Modules with Interfaces for Dynamic Linking and Communication

Yu David Liu Scott F. Smith

Department of Computer Science
The Johns Hopkins University
{yliu, scott}@cs.jhu.edu

Abstract. Module systems are well known as a means for giving clear interfaces for the static linking of code. This paper shows how adding explicit interfaces to modules for 1) dynamic linking and 2) cross-computation communication can increase the declarative, encapsulated nature of modules, and build a stronger foundation for language-based security and version control. We term these new modules *Assemblages*.

1 Introduction

Module systems traditionally excel in static linking. In typical module systems such as ML functors [Mac84], mixins [BC90,DS96,HL02], Units [FF98] and Jiazzi [MFH01], each module has a list of features (including functions, classes, types, submodules, *etc*) as exports, and a list of imported features. Applications can then be built by statically linking together a collection of modules. Definitions of imported features are unknown when the module is written, but all names must be resolved the moment the module is loaded and executed.

The rapid evolution of the Internet has changed the landscape of software design, and now it is more common that encapsulated code segments contain name references that can only be resolved *at runtime*. Applications of this nature can generally be placed into two distinct categories, which we term *dynamic plugins* and *reactive computations*, respectively.

Dynamic Plugins A dynamic plugin is our term for dynamically linked code: a piece of code is dynamically plugged into an already running computation. Dynamic plugins are ubiquitous in modern large-scale software designs, from browser and operating system plugins, to incremental as-needed loading of application features, and for dynamic update of critical software systems demanding non-stop services. When the main application is loaded, future dynamic plugins may be completely unknown, and name references to them must be bound at runtime.

Reactive Computations Reactive computations are the collection of autonomous coarse-grained computations that are reactive to requests in a distributed environment. A reactive computation has its own collection of objects;

so, a thread in a JVM is not a reactive computation, but a whole JVM is, as is an application domain in the Microsoft CLR. In the grid computing paradigm, each grid cell can be viewed as a reactive computation communicating with other cells. The communication between a Java applet and its loading virtual machine can also be viewed as one between two reactive computations. Compared to a dynamic plugin, reactive computations are much more loosely coupled since object references cannot be shared; this is because the computations are on different nodes or involve parties with limited trust in each other.

As with dynamic plugins, cross-computation invocation also requires name binding at runtime: when a computation is loaded, it cannot yet know about the existence of other computations it intends to communicate with.

Previous research on dynamic linking [Blo83,LB98,SPW03,DLE03] and software updating [HMN01,BHSS03] also focuses on the dynamic plugin problem, and numerous projects on remote invocation such as RPC and RMI have targeted reactive computations. In this paper we develop a new *module-centered* approach to these two forms of computation which offers advantages over the existing approaches by making interfaces more explicit, and increasing understanding and expressiveness of the code. In particular, we develop a new module theory that, along with a standard form of interface for static linking, also incorporates explicit interfaces for dynamic plugins and reactive computations.

2 Our Approach: Assemblages

In this paper, we present a type safe module calculus with a single notion of module that supports static linking, dynamic linking, and communication between reactive computations. Our modules are called *assemblages*. Assemblages are code blocks that can be statically linked with one another to form bigger assemblages. When an assemblage is loaded into memory, it becomes an *assemblage runtime* (or *runtime* for short), which serves as a reactive computation with an explicit interface for cross-computation communication and an explicit interface for dynamically plugging other assemblages into its current runtime. All interactions of a runtime should be through these interfaces alone, giving complete encapsulation.

2.1 Basic Model

Fig. 1 shows the three fundamental processes our calculus addresses. We now introduce them separately, together with concepts and terms we will use throughout the paper.

Static Linking Our module calculus comes with a fairly standard notion of static linking. Fig. 1 (a) shows expression $A + A'$, which represents the static linking of assemblages A and A'. One somewhat unusual feature is that assemblages themselves are first-class values in our calculus, and static linking is also

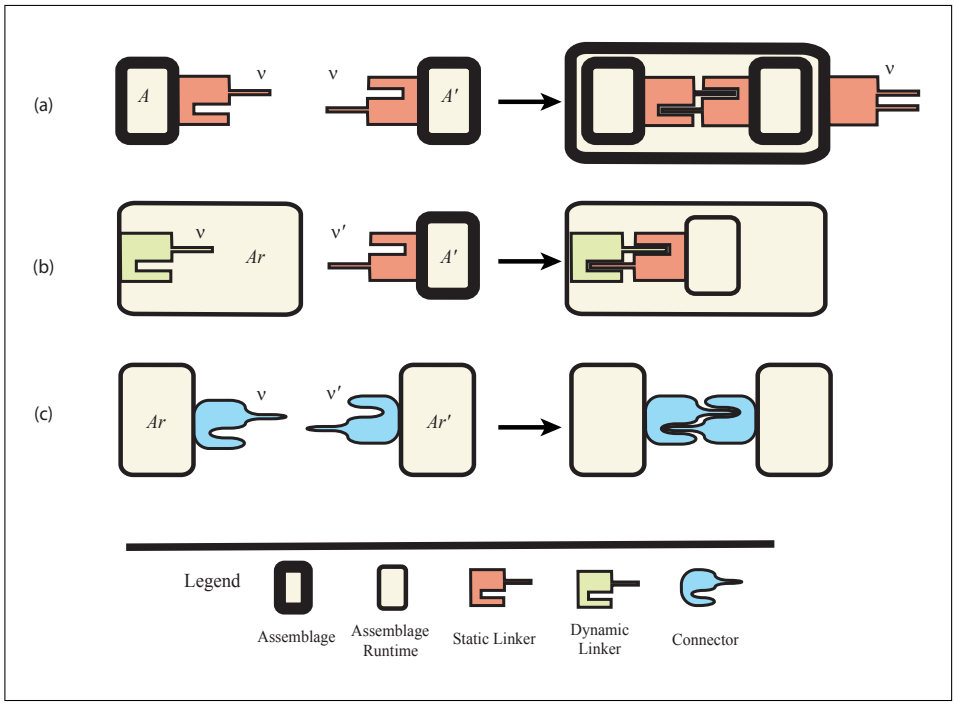

Fig. 1. Three Fundamental Processes (a) Static Linking (b) Dynamic Linking (c) Cross-computation Communication

a first-class expression: linking can happen anywhere in a program. Each assemblage is associated with a list of *static linkers*, each of which defines what features it imports and what ones it exports. In the static linking process, suppose A and A' each have a static linker of the name ν. The pair of static linkers are thus matched against each other, where imports of A are satisfied by exports of A' and *vice versa.* In the resulting assemblage, a new static linker with name ν will be created, which bears the exported features of both static linkers. Static linkers that are not matched will also be carried to the composed assemblage (suppose A has a static linker named ν' but A' does not). Notice that A and A' are stateless code entities. Units, mixin module systems, and various other calculi for the static linking of code fragments [Car97,DEW99,WV00,AZ02] all work in a related way, so this aspect of our theory is not particularly unique.

Dynamic Linking Dynamic linking is illustrated in Fig. 1 (b); expression $\mathbf{plugin}_{\nu \mapsto \nu'}\ A'$ in our calculus triggers a dynamic linking. Assemblage A' is loaded and linked into the current assemblage runtime Ar where the **plugin** expression is defined. The interesting thing here is not only that A' is an assemblage, but also Ar is an *assemblage* runtime. Thus, unlike some projects [Blo83,FF98,SPW03] which recognize dynamic plugins can be modelled as mod-

ules, we also recognize dynamic linking happens *between a module runtime and a module*. By equipping Ar with interfaces describing its dynamic linking behaviors (which we call *dynamic linkers*), a successful dynamic linking process can thus be conceived as a *bi-directional* interface matching between the plugin initiator's codebase module and the module representing the dynamic plugin. We believe this is more precise and explicit than the unidirectional notion of dynamic linking found in other module systems, where the initiating computation has no explicit interface to the module being linked.

A *dynamic linker* is the dynamic linking interface of an assemblage. It specifies the dynamic linking behaviors after the assemblage is loaded to become an assemblage runtime. Specifically, it defines what type of dynamic plugins the assemblage expects. In Fig. 1 (b), the **plugin** initiating assemblage runtime has a dynamic linker named ν, and when expression $\mathbf{plugin}_{\nu \mapsto \nu'}\ A'$ is evaluated, dynamic linker ν of the initiating assemblage runtime is matched with the *static linker* ν' of A'. This may seem like a mismatch, linking a static linker and a dynamic linker, but the linking is in fact occurring across sorts: a runtime is being linked with a piece of code. The result of a **plugin** expression is for the runtime to be the original runtime plus the runtime form of the newly added plugin. Dynamic linking here does not increase the number of runtimes, because dynamic linking is a tightly-coupled interaction. A non-example is Java applets; they are not tightly coupled with the loading JVM, they are best *not* viewed as dynamically linked code. Individual applets are better modeled as distinct runtimes communicating with its inhabiting VM, which can be better modelled by the cross-computation communication we introduce next.

