Sharing in Typed Module Assembly Language

Dominic Duggan

Department of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07030.
dduggan@cs.stevens-tech.edu

Abstract. There is a growing need to provide low-overhead software-based protection mechanisms to protect against malicious or untrusted code. Type-based approaches such as proof-carrying code and typed assembly language provide this protection by relying on untrusted compilers to certify the safety properties of machine language programs. Typed Module Assembly Language (TMAL) is an extension of typed assembly language with support for the type-safe manipulation of dynamically linked libraries. A particularly important aspect of TMAL is its support for shared libraries.

1 Introduction

Protection of programs from other programs is an old and venerable problem, given new urgency with the growing use of applets, plug-ins, shareware software programs and ActiveX controls (and just plain buggy commercial code). Historically the conventional approach to providing this protection has been based on hardware support for isolating the address spaces of different running programs, from each other and from the operating system. The OS kernel and its data structures sit in a protected region of memory, and machine instructions are provided to "trap" into the kernel in a safe way to execute kernel code.

While this approach to protection is widely popularized by "modern" operating systems such as Windows 2000 and Linux, there is a growing desire to find alternatives. The problem is that this technique is a fairly heavyweight mechanism for providing protection, relying on expensive context switching between modes and between address spaces. Although application designers have learned to program around this expensive context switching (for example, buffering I/O in application space), this approach breaks down very quickly in software systems composed of separately authored subsystems that do not place much trust in each other, and where context switches may occur much more frequently than in an OS/application scenario [39].

In the OS research community, investigation of alternatives has been motivated by the demands of modular micro-kernel operating systems, where OS modules outside the kernel might not be trusted. Software fault isolation (where the loader inserts software sandboxing checks into machine code [38]) and the SPIN project (where type-safe OS modules are compiled by a trusted compiler [30]) are examples of approaches to providing protection in software rather than hardware. Sandboxing in Java VMs has also been motivated by the expense of hardware-based memory protection for applets

[39]. The commercial world is seeing an explosion in the use of component technology, exemplified by Java Beans and ActiveX controls. This use of component technology again motivates the need to find some relatively lightweight software-based approach to protection in running programs.

Proof-carrying code [32, 31] and typed assembly language [29, 28] are type-based approaches to providing this protection at low run-time cost. These approaches are examples of *self-certifying code*. A compiler produces a certificate that a program satisfies some safety property, for example, that the program is well-typed. The user of a compiled program can check that the certificate supplied with a program is valid for that program. If the check succeeds, the program can be run without run-time checks for the safety properties verified by the certificate. This approach has the advantage of moving the compiler out of the trusted computing base, and limiting the cost of safety checking to a single pass over the code at load time.

Typed assembly language (TAL) enforces a type discipline at the assembly language level, ensuring that malicious or carelessly written components cannot use "tricks" such as buffer overflows or pointer arithmetic to corrupt the data structures in a running program. Unlike the typed machine language underlying the JVM, TAL is not tied to a particular language's type system or interpreter architecture. Instead the type system is a moderately generic high-level type system with procedures, records and parametric polymorphism, while the target assembly language is any modern RISC or CISC assembly language. The type system is designed to be rich enough that a compiler can produce well-typed low-level assembly language, while at the same time having as much freedom as possible in its choice of code optimizations, parameter-passing conventions, and data and environment representations [9].

Given the importance of component technology as a motivating factor for TAL, it is clear that there should be support for manipulating components in a type-safe but flexible manner. Modular Typed Assembly Language (MTAL) extends TAL to typed object files and type-safe linking [15]. However this is limited by the assumption that all of a program is linked together before the program is run, with linking happening outside of the program itself. Any program that uses a GUI for example must make use of dynamic linking of libraries, both to avoid loading an entire library when only a small part of the library will be needed, and also to allow several processes to share a commonly used library. Indeed one can consider the operating system itself as a shared library, one that is made available in a protected region of memory to all running programs.

Our interest is in extending TAL with support for dynamic linking and shared libraries. Glew and Morrisett [15] consider some alternative approaches to extending MTAL with dynamic linking[1], but this consideration is only informal. One issue that they do not consider, that is central to our work, is what model dynamic linking should use for software components and and for linking components.

[1] Glew and Morrisett refer to "dynamic linking" as the process of linking an executable with libraries when it is first invoked, while they refer to "dynamic loading" as the linking and loading of libraries at an arbitrary point during execution. Our use of the generic term *dynamic linking* is meant in the latter sense. We provide separate operations for "loading" a module (reflecting it from the core language to the module language) and for "linking" (linking together two modules).

An obvious candidate is the ML module system [27], which provides fairly sophisticated support for type-safe linking as a language construct [24, 5]. Indeed this is the philosophy underlying MTAL, which relies on a phase-splitting transformation to translate ML modules to TAL object files. However the problem with this approach is that it leads to two different models of linking:

1. At the source language level, linking is based on applying parameterized modules. Higher-order parameterized modules may be useful for separate compilation [17, 16, 19, 25, 20], but there are still problems with supporting recursive modules [7] (as are found in Java and C).
2. At the assembly language level, linking is based on a type-safe version of the Unix `ld` command. Circular imports present no problem at this level, but much of the sophistication of the type system for ML modules is lost. This is unfortunate, since there are many lessons to be learned from ML that could fruitfully be applied to develop rich linking operations for languages such as Java.

This article describes Typed Module Assembly Language (TMAL), an extension of TAL with run-time support for loading, linking and running modules. TMAL pursues a model of linking that is closer to the MTAL approach than the ML approach, because it is closer to the form of linking used by popular languages such as Java. TMAL enriches the MTAL approach in several ways, drawing lessons from the ML experience, but also limiting the ML approach in some ways that are not limiting for Java applications, but do avoid problems with extending ML modules to support Java.

We make the following contributions to TAL:

1. We enrich TAL with coercive interface matching, which allows a module to be coerced to an expected type that makes some fields of the module "private." This is present in for example the ML module system, but not in MTAL. On the other hand, ML does not provide the same linking primitives as MTAL.
2. We enrich TAL with support for shared libraries. This is supported in the ML module language but not in MTAL. On the other hand, ML does not support recursive modules, which are present in MTAL and which complicate the definition of shared libraries.
3. We extend TAL with primitives for type-safe dynamic linking. Our approach resolves some open problems with dynamic linking and abstract data types.

TMAL arises out of work on a high-level module language, incorporating ideas from ML but with application to languages such as Java, including support for recursive DLLs and shared libraries [13]. It can be viewed as a demonstration of how the semantics of that module language can be incorporated into typed assembly language. A central aspect of this scheme is the proper treatment of shared libraries, an important issue that is addressed in the ML module language but not in more low-level typed module languages [6, 15]. A related issue is a *phase distinction* in module languages, between the link-time phase of a module and the run-time phase of a module. This issue is not often explicitly acknowledged in the literature. In TMAL it is recognized by an explicit initialization operation, `dlopen`, that provides the demarcation point between these two phases in the lifetime of a module.

In Sect. 2 we give a brief review of TAL and MTAL. In Sect. 3 we reconsider the approach used in MTAL to represent abstract types that are exported by typed object files, and in particular how type equality and type definitions are handled. In Sect. 4 we give an overview of TMAL. The next four sections describe the operations of TMAL in more detail. In Sect. 5 we describe TMAL's support for coercive interface matching. In Sect. 6 we describe how types and values can be dynamically obtained from a module in TMAL. In Sect. 7 we describe how shared libraries can be constructed in TMAL. In Sect. 8 we describe how DLLs are loaded in a type-safe manner in TMAL. Finally Sect. 10 provides our conclusions.

2 Modular Typed Assembly Language

$$
\begin{array}{rcl}
K \in \text{Kind} & ::= & ty \mid (K_1 \rightarrow K_2) \\
A, B \in \text{Type Cons} & ::= & t \mid int \mid (\lambda t : K.A) \mid A(B) \mid \\
 & & \forall[t_1,\ldots,t_m]\Gamma \mid \langle A_1^{i_1},\ldots,A_k^{i_k}\rangle \\
\Phi \in \text{Type Heap IFace} & ::= & \{t_1 : K_1,\ldots,t_k : K_k\} \\
\Psi \in \text{Value Heap IFace} & ::= & \{x_1 : A_1,\ldots,x_k : A_k\} \\
\Gamma \in \text{Register File Type} & ::= & \{r_1 : A_1,\ldots,r_k : A_k\} \\
\Delta \in \text{Type Var Context} & ::= & t_1 : K_1,\ldots,t_k : K_k \\
 & & \\
h \in \text{Heap Value} & ::= & \texttt{code}[t_1,\ldots,t_m]\Gamma.I \mid \langle w_1,\ldots,w_k\rangle \\
r \in \text{Register Name} & ::= & \texttt{r0},\texttt{r1},\ldots \\
w \in \text{Word Value} & ::= & n \mid x \mid \ldots \\
v \in \text{Small Value} & ::= & w \mid r \\
TH \in \text{Type Heap} & ::= & \{t_1 \mapsto A_1,\ldots,t_k \mapsto A_k\} \\
VH \in \text{Value Heap} & ::= & \{x_1 \mapsto h_1,\ldots,x_k \mapsto h_k\} \\
R \in \text{Register File} & ::= & \{r_1 \mapsto w_1,\ldots,r_k \mapsto w_k\} \\
I \in \text{Instruction Sequence} & ::= & i_1;\ldots;i_k \\
i \in \text{Instruction} & ::= & \texttt{add}\ r_1, r_2, v \mid \texttt{malloc}\ r[\overline{A}] \mid \texttt{jmp}\ v \mid \ldots \\
 & & \\
Int \in \text{Interface} & ::= & (\Phi,\Psi) \\
O \in \text{Object File} & ::= & [Int_I \Rightarrow (TH,VH) : Int_E] \\
E \in \text{Executable} & ::= & (TH,VH,x) \\
P \in \text{Program State} & ::= & (TH,VH,R,I)
\end{array}
$$

Fig. 1. Syntax of MTAL

In this section, we review Typed Assembly Language (TAL) and Modular Typed Assembly Language (MTAL). This review is largely based on descriptions in the literature [29, 9, 15]. The syntax of MTAL is given in Fig. 1. Typed Assembly Language can be explained as the result of carrying types in a high-level language through the compilation process, all the way to the final output of assembly language in the backend. Starting with a high-level language, say with procedures and records, programs are translated to an assembly language where procedures have been translated to code segments (with code for environment-handling) and records have been translated to heap blocks. Thus for example the procedure definition:

```
int fact (int x) {
   int y = 1;
   while (x != 0) y = (x--) * y;
   return y;
}
```

is translated to the code segment:

```
fact: code[]{a0:int,ra:∀[]{v0:int}}.
      mov   v0,1
      jmp   loop
loop: code[]{a0:int,v0:int,ra:∀[]{v0:int}}.
      bz    a0,ra
      mul   v0,a0,v0
      sub   a0,a0,1
      jmp   loop
```

The register `ra` is the continuation or return address register, pointing to the code to be executed upon return. The `fact` procedure expects an integer in the argument register `a0`, and returns to its caller with an integer in the value return register `v0`. We use MIPS gcc calling conventions to name the registers in examples.

