Type-Safe Linking and Modular Assembly Language [*]

Neal Glew and Greg Morrisett

Department of Computer Science
Cornell University

Abstract

Linking is a low-level task that is usually vaguely specified, if at all, by language definitions. However, 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 linking and loading as a fundamental part of their operation, so it is critical to precisely define the consistency checks a linker, be it static or dynamic, 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 (Java Virtual Machine 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 compatibility. Unfortunately, this component is vaguely specified and has been a source of well publicised 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 compatibility [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.

In previous work, we presented a Typed Assembly Language (TAL) [MWCG98] that was suitable for compiling high-level core languages, and it seemed natural to extend this work 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, we found that it abstracts from many important low-level details such as binding and alpha-variance of labels, and omits certain critical features, notably support for cyclic inter-object file references, user-defined type abstraction, and dynamic linking. Hence, the goal of this paper is to build upon Cardelli's work and provide a suitable treatment of these issues. In particular, we extend core TAL with a language of typed object files and formalise the concepts of linking and link compatibility. The goals of the design were to model important properties of conventional object files and linkers (*e.g.*, Unix's `ld` or Win32's `link`), and 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.

Our design for typed object files borrows heavily from the previous work on modules for high-level languages and hence there are relatively few important innovations. However, we 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.

---

[*] This material is based on work supported in part by the AFOSR grant F49620-97-1-0013 and ARPA/RADC grant F30602-1-0317. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

To appear in the 26th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages, January 1999, San Antonio, Texas, USA.

We have implemented the resulting design as part of an on-going language-based security project. The implementation provides a set of tools for defining and type-checking a variant of IA32 (Intel 80x86) object files and interfaces. These tools, together with a compiler that maps a variant of type-safe C code to type-safe object files 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 compatibility 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. Finally, in Section 5 we discuss extending our model to include dynamic linking and dynamic loading.

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. We model execution as a one-step reduction relation between program states. In our model, 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. 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 such as integers or labels. 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. We also consider registers to be small values to simplify the syntax of operands.[1] A register file is a finite map from registers, r, to small values. The reduction relation simulates the execution of one instruction, such as:

$$\begin{array}{l}(H, \{\mathtt{r2} \mapsto 6\}, \mathtt{add}\, \mathtt{r1}, \mathtt{r2}, 3; I) \\ \longmapsto \\ (H, \{\mathtt{r1} \mapsto 9, \mathtt{r2} \mapsto 6\}, I)\end{array}$$

Here, the initial program state includes a register file that maps `r2` to the value 6, and the program counter points to an instruction sequence where the first instruction is intended to add 3 to the contents of register `r2`, placing the resulting value in `r1`. Hence, the reduction relation takes us to a program state with the same heap, an updated register file that maps `r1` to the value 9, and an updated program counter that points to the next instruction in the sequence to be executed.

2.2 Object Files

Abstractly, an object file 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, in order to provide specific examples, we will use the untyped assembly language of the previous section.

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 labels occurring in a term e:

$$\frac{\begin{array}{c}\mathcal{E} \subseteq \mathrm{dom}(H) \\ \mathcal{I} \cap \mathrm{dom}(H) = \emptyset \\ \forall \ell \in \mathrm{dom}(H). FL(H(\ell)) \subseteq \mathrm{dom}(H) \cup \mathcal{I}\end{array}}{\vdash [\mathcal{I} \Rightarrow H : \mathcal{E}]}$$

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 (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 a fashion similar to DeBruijn indices. 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* $O_1 \overset{\mathrm{lc}}{\leftrightarrow} O_2$*, iff* $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$ *(their exported labels are disjoint).*

---

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

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

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

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

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

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

Because of the condition $\mathrm{dom}(H_1) \cap \mathrm{dom}(H_2) = \emptyset$ (technically a side condition), 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 $\vdash O_1$ link $O_2 \rightsquigarrow O$ then $\vdash O$ (*i.e.*, the resulting object file is well-formed.)