Cross-Computation Communication Fig. 1 (c) demonstrates how expression $\mathbf{connect}_{\nu \mapsto \nu'}\ Ar'$ sets up a cross-computation connection between two assemblage runtimes, the runtime containing the **connect** expression (Ar) and the runtime Ar'. In a similar vein to dynamic linking, both the party *receiving* the cross-computation invocation and the party *initiating* the invocation must have explicit interfaces declaring the form of this interaction on their codebase assemblages, which we call *connectors*. A successful communication process is thus a *bi-directional* interface matching between the initiator module runtime and the receiver module runtime.

A *connector* specifies the cross-computation communication interface of an assemblage runtime. Specifically, it defines what type of assemblage runtimes the assemblage expects to communicate with. In Fig. 1 (c), the **connect**-initiating assemblage runtime has a connector named ν, and when its $\mathbf{connect}_{\nu \mapsto \nu'}\ Ar$ expression is evaluated, connector ν of the initiating assemblage runtime is matched with connector ν' of the assemblage runtime Ar. The **connect** expression will not lead to merging of runtimes, and since the runtimes must remain distinct, all parameters passed between the two must be passed by (deep) copy. These runtimes may also be on the same or different network nodes.

Before presenting further details of the system, we introduce a simple real-world example.

```
VolcanoMain = assemblage {
    static linker NetLib{
    ... statically linked some network library ...
    }
    dynamic linker DetectorPlugin{
        import detectMethod()
        export getEnv == ... get current environment snapshot ...
    }
    connector CodeUpdate{
        import getDetectCode();
        export check(condition) == ... check applicability of detect model ...
    }
    ⋮
    // local feature implementation
    updateDetector == λx.( let cb = connect_{CodeUpdate↦Code} x in
                           let code = cb ▷ getDetectCode ← () in
                           let comp = plugin_{DetectorPlugin↦Detect} code in
                           comp.detectMethod())
    ⋮
}
```

Fig. 2. A Sensor Network Example

2.2 A Real World Example

In this section, we introduce a simplified volcano sensor network example to demonstrate the basic ideas of assemblages. The example is in Fig. 2. For the purpose of improved readability, we here use sugared syntax slightly different from our calculus; type declarations are also omitted here for brevity. Initially we define an example with core features only, and will extend it below to illustrate more advanced features of our calculus.

In a typical sensor network [HSW+00] for a volcano sensing application, a number of smart tiny sensor nodes are scattered in the crater of a volcano, each of which functions as an independent computer. In addition to the parts for regular computation, each sensor node is also equipped with sensing device to collect environment information, such as temperature, electromagnetism, *etc.* Different sensor nodes can communicate with each other; at least some sensor node can communicate with base station situated out of the volcano to report data or receive control information.

One critical necessity in the design of such a system is that, once sensor nodes are physically deployed, they are not likely to be reclaimable for purposes such as software upgrade. In the example shown in Fig. 2, function updateDetector provides support to dynamically update the mathematical model of volcano detection used in the sensor: in reality, it is not uncommon for scientists to adjust the mathematical models after the sensors have been deployed. Without getting into too much detail, the function works as follows: it first sets up a connection cb with base station represented by function argument x, via $\mathbf{connect}_{\mathsf{CodeUpdate}\mapsto\mathsf{Code}}\ x$. A new version of the detection model code is thus

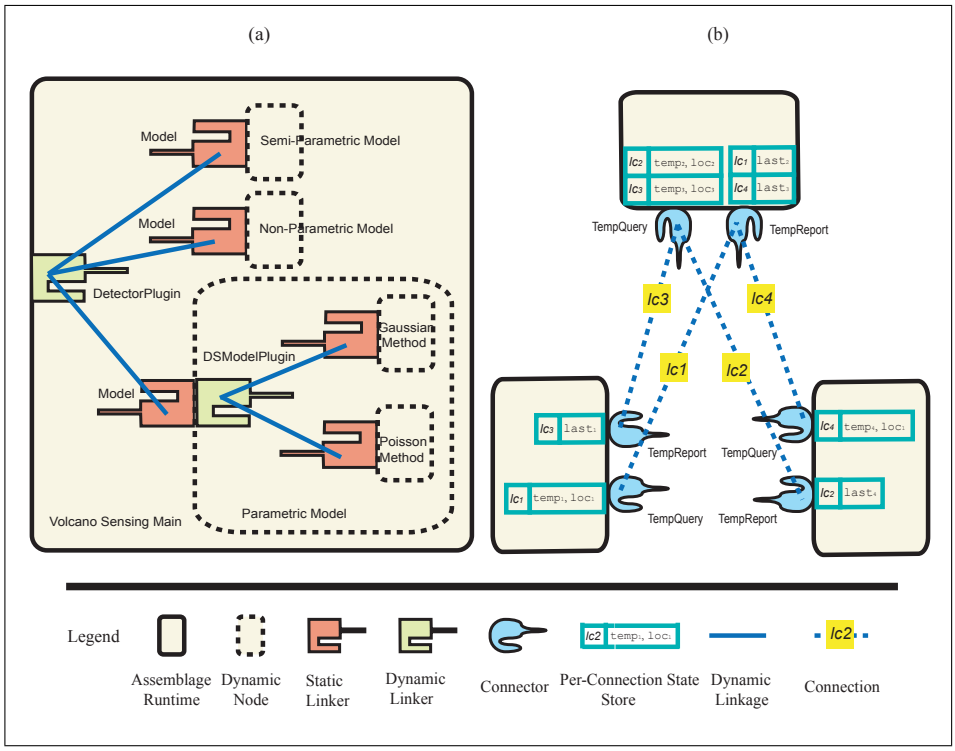

Fig. 3. An Example on (a) Rebindable Dynamic Linkers (b) Generative Stateful Connectors

acquired by invoking $cb \triangleright \mathsf{getDetectCode} \leftarrow ()$; subsequently, it is dynamically plugged into the current runtime via $\mathbf{plugin}_{\mathsf{DetectorPlugin} \mapsto \mathsf{Detect}}\ code$. The up-to-date detection method may then be invoked via $comp.\mathsf{detectMethod}()$.

This example demonstrates some basic uses of dynamic linkers and connectors. Observe how the dynamic linker and connector are both bi-directional interfaces: they import some features and export others. Interface matching is a component of both dynamic linking and connection. The $\mathbf{connect}_{\mathsf{CodeUpdate} \mapsto \mathsf{Code}}\ x$ indicates the CodeUpdate connector of the current assemblage runtime is connected to the Code connector of x. A connection can be successfully established only if each party exports what the other party needs to import (extra exports can be present and are ignored). We also show a standard static linker NetLib, expecting some network libraries; this importer will need to be satisfied before the sensor program is up and running.

2.3 More on Assemblages

With the basic model introduced and a simple example given, we now look at more advanced features of our calculus.

Rebindable Dynamic Linkers In the basic model presented in Fig. 1, dynamic linking was presented as a one-to-one relation between the initiating assemblage runtime and the dynamic plugin. This is however an oversimplified view of the calculus, and does not completely coincide with reality. Consider the sensor example again. Scientists might prefer to run multiple detection models at the same time, and compare the results of different models. The current VolcanoMain assemblage however only has one dynamic linker. In a one-to-one dynamic linking model, plugging in one detection assemblage would lead to the invalidation of previous dynamic plugins on the same dynamic linker.

For this reason, dynamic linkers in our calculus are rebindable. This implies different assemblages may be plugged in to the same dynamic linker at the same time, and not interfere with each other. Fig. 3 (a) shows this one-to-N relation. Here the Volcano Sensing Main assemblage runtime is the runtime form of assemblage VolcanoMain after its static linker is satisfied; it is here dynamically linked to three different dynamic plugins representing three different detection mathematical models; interestingly, since dynamic plugins themselves can have dynamic linkers, it might plug in other assemblages as *its* plugins. In Fig. 3 (a), a dynamic plugin of the Volcano Sensing Main assemblage runtime, one representing say a parametric probabilistic detection approach, further plugs in different distribution models to its DSModelPlugin dynamic linker, such as a Gaussian distribution or Poisson distribution method.

The dynamic linking established by a **plugin** expression is called a *dynamic linkage*, and the expression returns a value we call a *dyanmic linkage handle*; programmers can use it to refer to the particular assemblage just plugged in.

Generative Connectors Our connectors are also more nuanced than as presented in Fig. 1: connectors also need to be rebound, as we just saw for dynamic linkers. As shown in Fig. 2.3 (b), a task such as temperature measurement in a typical volcano sensor network is achieved by the collaboration of a number of sensors; each one of them usually communicates with its neighbors to exchange data such as temperature information. Since the configuration of the network is not fixed until sensors are scattered in the crater, the program developer can not define an *a priori* list of connectors, each of which is assigned to one neighbor. Instead, each sensor must only be equipped with rebindable connectors like TempQuery and TempReport given in Fig. 3 (b), where at any moment the TempQuery connector of a sensor may be connected with multiple TempReport connectors of its neighbors, and *vice versa*.

