Type-Safe Linking and Modular Assembly Language

Neal Glew and Greg Morrisett
Department of Computer Science
Cornell University

23 July 1998

Abstract

Linking is a low-level task that is usually vaguely specified, if at all, by language definitions. But the security of web browsers and other extensible systems depends crucially upon a set of checks that must be performed at link time. Building upon the simple, but elegant, ideas of Cardelli, and module constructs from high-level languages, we present a formal model of *typed object files* and a set of inference rules that are sufficient to guarantee that type safety is preserved by the linking process.

Whereas Cardelli's link calculus is built on top of the simply-typed lambda calculus, our object files are based upon *typed assembly language* so that we may model important low-level implementation issues. Furthermore, unlike Cardelli, we provide support for abstract types and (higher-order) type constructors—features critical for building extensible systems or modern programming languages such as ML.

1 Introduction

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, extensible systems [LY97, WLAG93, BSP+95, Nec97, Koz98, HCC+98] are being built based upon the strong safety guarantees that language definitions provide. These systems use dynamic linking and loading as a fundamental part of their operation, so it is critical to precisely define the consistency checks a linker must perform.

For example, Java-extensible web browsers, such as Netscape's Navigator and Microsoft's Internet Explorer, rely on static type systems with link checks to enforce a wide class of important safety properties. Extensions (applets) are written in a high-level language such as Java, and then compiled by an untrusted compiler to a target language (JVM bytecode). The integrity of the browser does not depend on type soundness of Java, but rather on type soundness (among other properties) of the JVM. 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 compatability. Unfortunately, this component is vaguely specified and has been a source of some security holes [MF96, Sar97].

Recently, Cardelli proposed a calculus of compilation units for the simply-typed lambda-calculus and presented a set of rules for determining link compatability [Car97]. 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 program was well-formed and hence would not "go wrong" when evaluated.

Whereas Cardelli's calculus is an elegant formulation of some of the issues involved in linking, it keeps abstract many important details such as binding and alpha-variance of labels, and leaves unresolved certain critical features, notably support for cyclic references, user-defined type abstraction, and dynamic loading. The goal of this paper is to build upon Cardelli's work and provide a suitable treatment of the typing issues for low-level compilation units (object files), the process of linking them to produce a well-formed executable, and the operation of dynamically loading code into a running program.

In principle, the compilation units we propose are orthogonal to language types and term structures. However, as we are concerned with low-level implementation details, we use a variant of the Typed Assembly Language (TAL) of Morrisett et al. [MWCG98]. Operationally, TAL is based on a conventional RISC-style assembly language. However, as with high-level languages (like ML), TAL has a static type system that prevents runtime type errors and provides other properties on which security systems might depend. Morrisett et al. presented the syntax, static and dynamic semantics of the TAL core language, established the soundness of the static semantics, and showed how a high-level core functional language based on ML could be compiled to TAL [MWCG97]. In this paper, we extend core TAL with *typed object files* and formalise the concepts of linking and link compatability. We also sketch how high-level language constructs, such as the signatures, structures, and functors of SML might be mapped to typed object files. The ideas of this paper are implemented as part of a project to implement TAL, which is available at `http://www.cs.cornell.edu/home/walker/talnet/tal.html#talc`.

We proceed as follows: In section 2, we present the abstractions of conventional untyped object files and linkers (*e.g.*, Unix's `ld`) and discuss the issues of link compatability and dynamic loading in this simplified setting. In section 3, we briefly review the type system of TAL and introduce a simple module language MTAL_0, which, in the spirit of Cardelli, provides support for separate compilation, separate type-checking, and a stronger notion of link compatibility. We extend MTAL_0 in section 4 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 may compile ML-style modules, including functors, to the target language.

2 Untyped Object Files and Linkers

We begin with a model of typical untyped object files and the process of linking.

2.1 Untyped Assembly Language

Figure 1 shows a simple, operational model an of untyped assembly language. The state of an assembly language program consists of three components: a heap representing memory, a register file, and an instruction sequence representing the program counter. A heap, H, is a finite mapping of labels (symbolic addresses), ℓ, to heap values, h. In our formulation, there are two types of heap values: code sequences, `code` I, where I is an instruction sequence, and tuples, $\langle v_1, \ldots, v_n \rangle$, where the v_i are small values. Instructions include typical RISC instructions (*e.g.*, `add` r_d, r_s, v) and one special instruction, `malloc` $r[n]$. This instruction allocates in the heap a new tuple with n uninitialised entries and places the new label in r. Small values are the word sized objects of the machine, that is, objects that fit in a register. They include integers i, labels, and uninitialised

$$
\begin{array}{lrcl}
\text{Small Values} & v & ::= & i \mid r \mid \ell \mid ? \mid \ldots \\
\text{Heap Values} & h & ::= & \texttt{code}\ I \mid \langle v_1, \ldots, v_n \rangle \\
\text{Instructions} & \iota & ::= & \texttt{add}\ r_d, r_s, v \mid \texttt{malloc}\ r[n] \mid \texttt{jmp}\ v \mid \texttt{halt} \mid \ldots \\
\text{Heaps} & H & ::= & \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \\
\text{Register Files} & R & ::= & \{r_1 \mapsto v_1, \ldots, r_n \mapsto v_n\} \\
\text{Instruction Sequences} & I & ::= & \iota_1; \ldots; \iota_n \\
\text{Program States} & P & ::= & (H, R, I)
\end{array}
$$

$$(H, R, I) \longmapsto P' \quad \text{where:}$$

I	P'	Conditions
$\texttt{add}\ r_d, r_s, v; I'$	$(H, R\{r_d \mapsto R(r_s) + \hat{R}(v)\}, I')$	
$\texttt{malloc}\ r[n]; I'$	$(H\{\ell \mapsto \langle \underbrace{?, \ldots, ?}_{n} \rangle\}, R\{r \mapsto \ell\}, I')$	$\ell \notin \text{dom}(H)$
$\texttt{jmp}\ v$	(H, R, I')	$\hat{R}(v) = \ell \wedge H(\ell) = \texttt{code}\ I'$

$$\hat{R}(v) = \begin{cases} i & v = i \\ R(r) & v = r \\ \ell & v = \ell \\ ? & v =? \\ \vdots & \end{cases}$$

Figure 1: Untyped Assembly Language

values, `?`. Registers are also small values to simplify the syntax of operands.[1] A register file is just a finite map from registers, r, to small values. We model the execution of assembly language programs by a reduction relation that simulates the execution of one instruction, for example $(H, \{\texttt{r1} \mapsto 5, \texttt{r2} \mapsto 6\}, \texttt{add r1, r2, 3}; I) \longmapsto (H, \{\texttt{r1} \mapsto 9, \texttt{r2} \mapsto 6\}, I)$.

2.2 Object Files

Abstractly, an object files consists of three components:

1. *A heap H.*
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.

While this description of object files is generic, as heaps could map labels to the terms of any language, we will use the untyped assembly language of the previous section for concreteness.

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

$$\frac{\mathcal{E} \subseteq \text{dom}(H) \quad \mathcal{I} \cap \text{dom}(H) = \emptyset \quad \forall \ell \in \text{dom}(H).FL(H(\ell)) \subseteq \text{dom}(H) \cup \mathcal{I}}{\vdash [\mathcal{I} \Rightarrow H : \mathcal{E}]}$$

[1] A proper account would need to distinguish word values taht may be stored in the heap or in a register from values that may be operands (which include registers).

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 (alpha-conversion) of local labels. The justification for this implicit alpha-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* alpha-vary so that the linker may resolve cross references among object files.

2.3 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, hence, the notion of *link compatibility*.