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, denoted (H, ℓ). Thus, an executable is well-formed according to the following rule:

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

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 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, ignoring the parameters, 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:

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

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. In the following sections, 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 assembly 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 make type checking easier (see [MWCG98] for details.) The allocation instruction `malloc` $r[\tau_1, \ldots, \tau_n]$ includes the types of the entries of the tuple to be allocated. It assigns r the tuple type $\langle \tau_1^0, \ldots, \tau_n^0 \rangle$

---

[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.

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 and is used both to type-check individual instructions and to assign a type to instruction sequences. In general, TAL code sequences have the form $\texttt{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I$ where the code is polymorphic over the type variables $\alpha_1, \ldots, \alpha_n$, 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 is assigned the type $\forall[\alpha_1, \ldots, \alpha_n]\Gamma$. When the code is monomorphic, we abbreviate it as $\texttt{code}\Gamma.I$ and abbreviate its label's type as simply Γ.

An increment function might be written in TAL as:

$$\texttt{inc} \mapsto \texttt{code}\{\texttt{r1}{:}\mathbf{int}, \texttt{ra}{:}\{\texttt{r1}{:}\mathbf{int}\}\}.\texttt{add r1}, \texttt{r1}, 1; \texttt{jmp ra}$$

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`. Thus, the label `inc` is assigned the type:

$$\{\texttt{r1}{:}\mathbf{int}, \texttt{ra}{:}\{\texttt{r1}{:}\mathbf{int}\}\}$$

Finally, TAL assigns to heaps 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, $\Psi \vdash H : \Psi$, as done in Harper's system for mutable references [Har94]. Note that this allows circular references such as $\{\ell \mapsto \langle 1, \ell\rangle\}$ and thus extends Cardelli's work, which considered only DAG-like heaps, to general graphs.

3.2 Type Safety

The main motivation for static typing is 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. We desire that the link operation is type preserving, so the resulting object file will be well formed. Similarly, we desire that 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 an extension is checked against 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 paper, but Leroy and Rouaix [LR98] provide a discussion of some of these issues.

fact.tal:

```
  export fact:{r1 : int, ra : {r1:int}}
  fact↦ code{r1:int, ra:{r1:int}}.
        mov   r2,r1
        mov   r1,1
        jmp   loop
  loop↦ code{r1:int, r2:int, ra:{r1:int}}.
        bz    r2,ra
        mul   r1,r2,r1
        sub   r2,r2,1
        jmp   loop
```

main.tal:

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

Figure 2: Modular Factorial Example

3.3 Object Files and Interfaces

MTAL$_0$'s object files extend untyped object files with types. A MTAL$_0$ object file is a triple $[\Psi_I \Rightarrow H : \Psi_E]$ where H is a TAL heap, Ψ_I is an import interface, and Ψ_E is an export interface. 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:

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

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

---

[3] $\vdash \Psi_A \leq \Psi_E$ means that Ψ_A is a subinterface of Ψ_E, formally, $\forall \ell \in \mathrm{dom}(\Psi_E) : \Psi_A(\ell) = \Psi_E(\ell)$. Note that in a subtype setting, this could be extended to $\forall \ell \in \mathrm{dom}(\Psi_E) : \Psi_A(\ell) \leq \Psi_E(\ell)$.

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:[4]

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

Given interface compatibility, link compatibility is easily defined:

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

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 a linking operation.

MTAL$_0$ 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' 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 $\tau_e = \forall[\,]\{\texttt{r1} : \textbf{int}\}$. 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.

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

```
; Client
import type file: T
; Since file is abstract the client cannot coerce
; file to ⟨int, int⟩ or vice versa.
```

Figure 3: File Example

The production of an executable and the process of execution is the same as in the untyped calculus. However, the consistency checks are sound: the formation of an executable implies the executable is well formed, and the formation of an initial state implies the initial state is well formed. (TAL program state well formedness and reduction are described in detail in [MWCG97].)

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 very closely models the tasks of real 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] In MTAL$_0$, compatible types are equal types. In a subtype setting, this could be weakened.

4.1 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 $\mathbf{string} \rightarrow \langle \mathbf{int}, \mathbf{int} \rangle$. Because clients see this type, not a type like $\mathbf{string} \rightarrow \texttt{file}$, they can ignore the abstraction and use integer operations to directly modify the access rights. Following high-level module designs which address this issue, we add to MTAL the ability to declare abstract types in interfaces and use them in the types given to labels.

