsf{T}$
4. $c = \mathsf{sptr}(c')$ *implies* $w = \mathsf{sptr}(|c'|)$
5. $\mathrm{unroll}_{CH;\epsilon}(c) = c'$ *implies* $w = \mathsf{roll}^{c''}(w')$, $\Phi; \epsilon \vdash c = c'' : \mathsf{T}$, $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w' : c'$, *and the conditions of this lemma hold for* c' *and* w'.
6. $c = \mathsf{self}\ \alpha{:}\mathsf{T}.c_1$ *implies* $w = \mathsf{pack}^{c_2}(\mathsf{roll}^{c_3}(w))$, $\Phi; CH; \Psi; \epsilon; \epsilon \vdash \mathsf{roll}^{c_3}(w) : c_3$, *and* $\Phi; \epsilon; \epsilon \vdash \mathrm{unroll}_{CH;\epsilon}(c_3) \leq c_1\{\alpha := c_3\} : \mathsf{T}$

Proof: First strengthen the lemma so that instead of "$c = \mathsf{int}$" it says "the canonical form of c is int" and similarly for the other items. The proof procceeds by induction on the derivation of $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w : c$. Consider the various cases for the last rule used in the derivation:

case (sv-int), (sv-ns), or (sv-sptr): Immediate.

case (svname): By inspection of the rule (VH) it must be that $c = \Psi(\ell)$ and $\Phi; CH; \Psi \vdash_{\mathsf{h}} VH(\ell) : c$. The rest follows from Canonical Heap Forms.

case (subsume): In case $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w : c'$ and $\Phi; \epsilon; \epsilon \vdash c' \leq c : \mathsf{T}$. By the induction hypothesis, the items hold for c' and w. By Lemma A.5 the normal forms of c' and c have the same outer form, therefore the same cases apply for c as for c'. If the normal form of c' is $\mathsf{sptr}(c''')$ and the normal form of c is $\mathsf{sptr}(c'')$, then by the induction hypothesis $w = \mathsf{sptr}(|c'''|)$. By the (sub-sptr) rule $\Phi; \epsilon; \epsilon \vdash c''' \leq c'' : \mathsf{S}$. Inspection of the subtyping rules and induction show that $|c''|$ is defined and equal to $|c'''|$. Thus $w = \mathsf{sptr}(|c''|)$ as required. The other cases are similar.

case (coerce): In this case $w = \delta(w')$, $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w' : c'$, and $\Phi; CH; \epsilon; \epsilon \vdash c' : c \Rightarrow$. Consider the various cases for δ:

case $\delta = [c'']$: In this case $c' = \forall t.c'''$, $\Phi; CH; \epsilon; \epsilon \vdash c'' \rhd t \Rightarrow \varrho$, and $c = c'''\{\varrho\}$. By the induction hypothesis, either no items apply to c and w or item 3 applies to c' and w'. The conditions for item 3 for c and w are easy to establish from those for c' and Type Substitution.

case $\delta = \mathsf{roll}^c$**:** Item 5 holds in this case; the conditions are easy to establish given the induction hypothesis.

case $\delta = \mathsf{unroll}$**:** This case holds by the induction hypothesis, item 5, and Lemma A.9.

case $\delta = \mathsf{pack}^c$**:** In this case $c = \mathsf{self}\ \alpha{:}\mathsf{T}.c_1$, $\mathrm{unroll}_{CH;\epsilon}(c')c_2$ and $\Phi;\epsilon;\epsilon \vdash c_2 \le c_1\{\alpha := c'\} : \mathsf{T}$. The only types for which $\mathrm{unroll}_{CH;\epsilon}(c')$ is defined are type names and recursive types. For both of these, $\mathrm{roll}(c')$ is true, so by the induction hypothesis $w' = \mathsf{roll}^{c'} w''$. This establishes the conditions for item 6; none of the other items apply.

$\square$

Theorem A.17 (Preservation) *If* $\vdash_{\mathsf{P}} P_1$ *and* $P_1 \mapsto P_2$ *then* $\vdash_{\mathsf{P}} P_2$.