In general heap values h have the form:

1. A code segment $\texttt{code}[t_1,\ldots,t_m]\Gamma.I$, with register file type $\Gamma = \{r_1 : A_1,\ldots,r_n : A_n\}$. This is a code segment parameterized over m type variables $t_1,\ldots,t_m$ and expecting its n arguments in the argument registers $r_1,\ldots,r_n$. The types of the values in the argument registers are specified in the register file type. I is the sequence of assembly instructions for the code sequence. This segment has the code type $\forall[t_1,\ldots,t_m]\Gamma$.
2. A heap block $\langle w_1,\ldots,w_k\rangle$ where the k values $w_1,\ldots,w_k$ are word values. Such a heap block has a heap block type $\langle A_1^{i_1},\ldots,A_k^{i_k}\rangle$, where each $i_j \in \{0,1\}$ indicates if the jth slot has been initialized.

Parametric polymorphism is used in an essential way to abstract over the call stack in typing a procedure definition. For example the most general definition of `fact` is:

```
fact: code[EnvT]{a0:int,sp:EnvT,ra:∀[]{v0:int,sp:EnvT}}. ...
```

where the `sp` register points to the environment of the calling procedure. The type parameter `EnvT` ensures that the continuation is passed the calling procedure's environment upon return.

An operational semantics is specified using program states of the form (VH, R, I), where

1. $VH = \{\overline{x \mapsto h}\}$ is a value heap, a mapping from labels to heap values h;
2. $R = \{\overline{r \mapsto h}\}$ is a register file, a mapping from register names to values; and
3. I is a sequence of typed assembly instructions.

Configurations are typed using register file types $\Gamma = \{\overline{r : A}\}$ and heap types $\Psi = \{\overline{x : A}\}$, where the latter maps from labels to types. The heap contents are unordered and may contain circular references.

Modular TAL (MTAL) extends these concepts to object files for independent compilation and type-safe linking. An untyped object file imports some values and exports some values, identified by labels pointing into the object file heap. A MTAL object file places types on the imported and exported labels. Furthermore, to support the exportation of abstract data types, a MTAL object file imports and exports types and type operators, identified by labels pointing into a type heap in the object file. An object file O in MTAL has the form

$$[(\Phi_I, \Psi_I) \Rightarrow (TH, VH) : (\Phi_E, \Psi_E)]$$

where Φ_I and Φ_E are type interfaces mapping labels to kinds, Ψ_I and Ψ_E are value interfaces mapping labels to types, $TH = \{\overline{t \mapsto A}\}$ is a type heap mapping labels to type and type operator definitions and $VH = \{\overline{x \mapsto h}\}$ is a value heap mapping labels to initial values. Φ_I and Ψ_I provide the interfaces for imported types and values, while Φ_E and Ψ_E provide the interfaces for exported types and values. An interface is a pair $Int = (\Phi, \Psi)$ of type and value heap interfaces.

There are three operations in the MTAL module language:

1. Linking $O_1 \;\texttt{link}\; O_2 \leadsto O$ combines the object files O_1 and O_2 into the single object file O. Imports in O_1 and O_2 may be resolved during linking. Interface checking ensures that resolved imports have the correct type.
2. Executable formation $(O, x) \overset{\text{prg}}{\leadsto} E$ identifies the label for executing the code of the object file. Type-checking ensures that this label is bound in the value heap, and that all imports have been resolved.
3. Execution of an executable $E \overset{\text{exec}}{\leadsto} P$ produces a program state of the operational semantics from an executable. Program states are extended to include a type heap, and have the form (TH, VH, R, I).

3 Type Heap Reconsidered

Before giving a description of TMAL, it is useful to explain how our treatment of the type heap and type identity differs from that of MTAL. In MTAL there are two views of a type:

1. Within an object file, a type exported by that object file is completely transparent. The definition of a type label is given by its binding in the type heap, TH. Because the type heap may contain circular bindings, there are word value operations

$\texttt{unroll}(w)$ and $\texttt{roll}^t(w)$ that unfold and fold the definition of a type label in the type of w, respectively. For example if a file system module defines a file abstract type t as $\langle \texttt{int}^1, \texttt{int}^1 \rangle$, and w is a word value with this type, then $\texttt{roll}^t(w)$ gives a word value with type t, that is with the concrete type folded to the defined type. This means that all types defined in object files are datatypes.
2. Outside of an object file, a type exported by that object file is completely opaque. The interface only provides the kind, and the type heap is only visible within the module.

The advantage of requiring all defined types to be datatypes is that recursive types are assured to be iso-recursive types[2], thus greatly simplifying the problem of type-checking. The problem with this approach is that it does not adequately handle type sharing for shared libraries. This is explained in more detail in [13]. Consider for example the following Objective ML code [33]:

```
module type S = sig type t; val x:t end
module S1 : S = struct type t = C; val x = C end
module S2 : (S where type t = S1.t) = S1
if true then S1.x else S2.x
```

The last conditional type-checks because `S2.x` has type `S2.t`, and the type of `S2` includes the constraint `t=S1.t`, which is also the type of `S1.x`. The structure `S1` is an example of a *shared library*, in the sense that the identity of its (abstract) type component `S1.t` is shared with `S2.t`. The datatype restriction, on the other hand, requires the insertion of marshalling and unmarshalling code at the interface of a shared library, severely curtailing its usability.

It is informally mentioned in the description of MTAL that the implementation includes singleton kinds to expose type definitions to clients of object files. However this is not formalized in the type system and therefore several important issues are left unresolved. For example it is not hard, using singleton kinds, to define two mutually recursive types in separate object files, and linking those files then results in equ-recursive types. This problem can be avoided by only allowing singleton kinds to contain type labels, where the definitions remain encapsulated in the type heap in the object file. In terms of the type system presented here, this amounts to only allowing type sharing constraints in the interface, and not allowing type definitions to be exposed.

In our type system we allow both exposure of type definitions, and type sharing, to be expressed in module interfaces. This is done without allowing equ-recursive types in the type system. This is done by separating these two uses of type information in the interface:

1. Exposure of type definitions is expressed using *box kinds*. Box kinds differ from singleton kinds in the following way: whereas singleton kinds allow implicit equality of a type identifier with the type in its singleton kind, box kinds require explicit

[2] Harper, Crary and Puri [7] make the distinction between *iso-recursive* and *equ-recursive* types. The latter require an equality theory for types that includes a rule for implicitly unrolling a recursive type. The former do not require this equality, and instead rely on operations in the language for explicitly folding and unfolding recursive types.

coercions in the term language between a type identifier and the type in its box kind.
2. Type sharing is expressed using *type sharing constraints*. The type system includes an equality theory that is merely the congruence closure of an equality between type identifiers defined by a context of type sharing constraints. Since equality is only between identifiers, there is no problem with analysing recursive constraints. This is particularly important when we consider dynamic type-checking of DLLs.

TMAL replaces the $\mathtt{roll}^t$ and `unroll` operations of MTAL, with operations for constructing and deconstructing values of types with box kind:

	Introduction	Elimination
MTAL Expression	$\mathtt{roll}^t(w)$	$\mathtt{unroll}(w)$
MTAL Side-Condition	$w : A,\ TH(t) = A$	$w : t,\ TH(t) = A$
TMAL Expression	$\mathtt{fold}_t(w)$	$\mathtt{unfold}_t(w)$
TMAL Side-Condition	$t : \boxtimes A,\ w : A$	$t : \boxtimes A,\ w : t$

Because the TMAL operations are typed independently of the type heap, box kinds can be used to expose type definitions in the interface of an object file. In contrast with singleton kinds, because explicit coercions are required between a type with box kind and the type in its kind, recursive types are guaranteed to be iso-recursive types.

4 Typed Module Assembly Language

Fig. 2 provides the syntax of Typed Module Assembly Language. In comparison with MTAL, the major changes in module interfaces are:

1. We enrich kinds with box kinds $\boxtimes A$. For simplicity we only consider simple types in this account. Box kinds generalize to type operators with some care [13].
2. We enrich import and export interfaces *Int* with a type sharing context Ξ. This is a context of equality constraints between type identifiers.
3. To support coercive interface matching, we add external labels to type and value heap interfaces. As explained in the next section, this allows some of the fields in a module to be safely made private, whereas allowing private fields in MTAL leads to the possibility of run-time name clashes.