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 kinding, is a finite map from *type-labels* to kinds.[6] 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 implicitly treats ℓ and $TH(\ell)$ as equal types. This makes a decision procedure for type equality considerably more complex. We choose the second way and introduce explicit roll and unroll operations that witness the isomorphism $\ell \cong TH(\ell)$. Roll coerces an object from a concrete type to an abstract type; 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 directly manipulate members of ℓ and thus ℓ really is abstract. Consequently, the roll and unroll operations are used not only to mediate recursive types, but also to provide explicit coercions to and from abstract types.

In this respect, our treatment of label types is similar to the "generative" datatypes of SML. Unlike SML, however, our 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 Mixins [DS98, FF98]. 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.

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 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[7] and power kinds [Car88, Car91] respectively. However, as we only support globally-scoped type labels, the setting is greatly simplified because we do not need both internal and external names for types as in Harper and Lillibridge. 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, we felt that the symmetry at the type-level, together with the simplification of these language features justified the cost. A promising approach to alleviate the software engineering problems of a flat name space is to provide a means to restrict the scope of a type- or value-label during the linking process, just as the restriction operator of the pi-calculus limits the scope of a channel [Mil91]. Operationally, a restriction operator could be implemented by a tool that systematically turned global labels into local labels (*i.e.*, an explicit alpha-conversion.)

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 (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 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

---

[5] Technically 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.

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

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

Interface:

`type stack$t:` $\mathbf{T} \to \mathbf{T}$
`val stack$empty:` $\forall[\alpha{:}\mathbf{T}]\{\mathtt{ra} : \{\mathtt{r1} : \mathtt{stack\$t}\,\alpha\}\}$
`val stack$isempty:` $\forall[\alpha{:}\mathbf{T}]\{\mathtt{r1} : \mathtt{stack\$t}\,\alpha, \mathtt{ra} : \{\mathtt{r1} : \mathbf{bool}\}\}$
`val stack$push:` $\forall[\alpha{:}\mathbf{T}]\{\mathtt{r1} : \alpha, \mathtt{r2} : \mathtt{stack\$t}\,\alpha, \mathtt{ra} : \{\mathtt{r1} : \mathtt{stack\$t}\,\alpha\}\}$
`val stack$pop:` $\forall[\alpha{:}\mathbf{T}]\{\mathtt{r1} : \mathtt{stack\$t}\,\alpha, \mathtt{ra} : \{\mathtt{r1} : \mathtt{stack\$t}\,\alpha\}\}$
`val stack$top:` $\forall[\alpha{:}\mathbf{T}]\{\mathtt{r1} : \mathtt{stack\$t}\,\alpha, \mathtt{ra} : \{\mathtt{r1} : \alpha\}\}$

Figure 4: Stack Example

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, Sha98] to compile functor systems into MTAL.

5 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.

We believe that our model should extend to incorporate dynamic loading. In fact, during the course of this work, we sketched several possible extensions, each resulting from different choices in resolving specific issues. But in contrast to our previous development, none of these extensions seemed to be "the right" model. In particular, 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.

6 Related Work and Implementation

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 describes 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 also related to 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 principals 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, based on the instruction set of IA32 (Intel's 80x86 architecture), scales the ideas of TAL up to handle real languages, and uses the ideas of MTAL for its module language. As well as TAL's tuples and code, Talx86 includes sums, arrays, recursive types, exceptions, abstract types, translucent types, 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.

7 Conclusion

This paper presents MTAL a calculus that formalises a low level notion of linking similar to the linker tools of modern operating systems. 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 modeling dynamic linking and loading. MTAL unifies previous work on typed assembly languages [MWCG98, MWCG97, MCGW98] with previous work on the type theory of modules [Ler94, HL94, FF98].

Our account is a straightforward combination of the previous work of Cardelli and on typed assembly languages. The contribution of our paper is a complete account of type-safe linking for a realistic, low-level language.

References

[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 *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.

[DE98a] 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/`.