Proof: The proof is very similar to the ones given by Morrisett et al. [MWCG99, MCGW98b], so I will show only the new cases and the cases for load and store, since MooTal has polymorphic tuples unlike Tal or Stal. P_1 has the form $(CH, VH, R, \iota; I)$ or $(CH, VH, R, \mathsf{jmp}\ v)$. Let TD be the derivation of $\vdash_{\mathsf{P}} P_1$. Consider the following cases for ι:

case $\mathsf{mov}\ r_d, [r_s + i]$: TD has the form:

$$\dfrac{\begin{array}{c}\vdash_{\mathsf{Int}} (\Phi,\Psi) \\ \Phi \vdash_{\mathsf{CH}} CH : \Phi \\ \Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi \\ \Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma\end{array} \quad \dfrac{\dfrac{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash r_s : *\mathsf{tag}(c^\phi)\langle c_0^{\phi_0},\ldots,c_n^{\phi_n}\rangle}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ r_d, [r_s+i] : \epsilon; \Gamma\{r_d{:}\tau_i\}} \quad \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma\{r_d{:}\tau_i\} \vdash_{\mathsf{I}} I}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{I}} \iota; I}}{\vdash_{\mathsf{P}} P}$$

By the operational semantics $P_2 = (CH, VH, R\{r_d \mapsto w_i\{\vec{\alpha} := \vec{c'}\}, I\}$ where $R(r_s) = [c'_m](\cdots[c'_1](\ell)\cdots)$, $VH(\ell) = \mathsf{tag}(c')\Lambda[t_1,\ldots,t_m]\langle w_0,\ldots,w_n\rangle$, and $0 \le i \le n$. By $\hat{R}$ Typing, Canonical Forms, Canonical Heap Forms, Subtyping, and Type Substitution $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w_i\{\vec{\alpha} := \vec{c'}\} : c_i^+$. By inspection of the rules for word values having varianced typing, it must be that $\Phi; CH; \Psi; \epsilon; \epsilon \vdash w_i\{\vec{\alpha} := \vec{c'}\} : c_i$. By Register File Update $\Phi; CH; \Psi \vdash_{\mathsf{R}} R\{r_d \mapsto w_i\{\vec{\alpha} := \vec{c}\}\} : \Gamma\{r_d{:}c_i\}$ and $\vdash_{\mathsf{P}} P_2$.

case $\mathsf{mov}\ [r_d + i], r_s$**:** For this case let:

$$\begin{array}{rcl} c_1 &=& \langle c_0^{\phi_0},\ldots,c_n^{\phi_n}\rangle \\ c_2 &=& \begin{cases} \langle c_0^{\phi_0},\ldots,c_{i-1}^{\phi_{i-1}}, c_i^{\circ}, c_{i+1}^{\phi_{i+1}},\ldots,c_n^{\phi_n}\rangle & \phi_i = 0 \\ c_1 & \phi_i \le - \end{cases} \\ c_3 &=& \langle {c'_0}^{\phi'_0},\ldots,{c'_n}^{\phi'_n}\rangle \\ c_4 &=& \begin{cases} \langle {c'_0}^{\phi'_0},\ldots,{c'_{i-1}}^{\phi'_{i-1}}, {c'_i}^{\circ}, {c'_{i+1}}^{\phi'_{i+1}},\ldots,{c'_n}^{\phi'_n}\rangle & \phi_i = 0 \\ c_3 & \phi_i \le - \end{cases} \\ \Gamma' &=& \Gamma\{r_d{:}*\mathsf{tag}(c^\phi)c_2\} \end{array}$$

TD has the form:

$$\dfrac{\begin{array}{c}\vdash_{\mathsf{Int}} (\Phi,\Psi) \\ \Phi \vdash_{\mathsf{CH}} CH : \Phi \\ \Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi \\ \Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma\end{array} \quad \dfrac{\dfrac{\begin{array}{c}\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash r_d : *\mathsf{tag}(c^\phi)c_1 \\ \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash r_s : c_i\end{array}}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{i}} \mathsf{mov}\ [r_d+i], r_s : \epsilon; \Gamma'} \quad \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma' \vdash_{\mathsf{I}} I}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{I}} \iota; I}}{\vdash_{\mathsf{P}} P}$$