There are two forms of module values in TMAL:

1. Modules or object files $O \equiv [Int_I \Rightarrow (TH, VH) : Int_E]$. This defines a type heap *TH* and a value heap *VH*, that may be linked with other such heaps using the TMAL operations. $Int_I \equiv (\Phi_I, \Psi_I, \Xi_I)$ is the interface of symbols imported by the module, while $Int_E \equiv (\Phi_E, \Psi_E, \Xi_E)$ is the interface of symbols exported to clients of the module.

$$
\begin{array}{rcl}
K \in \text{Kind} & ::= & ty \;\mid\; \boxtimes A \\
A, B \in \text{Type Cons} & ::= & t \;\mid\; int \;\mid\; \forall[t_1 : K_1, \ldots, t_m : K_m]\Gamma \\
 & \mid & \langle A_1^{i_1}, \ldots, A_k^{i_k} \rangle \;\mid\; \langle\langle\,\rangle\rangle \;\mid\; OT \;\mid\; Int \\
\Phi \in \text{Type Heap Interface} & ::= & \{t_1 :: \mathbf{t_1} : K_1, \ldots, t_k :: \mathbf{t_k} : K_k\} \\
\Psi \in \text{Value Heap Interface} & ::= & \{x :: \mathbf{x}_1 : A_1, \ldots, x :: \mathbf{x}_k : A_k\} \\
\Xi \in \text{Type Sharing Cons} & ::= & \{t_1 \cong t_1' \in K_1, \ldots, t_k \cong t_k' \in K_k\} \\
\Gamma \in \text{Register File Type} & ::= & \{r_1 : A_1, \ldots, r_k : A_k\} \\
\Delta \in \text{Type Var Context} & ::= & t_1 : K_1, \ldots, t_k : K_k \\
 & & \\
h \in \text{Heap Value} & ::= & \texttt{code}[t_1 : K_1, \ldots, t_m : K_m]\Gamma.I \\
 & \mid & \langle w_1, \ldots, w_k \rangle \;\mid\; \langle\langle w, \mathcal{R}(OT) \rangle\rangle \;\mid\; O \;\mid\; ST \\
r, r^m, r^s \in \text{Register Name} & ::= & \texttt{r0}, \texttt{r1}, \ldots \\
w \in \text{Word Value} & ::= & n \;\mid\; x \;\mid\; w[\overline{A}] \;\mid\; \ldots \\
v \in \text{Small Value} & ::= & w \;\mid\; r \\
TH \in \text{Type Heap} & ::= & \{t_1 :: \mathbf{t_1} : K_1 \mathcal{B}_1^A, \ldots, t_k :: \mathbf{t_k} : K_k \mathcal{B}_k^A\} \\
\mathcal{B}^A \in \text{Type Binding} & ::= & \triangleq A \quad \text{(Type Definition)} \\
 & \mid & \cong t \quad \text{(Shared Type Binding)} \\
VH \in \text{Value Heap} & ::= & \{x_1 :: \mathbf{x_1} : A_1 \mathcal{B}_1^e, \ldots, x_k :: \mathbf{x_k} : A_k \mathcal{B}_k^e\} \\
\mathcal{B}^e \in \text{Value Binding} & ::= & \triangleq h \quad \text{(Value Definition)} \\
 & \mid & \cong x \quad \text{(Shared Value Binding)} \\
R \in \text{Register File} & ::= & \{r_1 \mapsto w_1, \ldots, r_k \mapsto w_k\} \\
\rho \in \text{Renaming Substitution} & ::= & \{\mathbf{n}_1 \mapsto \mathbf{n}_1', \ldots, \mathbf{n}_k \mapsto \mathbf{n}_k'\} \\
I \in \text{Instruction Sequence} & ::= & i_1; \ldots; i_k \\
i \in \text{Instruction} & ::= & \texttt{add}\, r_1, r_2, v \;\mid\; \texttt{malloc}\, r[\overline{A}] \;\mid\; \texttt{jmp}\, v \;\mid\; \ldots \\
 & & \\
Int \in \text{Interface} & ::= & (\Phi, \Psi, \Xi) \\
OT \in \text{Object File Type} & ::= & [Int_I \Rightarrow Int_E] \\
O \in \text{Object File} & ::= & [Int_I \Rightarrow (TH, VH) : Int_E] \\
ST \in \text{Symbol Table} & ::= & \{\overline{\mathbf{t}} \mapsto \overline{t},\ \overline{\mathbf{x}} \mapsto \overline{y}\} \\
P \in \text{Program State} & ::= & (TH, VH, R, I)
\end{array}
$$

Fig. 2. Syntax of TMAL

Purpose	Instruction	Semantics
Linking,	`dllink` r_1^m, r_2^m, r_3^m	Link modules
interface	`dlcoerce` r_1^m, r_2^m, OT	Coerce to interface
matching	`dlrename` r_1^m, r_2^m, ρ	Rename external labels
Dynamic	`dlopen` r^s, r^m	Initialize module
imports	`dlsym_t` $[t] r_1^s, r_2^s, \mathbf{t}$	Import type
	`dlsym_v` $r, r^s, \mathbf{x}$	Import value
Shared	`dlsetsym_t` $r_1^m, r_2^m, t, \mathbf{t}$	Set shared type
definitions	`dlsetsym_v` $r_1^m, r_2^m, r, \mathbf{x}$	Set shared value
Dynamic	`dldynamic` $r, v, \mathcal{R}(OT)$	Construct DLL
linking	`dlload` $r^m, r_1, r_2, \mathcal{R}(OT)$	Extract module

Fig. 3. Summary of TMAL instructions

2. Symbol tables $ST \equiv \{\bar{\mathbf{t}} \mapsto \bar{t}, \bar{\mathbf{x}} \mapsto \bar{x}\}$. A symbol table arises from the initialization of a module. Initializing a module adds its type and value definitions to the type and value heaps, respectively, of the running program. The symbol table provides mappings from the external labels of the module to the heap addresses of its definitions. TMAL provides operations for dynamically importing these addresses into a running program, using a symbol table to perform a run-time lookup based on external labels.

A type heap definition $t :: \mathbf{t} : K\mathcal{B}^A$ has one of two forms:

1. A definition of the form $t :: \mathbf{t} : K \triangleq A$ defines a branded type t with external name $\mathbf{t}$ and definition A. External names are explained in the next section. The most general kind for such a type is $\boxtimes A$, revealing the structure of the type definition. This is a subkind of ty, the kind of simple types that makes type definitions opaque.
2. A definition of the form $t :: \mathbf{t} : K \cong t'$ defines a shared type t that is equated to the type t'. Such a type sharing definition can be exposed in an interface by a type sharing constraint $t \cong t' \in K$.

Similarly a value heap definition $x :: \mathbf{x} : A\mathcal{B}^e$ has one of the two forms $x :: \mathbf{x} : A \triangleq h$ (analogous to a value heap definition $x \mapsto h$ in MTAL) or $x :: \mathbf{x} : A \cong y$ (a value sharing definition). Module initialization transforms a value sharing definition to a value heap definition $x :: \mathbf{x} : A \triangleq h$ by looking up the definition of y in the heap. Initialization may detect circular value sharing definitions, which correspond to values with no clearly defined initial values.

In TMAL, modules are manipulated (loaded, coerced and linked) at run-time. This does not necessarily require modules as first-class values, and indeed TMAL is based on a module language where there is a strict separation between module values and simple values [13]. Nevertheless a critical part of the transition from a high-level language to TAL is closure conversion, where environment slots are allocated for local variables in a procedure, and the contents of the register file are saved to the environment on a procedure call. Since some local variables may be bound to module values, it is necessary in TMAL to make modules into first-class values. For example, the kernel language described in [13] includes a `letmod` construct for binding a local module identifier to a module:

$$\texttt{letmod}\ s = \mathit{Mod}\ \texttt{in}\ \mathit{Expr}$$

where *Mod* is a module language expression and *Expr* a core language expression. Closure conversion then requires that an environment slot be allocated for the free module identifier s, leading to the need for first-class modules.

This potentially has some unpleasant consequences. For example Lillibridge [23] has demonstrated that type-checking is undecidable for a type system with first-class modules. The source of this undecidability is a subtype relation between modules that allows fields to be made private, and allows type definitions to be made opaque. There is no such subtype relation in the core language of TMAL, and therefore no such subtyping for modules. This makes "first-class" modules in TMAL strictly less powerful than general first-class modules. For example with general first-class modules, it is possible for the two arms of a conditional to return modules with different interfaces, by having the result interface contain the intersection of the fields of the two modules. However the weak type system for modules in TMAL is sufficient for the purposes of closure conversion, and avoids the undecidability problems with more general type systems.

Rather than allowing type subsumption for modules, TMAL has a `dlcoerce` instruction for explicitly coercing a module to a required type. This coercion operation requires that the module's type be a subtype of the required type:

$$\begin{array}{rcl}
OT \preceq OT' & \iff & OT \equiv [Int_I \Rightarrow Int_E],\ OT' \equiv [Int'_I \Rightarrow Int'_E], \\
& & Int'_I \preceq Int_I \text{ and } Int_E \preceq Int'_E \\
Int \preceq Int' & \iff & Int \equiv (\Phi, \Psi, \Xi),\ Int' \equiv (\Phi', \Psi', \Xi'), \\
& & \Phi \leq \Phi',\ \Psi \leq \Psi', \text{ and } \Xi \text{ entails } \Xi' \\
\Phi \leq \Phi' & \iff & \Phi \equiv \{\overline{t_k :: \mathbf{t_k} : IK_k}\},\ \Phi' \equiv \{\overline{t_m :: \mathbf{t_m} : IK_m}\},\ k \geq m,\ \overline{IK_m} \leq \overline{IK'_m} \\
\Psi \leq \Psi' & \iff & \Psi \equiv \{\overline{x_k :: \mathbf{x_k} : A_k}\},\ \Psi' \equiv \{\overline{x_m :: \mathbf{x_m} : A_m}\},\ k \geq m,\ \overline{A_m} = \overline{A'_m}
\end{array}$$

So interface containment reduces to kind containment (where the only containments are of the form $\boxtimes A \leq ty$) and equality between types. The latter equality relation is the congruence closure of the equalities between type identifiers given by the sharing constraints (type operators add β-conversion).