Another important issue is that each connection will need to keep its per-connection data: sensors need to record collected temperature information from its neighbors, together with the location information on where the temperature is sampled. This kind of information varies from connection to connection; a global state of assemblage for this purpose is not enough. In Fig. 3 (b), each connector is associated with a *per-connection state store*, which records the private generative states associated with each connections. The index of the store is the connection ID generated when connection is established via a **connect** expression.

Our calculus supports generative connectors where per-connection states are supported. It implicitly also supports rebindability. Each successful **connect** expression creates a *connection*, and the expression returns a value that is a *connection handle*; with the handle, programmers can refer to different connections (and the private per-connection state) on the same connector in the same program. One additional advantage of using handles is there is no problem with programs trying to access features on a non-connected connector—there is no name by which to refer to the features on the connector until there is a handle. Since multiple handles can be active and each connection has a unique state, there is an analogy of a connection definition with a class, and each active connection with an object, with the connection handle being the reference to the object. These "objects" are something like facades in the facade design pattern—they are the external interface to the component.

Typed Calculus and Types as Features Our calculus is typed, and the type system has several pleasant properties such as soundness and decidability of type checking. There is no runtime error associated with attempting to use a connector that is not connected to anything, because connectors are only accessed via handles which only exist because a connection was created. The only error possible is the handle could be stale because the connection has terminated. We have also explored the possibilities that types are themselves treated as features that are imported and exported across static linkers, dynamic linkers and connectors. In this presentation however, we do not focus on these aspects due to limited space. Interested readers can refer to [LS04] for details.

2.4 Why Dynamic Linkers and Connectors?

Static linkers, dynamic linkers and connectors together form the interfaces of assemblages. Before proceeding to the formalization, we address an important question concerning the purpose of the paper: Why dynamic linkers and connectors?

First, *modules with fully declarative interfaces lead to a more complete program specification.* Assemblages are highly declarative; in fact, all of an assemblage's potential for interaction with outside the codebase can be read off of the interfaces. This is obviously a good thing for language design, leading to provable type safety without obscurity. The idea also has impact on paradigms of software development. Indeed, interfaces (static linkers, dynamic linkers and connectors) can be defined at the design phase, reflecting designer's intention of the assemblage to interact with other parties. A declaration of DetectorPlugin dynamic linker coincides with the designer's intention that VolcanoMain module will dynamically link to some detection model plugins; the CodeUpdate connector coincides with the designer's expectation that the module will communicate with some codebase. In a large-scale software development process, software design and software implementation are typically accomplished by different people. The module calculus' type system ensures the implementation will faithfully follow

the intention of the designer. For instance, a compile-time type mismatch would occur if implementor of VolcanoMain desires to communicate with a codebase which does not provide a getDetectCode function.

Second, *these interfaces provide crucial support for extending the calculus to other important language features*. Since all of the external interactions are declared on the static linkers, dynamic linkers, and connectors, new modes of external interaction can easily be layered on top of these existing notions. Examples include security (on connectors), transaction management (on connectors), and version control (on dynamic linkers). For example, for security, since every cross-computation invocation will need to be directed through connectors, access control on connectors is enough to secure assemblage runtimes from unauthorized nonlocal access. We do not directly address these topics here, but the module theory is designed with them in mind.

Third, *dynamic linkers and connectors better model the complex interactions between different parties* than is possible in systems without them. A naive implementation for dynamic plugins can be achieved by direct dynamic loading, and invocations can thus be made on exported functions of the plugins. An obvious problem of this approach however is when there is a need for callback functions. For reactive computations, distributed protocols often involve message exchanges back and forth. If RMI or RPC is used, this interaction protocol will be completely submerged in the code in the various methods, and no single point in the program will indicate the protocol as a whole. In our example, the CodeUpdate connector specifies all the possible interactions between a code provider and a code client. The whole code updating protocol, although simple in this example, is captured by CodeUpdate connector, giving a clear protocol specification.

3 Syntax

The syntax of our calculus is shown in Fig. 4. It differs from the syntax used in Sec. 2, but in a trivial manner only: in the calculus syntax we remove the keywords (such as **assemblage**, **import** and **export**) used in the sugared syntax; otherwise the two forms of syntax are identical. Notation $\overline{m}$ is used to represent a sequence of entities $m_1, \ldots, m_n$, and we take the empty sequence as a special value ϕ. The $\uplus$ operator denotes the concatenation of two sequences; for the empty sequence, $\overline{m} \uplus \phi = \phi \uplus \overline{m} = \overline{m}$ for any $\overline{m}$. Since in this presentation, each m_i can only take one of the three syntactical forms $a \mapsto b$, $a == b$ and $a : b$, we also view $\overline{m}$ as a mapping and call it well-formed if it is a function, *i.e.* there does not exist $a_1 = a_2$ but $b_1 \neq b_2$; in this case when no confusion arises, we also define $\overline{m}(a) = b$. $\overline{m}(a) = \bot$ iff $a \mapsto b$ (or $a == b$, or $a : b$) is not present for any b, or $\overline{m}$ is not well-formed. $\overline{m}\{a \mapsto b\}$ denotes a mapping update; it is a mapping the same as $\overline{m}$, except $\overline{m}\{a \mapsto b\}(a) = b$ while $\overline{m}(a)$ could be other values.

An assemblage (A) is composed of a well-formed sequence of static linkers ($\overline{S}$), dynamic linkers ($\overline{D}$), connectors ($\overline{C}$) and its local private code ($\overline{L}$). Each static

$$
\begin{array}{lcll}
A & ::= & \langle \overline{S}; \overline{D}; \overline{C}; \overline{L} \rangle & \textit{assemblage} \\
S, D & ::= & \nu \mapsto \langle \overline{I}; \overline{E} \rangle & \textit{static linker}, \textit{dynamic linker} \\
C & ::= & \nu \mapsto \langle \overline{I}; \overline{E}; \overline{J} \rangle & \textit{connector} \\
I & ::= & \alpha : \tau & \textit{imported feature} \\
E & ::= & \alpha == \lambda x.e : \tau & \textit{exported feature} \\
L & ::= & \alpha == F : \tau & \textit{local feature} \\
J & ::= & \alpha == \textbf{ref}\ F : \tau & \textit{per-connection state} \\
F & ::= & \textit{cst} \mid A \mid \lambda x.e \mid \textbf{ref}\ F & \textit{feature} \\
\\
e & ::= & () \mid x \mid \textit{cst} \mid \textbf{thisc} \mid \textbf{thisd} & \textit{null value}, \textit{variable}, \textit{const} \\
 & & \mid A : \tau \mid e + e & \textit{first class assemblage}, \textit{sum} \\
 & & \mid \textbf{plugin}_{\nu \mapsto \nu'}\ e \mid \textbf{connect}_{\nu \mapsto \nu'}\ e & \textit{dynamic plugin}, \textit{connect} \\
 & & \mid \alpha@\texttt{local} \mid \alpha@\nu \mid e.\alpha \mid e \triangleright \alpha \mid e \triangleright \alpha \leftarrow e & \textit{feature access} \\
 & & \mid \lambda x.e : \tau \mid e\ e & \textit{first class function}, \textit{app} \\
 & & \mid \textbf{ref}\ e \mid !\, e \mid e := e & \textit{state} \\
\\
\nu & & & \textit{interface name} \\
\alpha & & & \textit{feature name} \\
\tau & & & \textit{type}, \textit{defined in Fig.8} \\
\textit{cst} & & & \textit{integer constant} \\
x & & & \textit{variable}
\end{array}
$$

Fig. 4. Assemblage Language Syntax

linker or dynamic linker has a name (ν), a well-formed sequence of imported features ($\overline{I}$) and a well-formed sequence of exported features ($\overline{E}$); each connector, in addition, is associated with a well-formed sequence of per-connection states ($\overline{J}$). $\overline{I} \uplus \overline{E}$ (and in the connector case $\overline{I} \uplus \overline{E} \uplus \overline{J}$) also must be well-formed: we disallow the case where the same feature is imported and exported on the assemblage. Features are chosen from constants (*cst*), functions ($\lambda x.e$), references (**ref** F) and nested assemblages (A). This particular choice is made to preserve a balance between functional features and imperative features. With references and functions around, primitive classes and objects can also be modelled with widely known encoding techniques, and are left out of the calculus for simplicity. $\alpha : \tau$ denotes importing a feature named α with type τ and $\alpha == F : \tau$ denotes exporting a feature named α, defined to be F with type τ. Features to be imported/exported on interfaces must be functions. This function-only restriction however does not restrict the expressiveness of the calculus: importing/exporting references can be encoded as importing/exporting a pair of getter function and a setter function; importing/exporting assemblages can be encoded as importing/exporting a function taking a null value and returning the assemblage. Per-connection state (J) is defined via feature $\alpha == \textbf{ref}\ F : \tau$, which means the state is named α, **ref** F will be its initial value, and τ is its type.