By the operational semantics, $P_2 = (CH, VH\{\ell \mapsto \mathsf{tag}(c')\Lambda[t_1,\ldots,t_m]\langle w'_0,\ldots,w'_n\rangle\}, R, I)$ where $R(r_s) = [c''_m](\cdots[c''_1](\ell)\cdots)$, $VH(\ell) = \mathsf{tag}(c')\Lambda[t_1,\ldots,t_m]\langle w_0,\ldots,w_n\rangle$, $0 \le i \le n$, $w'_j = w_j$ if $j \ne i$, and $w'_i = R(r_s)$. By $\hat{R}$ Typing, Canonical Forms, Canonical Heap Forms, and Subtyping, $\Psi(\ell) = \forall[t_1,\ldots,t_m]{*}\mathsf{tag}(c'^{\circ})c_3$, $\phi'_j \ne +$ implies $\Phi;\epsilon \vdash_{\mathsf{tc}} c'_j : \mathsf{T}$, $\Phi; CH; \Psi; \Delta; B \vdash w_i : c_i'^{\phi'_i}$, and $\Phi;\epsilon;\epsilon \vdash c_i'^{\phi'_i}\{\vec{\alpha} := \vec{c''}\} \le c_i^{\phi_i} : \mathsf{T}$. Since $\phi_i \le -$ or $\phi_i = 0$, $\phi_i \ne +$, and by Subtyping $\phi'_i \ne +$, so $\Phi;\epsilon;\epsilon \vdash_{\mathsf{tc}} c''_i : \mathsf{T}$. By Derived Judgements and Subtyping, $\Phi;\epsilon;\epsilon \vdash c_i \le c''_i : \mathsf{T}$. By $\hat{R}$ Typing and subsumption, $\Phi; CH; \Psi; \epsilon; \epsilon \vdash R(R_s) : c''_i$, and $\Phi; CH; \Psi; \Delta; B \vdash_{\hat{\mathsf{h}}} \langle w'_0,\ldots,w'_n\rangle : c_4$ where $\Delta = \Delta(t_1,\ldots,t_m)$ and $B = B(t_1,\ldots,t_m)$. Now it is easy to establish that $\Phi;\Delta;B \vdash c_3 \le c_4 : \mathsf{M}$, so $\Phi;\epsilon;\epsilon \vdash \forall[t_1,\ldots,t_m]{*}\mathsf{tag}(c'^{\circ})c_4 \le \Psi(\ell) : \mathsf{T}$. Let $\Psi' = \Psi\{\ell{:}\forall[t_1,\ldots,t_m]{*}\mathsf{tag}(c'^{\circ})c_4\}$. Then, by Heap Update $\vdash_{\mathsf{Int}} (\Phi,\Psi')$, $\Phi; CH; \Psi' \vdash_{\mathsf{R}} R : \Gamma$, and $\Phi; CH; \Psi' \vdash_{\mathsf{VH}} VH\{\ell \mapsto \mathsf{tag}(c')\Lambda[t_1,\ldots,t_m]\langle w'_0,\ldots,w'_n\rangle\} : \Psi'$. It is easy to establish that $\Phi;\epsilon;\epsilon \vdash c_4\{\vec{\alpha} := \vec{c''}\} \le c_2 : \mathsf{M}$ and $\Phi; CH; \Psi'; \epsilon; \epsilon \vdash R(r_d) : {*}\mathsf{tag}(c^{\phi})c_2$. By Register File Update $\Phi; CH; \Psi' \vdash_{\mathsf{R}} R : \Gamma\{r_d{:}{*}\mathsf{tag}(c^{\phi})c_2\}$ and $\vdash_{\mathsf{P}} P_2$.