Definition 2.1 (Link Compatibility) *Well formed object files* $O_1 = [\mathcal{I}_1 \Rightarrow H_1 : \mathcal{E}_1]$ *and* $O_2 = [\mathcal{I}_2 \Rightarrow H_2 : \mathcal{E}_2]$ *are link compatible, written* $O_1 \overset{\text{lc}}{\leftrightarrow} O_2$*, iff* $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$ *(their exported labels are disjoint).*

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

$$\frac{\vdash [\mathcal{I}_1 \Rightarrow H_1 : \mathcal{E}_1] \quad \vdash [\mathcal{I}_2 \Rightarrow H_1 : \mathcal{E}_2] \quad \vdash [\mathcal{I}_1 \Rightarrow H_1 : \mathcal{E}_1] \overset{\text{lc}}{\leftrightarrow} [\mathcal{I}_2 \Rightarrow H_2 : \mathcal{E}_2] \quad \text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset}{\vdash [\mathcal{I}_1 \Rightarrow H_1 : \mathcal{I}_1] \text{ link } [\mathcal{I}_2 \Rightarrow H_2 : \mathcal{I}_2] \rightsquigarrow [(\mathcal{I}_1 \cup \mathcal{I}_2) \setminus (\mathcal{E}_1 \cup \mathcal{E}_2) \Rightarrow (H_1 \cup H_2) : (\mathcal{E}_1 \cup \mathcal{E}_2)]}$$

Because of the condition $\text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset$, in applying the link rule, alpha-variants of the object files must be chosen such that their local labels are disjoint. It follows from the definitions that if $\vdash O_1 \text{ link } O_2 \rightsquigarrow O$ then $\vdash O$.

2.4 Static Executables

The final operation the linker performs is to produce an executable. An executable is just a closed heap paired with an entry label defined in that heap. We write (H, ℓ) to denote an executable. An executable is well-formed according to the following rule:

$$\frac{\ell \in \text{dom}(H) \quad \forall \ell \in \text{dom}(H).FL(H(l)) \subseteq \text{dom}(H)}{\vdash (H, \ell) \text{ executable}}$$

Given a well-formed object file and a distinguished label from its export set, we may produce a well-formed executable only when the import set of the object file is empty:

$$\frac{\vdash [\mathcal{I} \Rightarrow H : \mathcal{E}] \quad \mathcal{I} = \emptyset \quad \ell \in \mathcal{E}}{\vdash [\mathcal{I} \Rightarrow H : \mathcal{E}], \ell \overset{\text{prg}}{\rightsquigarrow} (H, \ell)}$$

To 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, an executable is mapped to an initial program state 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:

$$\frac{\vdash (H, \ell) \text{ executable}}{\vdash (H, \ell) \overset{\text{exec}}{\rightsquigarrow} (H, \emptyset, \texttt{jmp}\ \ell)}$$

[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.

2.5 Dynamic Linking and Loading

Modern operating systems also provide some form of dynamic linking and loading. Dynamic linking essentially delays the complete linking from the time an executable is produced to the time that it is loaded. A *dynamic executable*, denoted $(O, \ell, ((L_1, \mathcal{E}_1), \ldots, (L_n, \mathcal{E}_n)))$, consists of an incomplete object file, an entry label, and a list of libraries to be dynamically linked. The L_i name libraries to dynamically link, and $\mathcal{E}_i$ gives the labels to import from L_i. A dynamic executable is well formed if its imports are contained within the named libraries:

$$\frac{\vdash [\mathcal{I} \Rightarrow H : \mathcal{E}] \quad \mathcal{I} \subseteq \cup_{i=1}^{n} \mathcal{E}_i \quad \ell \in \mathcal{E} \quad \mathcal{E}, \mathcal{E}_1, \ldots, \mathcal{E}_n \text{ are disjoint}}{\vdash ([\mathcal{I} \Rightarrow H : \mathcal{E}], \ell, ((L_1, \mathcal{E}_1), \ldots, (L_n, \mathcal{E}_n)))}$$

Dynamic libraries are an object file paired with an entry label for initialising the library. To execute a dynamic executable the operating system locates libraries $((O_1, \ell_1), \ldots, (O_n, \ell_n))$ for $L_1, \ldots, L_n$. Then it links the object files together $\vdash O_1$ link $\ldots$ link O_n link $O \rightsquigarrow [_ \Rightarrow H : _]$, and forms an initial state $(H, \emptyset, \texttt{jal}\ \ell_1; \ldots; \texttt{jal}\ \ell_n; \texttt{jmp}\ \ell)$.[3] Note that operating systems with dynamic linking usually share the libraries between the processes in the system so that only one copy of a library exists in memory. We ignore such details in our model.

Dynamic loading delays linking until runtime. For example, in an extensible system, an extension is an object file and an entry label to run/initialise the extension. To process a request to load an extension, the extensible system verifies or instruments the object file, then asks the operating system to link it, and finally calls the entry label. We model extensions as another kind of heap value (O, ℓ) consisting of an object file O and an entry label ℓ. We model the dynamic loading operation as an "instruction" $\texttt{dynlink}\ r_d, r_s$ where r_s contains a label whose heap value is an extension. The semantics of this instruction is to link the object file into the running program and place the entry label into r_d:

$$(H, R, \texttt{dynlink}\ r_d, r_s; I) \longmapsto (H', R\{r_d \mapsto \ell\}, I)$$
where $R(r_s) = \ell_1$, $H(\ell_1) = (O, \ell)$, and $\vdash [\mathrm{dom}(H) \Rightarrow H : \emptyset]$ link $O \rightsquigarrow [_ \Rightarrow H' : _]$

In an extensible web browser the entry label would run the applet. In an extensible web server or extensible kernel the entry label would initialise the extension registering its services with the server or kernel.

3 MTAL_0

The goal of this paper is to formalise typed object files combining the development in section 2 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 calculus; the next section will extend MTAL_0 to our full calculus by adding (higher order) type abstraction. While our module calculus is independent of the core calculus, MTAL_0 is based on TAL [MWCG97, MCGW98] for concreteness. We briefly review TAL and the benefits of type safety, and then build a notion of typed object files on top of TAL.

3.1 TAL

TAL is essentially a typed version of the language of section 2.1. Its types include integers, tuples, and code types. TAL also includes some "typing" instructions that have no operational effect but

[3] The instruction sequence $\texttt{jal}\ \ell_1; \ldots; \texttt{jal}\ \ell_n; \texttt{jmp}\ \ell$ calls each of the entry poitns ℓ_i in turn and then jumps to the entry point ℓ.

make type checking easier (*e.g.*, `unpack`). The allocation instruction `malloc` $r[\tau_1, \ldots, \tau_n]$ includes the types of the entries of the tuple to be allocated. It gives r the tuple type $\langle \tau_1^0, \ldots, \tau_n^0 \rangle$ where the 0s indicate possibly uninitialised slots. Storing into the first slot would give r the type $\langle \tau_1^1, \tau_2^0, \ldots, \tau_n^0 \rangle$ where 1 indicates a definitely initialised slot. A slot may be loaded only if it is definitely initialised. A register file type $\Gamma = \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$ states that r_i contains a value of type τ_i. TAL code has the form $\texttt{code}[\Delta]\Gamma.I$; it is polymorphic over the type variables Δ, and I is a typed instruction sequence whose first instruction expects the registers to have register file type Γ. A label mapped to this code sequence will have type $\forall[\Delta]\Gamma$.