[DE98b] Sophia Drossopoulou and Susan Eisenbach. *Towards an Operational Semantics and a Proof of Type Soundness for Java*. Springer-Verlag, 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 *Fourth ACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, April 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.

[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.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.

[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 *the Twenty-First ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland Oregon, USA, January 1994.

[HMM90] Robert Harper, Eugenio 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 separate 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. Published as LNCS volume 1473, pages 28–52, Springer-Verlag.

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

[Mil91] Robert Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LCFS-91-180, Edinburgh University, 1991. Reprinted in *Logic and Algebra of Specification*, F. Brauer, W. Brauer, and H. Schwichtenberg, Eds, Springer Verlag, 1993, 203–246.

[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 Gún 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.

[Sha98] Zhong Shao. Typed cross-module compilation. In *Third ACM SIGPLAN International Conference on Functional Programming*, pages 141–152, Baltimore, MD, USA, September 1998.

[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.

kinds	κ	::=	$\mathbf{T} \mid \kappa_1 \to \kappa_2 \mid \langle \kappa_1, \ldots, \kappa_n \rangle$
type constructors	c	::=	$\alpha \mid \ell \mid \lambda\alpha{:}\kappa.c \mid c_1\ c_2 \mid \langle c_1, \ldots, c_n \rangle \mid c.i \mid \ldots$
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, ℓ)
program states	P	::=	(CH, VH, R, I)

Figure 5: Syntax of MTAL

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].

The syntax for MTAL appears in Figure 5. The core language constructs fit in the ellipses 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 mapping M. All constructs are equivalent up to alpha conversion and reordering of unordered maps. The capture avoiding substitution of z for y in x is denoted $x[y := z]$.

The judgements for the static semantics are summarised in Figure 6, and the typing rules appear in Figure 7. The rules for type constructor kinding are standard except for the rule for type labels:

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

The rules for type constructor equality are the usual reflexive, transitive, and congruence rules, and rules for the core calculus. The rules for heap values, word values (except for roll and unroll), and instructions are given by the core calculus. The rules for the judgements for type constructor and value heap typings are similar as they are both similar to record types. A heap typing is well formed when the labels defined are disjoint, and kinds/types are well formed. Two heap typings are equal when they define the same set of labels, and, map a given label to equal kinds/types. A heap typing is a heap subtype of another heap typing when it defines at least the labels of the later, the common labels are mapped to equal kinds/types, and the extra labels are mapped to well formed kinds/types. Two heap typings are disjoint when they define disjoint sets of labels. Two heap typings are compatible when they are both well formed, and labels common to both are mapped to equal kinds/types. Interface well formedness, equality, subinterfacing, disjointness, and compatibility are defined pairwise in terms of the same judgements on type constructor heap and value heap typings. The rules for roll and unroll are the standard ones except they are in terms of type labels rather than recursive types. The rules for type constructor heaps, value heaps, and register files are all similar to the rule for records. The labels must be distinct and the type maps the same labels to the types that correspond to the items.

The interesting typing rules are those for object files, $\vdash O$, executables, $\vdash E$ executable, and program states, $\vdash P$ state. An object file is well formed if there is an actual interface for the heaps that is disjoint from the import interface and a subinterface of the export interface. The heaps must have types matching the actual interface, but are checked in a context that includes the imports. The value heap is checked in a context that also includes the type constructor heap, as the latter defines abstract types the object file implements. The rules for executables and program states are similar. They require an interface that describes their heaps but this time their are no imports. In the case of executables the entry label must satisfy the entry convention. In the case of program states the register file must also have a type and the instruction sequence must be well formed.