Most of the expressions e have been explained in Sec. 2. Additionally, we use α@`local` to refer to a feature α defined in locally (in $\overline{L}$). $\alpha@\nu$ refers to a feature α defined in static linker ν. **thisd** refers to the *current* dynamic linkage, and **thisc** refers to the *current* connection. The meaning of "current" depends on the dynamic linkage handle or connection handle which invokes the function **thisd**.α expression is situated in. Because of the rebindable nature of dynamic

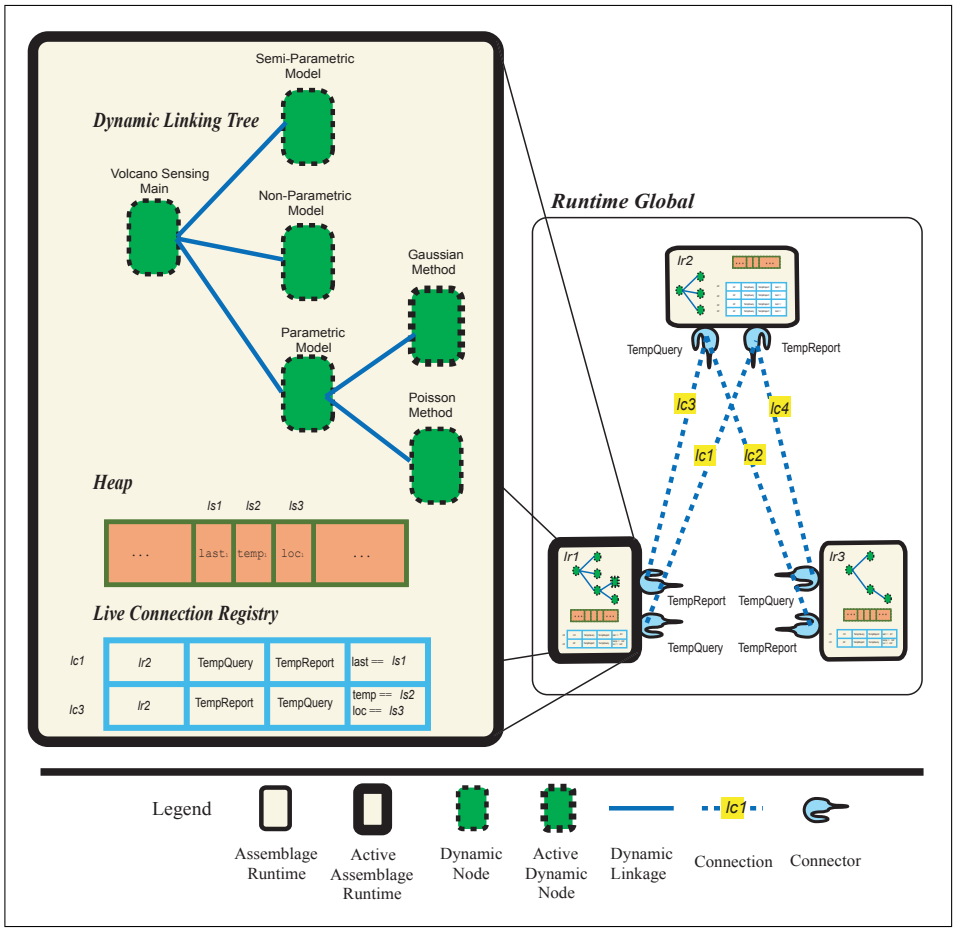

Fig. 5. The Big Picture

linking, we disallow a syntax like $\alpha@\nu$ where ν is a dynamic linker name: it would be ambiguous which version of the dynamic linkages is referred to if there were more than one dynamic linkage created from the same dynamic linker, and would be undefined if none were present. This restriction on syntax also holds for connectors for similar reasons. $e.\alpha$ is used to refer to a feature α in dynamic linkage handle e. Syntax $e \triangleright \alpha$ refers to a per-connection state α in connection handle e. $e \triangleright \alpha \leftarrow e$ denotes an invocation of a function defined in a connection handle. This expression can potentially denote a cross-computation invocation, and has a different semantics from regular function application; we therefore use different syntax for the two cases. () is used to denote a null value of **unit** type, as in ML. **let** is encoded by function application.

$$
\begin{array}{rcll}
\iota r & \in & \mathbb{R} & \textit{assemblage runtime ID} \\
\iota s & \in & \mathbb{S} & \textit{store ID} \\
\iota n & \in & \mathbb{N} & \textit{dynamic node ID} \\
\iota d & \in & \mathbb{D} & \textit{dynamic linkage ID} \\
\iota c & \in & \mathbb{C} & \textit{connection ID} \\
\\
G & ::= & \overline{R} & \textit{runtime global} \\
R & ::= & \iota r \mapsto \langle T; H; Y \rangle & \textit{assemblage runtime} \\
T & ::= & \langle \overline{N}; \overline{K} \rangle & \textit{dynamic linking tree} \\
H & ::= & \overline{\iota s \mapsto v} & \textit{heap} \\
Y & ::= & \overline{\iota c \mapsto \langle \iota r; \nu_1; \nu_2; \overline{Jr} \rangle} & \textit{live connection registry} \\
N & ::= & \iota n \mapsto \langle \overline{Sr}; \overline{Dr}; \overline{Cr}; \overline{Lr} \rangle & \textit{dynamic node} \\
K & ::= & \iota d \mapsto \langle \iota n_1; \nu_1; \iota n_2; \nu_2 \rangle & \textit{dynamic linkage} \\
Jr & ::= & \alpha \mapsto \iota s & \textit{per} - \textit{connection state} \\
fv & ::= & cst \mid A \mid \mathbf{fun}(\iota r, \iota n, \lambda x.e) \mid \iota s & \textit{feature value} \\
v & ::= & () \mid \iota r \mid \iota d \mid \iota c \mid fv & \textit{value} \\
e & ::= & \cdots \mid v \mid \mathbf{inR}(\iota r, \iota n, e) & \textit{expression at runtime} \\
\\
\mathbf{E} & ::= & [\,] \mid \mathbf{plugin}_{\nu \mapsto \nu'}\ \mathbf{E} \mid \mathbf{connect}_{\nu \mapsto \nu'}\ \mathbf{E} & \textit{evaluation context} \\
 & & \mid \mathbf{E} + e \mid v + \mathbf{E} & \\
 & & \mid \mathbf{E}\ e \mid v\ \mathbf{E} \mid \mathbf{ref}\ \mathbf{E} \mid !\mathbf{E} \mid \mathbf{E} := e \mid v := \mathbf{E} & \\
 & & \mid \mathbf{E}.\alpha \mid \mathbf{E} \triangleright \alpha \mid \mathbf{E} \triangleright \alpha \leftarrow e \mid v \triangleright \alpha \leftarrow \mathbf{E} \mid \mathbf{load}\ \mathbf{E} & \\
\\
cst,\ A,\ \alpha,\ J & & & \textit{defined in Fig. 4} \\
Sr,\ Dr,\ Cr\ , Lr & & & \textit{see Sec. 4.2}
\end{array}
$$

Fig. 6. Operational Semantics Auxiliary Definitions

4 Operational Semantics

In this section we present the dynamic semantics of our calculus. We first informally explain the big picture of how a typical application appears at run-time, and then we discuss formal details of the operational semantics.

4.1 The Big Picture

Sec. 2.3 gave an example of a temperature measurement application for sensor networks; the illustrations used in that section (Fig. 3), however, are only intended to target high-level concepts. In this section, the *precise* runtime snapshot of the same application is illustrated in Fig. 5. Here the whole network is composed of multiple sensor nodes in the form of Fig. 3 (b) to perform temperature measurement, while individual nodes are experimenting with different computational models in the form of Fig. 3 (a).

At runtime, the entire application space, with all reactive computations of concern, is called a *runtime global*; the whole temperature measurement network is a runtime global for instance. Inside it, independently deployed assemblage runtimes are running on potentially distributed locations. Each of them can be created by explicit **load** expressions, during which a static assemblage is loaded into memory. The first assemblage runtime in the runtime global is loaded via a *bootstrapping* process. At load time, each runtime is associated with an ID. In Fig. 5, we have three runtimes with IDs ιr_1, ιr_2 and ιr_3. Assemblage runtimes

communicate with each other via connections over paired connectors, which also have connection ID's, their *connection handle*. In the figure, runtimes with ID's ιr_1 and ιr_2 are communicating via two connections; the connection with ID ιc_1 is between the connector TempQuery of ιr_1 and connector TempReport of ιr_2.