An increment function might be written in TAL as:

$$\texttt{inc} \mapsto \texttt{code}[\,]\{\texttt{r1}{:}\textbf{int}, \texttt{ra}{:}\forall[\,]\{\texttt{r1}{:}\textbf{int}\}\}.\texttt{add r1, r1, 1; jmp ra}$$

To call `inc`, the caller must place an integer in `r1` and a return label in `ra`, the return label must accept an integer in `r1`. In general, the return point will be a continuation which requires an environment, and `inc` must preserve this environment across the call. Our examples will use register `re` to hold the continuation's environment; under this convention `inc` would have the type:

$$\forall[\alpha]\{\texttt{r1}{:}\textbf{int}, \texttt{re}{:}\alpha, \texttt{ra}{:}\forall[\,]\{\texttt{r1}{:}\textbf{int}, \texttt{re} : \alpha\}\}$$

One more detail of TAL's type system will be important in later sections: TAL heaps are typed with heap typings, which are finite maps from labels to types, $\{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$. The judgement $\Psi_1 \vdash H : \Psi_2$ asserts that H has heap typing Ψ_2 where free labels are typed by Ψ_1. TAL heaps are unordered maps and a heap value may refer to its own label directly or indirectly. Thus, TAL's type checker uses the final heap type during checking of the heap, as in, $\Psi \vdash H : \Psi$. Note that TAL heaps extend Cardelli's work which considered only acyclic references.

3.2 Type Safety

The main motivation for statically typing is the benefit of various guarantees implied by the static checks, particularly the guarantee of type safety—the property that a well typed program never performs an illegal operation. Consider a set of well formed and mutually link compatible object files that together with a label are a complete program. As we shall see, the link operation is type preserving, so the resulting object file will be well formed. Similarly, formation and creation of an initial program state are type preserving, so the initial program state resulting from the execution of the linked object file is well formed. TAL's type safety means that the execution starting in this well formed initial state will never perform an illegal operation. Thus if we define typed linking, program formation, and formation of an initial state to be type preserving then MTAL_0 will be type safe.

An extensible system writer may desire other guarantees from MTAL_0. For example, if extension is checked against a fixed import interface, the MTAL_0 type system guarantees that the extension can directly 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 paper. Leroy and Rouaix [LR98] provide a discussion of some of the issues.

3.3 Object Files and Interfaces

MTAL_0 objects files extend untyped object files with types. A MTAL_0 object file is a triple $[\Psi_I \Rightarrow H : \Psi_E]$ where H is a TAL heap, Ψ_I is an import interface, and Ψ_E is an export interface.

fact.tal:

```
export fact:  ∀[α]{r1 : int, re : α, ra : ∀[]{r1:int, re:α}}
fact↦ ∀[α]{r1:int, re:α, ra:∀[]{r1:int, re:α}}.
      mov   r3,r1
      mov   r1,1
      jmp   loop[α]
loop↦ ∀[α]{r1:int, re:α, r3:int, ra:∀[]{r1:int, re:α}}.
      bz    r3,ra
      mul   r1,r3,r1
      sub   r3,r3,1
      jmp   loop[α]
```

main.tal:

```
import fact:  ∀[α]{r1 : int, re : α, ra : ∀[]{r1:int, re:α}}
export main:  ∀[]{r1 : int}
main↦ ∀[]{r1:int}.
      mov   re,0
      mov   ra,ret1
      jmp   fact[int]
ret1↦ ∀[]{r1:int, re:int}
      halt[int]
```

Figure 2: Modular Factorial Example

An interface is a heap typing, *i.e.*, $\{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$. Figure 2 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 MTAL_0 object files requires type checking:

$$\frac{\vdash \Psi_I \quad \vdash \Psi_A \leq \Psi_E \quad \Psi_I \cup \Psi_A \vdash H : \Psi_A}{\vdash [\Psi_I \Rightarrow H : \Psi_E]} \ (\text{dom}(\Psi_I) \cap \text{dom}(\Psi_A) = \emptyset)$$

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, $\vdash \Psi_A \leq \Psi_E$.[4]

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-

[4] $\vdash \Psi_A \leq \Psi_E$ means that $\forall \ell \in \text{dom}(\Psi_E) : \Psi_A(\ell) = \Psi_E(\ell)$. In a subtype setting, this could be extended to $\forall \ell \in \text{dom}(\Psi_E) : \Psi_A(\ell) \leq \Psi_E(\ell)$.

level interfaces for other modules, then it can be compiled to a typed object file without needing the implementations of the other modules.

3.4 Linking

Crucial to typed link compatibility is *interface compatibility*, $\vdash \Psi_1 \sim \Psi_2$. In particular, if two interfaces mention the same label then they must give it compatible types:[5]

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

Given interface compatibility, link compatibility is easily defined:

$$\frac{\vdash \Psi_{I1} \sim \Psi_{I2} \quad \vdash \Psi_{I1} \sim \Psi_{E2} \quad \vdash \Psi_{I2} \sim \Psi_{E1}}{\vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}] \overset{\mathrm{lc}}{\leftrightarrow} [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}]} \ (\mathrm{dom}(\Psi_{E1}) \cap \mathrm{dom}(\Psi_{E2}) = \emptyset)$$

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 $\vdash O_1$ link $O_2 \rightsquigarrow O$ then $\vdash O$. This theorem is much stronger than in the untyped case as it asserts that no type errors are introduced by the linking operation.

Again, MTAL_0 has a separate type-checking property. Link compatibility is defined entirely in terms of the imported and exported interfaces of the two modules and is independent of the modules' 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.

3.5 Programs and Execution

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

$$\frac{\vdash \Psi \quad \Psi \vdash H : \Psi}{\vdash (H, \ell) \text{ executable}} \ (\Psi(\ell) = \tau_e)$$

where τ_e is the type the entry convention gives the entry label. The factorial example's intended entry convention has $\tau_e = \forall[\,]\{\mathtt{r1} : \mathbf{int}\}$. The entry convention is an important low-level detail of how programs get executed, which we can formally specify as a MTAL_0 type.

As before, we can check when an object file is complete:

$$\frac{\vdash [\emptyset \Rightarrow H : \Psi_E]}{\vdash [\emptyset \Rightarrow H : \Psi_E]; \ell \text{ complete}} \ (\Psi_E(\ell) = \tau_e)$$

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 τ_e. We formalise these checks as a judgement and prove a correctness theorem in appendix A.

The production of an executable and the process of execution is the same as in the untyped calculus. However, the consistency checks are sound: $\vdash O, \ell \overset{\mathrm{prg}}{\rightsquigarrow} P$ implies $\vdash P$ state and $\vdash \ P \overset{\mathrm{exec}}{\rightsquigarrow}$

[5] In MTAL_0, compatible types are equal types. In a subtype setting, compatible types are those whose meet exists.

```
export type file:  T;
export val open:  ∀[α : T]{r1 : string, re : α, ra : ∀[ ]{r1 : file, re : α}}
export val readline:  ∀[α : T]{r1 : file, re : α, ra : ∀[ ]{r1 : string, re : α}}
...
; A file is access rights plus O/S handle
; Access rights:  bit 0 read, bit 1 write, ...
file↦ ⟨int^1, int^1⟩
open↦ code[α : T]{r1 : string, re : α, ra : ∀[ ]{r1 : file, re : α}}.
  ; Call O/S open on r1 putting result in r3 preserving re and ra
  ; Determine access rights and store in r4 preserving re, r3, and ra
  malloc r1[int, int]
  st     r1.0, r4
  st     r1.1, r3
  coerce roll^file(r1)
  jmp    ra
readline↦ code[α : T]{r1 : file, re : α, ra : ∀[ ]{r1 : string, re : α}}.
  coerce unroll(r1)
  ld     r3, r1.0
  and    r3, 1
  bz     r3,error
  ld     r1, r1.1
  ; Call O/S read line on r1 putting result in r1 preserving re and ra
  jmp    ra
...
```

Figure 3: File Example

PS implies $\vdash PS$ state. (TAL program state well formedness and reduction are described in detail in [MWCG97].)

Developments similar to those for untyped dynamic linking and loading are possible. However, there are several choices of rules for dynamic loading and it is unclear which is correct. We leave this part to future work.