case tagcmp: TD has the form:

$$\dfrac{\begin{array}{c}\vdash_{\mathsf{Int}} (\Phi,\Psi)\\ \Phi \vdash_{\mathsf{CH}} CH : \Phi\\ \Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi\\ \Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v_1 : {*}\mathsf{tag}(c_1^-)c'_1\\ \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v_1 : {*}\mathsf{tag}(c_2^+)c'_2 \quad A\end{array}}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{i}} \mathsf{tagcmp}\ v_1, v_2, v_b : \epsilon; \Gamma} \quad \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{I}} I}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{I}} \iota; I}}{\vdash_{\mathsf{P}} P}$$

where A is:

$$\begin{array}{c}\Phi;\epsilon;\epsilon \vdash c_1 \le c_2 : \kappa\\ \Rightarrow\\ \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v_b : {*}\mathsf{tag}(c_b^{\phi})\mathsf{code}\ \Gamma\end{array}$$

There are two cases. If $P_2 = (CH, VH, R, I)$ then clearly $\vdash_{\mathsf{P}} P_2$. Otherwise $P_2 = (CH, CH, R, I'\{\vec{\alpha} := \vec{c}\})$, $\hat{R}(v_1) = Y_1(\ell')$, $\hat{R}(v_2) = Y_2(\ell')$, $\hat{R}(v_b) = Y(\ell)$, and $VH(\ell) = \mathsf{tag}(c)\Lambda[t_1,\ldots,t_n]\mathsf{code}\ I'$. By Canonical Forms, $VH(\ell') = \mathsf{tag}(c')\Lambda[t'_1,\ldots,t'_m]\hat{h}$, $\Phi;\epsilon;\epsilon \vdash c_1 \le c' : \kappa$, and $\Phi;\epsilon;\epsilon \vdash c' \le c_2 : \kappa$. By rule (sub-trans) and A, $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v_b : {*}\mathsf{tag}(c_b^{\phi})\mathsf{code}\ \Gamma$. By $\hat{R}$ Typing, Canonical Forms, Canonical Heap Forms, Type Substitution, and Context Strengthening , $\Phi; CH; \Psi; \epsilon, \epsilon, \Gamma \vdash_{\mathsf{I}} I'\{\vec{\alpha} := \vec{c}\}$. Hence $\vdash_{\mathsf{P}} P_2$.

case unpack: TD has the form:

$$\dfrac{\begin{array}{c}\vdash_{\mathsf{Int}} (\Phi,\Psi)\\ \Phi \vdash_{\mathsf{CH}} CH : \Phi\\ \Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi\\ \Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma\end{array} \qquad \dfrac{\dfrac{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v : \mathsf{self}\ \alpha{:}\mathsf{T}.c}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{i}} \mathsf{unpack}\ \alpha, r, v : \alpha{:}\mathsf{T}; \alpha \le c; \Gamma\{r{:}\alpha\}} \quad \Phi; CH; \Psi; \alpha{:}\mathsf{T}; \alpha \le c; \Gamma\{r{:}\alpha\} \vdash_{\mathsf{I}} I}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{I}} \iota; I}}{\vdash_{\mathsf{P}} P}$$

By the operational semantics, $P_2 = (CH, VH, R\{r{:}\mathsf{roll}^{c''}(w)\}, I\{\alpha := c''\})$, where $\hat{R}(v) = \mathsf{pack}^{c'}(\mathsf{roll}^{c''}(w))$. By $\hat{R}$ Typing and Canonical Forms, $\Phi; CH; \Psi; \epsilon; \epsilon \vdash \mathsf{roll}^{c''}(w) : c''$ (1) and $\Phi;\epsilon;\epsilon \vdash \mathrm{unroll}_{CH;\epsilon}(c'') \le c\{\alpha := c''\} : \mathsf{T}$ (2). By (1) and Register File Update, $\Phi; CH; \Psi \vdash_{\mathsf{R}} R\{r \mapsto \mathsf{roll}^{c''}(w)\} : \Gamma\{r{:}c''\}$. By (2) and Type Substitution, $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma\{r{:}\alpha\}\{\alpha := c''\} \vdash_{\mathsf{I}} I\{\alpha := c''\}$. By Derived Judgements, $\Phi;\epsilon \vdash_{\mathsf{RT}} \Gamma$ so $\Gamma\{r{:}\alpha\}\{\alpha := c''\} = \Gamma\{r{:}c''\}$. Thus $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma\{r{:}c''\} \vdash_{\mathsf{I}} I\{\alpha := c''\}$ and $\vdash_{\mathsf{P}} P_2$.