Internally, each runtime contains a *dynamic linking tree*, a *heap* and a *live connection registry*. The heap is standard and holds the multable data; it is defined as a sequence of stores, each of which is associated with a store ID. In Fig. 5, the heap of runtime ιr_1 currently has a store ιs_2 which holds a constant value $\texttt{temp}_1$. A runtime's live connection registry holds the currently active connections. It is defined as a table indexed by connection IDs; each entry contains information such as what parties are involved in the connection (runtime IDs and connector names), and the per-connection state store. For instance, the first row of the live connection registry of runtime ιr_1 shown in Fig. 5 indicates it currently has a connection ιc_1 on its TempQuery connector, and that connection is to TempReport of runtime ιr_2. The last column indicates the per-connection field last has a reference value ιs_1: the value is a reference since per-connection states always contain mutable data, just as object fields in object-oriented languages are mutable. The meanings of connector-related expressions of our calculus, such as $\mathbf{connect}_{\nu \mapsto \nu'}\ e$, are related to operations on live connection registries. For instance, when a connection is established via a **connect** expression, both involved parties of the connection have one entry added to their live connection registry. Per-connection states are allocated and initialized at connection establishment time.

A dynamic linking tree is used to reflect the rebindable nature of dynamic linkers, as described informally in Sec. 2.3. Indeed, if rebindability were not supported, a **plugin** expression could just merge the codebase of the current assemblage runtime with the code of dynamic plugin, in the same manner as static linking $A_1 + A_2$. However due to rebindability, each dynamic linker of the current runtime can be associated with multiple independent dynamic plugins at the same time. The data structure is in general a tree: when the main assemblage is first loaded to memory, it creates a root node, with all application logic of the main assemblage defined inside. Each **plugin** expression executed in the root will result in the creation of a tree node representing the dynamic plugin (with all application logic of the dynamic plugin defined inside), and the newly created node becomes a child of the root. Since child nodes can themselves plug in code (the plugin of a plugin), the tree can in general have depth greater than two. The fact that the data structure is a tree other than a DAG or some random graph can be easily proved by the way it is constructed. In Fig. 5, the dynamic linking tree is the internal representation of the application whose high-level requirement is shown in Fig. 3 (a). A dynamic linking tree is composed of a series of *dynamic nodes* (the tree nodes) and *dynamic linkages* (the tree edges). Each dynamic linkage is referenced by its dynamic linkage ID ιd, which is the realization of the *dynamic linkage handle* of the previous section. The behaviors of dynamic linker-related expressions, such as $\mathbf{plugin}_{\nu \mapsto \nu'}\ e$, are operations on dynamic linking trees. For instance, when a plugin is assembled via a **plugin**

expression, the dynamic plugin will becomes a dynamic node that is a leaf of the initiator runtime's dynamic linking tree.

Since concurrency is not the focus of this paper, our calculus assumes for simplicity that at any moment only one assemblage runtime is active, and only one dynamic node in this active runtime is performing the reduction. This fact is shown in Fig. 5, where distinct notation is used for active runtimes and active dynamic nodes.

4.2 A Formal Overview of Dynamic Semantics

Fig. 6 defines the relevant data structures that play a part in defining the dynamic semantics. Most of them have been explained with the example in Fig. 5. The rest is explained below.

Formal Details of the Dynamic Linking Tree Each dynamic node is associated with an ID ιn. Given a static assemblage $A = \langle \overline{S}; \overline{D}; \overline{C}; \overline{L} \rangle$, its corresponding dynamic node form $N = \iota n \mapsto \langle \overline{Sr}; \overline{Dr}; \overline{Cr}; \overline{Lr} \rangle$ is almost identical to A, except that it has an ID ιn, and the features defined in $\overline{Sr}$, $\overline{Dr}$, $\overline{Cr}$, $\overline{Lr}$ are slightly different in form from its static counterparts due to function closure and mutable states, which will be made clear when we explain feature values shortly. A dynamic linkage is of the form $\iota d \mapsto \langle \iota n_1; \nu_1; \iota n_2; \nu_2 \rangle$, denoting a tree edge with an ID ιd linking dynamic linker ν_1 of dynamic node ιn_1 with static linker ν_2 of dynamic node ιn_2. We use $root(T)$ to denote the root node of the dynamic linking tree T.

Feature Values At the source code level, our language supports four kinds of features; see Fig. 4. At runtime, not all of them are values; the possible feature values fv are defined in Fig. 6. **ref** F features are not values, since this indicates a heap allocation; the corresponding value is the store ID where the value is allocated on the heap. Function values are closures $\mathbf{fun}(\iota r, \iota n, \lambda x.e)$, where ιr and ιn are the IDs of the runtime and the dynamic node where the function is defined. The reason why $\lambda x.e$ is not a value is that the body e might refer to other features such as α@`local`. At runtime, if functions as first-class values are passed from one dynamic node to another, the meaning of α@`local` would not be preserved if the defining dynamic node were not recorded; passing around closures would make parameter passing of first-class functions have a consistent meaning universally. Our language does not allow functions to be passed from one runtime to another, so theoretically, the ιr information in function closures could be removed. We keep it here to show our language could easily support function passing across runtimes without technical difficulty, which also implies mechanisms like RMI could also be easily supported. The reason we do not support function passing across runtimes is that it gives an indirect access to a runtime that is not explicit in an interface; this topic is elaborated in Sec. 5.1.

Source-code level features are converted to feature values when assemblages are loaded either through bootstrapping process, or loaded via an explicit **load** expression, or added to the current runtime via a **plugin** expression.

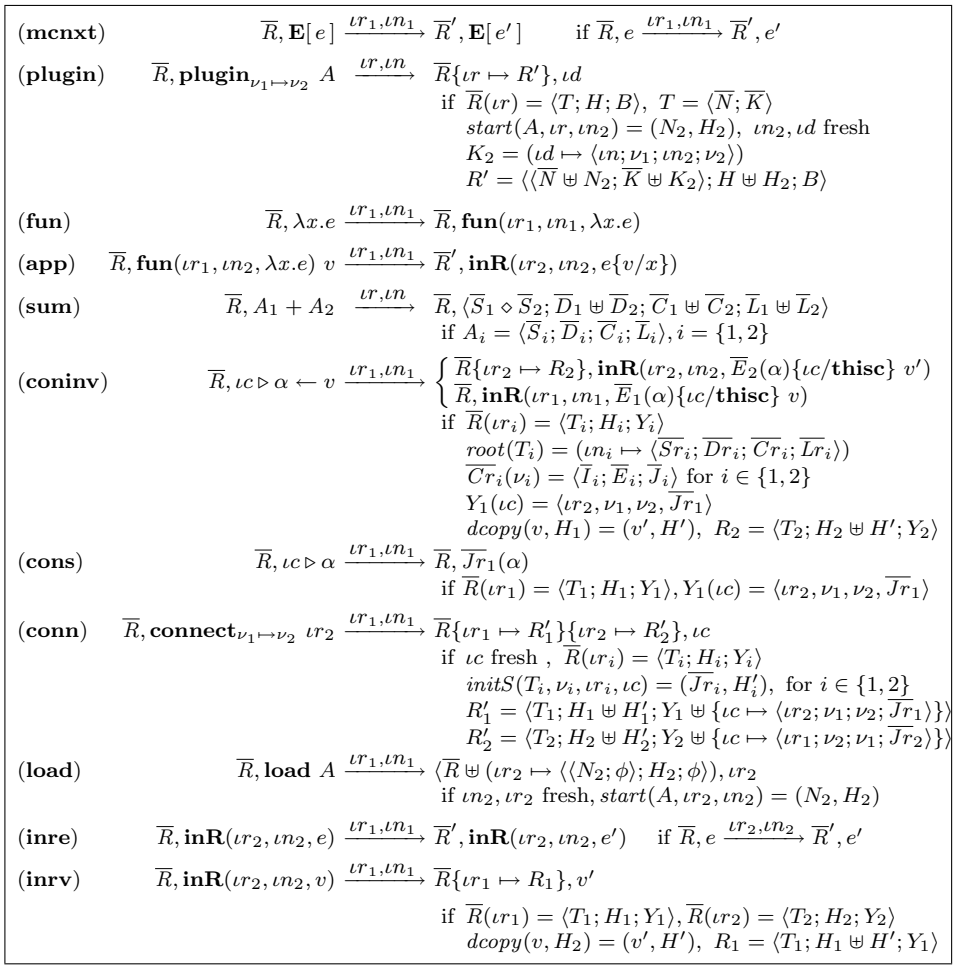

Fig. 7. Selected Reduction Rules

Values and Expressions at Runtime Values in our calculus can be feature values; assemblage runtime IDs ιr (which serve as handles to assemblage runtimes, returned from **load** expressions); dynamic linkage IDs ιd (which serve as handles to dynamic linkages, returned from **plugin** expressions); or, connection IDs ιc (which serve as handles to connections, returned from **connect** expressions).

We extend the expressions given in Fig. 4 with new syntax in Fig. 6 (see e definition) to aid in implementing the operational semantics. $\mathbf{inR}(\iota r, \iota n, e)$ is an auxillary expression defining a code context switch, meaning e is evaluated in runtime with ID ιr and dynamic node with ID ιn; the expression is particularly useful to model function invocations, during which the current execution point is switched.