4 MTAL

MTAL_0 is a typed low-level calculus with a formalised notion of link compatibility. It extends the work of Cardelli making important low level concerns explicit, and it is very closely models the tasks of real operating system linkers. However, MTAL_0 does not address other shortcomings of Cardelli's calculus. We will progressively add constructs to MTAL_0 in the following sections to obtain our full calculus MTAL. A complete description of MTAL, including its syntax and static semantics, appears in appendix A and forms the basis of our implementation.

4.1 Abstract Types

MTAL_0 provides many type safety gaurantees but does not does not provide type abstraction guarantees.[6] Consider a security monitor for file access: It exports an operation `open` that takes a string and returns a "file handle", a pair of the extension's access rights and an operating system file handle. In a system without type abstraction, `open` must have the type $\textbf{string} \to \langle \textbf{int}, \textbf{int} \rangle$ which allows the extension to modify the access rights. To solve this problem, we add to MTAL the ability to declare abstract types in interfaces and use them in the types given to labels. Since we already use labels to name values, we use labels to name abstract types as well.

A MTAL interface, 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 labels to kinds.[7] Object files are still a triple $[Int_I \Rightarrow (TH, VH) : Int_E]$ 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. Program states are also extended to include a type heap.

The file example is shown in figure 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 2 and 3 can be repeated for MTAL; we mention just the highlights.

Just as value heaps can contain cyclic references we also allow cyclic type heap references, introducing the possibility of recursive types. Following standard type theory, in a type heap, TH, a type label, ℓ, is isomorphic to $TH(\ell)$. There are two ways to reflect this isomorphism in the type system. The first way makes ℓ and $TH(\ell)$ equal types. This makes a decision procedure for type equality considerably more complex. We choose the second way and introduce roll and unroll operations that witness the isomorphism $\ell \cong TH(\ell)$. Roll coerces an object from a concrete type to an abstract type; its typing rule is:

$$\frac{\Phi; TH; \Psi; \Delta; \Gamma \vdash v : \tau}{\Phi; TH; \Psi; \Delta; \Gamma \vdash \texttt{roll}^{\ell}(v) : \ell} \ (TH(\ell) = \tau)$$

Unroll does the opposite.

The value heap of an object file is checked using its type heap. Thus, the $\texttt{roll}^{\ell}(\cdot)$ can be used if and only if the type heap defines ℓ. An object file implementing ℓ will have a definition for ℓ in its type heap. A client of the abstract type ℓ, however, will import ℓ, and since the import interface is disjoint from the type heap, the latter will not contain a definition for ℓ. Thus, neither roll nor unroll can be applied in the client code, that is, the client cannot create or manipulate members of ℓ—ℓ really is abstract.

Our implementation 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 a translucent type, as in Harper and Lillibridge [HL94] and Leroy [Ler94]. This 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 type 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[8] and power kinds [Car88, Car91] respectively.

[6] Technically TAL has existential and polymorphic types which can be used to implement type abstractions. However, becuase of the "local" nature of `unpack` this is too cumbersome in practice.

[7] For now there is only one kind, the kind $\mathbf{T}$ of types. Kinds are included in anticipation of future developments.

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

Interface:

$$\begin{array}{l}
\texttt{type stack\$t:}\quad \mathbf{T} \to \mathbf{T} \\
\texttt{val stack\$empty:}\quad \forall[\alpha{:}\mathbf{T}, \gamma{:}\mathbf{T}]\{\texttt{re} : \gamma, \texttt{ra} : \forall[\,]\{\texttt{r1} : \texttt{stack\$t}\ \alpha, \texttt{re} : \gamma\}\} \\
\texttt{val stack\$isempty:}\quad \forall[\alpha{:}\mathbf{T}, \gamma{:}\mathbf{T}]\{\texttt{r1} : \texttt{stack\$t}\ \alpha, \texttt{re} : \gamma, \texttt{ra} : \forall[\,]\{\texttt{r1} : \mathbf{bool}, \texttt{re} : \gamma\}\} \\
\texttt{val stack\$push:}\quad \forall[\alpha{:}\mathbf{T}, \gamma{:}\mathbf{T}]\{\texttt{r1} : \alpha, \texttt{r2} : \texttt{stack\$t}\ \alpha, \texttt{re} : \gamma, \texttt{ra} : \forall[\,]\{\texttt{r1} : \texttt{stack\$t}\ \alpha, \texttt{re} : \gamma\}\} \\
\texttt{val stack\$pop:}\quad \forall[\alpha{:}\mathbf{T}, \gamma{:}\mathbf{T}]\{\texttt{r1} : \texttt{stack\$t}\ \alpha, \texttt{re} : \gamma, \texttt{ra} : \forall[\,]\{\texttt{r1} : \texttt{stack\$t}\ \alpha, \texttt{re} : \gamma\}\} \\
\texttt{val stack\$top:}\quad \forall[\alpha{:}\mathbf{T}, \gamma{:}\mathbf{T}]\{\texttt{r1} : \texttt{stack\$t}\ \alpha, \texttt{re} : \gamma, \texttt{ra} : \forall[\,]\{\texttt{r1} : \alpha, \texttt{re} : \gamma\}\}
\end{array}$$

Figure 4: Stack Example

4.2 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. To handle such constructors, MTAL's types are extended to a type constructor language and its kinds are extended to include functions and products, resulting in a three tiered system very similar to F_ω [Gir71, Gir72].

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

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

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 appendix A.

Note that since we have essentially embedded F_ω into our calculus, we can use phase splitting [HMM90] to compile functor systems into MTAL.

5 Related Work, Implementation, and Future Work

Our work is closely related to Cardelli's [Car97] and builds on the type theory of high level modules including work by Leroy [Ler94] and Harper and Lillibridge [HL94]. More recently Flatt and Felleisen have proposed a new advanced module system [FF98]. 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. Units can be abstracted over and linking is a first class primitive. MTAL is similar but is constrained by and tries to explain what operating systems provide at the low level, whereas Flatt and Felleisen concentrate on source level module systems. Dean 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 [Dea97]. His work is a very abstract description of this interaction and does not describe actual linking and link compatibility.

Our work is related to other work on the security of extensible systems. We formalise the checks necessary for linking, but do not address the orthogonal security concerns such as determining what

interfaces prinicipals may link against or authenticating principals. Other systems, such as the SPIN project [SFPB96], have addressed these concerns and their ideas could be combined with MTAL's.

We have implemented the ideas presented in this paper in our compiler as part of a language called Talx86. Talx86 scales the ideas of TAL up to handle real languages, is based on the instruction set of IA32 (Intel's 80x86 architecture), and uses the ideas of MTAL for its module langauge. Talx86 includes tuples, sums, arrays, recursive types, exceptions, abstract types, and translucent types. We are currently implementing subtyping and bounded type labels. We have a type checker for Talx86 object files, a link compatibility and program completeness checker, and two front ends for toy languages meant to demonstrate the viability of Talx86 as a target language. We are also working on a front end for KML [Cra98], a variant of ML with higher order modules, full polymorphism, and subtyping among other features. We have found the type checker and link checker to be effective for our toy languages. In particular, type labels help to contain the otherwise verbose of type information at the target level. Our implementation is available at `http://www.cs.cornell.edu/home/walker/talnet/tal.html#talc`.

There are two major areas for further work. First, our description of dynamic linking is only a start. Dynamic loading is a fundamental part of some languages, for example Java. However, dynamic loading is less well understood than static linking and this has lead to problems in Java's security [MF96, Sar97]. We gave an adequate operational model of dynamic loading but further research is needed into typing rules for the dynamic loading operation. Of particular importance is finding a way to dynamically load new types. Second, we could investigate other more general linking schemes. Such schemes might allow an object file to expose a different interface to each of several other object files. For example, most systems are designed as a hierarchy of subsystems. The leaf subsystems are modules, which have interfaces that hide their internal details. Using these interfaces, the modules of a level 2 subsystem are linked together and then restricted to an interface that hides the details of that subsystem. The level 2 subsystems are linked together to form level 3 subsystems and so on.