□

Theorem A.18 (Progress) *If $\vdash_{\mathsf{P}} P$, then either P is a terminal configuration or P is reducible ($P \mapsto P'$ for some P').*

Proof: The proof is very similar to the ones given by Morrisett et al. [MWCG99, MCGW98b], so I will show only the new cases. Let $P = (CH, VH, R, I_{full})$ and TD be the derivation of $\vdash_{\mathsf{P}} P$.

case tagcmp: TD has the form:

$$\dfrac{\begin{array}{c}\vdash_{\mathsf{Int}} (\Phi, \Psi)\\ \Phi \vdash_{\mathsf{CH}} CH : \Phi \\ \Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi \\ \Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma\end{array} \quad \dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v_1 : *\mathsf{tag}(c_1^-)c_1' \\ \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v_1 : *\mathsf{tag}(c_2^+)c_2' \quad A\end{array}}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{i}} \mathsf{tagcmp}\ \ v_1, v_2, v_b : \epsilon; \Gamma} \\ \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{I}} I\end{array}}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{I}} I_{full}}}{\vdash_{\mathsf{P}} P}$$

where A is:

$$\begin{array}{c}\Phi; \epsilon; \epsilon \vdash c_1 \leq c_2 : \kappa \\ \Rightarrow \\ \Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v_b : *\mathsf{tag}(c_b^{\phi})\mathsf{code}\ \ \Gamma\end{array}$$

By Canonical Forms, $\hat{R}(v_1) = [c_{n_1}^1](\cdots[c_1^1](\ell_1)\cdots)$ and $\hat{R}(v_2) = [c_{n_2}^2](\cdots[c_1^2](\ell_2)\cdots)$. If $\ell_1 \neq \ell_1$ then $P \mapsto (CH, VH, R, I)$. Otherwise $\ell_1 = \ell_2$, so by Canonical Forms $VH(\ell_1) = \mathsf{tag}(c)\Lambda[t_1, \ldots, t_{n_1}]\hat{h}_1$, $\Phi; \epsilon; \epsilon \vdash c_1 \leq c : \kappa$, and $\Phi; \epsilon; \epsilon \vdash c \leq c_2 : \kappa$. By transitivity of subtyping and A, $\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v_b : *\mathsf{tag}(c_b^{\phi})\mathsf{code}\ \Gamma$. By Canonical Forms and Canonical Heap Forms, $\hat{R}(v_b) = [c_n''](\cdots[c_1''](\ell)\cdots)$ and $VH(\ell) = \mathsf{tag}(c_b')\Lambda[t_1'', \ldots, t_n'']\mathsf{code}$ I'. Therefore P reduces to $(CH, VH, R, I'\{\vec{\alpha''} := \vec{c''}\})$.

case unpack: TD has the form:

$$\dfrac{\begin{array}{c}\vdash_{\mathsf{Int}} (\Phi, \Psi)\\ \Phi \vdash_{\mathsf{CH}} CH : \Phi \\ \Phi; CH; \Psi \vdash_{\mathsf{VH}} VH : \Psi \\ \Phi; CH; \Psi \vdash_{\mathsf{R}} R : \Gamma\end{array} \quad \dfrac{\begin{array}{c}\dfrac{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash v : \mathsf{self}\ \alpha{:}\mathsf{T}.c}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{i}} \mathsf{unpack}\ \ \alpha, r, v : \alpha{:}\mathsf{T}; \alpha \leq c; \Gamma\{r{:}\alpha\}} \\ \Phi; CH; \Psi; \alpha{:}\mathsf{T}; \alpha \leq c; \Gamma\{r{:}\alpha\} \vdash_{\mathsf{I}} I\end{array}}{\Phi; CH; \Psi; \epsilon; \epsilon; \Gamma \vdash_{\mathsf{I}} I_{full}}}{\vdash_{\mathsf{P}} P}$$

By $\hat{R}$ Typing and Canonical Forms, $v = \mathsf{pack}^{c'}(\mathsf{roll}^{c''}(w))$. Hence $P \mapsto (CH, VH, R\{r \mapsto \mathsf{roll}^{c''}(w)\}, I\{\alpha := c''\})$.