4.3 A Guided Tour to Reduction Rules

Operational semantics of our language is given in Fig. 7. $G,\ e \xrightarrow{\iota r, \iota n} G', e'$ indicates a reduction of expression e in the presence of global runtime G, where the current active runtime has ID ιr, and the current active dynamic node in ιr has ID ιn. Evaluation contexts are defined in Fig. 6. Here we omit the rules for expressions **ref** e, $e := e'$, $!e$, $\alpha@\nu$, $\alpha@\texttt{local}$ and $e.\alpha$; these rules are relatively straightforward. Also note that some of the rules might get stuck on certain combinations of G and e. We largely omit the specifications of the faulty cases here, but claim that the static type system introduced in Sec. 5 will ensure these faulty cases never appear when real reductions happen at dynamic time. Details on omitted reduction rules, specifications of these faulty expressions, and proof to back up the forementioned claim can be found in [LS04].

We use $e\{e'/x\}$ to denote capture-free substitution. If e is an assemblage, the substitution does nothing: assemblages do not contain free variables, and even all import feature names need to be explicitly declared on their static linkers, dynamic linkers or connectors.

Assemblage Loading and Bootstrapping We now first explain how an assemblage runtime loads in another assemblage, and proceed to discuss the process of bootstrapping, where the first assemblage in runtime global is loaded.

The (**load**) rule in Fig. 7 shows how loading is simply the creation of a new assemblage runtime, and the result returned is a runtime handle, the ID of the new runtime. Function $\mathit{start}(A, \iota r, \iota n) = (N, H)$ prepares the initial dynamic node (N) out of a static assemblage (A), together with the initial heap (H). This function, whose formal definition is skipped here, is fairly straightforward according to the following rules:

- For every function feature in $\alpha == \lambda x.e$ form defined in A, its corresponding place in N is substituted with $\alpha == \mathbf{fun}(\iota r, \iota n, \lambda x.e)$.
- For every reference feature in $\alpha == \mathbf{ref}\ F$ form defined in A, its corresponding place in N is substituted with $\alpha == \iota s$, where ιs is a fresh store ID, and at the same time $\iota s \mapsto \mathit{fv}$ is in H. Here fv is the feature value form of F. Since reference feature could be in the form like **ref ref** 0, this process could lead to multiple stores defined in H.
- A and N are otherwise identical.

The (**load**) rule doesn't perform any initialization, because an initializing load can easy be defined using this primitive one; for example, one method could be

$$\begin{aligned}\mathbf{loadinit}\ A \ \stackrel{\text{def}}{=}\ & \mathbf{let}\ x_1 = \mathbf{load}\ A\ \mathbf{in} \\ & \mathbf{let}\ x_2 = \mathbf{connect}_{\texttt{InitIn}\mapsto\texttt{InitOut}}\ x_1\ \mathbf{in} \\ & x_2 \triangleright \texttt{Main} \leftarrow ()\end{aligned}$$

which assumes connectors `InitIn`/`InitOut` are present on loaders/loadees respectively, importing/exporting function `Main()`. Bootstrapping the first assemblage $A_{\text{boot}} = \langle \overline{S}; \overline{D}; \overline{C}; \overline{L} \rangle$ is accomplished by initiating execution in the state

$$\langle \iota r \mapsto \langle \langle N; \phi \rangle; H; \phi \rangle, \iota r \rangle, \quad \overline{C}(\texttt{InitOut})(\texttt{Main})()$$

where ιr, ιn are fresh, $start(A_{\text{boot}}, \iota r, \iota n) = (N, H)$.

Static Linking The (**sum**) rule shows how static linking of two first-class assemblages happens. It merges their dynamic linkers, connectors and local definitions, with preconditions that these parts do not clash by name. The fact that clash of local feature names would lead to stuck computations might be counter-intuitive: in reality, local features are supposed to be invisible from the outside, and therefore static linking of two assemblages with some shared local feature names *should* be a valid operation. To avoid this dilemma, we stipulate assemblages are freely α-convertible with regard to local feature names.

Static linkers are matched by name, according to the $\diamond$ operator. Given two sequences of static linkers $\overline{S}_1$ and $\overline{S}_2$, $\overline{S}_1 \diamond \overline{S}_2$ is the shortest sequence satisfying all of the following conditions:

- If $\overline{S}_1$ has a static linker S by name ν but $\overline{S}_2$ does not, or *vice versa.*, S is a static linker in $\overline{S}_1 \diamond \overline{S}_2$.
- If $\overline{S}_1$ and $\overline{S}_2$ both have a static linker by name ν whose bodies are $\langle \overline{I}_1; \overline{E}_1 \rangle$ and $\langle \overline{I}_2; \overline{E}_2 \rangle$ respectively, then $\overline{S}_1 \diamond \overline{S}_2$ also has a static linker named ν and a body $\langle \overline{I}; \overline{E}_1 \uplus \overline{E}_2 \rangle$, where $\overline{I}$ exactly include imported features whose names are listed in $\overline{I}_1$ but not $\overline{E}_2$, or listed in $\overline{I}_2$ but not $\overline{E}_1$.

Dynamic Linking The (**plugin**) rule dynamically links a new assemblage to the initiating runtime. A new dynamic node (N_2) is created out of the assemblage to be plugged in, via the *start*() function, and it is then added to the initiating assemblage runtime by adding the node and an edge with ID ιd to the runtime's dynamic linking tree. Note that in dynamic linking, no new runtime is created; the plugin will eventually become *part of* the initiating assemblage runtime. This can be illustrated by the way the *start*() function is used: the initiating runtime's ID is passed to create the new dynamic node, not a fresh runtime ID. The return value of the **plugin** expression is the ID of the newly created edge; this is the dynamic linkage handle to the plugin, and conceptually represents the link created out of the dynamic linking process. With this, features exported from the plugin or from the initiating party can thus be accessed via an $e.\alpha$ expression.

Cross-Computation Communication Connections are established by the (**conn**) rule, which adds an entry to the live connection registry (Y) of both connected parties. Per-connection states are also initialized at this point through a simple function *initS*(); since all per-connection states are mutable, this function predictably deals with initialization of reference features, which is detailed when we explained the *start*() function. Function features defined in connectors are invoked by expression $e \triangleright \alpha \leftarrow v$, and its semantics is defined by (**coninv**). Per-connection state can be referred to via expression $e \triangleright \alpha$; the related reduction rule is (**cons**).

τ	::=	$\mathbf{unit} \mid \mathbf{int} \mid \tau \to \tau \mid \tau\ \mathbf{ref}$	*primitive types*
	$\mid$	$\mathbf{Asm}(\overline{\mathcal{S}}, \overline{\mathcal{D}}, \overline{\mathcal{C}}, \overline{\mathcal{L}})$	*assemblage type*
	$\mid$	$\mathbf{Rtm}(\overline{\mathcal{C}})$	*runtime type*
	$\mid$	$\mathbf{Dlnk}(\overline{\mathcal{E}})$	*dynamic linkage type*
	$\mid$	$\mathbf{Cnt}(\overline{\mathcal{E}}, \overline{\mathcal{J}})$	*connection type*
$\mathcal{S}, \mathcal{D}$	::=	$\nu \mapsto \langle \overline{\mathcal{I}}; \overline{\mathcal{E}} \rangle$	*static linker/dynamic linker type*
$\mathcal{C}$	::=	$\nu \mapsto \langle \overline{\mathcal{I}}; \overline{\mathcal{E}}; \overline{\mathcal{J}} \rangle$	*connector type*
$\mathcal{I}, \mathcal{E}, \mathcal{J}, \mathcal{L}$	::=	$\alpha : \tau$	*feature type declaration*

Fig. 8. Type Syntax

In the (**coninv**) rule, $\iota c \triangleright \alpha \leftarrow v$ invokes a function named α on a previously established connection ιc, with v as the parameter. Since α could be defined by either of the two parties connection ιc connects, there are two possibilities: 1) α is exported in the assemblage runtime containing the expression; in this case, the invocation is an intra-runtime one. 2) α is imported in the assemblage runtime containing the expression; the invocation is thus an inter-runtime one. A deep copy of parameter v should be passed to the target runtime; specifically, when v is a store ID, the heap cells associated with v will be passed around, with store IDs refreshed. The underlying design principle for the copy semantics is object confinement: each assemblage runtime should have its own political boundaries and direct references across boundaries would cause many problems such as security. Function $dcopy(v, H) = (v', H')$ defines the value (v') and the heap cells (H') that need to be transferred if v under heap H needs to be passed across computations. v' is not always the same as v because stores are refreshed if v is a store ID. In both case 1) and case 2), substitution of ιc for **thisc** is needed to determine what the "current connection" means.

5 The Type System

In this section, we informally explain the type system of our calculus. We start with an overview which covers the major ideas behind the type system, then we explain the type-checking process in detail. Some properties of the type system are stated at the end. The complete formal type system with proofs of its properties can be found in a technical report [LS04].