6 Conclusion

This paper presents MTAL a calculus that formalises a low level notion of linking that is close to modern operating systems' linker tools. MTAL extends the earlier work of Cardelli [Car97], providing a better explanation of intermodule references, handling cyclic dependencies between modules, supporting abstract type constructors and the compilation of phase splitable functor systems, and modelling dynamic linking and loading. MTAL extends previous work on typed assembly languages [MWCG98, MWCG97, MCGW98] to seperate compilation and linking, and previous work on the type theory of modules [Ler94, HL94, FF98] to low level languages. We have continued to build the theoretical foundations of linking necessary for a full understanding of language implementation and language based security.

References

[BSP+95] Brain 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 *Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, December 1995.

[Car88] Luca Cardelli. Structural subtyping and the notion of power type. In *Fifteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, USA, January 1988.

[Car91] Luca Cardelli. Typeful programming. In *Formal Description of Programming Concepts*. Springer-Verlag, 1991.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, January 1997.

[Cra98] Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, August 1998.

[DE98] Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Theory and Practice of Object systems*, 1998. To appear, available at `http://www-dse.doc.ic.ac.uk/projects/slurp/`.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *FourthACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, April 1997.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998. To appear.

[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 Publishing Co., 1971.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, New Orleans, LA, USA, June 1998.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland Oregon, USA, January 1994.

[HMM90] Robert Harper, Eugene Moggi, and John Mitchell. Higher-order modules and the phase distinction. In *Seventeenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 341–354, January 1990.

[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.

[Ler94] Xavier Leroy. Manifest types, modules, and sparate compilation. In *Twenty-First ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland Oregon, USA, January 1994.

[LR98] Xavier Leroy and François Rouaix. Security properties of typed applets. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, San Deigo, California, USA, January 1998.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, pages 95–118, Kyoto, Japan, March 1998.

[MF96] Gary McGraw and Edward Felten. *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley and Sons, New York, USA, 1996.

[MWCG97] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language (extended version). Technical Report TR97-1651, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, November 1997.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, January 1998.

[Nec97] George Necula. Proof-carrying code. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.

[Sar97] Vijay Saraswat. Java is not type-safe. *URL:* `http://www.research.att.com/~vj/bug.html`, August 1997.

[SFPB96] Emin Gun Sireer, Marc Fiuczynski, Przemyslaw Pardyak, and Brian Bershad. Safe dynamic linking in an extensible operating system. In *Workshop on Compiler Support for System Software*, February 1996.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, December 1993.

A MTAL

This appendix gives a full technical description of our calculus MTAL as well as statements of important theorems. The calculus is fairly independent of the core language and does not have to be used at the assembly language level. The presentation style follows closely that of previous TAL papers [MWCG97] and it should be obvious how to use TAL as the core language.

The syntax for MTAL appears in figure 5. The core language constructs fit in the elipses of c, h, w, and I. The α in $\lambda\alpha{:}\kappa.c$ binds α in c. Type constructor heap typings, value heap typings, register file typings, type constructor heaps, value heaps, and register files are considered unordered finite maps. The notation $\mathrm{dom}(M)$ is the domain of the finite maping M. All constructs are equivalent up to alpha conversion and reordering of unordered maps. Included in alpha conversion are the internal labels of a compilation unit, formally defined as the least equivalence relation containing:

$$\begin{array}{rcl}
[(\Phi_I, \Psi_I) \Rightarrow (CH, VH) : (\Phi_E, \Psi_E)] & \equiv & [(\Phi_I, \Psi_I) \Rightarrow (CH[\ell_1 := \ell_2], VH[\ell_1 := \ell_2]) : (\Phi_E, \Psi_E)] \\
 & & \text{where } \ell_1 \in \mathrm{dom}(CH) - \mathrm{dom}(\Phi_E) \wedge \ell_2 \notin \mathrm{dom}(\Phi_I) \cup \mathrm{dom}(CH) \\
[(\Phi_I, \Psi_I) \Rightarrow (CH, VH) : (\Phi_E, \Psi_E)] & \equiv & [(\Phi_I, \Psi_I) \Rightarrow (CH, VH[\ell_1 := \ell_2]) : (\Phi_E, \Psi_E)] \\
 & & \text{where } \ell_1 \in \mathrm{dom}(VH) - \mathrm{dom}(\Psi_E) \wedge \ell_2 \notin \mathrm{dom}(\Psi_I) \cup \mathrm{dom}(VH) \\
(\ell \mapsto x)[\ell_1 := \ell_2] & = & \left\{ \begin{array}{ll} \ell & \ell \neq \ell_1 \\ \ell_2 & \ell = \ell_1 \end{array} \right\} \mapsto x[\ell_1 := \ell_2]
\end{array}$$

In general $x[y := z]$ denotes capture avoiding substitution of z for y in x.

The judgements for the static semantics are summarised in figure 6, and the typing rules appear in figures 7–9. The rules for type constructor equality include only the rules for constructors essential to the module system and are supplemented by rules for the core calculus. The rules for heap values, word values, and instructions are given by the core calculus.

The linking and execution judgements are in figure 10. The linking and execution operations are specified as type directed translations and include the conditions that specify when a linking operation is valid. $M - L$ is the mapping M with entries for labels in the set L removed; $M_1 - M_2$ denotes $M_1 - \mathrm{dom}(M_2)$; $(M_{11}, M_{12}) - (M_{21}, M_{22})$ denotes $(M_{11} - M_{21}, M_{12} - M_{22})$; $(M_{11}, M_{12}) \cup (M_{21}, M_{22})$ denotes $(M_{11} \cup M_{21}, M_{12} \cup M_{22})$.

The first operation is the linking of two object files: $\vdash O_1 \text{ link } O_2 \rightsquigarrow O$. Note: Alpha variants of the source constructor and value heaps are chosen so that they their domains are disjoint. This corresponds to the linker relocating source object files. Linking two object files results in a well formed object file, in other words, the operation is type correct.

Theorem A.1 *If* $\vdash O_1 \text{ link } O_2 \rightsquigarrow O$ *then* $\vdash O$.

Linking extends to multiple object files: $\vdash O_1 \text{ link } \cdots \text{ link } O_n \rightsquigarrow O$. From the rule for $\vdash O_1 \text{ link } \cdots \text{ link } O_n \rightsquigarrow O$ we can derive conditions sufficient for its applicability in terms of only the source compilation units not the intermediate results, $\vdash O_1 \overset{\text{lc}}{\leftrightarrow} \cdots \overset{\text{lc}}{\leftrightarrow} O_n$, and prove it has the desired property.

Theorem A.2 *If* $\vdash O_1$, ..., $\vdash O_n$, *and* $\vdash O_1 \overset{\text{lc}}{\leftrightarrow} \cdots \overset{\text{lc}}{\leftrightarrow} O_n$ *then there exists a* O *such that* $\vdash O_1 \text{ link } \cdots \text{ link } O_n \rightsquigarrow O$ *and* $\vdash O$.

The second operation is the formation of an executable from an object file and an entry label: $\vdash (O, \ell) \overset{\text{prg}}{\rightsquigarrow} E$. Executable formation is type correct.