□

Theorem A.19 (Type Soundness) *If $\vdash_{\mathsf{P}} P$ and $P \mapsto^* P'$ then P' is not stuck.*

Proof: By induction on the length of $P \mapsto^* P'$, Preservation, and Progress. □

A.3.9 Execution

An initial program state is formed from an executable (CH, VH, ℓ) by combining CH and VH with a register file that has an empty stack and an instruction sequence to jump to ℓ.

$$(\text{exec})\ \frac{\vdash_{\mathsf{E}}^{*\mathsf{tag}(c^{\phi})\mathsf{code}\{\mathsf{sp:se}\}} (CH, VH, \ell)}{\vdash (CH, VH, \ell) \overset{\mathsf{exec}}{\rightsquigarrow} (CH, VH, \{\mathsf{sp} \mapsto \mathsf{se}\}, \mathsf{jmp}\ \ell)}$$

Execution always produces a well formed executable.

Theorem A.20 *If* $\vdash E \overset{\mathsf{exec}}{\rightsquigarrow} P$ *then* $\vdash_{\mathsf{P}} P$.

Proof: By inspection of the various typing and linking judgement rules. □

A corollary of all the soundness theorems of this appendix is that well formed, complete, and link compatible object files will not get stuck when linked, formed into an executable, and executed. Since the link compatibility, completeness, and entry label checks require only the interfaces of the object files, MooTal is a safe and modular language.

Bibliography

[AC93] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, NY, USA, 1996.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 396–409, St. Petersburgh Beach, FL, USA, January 1996. ACM Press.

[BCP97] Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 415–438, Sendai, Japan, September 1997. Springer-Verlag. Also an invited lecture at FOOL 3, July 1996.

[BF98] Viviana Bono and Kathleen Fisher. An imperative, first-order calculus with object extension. In *5th International Workshop on Foundations of Object-Oriented Languages*, pages 8–1 to 8–13, San Diego, California, USA, January 1998.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David Turner. The ML Kit (version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[Bru94] Kim Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[BSP$^+$95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Sirer, Marc Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, USA, December 1995. ACM Press.

[BSvG95] Kim Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *9th European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 27–51. Springer-Verlag, 1995.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Car88a] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.

[Car88b] Luca Cardelli. Structural subtyping and the notion of power type. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 70–79, San Diego, CA, USA, January 1988. ACM Press.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 266–277, Paris, France, January 1997. ACM Press.

[CCH+89] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded quantification for object-oriented programming. In *4th International Conference on Functional Programming and Computer Architecture*, pages 273–280, London, UK, September 1989. ACM Press.

[CGL95] Giuseppe Castanga, Giorgio Ghelli, and Guiseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.

[Cha97] Craig Chambers. The Cecil language, specification and rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA, March 1997.

[CHC90] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 125–135, San Francisco, CA, USA, January 1990. ACM Press.

[CHW99] Karl Crary, Michael Hicks, and Stephanie Weirich. Type-safe dynamic linking of native code. Submitted for publication, contact: `weirich@cs.cornell.edu`, November 1999.

[Coo89] William Cook. *A Denotational Semantics of Inheritance*. Ph.D. dissertation, Brown University, Box 1910, Computer Science Department, Brown University, Providence, RI 02906, USA, May 1989. Available as technical report CS-89-33.

[Cra99] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report CMU-CS-99-100, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA, 1999.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *1999 ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, Paris, France, September 1999. ACM Press.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM SIGPLAN International Conference on Functional Programming*, pages 301–312, Baltimore Maryland, USA, September 1998. ACM Press.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *4th ACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, April 1997.

[DFWB97] Drew Dean, Edward Felten, Dan Wallach, and Dirk Balfanz. Java security: Web browsers and beyond. In Dorothy Denning and Peter Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*. ACM Press, October 1997.

[DS98] Dominic Duggan and Constantinos Sourelis. Parameterized modules, recursive modules and mixin modules. In *ACM SIGPLAN Workshop on ML*, pages 87–96, Baltimore, MA, USA, September 1998.