5.1 Overview

The Types The types are defined in Fig. 8. The assemblage type $\mathbf{Asm}(\overline{\mathcal{S}}, \overline{\mathcal{D}}, \overline{\mathcal{C}}, \overline{\mathcal{L}})$ contains type declarations of static linkers, dynamic linkers, connectors, and local features. It is used in two situations: top-level typechecking and typechecking of first-class assemblages. At the top level, each assemblage is a separate compilation unit in our type system and is given an assemblage type. In the second situation, first-class assemblages can appear anywhere as expressions, be passed as arguments, *etc.*

The runtime type $\mathbf{Rtm}(\overline{\mathcal{C}})$ is the type of assemblage runtimes. When an assemblage runtime is viewed by other assemblage runtimes, the only thing other runtimes care about is how to communicate with the runtime. Thus, a runtime type only contains the list of connector types. The dynamic linkage type $\mathbf{Dlnk}(\overline{\mathcal{E}})$ structurally is a sequence of type declarations for functions either exported from dynamic linking initiator's dynamic linker or the dynamic plugin's corresponding static linker. The connection type $\mathbf{Cnt}(\overline{\mathcal{E}}, \overline{\mathcal{J}})$ is structurally a sequence of type declarations for functions either exported from connection initiator's connector or connection receiver's connector, and $\overline{\mathcal{J}}$ is type declaration of per-connection states.

Interface Matching As introduced in Sec. 2, static linking, dynamic linking and connection establishment share one common trait: all three fundamental processes involve bi-directional interface matchings. This commonality is reflected in the typecheckings of three related expressions: $A_1 + A_2$, $\mathbf{plugin}_{\nu_1 \mapsto \nu_2}\ e$ and $\mathbf{connect}_{\nu_1 \mapsto \nu_2}\ e$; an *interface match* check is performed for all three typecheckings, between two static linkers in the first case, one dynamic linker and one static linker in the second case, and two connectors in the third case. By definition, each interface type, be it static linker type, dynamic linker type or connector type, is composed of a list of type declarations for imported features and a list for exported features. Two interface types, say i_1 and i_2, are considered a match iff

1. If i_1 exports a feature α of type τ, and if i_2 imports a feature α of type τ', then τ must be a subtype of τ'. The same should also hold if i_2 exports and i_1 imports.
2. i_1 and i_2 do not export features by the same name.
3. If i_1 and i_2 are both static linker types, they do not import features of the same name. If one of them is not a static linker type, then every imported feature in i_1 (or i_2) must match an exported feature of the same name in i_2 (or i_1).

Condition 1 is the most important one: features matched by name also match by type. The flexible part is that our type system does not demand exact matching of types; instead, it is acceptable if the export feature has a more precise type than what is expected from the import counterpart. Our subtyping relation is standard for primitive types; for types $\mathbf{Asm}(\overline{\mathcal{S}}, \overline{\mathcal{D}}, \overline{\mathcal{C}}, \overline{\mathcal{L}})$, $\mathbf{Rtm}(\overline{\mathcal{C}})$, $\mathbf{Dlnk}(\overline{\mathcal{E}})$, and $\mathbf{Cnt}(\overline{\mathcal{E}}, \overline{\mathcal{J}})$, subtyping is given the natural structural definition.

Condition 2 is used to avoid a feature name clash. For instance, if ιd is the result of $\mathbf{plugin}_{\nu \mapsto \nu'}\ e$, the meaning of expression $\iota d.\alpha$ would be ambiguous if both dynamic linker ν of the initiator and static linker ν' of the dynamic plugin exported the feature α. The restriction here might not correspond to reality: in real life, dynamic plugins might be developed independently, and such a name clash does have a chance to happen. However, such a clash can easily be avoided if the language supports either feature name renaming, or casting to remove some exported features. Our calculus currently does not include these operators, but they can easily be added without affecting the calculus core.

Condition 3 states that for dynamic linking and connection case, no dangling imports are allowed if interface match succeeds; for static linking, our calculus does allow some imports to not be satisfied, since the result (say A) of $A_1 + A_2$ can still be statically linked in the future, *e.g.*, by $A + A_3$.

Principle of Computation Encapsulation and Parameter Passing Across Computations One of the design principles of our calculus is computation encapsulation: reactive computations should only communicate with each other via *explicit* interfaces, in our context, connectors. Parameter passing across reactive computations, if not handled properly, could however violate this principle. The three types of parameters that cause troubles are function closures, dynamic linkage handles, and connection handles; our type system disallows the passing of these three types of values.

We first consider the problematic case of passing connection handles. Suppose assemblage runtime with ID ιr_1 contains a $\mathbf{connect}_{\nu \mapsto \nu'}\ \iota r_2$ expression which returns a connection handle ιc. Had we allowed ιc to be passed as a parameter, runtime ιr_1 could pass it to some runtime ιr_3 via some previously existing connection. Now although runtime ιr_3 does not have a connector ν (or ν'), it would still be able to use features associated with connection ιc via $\iota c \triangleright \alpha \leftarrow e$ expressions, meaning it is accessing a feature not through an explicit interface on its runtime, ιr_3. Similarly, passing dynamic linkage handles could allow assemblage runtimes to gain direct access to dynamic plugins they do not have interfaces to plug in to.

The case for passing function closures across assemblage runtimes suffers from a similar problem, but it is less obvious. In Sec. 4, we have already mentioned how there is no technical difficulty in passing function closures across assemblage runtimes; indeed we could just pass function closures in a manner similar to how Java RMI passes object references. Now let us consider why a mechanism like this would violate our encapsulation principle. Suppose assemblage runtime with ID ιr_1 has a function closure $\mathbf{fun}(\iota r_1, \iota n_1, \lambda x.e)$ and e contains an expression $\alpha@\nu$ to use a feature α exported from static linker ν of ιr_1. Had we allow this closure to be passed to another runtime, it could, by several indirections (ιr_1 to ιr_2, and ιr_2 to ιr_3 for instance), eventually be received by some runtime ιr_3 that has no direct communication with ιr_1. But, by applying the function, this assemblage runtime ιr_3 would be able to access to feature α of static linker ν of ιr_1, through a channel not explicit in ιr_3's interface.

The legal parameters that can be passed across computations are primitive values such as integers, runtime handles, references and first-class assemblages. Passing runtime handles is an important means for a runtime to "advertise" itself to other runtimes. References are passed by deep copy (recall the reduction rule (**coninv**) in Sec. 4). The exclusion of function closures from passable parameters across computations might appear to disallow the possibility of any code passing in our calculus, but in fact not. If passing code is needed, users can encapsulate the code as an assemblage and pass the assemblage; assemblages are completely

self-contained, without need for a closure, and so are nothing more than a kind of data.

To enforce the principle of computation encapsulation, our type system checks that for every imported function feature given in a connector type, its parameter and return value can not have the aforementioned types. This well-formedness property must hold for all connector types.

5.2 Details of Typechecking

We now explain how top-level typechecking is achieved, and how some important expressions are typechecked, in our type system.

Assemblage Typechecking At top level, assemblage $\langle \overline{S}; \overline{D}; \overline{C}; \overline{L} \rangle$ as a separate compilation unit is well typed if all exported features defined in its static linkers $\overline{S}$, dynamic linkers $\overline{D}$ and connectors $\overline{C}$, and all locally defined features $\overline{L}$, are well-typed. Well-typed assemblages have an assemblage type $\mathbf{Asm}(\overline{\mathcal{S}}, \overline{\mathcal{D}}, \overline{\mathcal{C}}, \overline{\mathcal{L}})$, which structurally corresponds to $\langle \overline{S}; \overline{D}; \overline{C}; \overline{L} \rangle$ in an intuitive way.

To typecheck an assemblage appearing inside a program as a first-class value is the same as top-level typechecking, with the only exception that if the assemblage is annotated with assemblage type $\mathbf{Asm}(\overline{\mathcal{S}}, \overline{\mathcal{D}}, \overline{\mathcal{C}}, \overline{\mathcal{L}})$ and typechecks, the type we give to this assemblage expression is $\mathbf{Asm}(\overline{\mathcal{S}}, \overline{\mathcal{D}}, \overline{\mathcal{C}}, \phi)$. Assemblages are encapsulated entities and on the outside local features should be invisible, just as with private fields of objects.

Static Linking To typecheck expression $A_1 + A_2$, the following conditions have to be satisfied:

- A_1 and A_2 both need to be well-typed, with type $\mathbf{Asm}(\overline{\mathcal{S}}_1, \overline{\mathcal{D}}_1, \overline{\mathcal{C}}_1, \phi)$, and $\mathbf{Asm}(\overline{\mathcal{S}}_2, \overline{\mathcal{D}}_2, \overline{\mathcal{C}}_2, \phi)$ respectively.
- If $\overline{\mathcal{S}}_1$ includes a static linker named ν and $\overline{\mathcal{S}}_2$ includes a static linker with the same name, the two linkers must match according to Sec. 5.1.
- No dynamic linkers in $\overline{\mathcal{D}}_1$ and $\overline{\mathcal{D}}_2$ can share the same name.
- No connectors in $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ can share the same name.