Theorem A.3 *If* $\vdash (O, \ell) \overset{\text{prg}}{\rightsquigarrow} E$ *then* $\vdash E$ *executable.*

kinds	κ	::=	$\mathbf{T} \mid \kappa_1 \to \kappa_2 \mid \langle \vec{\kappa} \rangle$
type constructors	c	::=	$\alpha \mid \ell \mid \lambda\alpha:\kappa.c \mid c_1\ c_2 \mid \langle \vec{c} \rangle \mid c.i \mid \ldots$
constructor heap types	Φ	::=	$\{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}$
value heap types	Ψ	::=	$\{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\}$
type variable contexts	Δ	::=	$\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n$
register file types	Γ	::=	$\{r_1{:}c_1, \ldots, r_n{:}c_n\}$
registers	r	::=	$\mathtt{r1} \mid \ldots$
heap values	h	::=	$\ldots$
word values	w	::=	$\mathtt{roll}^{\ell}(w) \mid \mathtt{unroll}(w) \mid \ldots$
constructor heaps	CH	::=	$\{\ell_1 \mapsto c_1, \ldots, \ell_n \mapsto c_n\}$
value heaps	VH	::=	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
register files	R	::=	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
instruction sequences	I	::=	$\ldots$
interfaces	Int	::=	(Φ, Ψ)
object files	O	::=	$[Int_I \Rightarrow (CH,\ VH) : Int_E]$
executables	E	::=	$(CH,\ VH, \ell)$
program states	P	::=	$(CH,\ VH, R, I)$

Figure 5: Syntax of MTAL

Again we can derive from the rules for multiple file linking and executable formation conditions sufficient for linking to result in a complete program, $\vdash O_1, \ldots, O_n; \ell$ complete, and prove it correct.

Theorem A.4 *If* $\vdash O_1$, *...,* $\vdash O_n$, *and* $\vdash O_1, \ldots, O_n; \ell$ *complete then there exists* O *such that* $\vdash O_1$ link $\cdots$ link $O_n \rightsquigarrow O$ *and* $\vdash O; \ell$ *complete.*

The last operation is the execution of an executable: $\vdash E \overset{\text{exec}}{\rightsquigarrow} P$. It is type correct.

Theorem A.5 *If* $\vdash E \overset{\text{exec}}{\rightsquigarrow} P$ *then* $\vdash P$ *state.*

Putting all the pieces together:

Corollary A.6 *If* $\vdash O_1$, *...,* $\vdash O_n$, $\vdash O_1 \overset{\text{lc}}{\leftrightarrow} \cdots \overset{\text{lc}}{\leftrightarrow} O_n$, *and* $\vdash O_1, \ldots, O_n; \ell$ *complete, then for any* O, E, *and* P *such that* $\vdash O_1$ link $\cdots$ link $O_n \rightsquigarrow O$, $\vdash (O, \ell) \overset{\text{prg}}{\rightsquigarrow} E$, *and* $\vdash E \overset{\text{exec}}{\rightsquigarrow} P$, *it is the case that* $\vdash P$ *state.*

Judgement	Meaning
$\Phi; \Delta \vdash c : \kappa$	c is a valid type constructor of kind κ
$\Phi; \Delta \vdash \Gamma$	Γ is a valid register file type
$\Phi; \Delta \vdash c_1 = c_2 : \kappa$	c_1 and c_2 are equal types
$\Phi; \Delta \vdash \Gamma_1 = \Gamma_2$	Γ_1 and Γ_2 are equal register file types
$\vdash \Phi$	Φ is a valid type constructor heap type
$\vdash \Phi_1 \leq \Phi_2$	Φ_1 is a type constructor heap subtype of Φ_2
$\vdash \Phi_1 \mid \Phi_2$	Φ_1 and Φ_2 are disjoint type constructor heap typings
$\vdash \Phi_1 \sim \Phi_2$	Φ_1 and Φ_2 are compatible type constructor heap typings
$\Phi \vdash \Psi$	Ψ is a valid value heap typing
$\Phi \vdash \Psi_1 = \Psi_2$	Ψ_1 and Ψ_2 are equal value heap typings
$\Phi \vdash \Psi_1 \leq \Psi_2$	Ψ_1 is a value heap subtype of Ψ_2
$\vdash \Psi_1 \mid \Psi_2$	Ψ_1 and Ψ_2 are disjoint value heap typings
$\Phi \vdash \Psi_1 \sim \Psi_2$	Ψ_1 and Ψ_2 are compatible value heap typings
$\vdash Int$	Int is a valid interface
$\vdash Int_1 = Int_2$	Int_1 and Int_2 are equal interfaces
$\vdash Int_1 \leq Int_2$	Int_1 is a subinterface of Int_2
$\vdash Int_1 \mid Int_2$	Int_1 and Int_2 are disjoint interfaces
$\vdash Int_1 \sim Int_2$	Int_1 and Int_2 are compatible interfaces
$\Phi; CH; \Psi \vdash h : c$ hval	the heap value h has type c
$\Phi; CH; \Psi; \Delta \vdash w : c$ wval	the word value w has type c
$\Phi; CH; \Psi; \Delta; \Gamma \vdash I$	I is a valid sequence of instructions
$\Phi_1 \vdash CH : \Phi_2$	the type constructor heap CH has type Φ_2
$\Phi; CH; \Psi_1 \vdash VH : \Psi_2$	the value heap VH has type Ψ_2
$\Phi; CH; \Psi \vdash R : \Gamma$	the register file R has type Γ
$\vdash O$	O is a valid object file
$\vdash E$ executable	E is a valid executable
$\vdash P$ state	P is a valid program state
$\vdash (Int_{I1}, Int_{E1}) \overset{\text{lc}}{\leftrightarrow} (Int_{I2}, Int_{E2})$	The two import and export interface pairs are link compatible
$\vdash O_1 \overset{\text{lc}}{\leftrightarrow} O_2$	O_1 and O_2 are link compatible object files
$\vdash O_1 \overset{\text{lc}}{\leftrightarrow} \cdots \overset{\text{lc}}{\leftrightarrow} O_n$	$O_1, \ldots, O_n$ are link compatible object files
$\vdash O_1$ link $O_2 \rightsquigarrow O_m$	Linking O_1 and O_2 results in O
$\vdash O_1$ link $\cdots$ link $O_n \rightsquigarrow O$	Linking $O_1, \ldots,$ and O_n results in O
$\vdash O; \ell$ complete	O is a complete object file with entry label ℓ
$\vdash O_1, \ldots, O_n; \ell$ complete	$O_1, \ldots, O_n$ are together are complete with entry label ℓ
$\vdash (O, \ell) \overset{\text{prg}}{\rightsquigarrow} E$	Forming O and ℓ into an executable results in E
$\vdash E \overset{\text{exec}}{\rightsquigarrow} P$	Executing E results in initial state P

Figure 6: Static Semantics of MTAL (judgements)

$\boxed{\Phi; \Delta \vdash c : \kappa \quad \Phi; \Delta \vdash \Gamma}$

$$\frac{\Phi; \Delta \vdash c = c : \kappa}{\Phi; \Delta \vdash c : \kappa} \qquad \frac{\Phi; \Delta \vdash \Gamma = \Gamma}{\Phi; \Delta \vdash \Gamma}$$