The type formation rules for modules (object files) and symbol tables are provided in App. A.

5 Coercive Interface Matching

MTAL assumes that all field names are globally defined, and interface matching is based on these global field names. Any "implicit" renaming of an identifier requires it to be rewritten globally. There is no notion (as in our approach) of differentiating between external and internal names, with internal names locally bound, and therefore allowing local renaming of these internal names to avoid name clashes during linking. As a consequence, if fields of an object file are made private in MTAL, there is no way to rename the private fields in order to avoid name clashes when this object file is linked with other object files.

We want to support run-time linking where a library is loaded from disk into the program address space and linked with other libraries. Type safety requires a run-time type check at some point in this scenario. This type check requires that the labels do not admit implicit renaming (such as alpha-conversion in the lambda-calculus). We do not expect that all labels of the loaded library are known, only those labels specified in the expected interface in the run-time type check. Following the MTAL approach, there is the potential for confusion of labels because some of the "hidden" labels in the loaded library may be the same as labels in the libraries it is linked with.

This is the motivation for generalizing labels in type and value heap interfaces to include external names $\mathbf{t}$ and $\mathbf{x}$. Type and value heap interfaces have the form

$$\Phi = \{\overline{t :: \mathbf{t}} : \overline{K}\} \text{ and } \Psi = \{\overline{x :: \mathbf{x}} : \overline{A}\}$$

The internal names t and x represent local (type and value) heap addresses. These names admit implicit renaming or alpha-conversion, corresponding to relocating symbols in a heap. The external names $\mathbf{t}$ and $\mathbf{x}$ represent external labels that allow reference to the internal contents of a heap component of a module from outside. To allow fields of a module to be made private, external type and value names in type and value heaps include the special symbol $\star$, the name of a private field. Fields in a module are made private using the `dlcoerce` instruction, that changes the external names of fields made private to $\star$. The private external name $\star$ should never appear in a type or value heap type.

Before the contents of a module can be used by a running program, its heaps must be combined with the program heaps. This combination ensures that the internal labels of the module heaps are distinct from the internal labels in the program heaps.

Following [13], we provide three operations for combining and adapting modules. The choice of these operations is informed by an analogy between module combination and process composition in process algebras such as CCS [26]:

Operation	TMAL	CCS
Linking	`dllink` r_1^m, r_2^m, r_3^m	$(P \mid Q)$
Coercion	`dlcoerce` r_1^m, r_2^m, OT	$(P \setminus x)$
Renaming	`dlrename` r_1^m, r_2^m, ρ	$P[\rho]$

The `dllink` instruction links together two modules, combining the type and value heaps. The modules being linked together are in the source registers r_2^m and r_3^m, and the result of linking is left in the destination register r_1^m. The exports of the resulting module are the union of the exports of the two modules, while the imports are the union of the imports of the linked modules minus any imports that are resolved by linking. To obtain a coherent result, the type rules require that the external labels of the exports of the two linked modules are distinct. To maintain this restriction, the external labels of a module must always be visible in the type of the module. The linking operation also requires that the internal labels of the exports of the modules be distinct. Since internal names are bound within a module, they can be renamed to avoid name clashes when merging the fields of the modules being linked. In a concrete implementation, this renaming is handled straightforwardly by relocating the internal addresses of two object files that are linked together.

The `dlcoerce` instruction is necessary because of the absence of a subsumption rule based on interface containment for modules. This latter subsumption rule is not allowable because of the requirement that the external labels of a module must always be visible in its type. The coercion operation performs a run-time adaptation of a module, removing some of its external labels. The corresponding definitions are no longer visible to external clients of the module, but are still accessible via their internal labels to other definitions within the module. The source module is in register r_2^m, while the result of coercion is left in the destination register r_1^m. The type to which the module is coerced is specified by the object file type OT. This type annotation is mostly only for type-checking, and can be removed before execution. The part of the annotation that must be preserved during execution is the association between external names and internal names; TMAL includes instructions for looking up a field in an initialized module based on its external name.

The `dlrename` instruction is a second operation for coercive interface matching, and renames some of the external labels in a label. A renaming substitution ρ is an injective mapping from external labels to external labels. Since external names are used at run-time, this renaming substitution must be applied at run-time.

6 Dynamic Imports

The instructions given in the previous section operate on values at the module language level. At the heart of the TMAL approach are the instructions that connect the module language level to the core language level. In the $\lambda_{\text{box}}^{\text{mod}}$ module language described in [13], this connection is provided by an `init` operation that initializes a module and introduces its definitions into a local scope in a core language program. In TMAL the `init` operation is realized by three instructions, for initializing a module and for importing its definitions into the scope of a running thread:

Operation	TMAL
Initialize module	`dlopen` r^s, r^m
Import type	`dlsym_t` $[t] r_1^s, r_2^s, \mathbf{t}$
Import value	`dlsym_v` $r, r^s, \mathbf{x}$

These operations allow a program to import some of the symbols from a DLL, using the external labels of a DLL to access its definitions.

```
// Assume s1 points to loaded file system module
dlopen   s2,s1              // Initialize module
dlsym_t  [FileT] s3,s2,File // Import file type
dlsym_v  s4,s3,open         // Import file open operation
mov      a0,file_name       // Load file name
mov      ra,retpt[FileT]    // Load continuation
jmp      s4[EnvT]           // Jump to file open operation

retpt:     code[FileT]{v0:FileT,sp:EnvT} ...
file_name: "/etc/passwd"
```

Fig. 4. Example of dynamic imports

Fig. 4 gives an example of the use of these operations. Assuming the `s1` register points to a module, the `dlopen` instruction initializes that module, addings its type and value heap definitions to those of the running program. The result of initialization is a pointer, in the `s2` register, to a symbol table mapping from the external labels of the module to the addresses of its definitions in the program heaps. The `dlsym_t` instruction imports a type definition into the local context of the current thread, while the `dlsym_v` instruction imports a value definition.

The `dlsym_v` operation imports (the heap address of) a value definition from a DLL into a register, using the external label of the value definition and the symbol table of the DLL to map to the internal label. Note that the internal label cannot be known statically; the internal label is chosen at the point where the DLL is initialized and its value definitions are added to the program's value heap. This is in contrast with MTAL, where heap locations are referenced by globally bound internal names, and where renaming to avoid name clashes is not possible. In TMAL, the internal label is chosen so that there is no clash with the labels already given to program heap contents. Since the complete contents of the program heap are not known until run-time, there is no way to know the internal label during type-checking.

The important proviso in the `dlsym_v` operation is that none of the free type variables in the type of a value definition are bound by the type heap definitions addressed by the symbol table. For example, recalling the example in Fig. 4, assume that the symbol table resulting from initializing the file system module has type:

```
type File::File : ty
val open::open :
    ∀[EnvT:ty]{a0:String,sp:EnvT,ra:∀[]{v0:[File],sp:EnvT}
```

The abstract file type `File` occurs free in the type of the `open` operation. Therefore the `dlsym_v` instruction cannot import this definition immediately. The reason is that the register file type resulting from this importation would have no binding for the type identifier `File` in the type of the `s4` register.

In order to import the `open` operation, the type identifier `File` that occurs free in its type must first be imported from the DLL. This is done using the `dlsym_t` operation.

In the example in Fig. 4, the `dlsym_t` instruction binds a local type identifier `FileT` to the abstract type `File` defined by the DLL. The `s3` register is bound to a new symbol table with type:

$$\texttt{val open::}\textbf{open}\texttt{ :}\\ \quad \forall\texttt{[EnvT:ty]}\{\texttt{a0:String,sp:EnvT,ra:}\forall\texttt{[]}\{\texttt{v0:}\boxed{\texttt{FileT}}\texttt{,sp:EnvT}\}$$

The abstract file type in the type of the `open` operation has been relocated to a type bound in the local context of the current thread, therefore it is now possible to import the `open` definition from the DLL.

7 Shared Libraries

Heaps in modules may contain shared type bindings $t :: \mathbf{t} : K \cong t'$ and shared value bindings $x :: \mathbf{x} : A \cong y$. If all linking is performed before a program runs, then shared bindings are unnecessary. However shared bindings become crucial in an environment where modules are initialized at run-time.

For example, consider a module implementing a network protocol. This implementation requires some operations and types that are only provided by the operating system. Module linking can be used to combine these modules into a single module implementing the operating system with that protocol:

```
// Assume s1 points to loaded OS module
// Assume s2 points to loaded protocol module
dllink   s3,s1,s2          // Link OS, protocol modules
dlopen   s4,s3             // Initialize module
dlsym_t  [Conn] s5,s4,Conn // Import connection type
dlsym_v  s6,s5,open        // Import conn open operation
```

However there is a difficulty with this approach: the operating system will have already been initialized when the program runs. In fact the operating system is really the first module to be initialized, and a running program is just another module that has been loaded and initialized by code defined in the operating system module.

Similar remarks apply to access to OS operations from a process. The process must somehow have access to labels into the OS type and value heaps[3], but it is unrealistic to expect a program to be linked with its own copy of the OS module before execution can begin. The OS is one example of a shared library, a library that is loaded and initialized once, and that is subsequently available to other libraries as they are loaded.

The following instructions allow a program to construct a shared library:

Operation	TMAL
Set shared type	`dlsetsym_t` $r_1^m, r_2^m, t, \mathbf{t}$
Set shared value	`dlsetsym_v` $r_1^m, r_2^m, r, \mathbf{x}$

[3] As mentioned in Sect. 1, approaches such as typed assembly language should be regarded as an alternative to current heavyweight protection mechanisms such as hardware-based memory protection and the use of library stubs to trap to the OS.