The linking and execution judgements are in Figure 8. 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: $\vdash O_1$ link $O_2 \rightsquigarrow O$. Linking is governed by link compatibility. Two object files are link compatible when their import

Judgement	Meaning	Judgement	Meaning
$\Phi; \Delta \vdash c : \kappa$	type constructor kinding	$\Phi; \mathit{CH}; \Psi \vdash h : c$ hval	heap value typing
$\Phi; \Delta \vdash \Gamma$	valid register file type	$\Phi; \mathit{CH}; \Psi; \Delta \vdash w : c$ wval	word value typing
$\Phi; \Delta \vdash c_1 = c_2 : \kappa$	type constructor equality	$\Phi; \mathit{CH}; \Psi; \Delta; \Gamma \vdash I$	valid instruction sequence
$\Phi; \Delta \vdash \Gamma_1 = \Gamma_2$	equal register file types	$\Phi_1 \vdash \mathit{CH} : \Phi_2$	type constructor heap typing
$\vdash \Phi$	valid type constructor heap type	$\Phi; \mathit{CH}; \Psi_1 \vdash \mathit{VH} : \Psi_2$	value heap typing
$\vdash \Phi_1 \leq \Phi_2$	type constructor heap subtyping	$\Phi; \mathit{CH}; \Psi \vdash R : \Gamma$	register file typing
$\vdash \Phi_1 \mid \Phi_2$	disjoint type constr. heap typings	$\vdash O$	valid object file
$\vdash \Phi_1 \sim \Phi_2$	compatible type constr. heap typings	$\vdash E$ executable	valid executable
$\Phi \vdash \Psi$	valid value heap typing	$\vdash P$ state	valid program state
$\Phi \vdash \Psi_1 = \Psi_2$	equal value heap typings	$\vdash (\mathit{Int}_{I1}, \mathit{Int}_{E1}) \overset{\text{lc}}{\leftrightarrow}$	link compatible
$\Phi \vdash \Psi_1 \leq \Psi_2$	value heap subtyping	$(\mathit{Int}_{I2}, \mathit{Int}_{E2})$	import/export interfaces
$\vdash \Psi_1 \mid \Psi_2$	disjoint value heap typings	$\vdash O_1 \overset{\text{lc}}{\leftrightarrow} O_2$	link compatible object files
$\Phi \vdash \Psi_1 \sim \Psi_2$	compatible value heap typings	$\vdash O_1$ link $O_2 \rightsquigarrow O_m$	linking
$\vdash \mathit{Int}$	valid interface	$\vdash O; \ell$ complete	program completeness
$\vdash \mathit{Int}_1 = \mathit{Int}_2$	equal interfaces	$\vdash (O, \ell) \overset{\text{prg}}{\rightsquigarrow} E$	executable formation
$\vdash \mathit{Int}_1 \leq \mathit{Int}_2$	subinterface	$\vdash E \overset{\text{exec}}{\rightsquigarrow} P$	execution
$\vdash \mathit{Int}_1 \mid \mathit{Int}_2$	disjoint interfaces		
$\vdash \mathit{Int}_1 \sim \mathit{Int}_2$	compatible interfaces		

Figure 6: Static Semantics of MTAL (judgements)

and export interface pairs are link compatible. The later holds when the imports of one are compatible with the imports of the other and vice versa, the imports of both are compatible, and the exports of both are disjoint. The actual rule for linking looks daunting but is actually straightforward. It requires two well formed and link compatible input object files. The output object file combines the heaps and exports of the input object files and imports what the input object files imported but did not export. Note that the side condition forces the choice of alpha variants of the source heaps that have disjoint domains. 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* $\vdash O_1$ link $O_2 \rightsquigarrow O$ *then* $\vdash O$.

The second operation is the formation of an executable from an object file and an entry label: $\vdash (O, \ell) \overset{\text{prg}}{\rightsquigarrow} E$. This operation requires the object file and entry label to be complete, that is, the object file to import nothing and the entry label to exist in the object file's exports at the type required by the entry convention (in this formulation $\forall[]\{\}$). Executable formation is type correct:

Theorem A.2 *If* $\vdash (O, \ell) \overset{\text{prg}}{\rightsquigarrow} E$ *then* $\vdash E$ *executable.*

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

Theorem A.3 *If* $\vdash E \overset{\text{exec}}{\rightsquigarrow} P$ *then* $\vdash P$ *state.*