[ESTZ95] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico. An interpretation of typed oop in a language with state. *Lisp and Symbolic Computation*, 8(4):357–397, December 1995.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, Montreal, Canada, June 1998. ACM Press.

[FHM94] Kathleen Fisher, F. Honsell, and John Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1:3–37, 1994.

[Fis96] Kathleen Fisher. *Type Systems for object-oriented programming languages*. Ph.D. dissertation, Computer Science Department, Stanford University, CA 94305, USA, May 1996. Available as technical report CS-TR-98-1602.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 171–183, San Diego, CA, USA, January 1998. ACM Press.

[FM95a] Kathleen Fisher and John Mitchell. A delegation-based object calculus with subtyping. In *10th International Conference on Fundamentals of Computation theory*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1995.

[FM95b] Kathleen Fisher and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1995.

[FM96] Kathleen Fisher and John Mitchell. Classes = objects + data abstraction. Technical Report STAN-CS-TN-96-31, Computer Science Department, Stanford University, CA 94305, USA, January 1996.

[FM98] Kathleen Fisher and John Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, 1998.

[FR99] Kathleen Fisher and John Reppy. The design of a class mechanism for Moby. In *1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, May 1999. ACM Press. Moby information is available at `http://www.cs.bell-labs.com/~jhr/moby`.

[Gal90] Jean Gallier. On girard's "candidats de reductibilité". In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, number 31 in The APIC Series, pages 123–204. Academic Press, 1990.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. dissertation, Université Paris VII, 1972.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, August 1996.

[Gle99a] Neal Glew. An efficient class and object encoding. Unpublished, author's contact: `glew@cs.cornell.edu`., August 1999.

[Gle99b] Neal Glew. Object closure conversion. In Andrew Gordon and Andrew Pitts, editors, *3rd International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, September 1999. Elsevier. `http://www.elsevier.nl/locate/entcs/volume26.html`.

[Gle99c] Neal Glew. Object closure conversion. Technical Report TR99-1763, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, August 1999.

[Gle99d] Neal Glew. Type dispatch for named hierarchical types. Technical Report TR99-1738, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, April 1999.

[Gle99e] Neal Glew. Type dispatch for named hierarchical types. In *1999 ACM SIGPLAN International Conference on Functional Programming*, pages 172–182, Paris, France, September 1999. ACM Press.

[GM99a] Neal Glew and Greg Morrisett. Type safe linking and modular assembly language. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 250–261, San Antonio, TX, USA, January 1999. ACM Press.

[GM99b] Dan Grossman and Greg Morrisett. Towards compiler-independent certifying compilation. Submitted for publication, contact: `danieljg@cs.cornell.edu`, November 1999.

[Hic99] Jason Hickey. Predicative type-theoretic interpretation of objects. Unpublished, author's contact: `jyh@cs.cornell.edu`, 1999.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 123–137, Portland, OR, USA, January 1994. ACM Press.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 130–141, San Francisco, CA, USA, January 1995. ACM Press.

[HMM90] Robert Harper, Eugenio Moggi, and John Mitchell. Higher-order modules and the phase distinction. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 341–354, San Francisco, CA, USA, January 1990. ACM Press.

[HS97] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA, June 1997.

[Int97] Intel Corporation. *Intel Architecuture Software Developer's Manual*, 1997. Three volumes.

[Kam88] Samuel Kamin. Inheritance in SMALLTALK-80: A denotational definition. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 80–87, San Diego, CA, USA, January 1988. ACM Press.

[Koz98] Dexter Kozen. Efficient code certification. Technical Report 98-1661, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, January 1998.

[Koz99] Dexter Kozen. Language-based security. In *Mathematical Foundations of Computer Science*, volume 1672 of *Lecture Notes in Computer Science*, pages 248–298. Springer-Verlag, September 1999.

[KR94] Samuel Kamin and Uday Reddy. Two semantic models of object-oriented languages. In Carl Gunter and John Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, Cambridge, MA 02142, USA, 1994.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 109–122, Portland, OR, USA, January 1994. ACM Press.