The `dlsetsym_t` instruction allows a reference to a type to be added to the export list of a module, while `dlsetsym_v` instruction allows a reference to a value to be added. Once such a shared library has been constructed, the `dllink` instruction allows it to be linked with other modules.

Returning to the example above of a protocol module, suppose that this module requires a type `ProtID` of protocol identifiers and an operation `deliver` from the OS. The latter operation is used by this protocol module to deliver a protocol data unit to the next protocol above it in the protocol stack.

```
// Assume s1 points to initialized OS module
// Assume s2 points to loaded protocol module (PM)
dlsym_t      [ProtId] s1,s1,ProtId  // Import prot id type
dlsym_v      s3,s1,deliver          // Import deliver operation
dlsetsym_t   s2,s2,ProtId,ProtId    // Export protocol id to PM
dlsetsym_v   s2,s2,s3,deliver       // Export deliver to PM
dlopen       s4,s2                  // Initialize PM
```

Alternatively, if the code for initializing the protocol module is in the OS itself, then this code can be defined as:

```
// Assume s2 points to loaded protocol module (PM)
dlsetsym_t   s2,s2,ProtId,ProtId     // Export protocol id to PM
dlsetsym_v   s2,s2,deliver,deliver   // Export deliver to PM
dlopen       s4,s2                   // Initialize PM
```

where `ProtId` and `deliver` are direct references into the type and value heaps, respectively, in the module implementing the OS.

For example, considering the example above of assigning the `ProtId` and `deliver` fields of a protocol module, assume that the protocol module has type:

```
import type ProtId::ProtId
import val deliver::deliver : ∀[EnvT]{a0:ProtId,...}
export type Conn::Conn
export val open::open : ∀[EnvT]{a0:String,...}
```

Then setting the `ProtId` field with the `ProtId` type defined in the OS module results in a module with type:

```
export type ProtId'::ProtId
import val deliver::deliver : ∀[EnvT]{a0:ProtId',...}
export type Conn::Conn
export val open::open : ∀[EnvT]{a0:String,...}
sharing type ProtId' ≅ ProtId
```

If the OS module has a value heap label `deliver` with type

$$\forall\texttt{[EnvT]}\{\texttt{a0:ProtId,...}\}$$

then the type sharing constraint allows this type to be equated with the type of the `deliver` heap label in the protocol module. This allows the `dlsetsym_v` instruction to be used to assign this value field.

8 Dynamic Linking

The final set of instructions are used to attach run-time type information to a DLL. This type information is used in a run-time type check, to ensure that a DLL that is loaded from disk or from the network has the required module type. There is an instruction `dldynamic` for bundling a value with a type description, and another instruction `dlload` for checking that a DLL has a specified type.

Operation	TMAL
Construct DLL	`dldynamic` $r, v, \mathcal{R}(OT)$
Extract module	`dlload` $r^m, r_1, r_2, \mathcal{R}(OT)$

The type $\langle\langle\ \rangle\rangle$ is the type of a DLL. The `dldynamic` instruction associates a type tag $\mathcal{R}(OT)$ with the heap address of a module in a DLL value $\langle\langle w, \mathcal{R}(OT)\rangle\rangle$, of DLL type. Although module values contain annotations of both import and export interfaces, type identifiers and type annotations are stripped before execution. Therefore $\mathcal{R}(OT)$ denotes a value representation of a module type [11, 10].

The `dlload` instruction extracts a module from a DLL. This instruction also requires the value representation of a module type, the type that is expected of the module in the DLL. The instruction performs a run-time interface containment check, and if this succeeds it coerces the module in the DLL to the required type. If the interface check fails, control transfers to the failure continuation in register r_2.

The interface check includes a check for entailment of type sharing constraints. The simple form of type sharing constraints, only relating type identifiers, and the fact that the bindings in the type heap are opaque, facilitate this entailment check. The fact that type heap bindings are opaque also has the benefit that the dynamic type check cannot violate encapsulation of abstract types; this is explained in more detail in [13].

An interesting issue arises if the type of a module encapsulated in a DLL contains free type identifiers. The type description $\mathcal{R}(OT)$ bundled in a DLL then requires run-time descriptors corresponding to these free type parameters. This can be done by, for example, making the run-time descriptor type $\mathcal{R}(OT)$ into a value type [11, 10].

However a simpler approach is possible if we require that, in the source language, all free type identifiers in module types are bound to type components of other modules. This means that ultimately all free type identifiers in TMAL type descriptors are references to the type heap. Some of these free type identifiers may be bound in the type context, but ultimately instantiated to type heap addresses, because of the `dlsym_t` instruction. Although we do not elaborate on it in this account, it is possible to extend the type heap with run-time type tags for types exported by modules. This involves extending value types with tag types $Tag(t)$, and extending values with type tags $a \in Tag(t)$. The representation of module types $\mathcal{R}(OT)$ is defined inductively on types, interfaces and modules types, with a base case defined by:

$$\mathcal{R}(t) = Tag(t)$$

9 Related Work

There has been a great deal of work on the semantics of MILs, particularly in the context of the ML module system [17, 16, 19, 20, 36]. The notion of separating external and internal field names, with the latter allowing renaming to avoid name clashes, originated with Harper and Lillibridge [16]. A related idea is used by Riecke and Stone to allow fields of an object to be made private, and the object then extended with a field with the same external name. Similar notions of internal and external names appear in the module calculi of Ancona and Zucca [4] and Wells and Vestergaard [40].

Cardelli [6] gives a semantics for Unix-style linking in terms of a simple λ-calculus, ensuring that all symbols in a program are resolved before it is executed. Flatt and Felleisen [14] and Glew and Morrisett [15] extend this work to consider typed module contents and circular import dependencies. It is not clear what the type of a module is in these approaches (linking simply resolves imports against exports in a type-safe way). Glew and Morrisett do not support shared libraries (type sharing) or dynamic linking. Flatt and Felleisen allow dynamic linking of units. However the *invoke* operation for initializing a unit returns a single core language value; there is no other way for a program to access the contents of a unit. The *invoke* operation takes as arguments types and values from the running program that can be provided as imports to a library before initialization. So there are really two linking operations with units, the linking operation for merging units and the more limited linking that is implicitly part of the semantics of initialization. Our approach provides a single linking operation, and addresses the problem of sharing type (and value) identity that is not considered by these other approaches.

Crary et al [7] give an explanation of recursive modules in terms of the structure calculus [17]. Their work is predicated on the assumption that module linking is based on functor instantiation, and phase-splitting allows this to be transformed to core-language function application. As discussed in [13], it is difficult to generalize this model of linking to the kinds of module operations we consider.

Work on dynamic linking has focussed on class loading in the Java virtual machine [22]. Java has the problem of a weak MIL. On the other hand, ML has a powerful MIL but no support for dynamic linking. The current work was originally motivated by the desire to bridge this gap. Work on dynamic linking in ML has focussed on dynamic types [2, 21, 1, 37, 12]. With these approaches a dynamic value tags a value with a runtime type tag, of type *Dynamic*. This is similar to our approach to dynamic linking, but extended to modules rather than simple values, as a way of reifying modules into the core language.

A perennial problem with dynamics is that they violate encapsulation, in the sense that the underlying representation type of a value with abstract type can be exposed, by first bundling the value as a dynamic and then using runtime type checks to examine the representation type. This is an artifact of the fact that types are bound at runtime using beta-reduction. As mentioned in Sect. 8, and explained more fully in [13], our approach to DLLs avoids this problem, because the bindings in the type heap remain opaque during program execution.

Russo [35] considers an approach to adding first-class modules to ML, based on converting module values to core language values and back again. Explicit type annotations for modules ensure there are no unpleasant interactions with type inference. Russo avoids the undecidability of type-checking with first-class modules by omitting type subsumption for modules converted to core language values. This is similar to our approach to ensuring decidability with first-class modules. Our reflective treatment of DLLs is different from Russo's treatment of first-class modules. A module reified into the core language in Russo's approach retains its type, though reified to a core language type. In contrast, our reification operation (for building a DLL) masks the type entirely, and there must then be a reflection operation (with a dynamic type check) that extracts a module from a DLL. Dynamic typing is not necessary with Russo's approach, since his purpose is not to provide DLLs.

Ancona and Zucca [4], building on earlier work in mixin modules [3], provide a primitive calculus of modules that supports circular dependencies. Types are restricted to branded types. They do not consider dynamic linking or shared libraries (and the resulting issues with recursive type constraints).

Wells and Vestergaard [40] present a calculus for equational reasoning about first-class modules. They do not place any restrictions on circular import dependencies (including dependencies between value components), allowing circular definitions that lazily unwind. They verify strong normalization and confluence for their calculus, relying on a lazy reduction semantics. They do not consider typing aspects of their calculus. So for example they do not consider the problem of equ-recursive versus iso-recursive types, and they provide no support for shared libraries. Finally as with Russo's work there is no consideration of narrowing a DLL to a specific interface, an important practical facility for dynamic linking.

Crary, Hicks and Weirich [8, 18] extend TAL with primitive operations for building type-safe DLLs, on top of which more expressive dynamic linking mechanisms can be constructed. For example they are able to provide a type-safe implementation of the Unix dynamic linking API, as well as an implementation of units. Their approach amounts to extending the TAL kernel with dynamics (as described above), providing a functionality analogous to the `IQuery` interface in COM [34]. This is undeniably a smaller extension of the kernel than that suggested here. On the other hand, their approach is vulnerable to the same deficiencies with dynamics as described above. Furthermore, although their approach is type-safe, it is also more low-level than the approach described here, and so many errors that are caught statically in our type system are only caught dynamically by the `typecase` construct of dynamics. The single type failure point in our calculus is the `dlload` operation, that reflects a DLL from the core language into the module language.

10 Conclusions

We have described Typed Module Assembly Language (TMAL), an extension of typed assembly language with instructions for manipulating modules at run-time. These instructions include support for coercive interface matching, dynamically importing definitions from a library, constructing shared libraries, and using DLLs in a type-safe manner. A possible application of these mechanisms is in component-based programming environments, as demonstrated by commercial platforms based on COM or Java.