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\vdash \Phi \quad \Phi \vdash \Psi}{\vdash (\Phi, \Psi)} \qquad \frac{\vdash \Phi \quad \Phi \vdash \Psi_1 = \Psi_2}{\vdash (\Phi, \Psi_1) = (\Phi, \Psi_2)} \qquad \frac{\vdash \Phi_1 \leq \Phi_2 \quad \Phi_1 \vdash \Psi_1 \leq \Psi_2}{\vdash (\Phi_1, \Psi_1) \leq (\Phi_2, \Psi_2)} \qquad \frac{\vdash \Phi_1 \mid \Phi_2 \quad \vdash \Phi_1 \mid \Phi_2}{\vdash (\Phi_1, \Psi_1) \mid (\Phi_2, \Psi_2)} \qquad \frac{\begin{array}{c}\vdash \Phi_1 \sim \Phi_2 \\ \Phi_1 \cap \Phi_2 \vdash \Psi_1 \sim \Psi_2\end{array}}{\vdash (\Phi_1, \Psi_1) \sim (\Phi_2, \Psi_2)}$$

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

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

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

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

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

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

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

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

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

Figure 7: Static Semantics of MTAL, Judgements for Heap Typing, Interfaces, and Object Files

$$\boxed{\vdash (\mathit{Int}_{I1}, \mathit{Int}_{E1}) \overset{\text{lc}}{\leftrightarrow} (\mathit{Int}_{I2}, \mathit{Int}_{E2}) \quad \vdash O_1 \overset{\text{lc}}{\leftrightarrow} O_2}$$

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

$$\frac{\vdash (\mathit{Int}_{I1}, \mathit{Int}_{E1}) \overset{\text{lc}}{\leftrightarrow} (\mathit{Int}_{I2}, \mathit{Int}_{E2})}{\vdash [\mathit{Int}_{I1} \Rightarrow (\mathit{CH}_1, \mathit{VH}_1) : \mathit{Int}_{E1}] \overset{\text{lc}}{\leftrightarrow} [\mathit{Int}_{I2} \Rightarrow (\mathit{CH}_2, \mathit{VH}_2) : \mathit{Int}_{E2}]}$$

$$\boxed{\vdash O_1 \text{ link } O_2 \leadsto O}$$

$$\frac{\begin{array}{c}\vdash [\mathit{Int}_{I1} \Rightarrow (\mathit{CH}_1, \mathit{VH}_1) : \mathit{Int}_{E1}] \\ \vdash [\mathit{Int}_{I2} \Rightarrow (\mathit{CH}_2, \mathit{VH}_2) : \mathit{Int}_{E2}] \\ \vdash [\mathit{Int}_{I1} \Rightarrow (\mathit{CH}_1, \mathit{VH}_1) : \mathit{Int}_{E1}] \overset{\text{lc}}{\leftrightarrow} [\mathit{Int}_{I2} \Rightarrow (\mathit{CH}_2, \mathit{VH}_2) : \mathit{Int}_{E2}]\end{array}}{\begin{array}{l}\vdash [\mathit{Int}_{I1} \Rightarrow (\mathit{CH}_1, \mathit{VH}_1) : \mathit{Int}_{E1}] \text{ link } [\mathit{Int}_{I2} \Rightarrow (\mathit{CH}_2, \mathit{VH}_2) : \mathit{Int}_{E2}] \leadsto \\ \quad [(\mathit{Int}_{I1} \cup \mathit{Int}_{I2}) - (\mathit{Int}_{E1} \cup \mathit{Int}_{E2}) \Rightarrow (\mathit{CH}_1 \cup \mathit{CH}_2, \mathit{VH}_1 \cup \mathit{VH}_2) : \mathit{Int}_{E1} \cup \mathit{Int}_{E2}]\end{array}} \; (1)$$

(1) is $\mathrm{dom}(\mathit{CH}_1) \cap \mathrm{dom}(\mathit{CH}_2) = \mathrm{dom}(\mathit{VH}_1) \cap \mathrm{dom}(\mathit{VH}_2) = \emptyset$.

$$\boxed{\vdash O; \ell \text{ complete} \quad \vdash O \overset{\text{prg}}{\leadsto} E \quad \vdash E \overset{\text{exec}}{\leadsto} P}$$

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

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

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

Figure 8: MTAL Linking and Execution Judgements