$\boxed{\Phi; \Delta \vdash c_1 = c_2 : \kappa \quad \Phi; \Delta \vdash \Gamma_1 = \Gamma_2}$

$$\frac{\Phi; \Delta \vdash c_2 = c_1 : \kappa}{\Phi; \Delta \vdash c_1 = c_2 : \kappa} \qquad \frac{\Phi; \Delta \vdash c_1 = c_2 : \kappa \quad \Phi; \Delta \vdash c_2 = c_3 : \kappa}{\Phi; \Delta \vdash c_1 = c_3 : \kappa}$$

$$\frac{}{\Phi; \Delta \vdash \alpha = \alpha : \kappa}\ (\Delta(\alpha) = \kappa) \qquad \frac{}{\Phi; \Delta \vdash \ell = \ell : \kappa}\ (\Phi(\ell) = \kappa)$$

$$\frac{\Phi; \Delta, \alpha : \kappa_1 \vdash c_1 = c_2 : \kappa_2}{\Phi; \Delta \vdash \lambda\alpha : \kappa_1.c_1 = \lambda\alpha : \kappa_1.c_2 : \kappa_1 \to \kappa_2}$$

$$\frac{\Phi; \Delta \vdash c_{11} = c_{21} : \kappa_1 \to \kappa_2 \quad \Phi; \Delta \vdash c_{12} = c_{22} : \kappa_1}{\Phi; \Delta \vdash c_{11}\ c_{12} = c_{21}\ c_{22} : \kappa_2}$$

$$\frac{\Phi; \Delta \vdash c_i = c'_i : \kappa_i}{\Phi; \Delta \vdash \langle c_1, \ldots, c_n \rangle = \langle c'_1, \ldots, c'_n \rangle : \langle \kappa_1, \ldots, \kappa_n \rangle}$$

$$\frac{\Phi; \Delta \vdash c = c' : \langle \kappa_1, \ldots, \kappa_n \rangle}{\Phi; \Delta \vdash c.i = c'.i : \kappa_i}\ (1 \le i \le n)$$

$$\frac{\Phi; \Delta \vdash \lambda\alpha : \kappa_1.c_1\ c_2 : \kappa_2}{\Phi; \Delta \vdash \lambda\alpha : \kappa_1.c_1\ c_2 = c_1[\alpha := c_2] : \kappa_2} \qquad \frac{\Phi; \Delta \vdash \langle c_1, \ldots, c_n \rangle.i : \kappa_i}{\Phi; \Delta \vdash \langle c_1, \ldots, c_n \rangle.i = c_i : \kappa_i}\ (1 \le i \le n)$$

$$\frac{\Phi; \Delta \vdash c_i = c'_i : \mathbf{T}}{\Phi; \Delta \vdash \{r_1{:}c_1, \ldots, r_n{:}c_n\} = \{r_1{:}c'_1, \ldots, r_n{:}c'_n\}}\ (r_1, \ldots, r_n \text{ are distinct})$$

Figure 7: Static Semantics of MTAL, Judgements for Type Constructors

$\boxed{\vdash \Phi \quad \vdash \Phi_1 \leq \Phi_2 \quad \vdash \Phi_1 \mid \Phi_2 \quad \vdash \Phi_1 \sim \Phi_2}$

$$\frac{}{\vdash \{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}} \ (\ell_1, \ldots, \ell_n \text{ are distinct})$$

$$\frac{}{\vdash \{\ell_1{:}\kappa_1, \ldots, \ell_m{:}\kappa_m\} \leq \{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}} \ (m \geq n \text{ and } \ell_1, \ldots, \ell_m \text{ are distinct})$$

$$\frac{}{\vdash \{\ell_1{:}\kappa_1, \ldots, \ell_m{:}\kappa_m\} \mid \{\ell'_1{:}\kappa'_1, \ldots, \ell'_n{:}\kappa'_n\}} \ (\{\ell_1, \ldots, \ell_m\} \cap \{\ell'_1, \ldots, \ell'_n\} = \emptyset)$$

$$\frac{\vdash \Phi_1 \quad \vdash \Phi_2 \quad \forall \ell \in \mathrm{dom}(\Phi_1) \cap \mathrm{dom}(\Phi_2) : \ \Phi_1(\ell) = \Phi_2(\ell)}{\vdash \Phi_1 \sim \Phi_2}$$

$\boxed{\Phi \vdash \Psi \quad \Phi \vdash \Psi_1 = \Psi_2 \quad \Phi \vdash \Psi_1 \leq \Psi_2 \quad \vdash \Psi_1 \mid \Psi_2 \quad \Phi \vdash \Psi_1 \sim \Psi_2}$

$$\frac{\Phi \vdash \Psi_1 = \Psi_2}{\Phi \vdash \Psi_1}$$

$$\frac{\Phi; \emptyset \vdash c_i = d_i : \mathbf{T}}{\Phi \vdash \{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\} = \{\ell_1{:}d_1, \ldots, \ell_n{:}d_n\}} \ (\ell_1, \ldots, \ell_n \text{ are distinct})$$

$$\frac{\forall 1 \leq i \leq n : \Phi; \emptyset \vdash c_i = d_i : \mathbf{T} \quad \forall n < i \leq m : \Phi; \emptyset \vdash c_i : \mathbf{T}}{\Phi \vdash \{\ell_1{:}c_1, \ldots, \ell_m{:}c_m\} \leq \{\ell_1{:}d_1, \ldots, \ell_n{:}d_n\}} \ (m \geq n \text{ and } \ell_1, \ldots, \ell_m \text{ are distinct})$$

$$\frac{}{\vdash \{\ell_1{:}c_1, \ldots, \ell_m{:}c_m\} \mid \{\ell'_1{:}c'_1, \ldots, \ell'_n{:}c'_n\}} \ (\{\ell_1, \ldots, \ell_m\} \cap \{\ell'_1, \ldots, \ell'_n\} = \emptyset)$$

$$\frac{\Phi \vdash \Psi_1 \quad \Phi \vdash \Psi_2 \quad \forall \ell \in \mathrm{dom}(\Psi_1) \cap \mathrm{dom}(\Psi_2) : \ \Phi; \emptyset \vdash \Psi_1(\ell) = \Psi_2(\ell) : \mathbf{T}}{\Phi \vdash \Psi_1 \sim \Psi_2}$$

$\boxed{\vdash \mathit{Int} \quad \vdash \mathit{Int}_1 = \mathit{Int}_2 \quad \vdash \mathit{Int}_1 \leq \mathit{Int}_2 \quad \vdash \mathit{Int}_1 \mid \mathit{Int}_2 \quad \vdash \mathit{Int}_1 \sim \mathit{Int}_2}$

$$\frac{\vdash \Phi \quad \Phi \vdash \Psi}{\vdash (\Phi, \Psi)} \qquad \frac{\vdash \Phi \quad \Phi \vdash \Psi_1 = \Psi_2}{\vdash (\Phi, \Psi_1) = (\Phi, \Psi_2)} \qquad \frac{\vdash \Phi_1 \leq \Phi_2 \quad \Phi_1 \vdash \Psi_1 \leq \Psi_2}{\vdash (\Phi_1, \Psi_1) \leq (\Phi_2, \Psi_2)} \qquad \frac{\vdash \Phi_1 \mid \Phi_2 \quad \vdash \Phi_1 \mid \Phi_2}{\vdash (\Phi_1, \Psi_1) \mid (\Phi_2, \Psi_2)}$$

$$\frac{\vdash \Phi_1 \sim \Phi_2 \quad \Phi_1 \cap \Phi_2 \vdash \Psi_1 \sim \Psi_2}{\vdash (\Phi_1, \Psi_1) \sim (\Phi_2, \Psi_2)}$$

Figure 8: Static Semantics of MTAL, Judgements for Heap Typing and Interfaces

$\boxed{\Phi; CH; \Psi; \Delta \vdash w : c \text{ wval}}$

$$\frac{\Phi; CH; \Psi; \Delta \vdash w : \tau \text{ wval}}{\Phi; CH; \Psi; \Delta \vdash \texttt{roll}^{\ell}(w) : \ell \text{ wval}} \ (CH(\ell) = \tau) \qquad \frac{\Phi; CH; \Psi; \Delta \vdash w : \ell \text{ wval}}{\Phi; CH; \Psi; \Delta \vdash \texttt{unroll}(w) : \tau \text{ wval}} \ (CH(\ell) = \tau)$$