The mechanisms described here can be used to enrich such environments with flexible but type-safe operations for interconnecting modules under program control.

References

1. Martin Abadi, Luca Cardeli, Benjamin Pierce, and Didier Remy. Dynamic typing in polymorphic languages. In Peter Lee, editor, *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, California, June 1992. Carnegie-Mellon University Technical Report CMU-CS-93-105.
2. Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.
3. David Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
4. David Ancona and Elena Zucca. A primitive calculus for module systems. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, Paris, France, September 1999. Springer-Verlag.
5. Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of standard ML. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 55–64, Orlando, Florida, January 1994. ACM Press.
6. Luca Cardelli. Program fragments, linking and modularization. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 266–277. ACM Press, January 1997.
7. Karl Crary, Robert Harper, and S. Puri. What is a recursive module? In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999. ACM Press.
8. Karl Crary, Michael Hicks, and Stephanie Weirich. Safe and flexible dynamic linking of native code. In *Workshop on Types in Compilation*, Lecture Notes in Computer Science, Montreal, Quebec, Canada, September 2000. Springer-Verlag.
9. Karl Crary and Greg Morrisett. Type structure for low-level programming languages. In *Proceedings of the International Conference on Automata, Languages and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
10. Karl Crary and Stephanie Weinrich. Flexible type analysis. In *Proceedings of ACM International Conference on Functional Programming*, Paris, France, September 1999. ACM Press.
11. Karl Crary and Stephanie Weirich. Intensional polymorphism in type-erasure semantics. In *Proceedings of ACM International Conference on Functional Programming*. ACM Press, September 1998.
12. Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Transactions on Programming Languages and Systems*, 21(1):11–45, January 1999.
13. Dominic Duggan. Type-safe dynamic linking with recursive DLLs and shared libraries. Technical report, Stevens Institute of Technology, 2000.
14. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.
15. Neal Glew and Greg Morrisett. Type-safe linking and modular assembly languages. In *Proceedings of ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999. ACM Press.

16. Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994. ACM Press.
17. Robert Harper, John Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 341–354. Association for Computing Machinery, 1990.
18. Michael Hicks and Stephanie Weirich. A calculus for dynamic loading. Technical Report MS-CIS-00-07, University of Pennsylvania, 2000.
19. Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994. acmp.
20. Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–163, San Francisco, California, January 1995. ACM Press.
21. Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.
22. Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*. ACM Press, October 1998.
23. Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1997. Technical Report CMU-CS-97-122.
24. David MacQueen. Using dependent types to express modular structure. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 277–286. ACM Press, 1986.
25. David MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.
26. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
27. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Revised Definition of Standard ML*. The MIT Press, 1997.
28. Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Workshop on Compiler Support for Software Systems (WCSSS)*, Atlanta, GA, May 1999.
29. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1998.
30. B. N.Bershad, S. Savage, P. Pardyak, E. G.Sirer, M. E.Fiuczynski, D. Becker, C. Chambers, and S. Egger. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating Systems Principles*, pages 267–283, Copper Mountain, CO, 1995. ACM Press.
31. George Necula. Proof-carrying code. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1997.
32. George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Operating Systems Design and Implementation*, 1996.
33. Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ml. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.
34. Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.
35. Claudio Russo. Adding first-class modules to Standard ML. In *European Symposium on Programming*, Berlin, Germany, April 2000. Springer-Verlag.

36. Zhong Shao. Transparent modules with fully syntactic signatures. In *Proceedings of ACM International Conference on Functional Programming*, Paris, France, September 1999.
37. Mark Shields, Tim Sheard, and Simon Peyton-Jones. Dynamic typing as staged type inference. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 289–302, San Diego, California, January 1998. ACM Press.
38. Robert Wahbe, Steven Lucco, Thomas E.Anderson, and Susan L.Graham. Efficient software-based fault isolation. In *Symposium on Operating Systems Principles*, pages 203–216. ACM Press, 1993.
39. Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Symposium on Operating Systems Principles*. ACM Press, 1997.
40. Joseph Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In *European Symposium on Programming*, Berlin, Germany, April 2000. Springer-Verlag.

A Type Rules for Modules and Symbol Tables

This appendix summarizes the type rules for modules and symbol tables. The type rules for values and heaps are specified using judgements of the form given in Fig. 5. The contexts of type and value heap bindings are defined by:

$$\widetilde{\Phi} = \{(t : K) \mid (t :: \mathbf{t} : K) \in \Phi\}$$
$$\widetilde{\Psi} = \{(x : A) \mid (x :: \mathbf{x} : A) \in \Psi\}$$

The type rules for modules (object files) require that the type heap satisfies the exported type heap interface, that the value heap satisfies the exported value heap interface, and that the exported type sharing constraints are entailed by the type sharing implied by the type heap, the type sharing context, and the type sharing constraints imposed on the imports.

$$\frac{\begin{array}{c} Int_I = (\Phi_I, \Psi_I, \Xi_I) \quad Int_E = (\Phi_E, \Psi_E, \Xi_E) \\ \widetilde{\Phi} \cup \widetilde{\Phi_I}; \Delta; \Xi \cup \Xi_I \cup SHARE(TH) \vdash TH : \Phi_E \\ \widetilde{\Phi} \cup \widetilde{\Phi_I} \cup TENV(TH); \Delta; \Xi \cup \Xi_I \cup SHARE(TH) \vdash \Xi_E \\ \widetilde{\Phi} \cup \widetilde{\Phi_I} \cup TENV(TH); \Delta; \Xi \cup \Xi_I; \widetilde{\Psi} \cup \widetilde{\Psi_I} \vdash VH : \Psi_E \end{array}}{\widetilde{\Phi}; \Delta; \Xi; \widetilde{\Psi} \vdash [Int_I \Rightarrow (TH, VH) : Int_E] : [Int_I \Rightarrow Int_E]} \quad (\text{VAL OBJECT FILE})$$

$$\frac{\begin{array}{c} k \leq m \quad \{\mathbf{t}_{k+1}, \ldots, \mathbf{t}_m\} = \{\star\} \quad \widetilde{\Phi'} = \widetilde{\Phi} \cup \{\overline{t_m} : \overline{K_m}\} \quad \widetilde{\Phi'}; \Delta \vdash \widetilde{\Phi'} \\ \widetilde{\Phi'}; \Delta \vdash \overline{K'_k} \quad \widetilde{\Phi'}; \Delta \vdash \overline{\mathcal{B}^A_m} : \overline{K_m} \quad \widetilde{\Phi'}; \Delta; \Xi \vdash \overline{K_k} \leq \overline{K'_k} \end{array}}{\widetilde{\Phi}; \Delta; \Xi \vdash \{\overline{t_m :: \mathbf{t_m} : K_m \mathcal{B}^A_m}\} : \{\overline{t_k :: \mathbf{t_k} : K'_k}\}}$$
(TYPE HEAP)

$$\frac{\begin{array}{c} \widetilde{\Phi}; \Delta \vdash \Xi \\ \widetilde{\Phi}; \Delta \vdash \overline{K'_k} \quad \widetilde{\Phi}; \Delta \vdash \overline{t_k, t'_k : K'_k} \quad \widetilde{\Phi}; \Delta; \Xi \vdash \overline{t_k} = \overline{t'_k} \in \overline{K'_k} \end{array}}{\widetilde{\Phi}; \Delta; \Xi \vdash \{\overline{t_k} \cong \overline{t'_k} \in \overline{K'_k}\}} \quad (\text{SHARE HEAP})$$

$\widetilde{\Phi};\Delta \vdash \diamond$	Type context formation
$\widetilde{\Phi};\Delta \vdash \Phi'$	Type heap interface
$\widetilde{\Phi};\Delta;\Xi \vdash \Phi' = \Phi''$	Type heap interface equality
$\widetilde{\Phi};\Delta;\Xi \vdash \Phi' \leq \Phi''$	Type heap interface containment
$\widetilde{\Phi};\Delta;\Xi \vdash TH : \Phi'$	Type heap
$\widetilde{\Phi};\Delta \vdash \mathcal{B}^A : K$	Type definition
$\widetilde{\Phi};\Delta \vdash \Xi$	Sharing heap interface
$\widetilde{\Phi};\Delta;\Xi \vdash \Xi'$	Entailment of type sharing constraints
$\widetilde{\Phi};\Delta \vdash \Psi$	Value heap interface
$\widetilde{\Phi};\Delta;\Xi \vdash \Psi' = \Psi''$	Value heap interface equality
$\widetilde{\Phi};\Delta;\Xi \vdash \Psi' \leq \Psi''$	Value heap interface containment
$\widetilde{\Phi};\Delta;\Xi;\widetilde{\Psi} \vdash VH : \Psi'$	Value heap
$\widetilde{\Phi};\Delta;\Xi;\widetilde{\Psi} \vdash \mathcal{B}^e : A$	Value definition
$\widetilde{\Phi};\Delta \vdash K$	Kind formation
$\widetilde{\Phi};\Delta;\Xi \vdash K = K'$	Kind equality
$\widetilde{\Phi};\Delta;\Xi \vdash K \leq K'$	Kind containment
$\widetilde{\Phi};\Delta \vdash A : K$	Type formation
$\widetilde{\Phi};\Delta;\Xi \vdash A = B \in K$	Type equality
$\widetilde{\Phi};\Delta;\Xi \vdash [Int_I \Rightarrow Int_E] \preceq [Int'_I \Rightarrow Int'_E]$	Module type containment
$\widetilde{\Phi};\Delta;\Xi;\widetilde{\Psi} \vdash h : A$	Type of heap value
$\widetilde{\Phi};\Delta;\Xi;\widetilde{\Psi} \vdash w : A$	Type of word value
$\widetilde{\Phi};\Delta \vdash \Gamma$	Register file type
$\widetilde{\Phi};\Delta;\Xi;\widetilde{\Psi} \vdash R : \Gamma$	Register file
$\widetilde{\Phi};\widetilde{\Psi};\Xi \vdash \{\Delta;\Gamma\}\ I\ \{\Delta';\Gamma'\}$	Instruction formation