[LR98] Xavier Leroy and François Rouaix. Security properties of typed applets. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 391–403, San Deigo, CA, USA, January 1998. ACM Press.

[LST99] Christopher League, Zhong Shao, and Valerey Trifonov. Representing java classes in a typed intermediate language. In *1999 ACM SIGPLAN International Conference on Functional Programming*, pages 183–196, Paris, France, September 1999. ACM Press.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, September 1996.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, Daivd Walker, Stephanie Weirich, and Steve Zdancewic. TALx86:

A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999. INRIA research report no. 0228.

[MCGW98a] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *2nd International Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Kyoto, Japan, March 1998. Springer-Verlag.

[MCGW98b] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language (extended version). Technical Report CMU-CS-98-185, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA, December 1998. A version with proofs was submitted to the Journal of Functional Programming.

[Mit90] John Mitchell. Toward a typed foundation for method specialization and inheritance. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 109–124, San Francisco, CA, USA, January 1990. ACM Press.

[Mor95] Greg Morrisett. *Compiling with Types*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, December 1995. Published as CMU Technical Report CMU-CS-95-226.

[MTC+96] Greg Morrisett, David Tarditi, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, Tucson, AZ, USA, February 1996.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 85–97, San Diego, CA, USA, January 1998. ACM Press.

[MWCG99] 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):528–569, May 1999.

[Nec98] George Necula. *Compiling with Proofs*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA, September 1998. Published as Carnegie Mellon University technical report CMU-CS-98-154.

[PHH+93] Simon Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *U.K. Joint Framework for Information Technology Technology Conference*, pages 249–257, Keele, U.K., March 1993. Somewhat out of date, see `http://www.haskell.org/ghc/`.

[Pie94] Benjamin Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.

[PT94] Benjamin Pierce and David Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[Red88] Uday Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *ACM Symposium on LISP and Functional Programming*, pages 289–297, Snowbird, UT, USA, July 1988. ACM Press.

[Rém94] Didier Rémy. Programming objects with ML-ART, an extension to ML with abstract and record types. In Masami Hagiya and John Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Science*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

[Rey74] John Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[RR96a] John Reppy and Jon Riecke. Classes in Object ML via modules. In *3rd International Workshop on Foundations of Object-Oriented Languages*, New Brunswick, NJ, USA, July 1996.

[RR96b] John Reppy and Jon Riecke. Simple objects for Standard ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–180, Philadelphia, PA, USA, May 1996. ACM Press.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 40–53, Paris, France, January 1997. ACM Press.

[SFPB96] Emin Gún Sireer, Marc Fiuczynski, Przemyslaw Pardyak, and Brian Bershad. Safe dynamic linking in an extensible operating system. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, Tucson, AZ, February 1996.

[SH00] Chris Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, Boston, MA, USA, January 2000. ACM Press.

[Sha97] Zhong Shao. An overview of the FLINT/ML compiler. In *1st International Workshop on Types in Compilation*, number BCCS-97-03 in the Boston College Computer Science Technical Reports, Amsterdam, The Netherlands, June 1997.

[SMS96] Andrew Shalit, David Moon, and Orca Starbuck. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, September 1996.

[TDMW97] Franklyn Turbak, Allyn Dimock, Robert Muller, and Joe Wells. Compiling with polymorphic and ployvariant flow types. In *1st International Workshop on Types in Compilation*, number BCCS-97-03 in the Boston College Computer Science Technical Reports, Amsterdam, The Netherlands, June 1997.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996. ACM Press.

[Vis98] Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *13th IEEE Symposium on Logic in Computer Science*, pages 380–391, Indianapolis, IN, USA, June 1998. IEEE Computer Society.

[Vou98] Jérôme Vouillon. Using modules as classes. In *5th International Workshop on Foundations of Object-Oriented Languages*, pages 4–1 to 4–10, San Diego, CA, USA, January 1998.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 1993. ACM Press.

[Wri95] Andrew Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4):343–355, December 1995.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

[ZGM99] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *1999 ACM SIGPLAN International Conference on Functional Programming*, pages 197–207, Paris, France, September 1999. ACM Press.