$\boxed{\Phi \vdash CH : \Phi \quad \Phi; CH; \Psi \vdash VH : \Psi \quad \Phi; CH; \Psi \vdash R : \Gamma}$

$$\frac{\Phi; \emptyset \vdash c_i : \kappa_i}{\Phi \vdash \{\ell_1 \mapsto c_1, \ldots, \ell_n \mapsto c_n\} : \{\ell_1{:}\kappa_1, \ldots, \ell_n{:}\kappa_n\}} \ (\ell_1, \ldots, \ell_n \text{ are distinct})$$

$$\frac{\Phi; CH; \Psi \vdash h_i : c_i \text{ hval}}{\Phi; CH; \Psi \vdash \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} : \{\ell_1{:}c_1, \ldots, \ell_n{:}c_n\}} \ (\ell_1, \ldots, \ell_n \text{ are distinct})$$

$$\frac{\Phi; CH; \Psi; \emptyset \vdash w_i : c_i \text{ wval}}{\Phi; CH; \Psi \vdash \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} : \{r_1{:}c_1, \ldots, r_n : c_n\}} \ (r_1, \ldots, r_n \text{ are distinct})$$

$\boxed{\vdash O \quad \vdash E \text{ executable} \quad \vdash P \text{ state}}$

$$\frac{\begin{array}{c}\vdash (\Phi_I, \Psi_I) \\ \vdash (\Phi_A, \Psi_A) \leq (\Phi_E, \Psi_E) \\ \vdash (\Phi_I, \Psi_I) \mid (\Phi_A, \Psi_A) \\ \Phi_I \cup \Phi_A \vdash CH : \Phi_A \\ \Phi_I \cup \Phi_A; CH; \Psi_I \cup \Psi_A \vdash VH : \Psi_A\end{array}}{\vdash [(\Phi_I, \Psi_I) \Rightarrow (CH, VH) : (\Phi_E, \Psi_E)]} \qquad \frac{\begin{array}{c}\vdash \Phi \quad \Phi \vdash \Psi \\ \Phi \vdash CH : \Phi \\ \Phi; CH; \Psi \vdash VH : \Psi\end{array}}{\vdash (CH, VH, \ell) \text{ executable}} \ (\Psi(\ell) = \forall[]\{\})$$

$$\frac{\vdash \Phi \quad \Phi \vdash CH : \Phi \quad \Phi \vdash \Psi \quad \Phi; CH; \Psi \vdash VH : \Psi \quad \Phi; CH; \Psi \vdash R : \Gamma \quad \Phi; CH; \Psi; \emptyset; \Gamma \vdash I}{\vdash (CH, VH, R, I) \text{ state}}$$

Figure 9: Static Semantics of MTAL, Judgements for Object Files

$$\boxed{\vdash (Int_{I1}, Int_{E1}) \overset{\text{lc}}{\leftrightarrow} (Int_{I2}, Int_{E2}) \quad \vdash O_1 \overset{\text{lc}}{\leftrightarrow} O_2 \quad \vdash O_1 \overset{\text{lc}}{\leftrightarrow} \cdots \overset{\text{lc}}{\leftrightarrow} O_n}$$

$$\frac{\vdash Int_{I1} \sim Int_{I2} \quad \vdash Int_{I1} \sim Int_{E2} \quad \vdash Int_{I2} \sim Int_{E1} \quad \vdash Int_{E1} \mid Int_{E2}}{\vdash (Int_{I1}, Int_{E1}) \overset{\text{lc}}{\leftrightarrow} (Int_{I2}, Int_{E2})}$$

$$\frac{\vdash (Int_{I1}, Int_{E1}) \overset{\text{lc}}{\leftrightarrow} (Int_{I2}, Int_{E2})}{\vdash [Int_{I1} \Rightarrow (CH_1, VH_1) : Int_{E1}] \overset{\text{lc}}{\leftrightarrow} [Int_{I2} \Rightarrow (CH_2, VH_2) : Int_{E2}]}$$

$$\frac{\forall 1 \leq i < j \leq n : \vdash O_i \overset{\text{lc}}{\leftrightarrow} O_j}{\vdash O_1 \overset{\text{lc}}{\leftrightarrow} \cdots \overset{\text{lc}}{\leftrightarrow} O_n}$$

$$\boxed{\vdash O_1 \text{ link } O_2 \rightsquigarrow O \quad \vdash O_1 \text{ link } \cdots \text{ link } O_n \rightsquigarrow O}$$

$$\frac{\begin{array}{c}\vdash [Int_{I1} \Rightarrow (CH_1, VH_1) : Int_{E1}] \\ \vdash [Int_{I2} \Rightarrow (CH_2, VH_2) : Int_{E2}] \\ \vdash [Int_{I1} \Rightarrow (CH_1, VH_1) : Int_{E1}] \overset{\text{lc}}{\leftrightarrow} [Int_{I2} \Rightarrow (CH_2, VH_2) : Int_{E2}]\end{array}}{\begin{array}{l}\vdash [Int_{I1} \Rightarrow (CH_1, VH_1) : Int_{E1}] \text{ link } [Int_{I2} \Rightarrow (CH_2, VH_2) : Int_{E2}] \rightsquigarrow \\ \quad [(Int_{I1} \cup Int_{I2}) - (Int_{E1} \cup Int_{E2}) \Rightarrow (CH_1 \cup CH_2, VH_1 \cup VH_2) : Int_{E1} \cup Int_{E2}]\end{array}} \quad (1)$$

(1) is $\text{dom}(CH_1) \cap \text{dom}(CH_2) = \text{dom}(VH_1) \cap \text{dom}(VH_2) = \emptyset$.

$$\frac{\begin{array}{c}\vdash O_1 \text{ link } O_2 \rightsquigarrow O_2' \\ \forall 1 < i < n : \vdash O_{i-1}' \text{ link } O_i \rightsquigarrow O_i'\end{array}}{\vdash O_1 \text{ link } \cdots \text{ link } O_n \rightsquigarrow O_n'}$$

$$\boxed{\vdash O; \ell \text{ complete} \quad \vdash O_1, \ldots, O_n; \ell \text{ complete} \quad \vdash O \overset{\text{prg}}{\rightsquigarrow} E \quad \vdash E \overset{\text{exec}}{\rightsquigarrow} P}$$

$$\frac{}{\vdash [(\emptyset, \emptyset) \Rightarrow (CH, VH) : (\Phi_E, \Psi_E)]; \ell \text{ complete}} \quad (\Psi_E(\ell) = \forall[]\{\})$$

$$\frac{\begin{array}{c}\forall 1 \leq j \leq n : O_i = [(\Phi_{Ii}, \Psi_{Ii}) \Rightarrow (CH_i, VH_i) : (\Phi_{Ei}, \Psi_{Ei})] \\ \forall 1 \leq i \leq n : \forall \ell \in \text{dom}(\Phi_{Ii}) : \exists 1 \leq j \leq n : \ell \in \text{dom}(\Phi_{Ej}) \\ \forall 1 \leq i \leq n : \forall \ell \in \text{dom}(\Psi_{Ii}) : \exists 1 \leq j \leq n : \ell \in \text{dom}(\Psi_{Ej}) \\ \exists 1 \leq j \leq n : \Psi_{Ej}(\ell) = \forall[]\{\}\end{array}}{\vdash O_1, \ldots, O_n; \ell \text{ complete}}$$

$$\frac{\vdash [(\emptyset, \emptyset) \Rightarrow (CH, VH) : (\Phi_E, \Psi_E)] \quad \vdash [(\emptyset, \emptyset) \Rightarrow (CH, VH) : (\Phi_E, \Psi_E)]; \ell \text{ complete}}{\vdash ([(\emptyset, \emptyset) \Rightarrow (CH, VH) : (\Phi_E, \Psi_E)], \ell) \overset{\text{prg}}{\rightsquigarrow} (CH, VH, \ell)}$$

$$\frac{\vdash (CH, VH, \ell) \text{ executable}}{\vdash (CH, VH, \ell) \overset{\text{exec}}{\rightsquigarrow} (CH, VH, \emptyset, \texttt{jmp}\ \ell)}$$

Figure 10: MTAL Linking and Execution Judgements