Fig. 5. Judgement Forms of TMAL

$$\frac{k \leq m \quad \{\mathbf{x}_{k+1},\ldots,\mathbf{x}_m\} = \{\star\} \quad \widetilde{\Psi'} = \widetilde{\Psi} \cup \{\overline{x_m : A_m}\} \quad \widetilde{\Phi};\Delta \vdash \widetilde{\Psi'} \quad \widetilde{\Phi};\Delta \vdash \overline{A'_k : ty} \quad \widetilde{\Phi};\Delta;\Xi;\widetilde{\Psi'} \vdash \overline{\mathcal{B}^e_m : A_m} \quad \widetilde{\Phi};\Delta;\Xi \vdash \overline{A_k} = \overline{A'_k} \in \overline{ty}}{\widetilde{\Phi};\Delta;\Xi;\widetilde{\Psi} \vdash \{\overline{x_m :: \mathbf{x_m}} : \overline{A_m \mathcal{B}^e_m}\} : \{\overline{x_k :: \mathbf{x_k}} : \overline{A'_k}\}} \quad \text{(VAL HEAP)}$$

The type rule for symbol tables is relatively straightforward. A symbol table is a mapping from type and value external names to type and value labels, respectively, in the global type and value heaps. The side-conditions that $\widetilde{\Phi'} \subseteq \widetilde{\Phi}$ means that, in checking the well-formedness of types and kinds, global type heap labels are chosen to be consistent with the internal type names used in the interface of the symbol table.

$$\frac{Int_E = (\Phi',\Psi',\Xi') \quad ST = \{\bar{\mathbf{t}} \mapsto \bar{t},\ \bar{\mathbf{x}} \mapsto \bar{x}\} \quad \widetilde{\Phi};\Delta \vdash \Phi' \quad \widetilde{\Phi};\Delta \vdash \Psi' \quad \widetilde{\Phi};\Delta;\Xi \vdash \Xi' \quad \Phi' = \{\overline{t :: \mathbf{t} : K}\} \quad \widetilde{\Phi'} \subseteq \widetilde{\Phi} \quad \Psi' = \{\overline{x :: \mathbf{x} : A}\}}{\widetilde{\Phi};\Delta;\Xi;\widetilde{\Psi} \vdash ST : Int_E} \quad \text{(VAL SYMBOL TABLE)}$$

The *TENV* and *SHARE* metafunctions are defined as follows:

$$\begin{aligned} TENV(TH) &= \{(t : K) \mid (t :: \mathbf{t} : K\mathcal{B}^A) \in TH\} \\ SHARE(TH) &= \{(t_1 \cong t_2 \in ty) \mid (t_1 :: \mathbf{t_1} : K \cong t_2) \in TH\} \end{aligned}$$

B Semantics of Module Linking Instructions

In this appendix we provide more details of the static and dynamic semantics of the instructions of TMAL. The reduction rules use configurations of the form

$$(\widetilde{TH},\widetilde{VH},R,I)$$

where R is a register file and I an instruction stream, and

$$\begin{aligned} \widetilde{TH} &= \{(x : A\ \mathcal{B}^e) \mid (x :: \mathbf{x} : A\ \mathcal{B}^e) \in TH\} \\ \widetilde{VH} &= \{(t : IK\ \mathcal{B}^A) \mid (t :: \mathbf{t} : IK\ \mathcal{B}^A) \in VH\} \end{aligned}$$

Since the type annotations are not necessary for the dynamic semantics, we sometimes abbreviate $(x : A \triangleq v)$ and $(t : IK \cong t')$ as $(x \mapsto v)$ and $(t \mapsto t')$, respectively. The type rules for `dllink`, `dlcoerce` and `dlrename` are similar to that for similar constructs described in [13]. The reduction rules for these instructions are given by:

$$\frac{\begin{array}{c} R(r^m_i) = x_i \text{ and } \widetilde{VH}(x_i) = [Int^i_I \Rightarrow (TH_i, VH_i) : Int^i_E],\ i = 2,3 \\ Int^1_E = (Int^2_E \cup Int^3_E) \quad Int^1_I = (Int^2_I \sqcup Int^3_I) - Int^3_E \\ TH_1 = TH_2 \cup TH_3 \quad VH_1 = VH_2 \cup VH_3 \quad x_1 \notin dom(\widetilde{VH}) \\ \widetilde{VH'} = \widetilde{VH} \cup \{x_1 \mapsto [Int^1_I \Rightarrow (TH_1, VH_1) : Int^1_E]\} \end{array}}{(\widetilde{TH},\widetilde{VH},R,(\texttt{dllink}\ r^m_1, r^m_2, r^m_3;\ I)) \longrightarrow (\widetilde{TH},\widetilde{VH'},R[r^m_1 \mapsto x_1],I)} \quad \text{(RED DL LINK)}$$

$$\frac{\begin{array}{c} R(r_2^m) = x_2 \text{ and } \widetilde{VH}(x_2) = [Int_I' \Rightarrow (TH', VH') : Int_E'] \\ OT = [Int_I \Rightarrow Int_E] \quad x_1 \notin dom(\widetilde{VH}) \\ \widetilde{VH'} = \widetilde{VH} \cup \{x_1 \mapsto [Int_I \Rightarrow (TH', VH') : Int_E]\} \end{array}}{(\widetilde{TH}, \widetilde{VH}, R, (\texttt{dlcoerce}\ r_1^m, r_2^m, OT;\ I)) \longrightarrow (\widetilde{TH}, \widetilde{VH'}, R[r_1^m \mapsto x_1], I)}$$

(RED DL COERCE)

$$\frac{\begin{array}{c} R(r_2^m) = x_2 \text{ and } \widetilde{VH}(x_2) = [Int_I \Rightarrow (TH', VH') : Int_E] \\ x_1 \notin dom(\widetilde{VH}) \quad \widetilde{VH''} = \widetilde{VH} \cup \{x_1 \mapsto [\rho(Int_I) \Rightarrow (\rho(TH'), \rho(VH')) : \rho(Int_E)]\} \end{array}}{(\widetilde{TH}, \widetilde{VH}, R, (\texttt{dlrename}\ r_1^m, r_2^m, \rho;\ I)) \longrightarrow (\widetilde{TH}, \widetilde{VH''}, R[r_1^m \mapsto x_1], I)}$$

(RED DL RENAME)

The following type rule and reduction rule explain the semantics of the `dlopen` operation. This operation expects register r^m to point to a module with type $[Int_I \Rightarrow Int_E]$, where $Int_I = (\{\}, \{\}, \{\})$. The operation leaves in register r^s a pointer to a symbol table with interface Int_E, after adding the heaps of the module to the program heaps:

$$\frac{\widetilde{\Phi}; \Delta; \widetilde{\Psi}; \Xi \vdash r^m : [(\{\}, \{\}, \{\}) \Rightarrow Int_E] \quad \Gamma' = \Gamma[r^s : Int_E]}{\widetilde{\Phi}; \widetilde{\Psi}; \Xi \vdash \{\Delta; \Gamma\}\ (\texttt{dlopen}\ r^s, r^m)\ \{\Delta; \Gamma'\}} \quad \text{(INSTR DL OPEN)}$$

$$\frac{\begin{array}{c} \widehat{R}(r^m) = x \text{ and } \widetilde{VH}(x) = [(\{\}, \{\}, \{\}) \Rightarrow (TH', VH') : Int_E] \\ ST = \{(\mathbf{t} \mapsto t) \mid (t :: \mathbf{t} : K) \in Int_E\} \cup \{(\mathbf{x} \mapsto x) \mid (x :: \mathbf{x} : A) \in Int_E\} \\ x' \notin idom(VH) \cup idom(VH') \quad idom(\widetilde{TH}) \cap idom(TH') = \{\} \quad \widetilde{TH''} = \widetilde{TH} \cup \widetilde{TH'} \\ idom(VH) \cap idom(VH') = \{\} \quad \widetilde{VH''} = CLOS(\widetilde{VH} \cup \widetilde{VH'} \cup \{x' \mapsto ST\}) \end{array}}{(\widetilde{TH}, \widetilde{VH}, R, (\texttt{dlopen}\ r^s, r^m;\ I)) \longrightarrow (\widetilde{TH''}, \widetilde{VH''}, R[r^s : Int_E \triangleq x'], I)}$$

(RED DL OPEN)

The $CLOS(VH)$ operation removes shared value bindings of the form $x : A \cong y$ from the value heap, by dereferencing y to its heap value definition:

$$CLOS(\widetilde{VH}) = \{(x : A \triangleq h) \mid (x : A\mathcal{B}^e) \in \widetilde{VH},\ h = DEREF_{\widetilde{VH}}(x)\}$$

$$DEREF_{\widetilde{VH}}(x) = \begin{cases} h \text{ if } (x : A \triangleq h) \in \widetilde{VH} \\ h \text{ if } (x : A \cong y) \in \widetilde{VH},\ h = DEREF_{\widetilde{VH}}(y) \end{cases}$$

The result of $CLOS(\widetilde{VH})$ is undefined if $\widetilde{VH}$ contains circular shared value bindings. This corresponds to an initialization failure due to cycles in the specification of initial values.

The type rule and reduction rule for the `dlsym_v` instruction are as follows:

$$\frac{\widetilde{\Phi};\Delta;\widetilde{\Psi};\Xi \vdash r^s : (\Phi',\Psi',\Xi') \quad \Psi' = (\Psi_1, x :: \mathbf{x} : A, \Psi_2) \quad FV(A) \cap idom(\Phi') = \{\} \quad \Gamma' = \Gamma[r : A]}{\widetilde{\Phi};\widetilde{\Psi};\Xi \vdash \{\Delta;\Gamma\}\ (\mathtt{dlsym\_v}\ r, r^s, \mathbf{x})\ \{\Delta;\Gamma'\}}$$

(INSTR DL SYMV)

$$\frac{\widetilde{VH}(\widehat{R}(r^s)) = ST}{(\widetilde{TH},\widetilde{VH},R,(\mathtt{dlsym\_v}\ r, r^s, \mathbf{x};\ I)) \longrightarrow (\widetilde{TH},\widetilde{VH},R[r \mapsto ST(\mathbf{x})],I)}$$

(RED DL SYMV)

$\widehat{R}(v)$ denotes the application of the register file R to the small value (register or word value) v:

$$\widehat{R}(v) = \begin{cases} w & \text{if } v = w \\ R(r) & \text{if } v = r \end{cases}$$

The type rule and reduction rule for the `dlsym_t` instruction are as follows:

$$\frac{\begin{array}{c}\widetilde{\Phi};\Delta;\widetilde{\Psi};\Xi \vdash r_2^s : (\Phi',\Psi',\Xi') \\ \Phi' = (\Phi_1, t :: \mathbf{t} : K, \Phi_2) \quad \widetilde{\Phi} \cup \widetilde{\Phi'};\Delta;\Xi \cup \Xi' \vdash K \leq K' \quad t \notin dom(\widetilde{\Phi}) \\ \Delta' = \Delta \cup \{t : K'\} \quad \Gamma' = \Gamma[r_1^s : (\Phi_1 \cup \Phi_2, \Psi', \Xi')]\end{array}}{\widetilde{\Phi};\widetilde{\Psi};\Xi \vdash \{\Delta;\Gamma\}\ (\mathtt{dlsym\_t}\ [t : K']r_1^s, r_2^s, \mathbf{t})\ \{\Delta';\Gamma'\}}$$

(INSTR DL SYMT)

$$\frac{\widetilde{VH}(\widehat{R}(r^s)) = ST \quad ST = ST' \cup \{\mathbf{t} \mapsto t\} \quad R' = R[r_1^s \mapsto ST']}{(\widetilde{TH},\widetilde{VH},R,(\mathtt{dlsym\_t}\ [t' : K']r_1^s, r_2^s, \mathbf{t};\ I)) \longrightarrow (\widetilde{TH},\widetilde{VH},R',\{t/t'\}I)}$$

(RED DL SYMT)

In the reduction rule, the local type identifier t' is bound to the global type heap address t of the type definition pointed to by the symbol table. This allows the remainder of the instruction stream I to access the value heap definitions, pointed to by the symbol table, that have references to this type heap address.

Type heap addresses and type identifiers serve only to support type-checking of the assembly code, and are stripped for run-time execution. The substitution $\{t/t'\}I$ is performed only in the abstract reduction semantics. Although we do not elaborate on it further here, the `dlsym_t` instruction can be generalized to import run-time type tags from a DLL, for languages such as Java and Modula-3 that associate type tags with some values.

The type rule for the `dlsetsym_v` instruction is reasonably straightforward. The only complication is that the type of the value field being assigned may have free type identifiers that are bound in the module. The typing rule relies on type sharing constraints in the module type that relate these locally bound type identifiers to global identifiers bound by the program type heap:

$$\frac{\begin{array}{c}\widetilde{\Phi};\Delta;\widetilde{\Psi};\Xi \vdash v : A \quad \widetilde{\Phi};\Delta;\widetilde{\Psi};\Xi \vdash r_2^m : [Int_I \Rightarrow Int_E] \\ Int_I = (\Phi_I,\Psi_I,\Xi_I) \quad Int_E = (\Phi_E,\Psi_E,\Xi_E) \quad (x :: \mathbf{x} : B) \in \Psi_I \\ \widetilde{\Phi} \cup \widetilde{\Phi_I} \cup \widetilde{\Phi_E};\Delta;\Xi \cup \Xi_I \cup \Xi_E \vdash A = B \in ty \\ \Gamma' = \Gamma[r_1^m \mapsto [(Int_I - \{x :: \mathbf{x} : B\}) \Rightarrow (Int_E \cup \{x :: \mathbf{x} : B\})]]\end{array}}{\widetilde{\Phi};\widetilde{\Psi};\Xi \vdash \{\Delta;\Gamma\}\ (\texttt{dlsetsym\_v}\ r_1^m, r_2^m, v, \mathbf{x})\ \{\Delta;\Gamma'\}} \text{(INSTR DL SETSYMV)}$$

$$\frac{\begin{array}{c}\widetilde{VH}(\widehat{R}(r_2^m)) = [Int_I \Rightarrow (TH',VH') : Int_E] \quad (x :: \mathbf{x} : A) \in Int_I \quad \widehat{R}(v) = y \\ Int_I' = Int_I - \{x :: \mathbf{x} : A\} \quad Int_E' = Int_E \cup \{x :: \mathbf{x} : A\} \\ z \notin dom(\widetilde{VH}) \quad \widetilde{VH''} = \widetilde{VH} \cup \{z \mapsto [Int_I' \Rightarrow (TH', VH' \cup \{x :: \mathbf{x} : A \cong y\}) : Int_E']\}\end{array}}{(\widetilde{TH},\widetilde{VH},R,(\texttt{dlsetsym\_v}\ r_1^m, r_2^m, v, \mathbf{x};\ I)) \longrightarrow (\widetilde{TH},\widetilde{VH''},R[r_1^m \mapsto z],I)}$$
(RED DL SETSYMV)

The `dlsetsym_t` instruction for assigning a type field in a module similarly relies on type sharing to equate any local type identifiers with global type identifiers in the kind of the type being assigned. Free type identifiers may appear free in the kind of a field with box kind. Once a type field has been assigned, a type sharing constraint is added to the export interface of the module, to allow subsequent value fields to be assigned:

$$\frac{\begin{array}{c}\widetilde{\Phi};\Delta \vdash t' : K \quad \widetilde{\Phi};\Delta;\widetilde{\Psi};\Xi \vdash r_2^m : [Int_I \Rightarrow Int_E] \\ Int_I = (\Phi_I,\Psi_I,\Xi_I) \quad Int_E = (\Phi_E,\Psi_E,\Xi_E) \quad (t :: \mathbf{t} : K') \in \Phi_I \\ \widetilde{\Phi} \cup \widetilde{\Phi_I} \cup \widetilde{\Phi_E};\Delta;\Xi \cup \Xi_I \cup \Xi_E \vdash K = K' \\ \Gamma' = \Gamma[r_1^m \mapsto [(Int_I - \{t :: \mathbf{t} : K'\}) \Rightarrow (Int_E \cup \{(t :: \mathbf{t} : K'),(t \cong t' \in K')\})]]\end{array}}{\widetilde{\Phi};\widetilde{\Psi};\Xi \vdash \{\Delta;\Gamma\}\ (\texttt{dlsetsym\_t}\ r_1^m, r_2^m, t', \mathbf{t})\ \{\Delta;\Gamma'\}}$$
(INSTR DL SETSYMT)

$$\frac{\begin{array}{c}\widetilde{VH}(\widehat{R}(r_2^m)) = [Int_I \Rightarrow (TH',VH') : Int_E] \quad (t :: \mathbf{t} : K) \in Int_I \\ Int_I' = Int_I - \{t :: \mathbf{t} : K\} \quad Int_E' = Int_E \cup \{(t :: \mathbf{t} : K),(t \cong t' \in K)\} \\ z \notin dom(\widetilde{VH}) \quad \widetilde{VH''} = \widetilde{VH} \cup \{z \mapsto [Int_I' \Rightarrow (TH' \cup \{t :: \mathbf{t} : K \cong t'\}, VH') : Int_E']\}\end{array}}{(\widetilde{TH},\widetilde{VH},R,(\texttt{dlsetsym\_t}\ r_1^m, r_2^m, t', \mathbf{t};\ I)) \longrightarrow (\widetilde{TH},\widetilde{VH''},R[r_1^m \mapsto z],I)}$$
(RED DL SETSYMT)

Finally the reduction rules for the instructions for creating a DLL, and for extracting a module from a DLL, are as follows:

$$\frac{x \notin dom(VH) \quad \widetilde{VH'} = \widetilde{VH} \cup \{x \mapsto \langle\langle \widehat{R}(v), \mathcal{R}(OT)\rangle\rangle\}}{(\widetilde{TH},\widetilde{VH},R,(\texttt{dldynamic}\ r, v, \mathcal{R}(OT);\ I)) \longrightarrow (\widetilde{TH},\widetilde{VH'},R[r \mapsto x],I)}$$
(RED DL DYNAMIC)

$$\frac{\begin{array}{c}\widehat{R}(r_1) = x \text{ and } \widetilde{VH}(x) = \langle\langle y, \mathcal{R}(OT)\rangle\rangle \\ OT = [Int_I \Rightarrow Int_E] \quad OT'' = [Int''_I \Rightarrow Int''_E] \quad \widetilde{VH}(y) = [Int'_I \Rightarrow (TH', VH') : Int'_E] \\ TENV(\widetilde{TH}); \{\}; SHARE(\widetilde{TH}) \vdash [Int_I \Rightarrow Int_E] \preceq [Int''_I \Rightarrow Int''_E] \\ z \notin dom(\widetilde{VH}) \quad \widetilde{VH''} = \widetilde{VH} \cup \{z \mapsto [Int''_I \Rightarrow (TH', VH') : Int''_E]\}\end{array}}{(\widetilde{TH}, \widetilde{VH''}, R, (\texttt{dlload}\, r^m, r_1, r_2, \mathcal{R}(OT'');\ I)) \longrightarrow (\widetilde{TH}, \widetilde{VH''}, R[r^m \mapsto z], I)}$$

(RED DL DYNAMIC)