Expression $A_1 + A_2$ has type $\mathbf{Asm}(\overline{\mathcal{S}}_1 \diamond_t \overline{\mathcal{S}}_2, \overline{\mathcal{D}}_1 \uplus \overline{\mathcal{D}}_2, \overline{\mathcal{C}}_1 \uplus \overline{\mathcal{C}}_2, \phi)$ if the above conditions are met. Here the $\diamond_t$ operator is the same as that of the $\diamond$ operator explained in Sec. 4.3, but changing $\overline{S}$, $\overline{I}$, $\overline{E}$ to $\overline{\mathcal{S}}$, $\overline{\mathcal{I}}$, $\overline{\mathcal{E}}$. Also notice that since first-class assemblages are given a type in which local feature types are set to ϕ, there can be no name clash checking on local features of A_1 and A_2. This is not a problem, however, since assemblages are α-convertible with respect to local feature names (see Sec. 4).

Dynamic Linking Expression $\mathbf{plugin}_{\nu_1 \mapsto \nu_2}\ e$, when well-typed in our type system, has a dynamic linkage type $\mathbf{Dlnk}(\overline{\mathcal{E}})$; this corresponds to the fact that the return value of a **plugin** expression is a dynamic linkage handle. It typechecks iff

– It appears in an assemblage whose type is $\mathbf{Asm}(\overline{\mathcal{S}}_1, \overline{\mathcal{D}}_1, \overline{\mathcal{C}}_1, \overline{\mathcal{L}}_1)$.
– Expression e, which will evaluate to an assemblage, has a type $\mathbf{Asm}(\overline{\mathcal{S}}_2, \overline{\mathcal{D}}_2, \overline{\mathcal{C}}_2, \phi)$.
– $\overline{\mathcal{D}}_1$ includes a dynamic linker type $\nu_1 \mapsto \langle \overline{\mathcal{I}}_1; \overline{\mathcal{E}}_1 \rangle$, and $\overline{\mathcal{S}}_2$ includes a static linker type $\nu_2 \mapsto \langle \overline{\mathcal{I}}_2; \overline{\mathcal{E}}_2 \rangle$, and the two types match according to Sec. 5.1.
– $\overline{\mathcal{E}} = \overline{\mathcal{E}}_1 \uplus \overline{\mathcal{E}}_2$.
– $\overline{\mathcal{C}}_2$ must be ϕ.
– Static linker ν_2 is the only static linker in $\overline{\mathcal{S}}_2$ which has imported features.

The last two conditions merit some further explanation. $\overline{\mathcal{C}}_2$ must be ϕ because if dynamic plugins had extra connectors, the assemblage runtime, after being plugged in with dynamic plugins of this kind, would be faced with a dilemma: it either needs to dynamically change its type to reflect some connectors that are dynamically added, or these new connectors are not exposed to outsiders and are *de facto* useless. Our calculus does not tackle this dilemma to preserve simplicity. The last condition is necessary because otherwise, the imported features not satisfied by dynamic linking would become dangling unresolved name references.

Cross-Computation Communication Expression $\mathbf{connect}_{\nu_1 \mapsto \nu_2}\ e$, when well-typed in our type system, has a connection type $\mathbf{Cnt}(\overline{\mathcal{E}}, \overline{\mathcal{J}}_1)$; this corresponds to the fact that the return value of a **connect** expression is a connection handle. It typechecks iff:

– It appears in an assemblage whose type is $\mathbf{Asm}(\overline{\mathcal{S}}_1, \overline{\mathcal{D}}_1, \overline{\mathcal{C}}_1, \overline{\mathcal{L}}_1)$.
– Expression e, which evaluates to an assemblage runtime handle, has a type $\mathbf{Rtm}(\overline{\mathcal{C}}_2)$.
– $\overline{\mathcal{C}}_1$ includes a connector type $\nu_1 \mapsto \langle \overline{\mathcal{I}}_1; \overline{\mathcal{E}}_1; \overline{\mathcal{J}}_1 \rangle$, and $\overline{\mathcal{C}}_2$ includes a connector type $\nu_2 \mapsto \langle \overline{\mathcal{I}}_2; \overline{\mathcal{E}}_2; \overline{\mathcal{J}}_2 \rangle$, and the two connector types match according to Sec. 5.1.
– $\overline{\mathcal{E}} = \overline{\mathcal{E}}_1 \uplus \overline{\mathcal{E}}_2$.

5.3 Properties of the Type System

We have proved soundness of our type system [LS04], in which we have shown the bootstrapping process preserves type, and the subject reduction property holds, *i.e.*, the $G,\ e \xrightarrow{\iota r, \iota n} G', e'$ reduction always preserves type. In addition, the typechecking process is decidable.

6 Related Work

In terms of static linking alone, our calculus is in the spirit of numerous module systems and calculi mentioned in Sec. 1 and Sec. 2. The calculus presented here supports first-class modules and static linking as first-class expressions, which some of the aforementioned projects, such as ML functors, do not support. In this presentation, we omitted how types can themselves be imported or exported as features, but type importing/exporting, including bounded parametric types and

cross-module recursive types is covered in the long version [LS04]. Previous works in this category, with the exception of Units, do not consider dynamic linking or cross-computation communication. For instance, ML modules do not themselves constitute a runtime, and even though structures have explicit interfaces for runtime interaction via $S.x$, this is for tightly-coupled interaction within a single runtime, not cross-computation invocation.

Dynamic linking is supported in Java. Although it does not support source-level dynamic linking expressions, classes are loaded dynamically [LB98,DLE03]. The classloader mechanism provides a very powerful way to customize the dynamic linking process, but its maximum expressiveness, particularly classloader delegation, requires much of the typechecking work to happen at dynamic link time. In addition, we believe the granularity of modules provides a better layer for dynamic linking, because explicit dynamic linking interfaces can be specified without too much labor, and users can have more programmatic control over the dynamic linking process. Dynamic linking of modules is explored in Argus [Blo83], in the **invoke** expressions of Units [FF98], and in the `dynamic export` declarations of MJ [CBGM03]. These projects only take advantage of the fact that dynamic plugins are modules and so the interface is unidirectional only: the running program has no explicit interface to the plugged-in code.

There have been many effective protocols developed for reactive computations, including RMI, RPC, and component architectures such as COM+ and CORBA. These protocols generally define a one-way communication interface only; the receiver has an interface, but not the sender. The bidirectional connector as a concept has existed in the software engineering community for some time, *e.g.* in [AG97], but those closest to our are two programming language efforts: ArchJava [ASCN03], and Cells [RS02]. In ArchJava, connectors are more low-level than ours. Each connector may have a `typecheck` method which maximize flexibility of typechecking, something we do not support. Connectors in our calculus share the same notion as in Cells language, but connectors in Cells do not consider rebindability and per-connection states. These projects in general do not have module system as a priority, and it therefore do not address type imports and exports.

Research on software components [Szy98] is diverse. A number of industrial component systems (such as COM+, Javabeans) have been successful in modelling reactive computations, and some support both static linking and cross-computation communication, such as CORBA CCM. They are only loosely related to our project, as they do not consider dynamic linking issue and type issues.

7 Conclusions and Future Work

The major contribution of this paper is a novel module system where static linking, dynamic linking and cross-computation communication are all defined in a uniform framework by declaring explicit, bi-directional interfaces. Explicit interfaces for dynamic linking and cross-computation communication provide

more declarative specifications of the interaction between parties, and also gives a stronger foundation for adding other critical language features such as security and version control. We have yet to see a fully bi-directional dynamic linking interface in the literature. Bi-directional communication interfaces are found for example in [ASCN03,RS02], but our work builds this feature into module systems. In the full version of this paper [LS04] we show how the calculus presented here can be extended to include types as features, and how they are useful for situations involving dynamic linking and cross-computation communication.

Since every cross-computation communication must be directed through connectors in our calculus, access control on connectors is enough to ensure the network security of the assemblage. Assemblages provide a good granularity for encapsulation, and our type system and semantics of the calculus restricts the types of data that can be transfered across computations, which also coincides with a proper policy of confinement in security. A future topic is to define a complete security architecture based on this calculus.

It is our belief that rebindable dynamic linkers can provide a strong initial basis upon which a rigorous theory of code version control can be built. Since dynamic plugins can be rebound, each successive binding at runtime is a new version of the code. Rebindability also allows multiple versions to co-exist. We are interested in building a version control layer on top of our calculus.

Acknowledgements We would like to acknowledge Ran Rinat for contributions at earlier stages of this project.

References

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[ASCN03] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In *Proceedings of the Seventeenth European Conference on Object-Oriented Programming*, June 2003.

[AZ02] D. Ancona and E. Zucca. A calculus of module systems. *Journal of functional programming*, 11:91–132, 2002.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of OOPSLA/ECOOP*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[BHSS03] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating, 2003.

[Blo83] Toby Bloom. Dynamic module replacement in a distributed programming system. Technical Report MIT/LCS/TR-303, 1983.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, 1997.

[CBGM03] John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: a rational module system for java and its applications. In *Proceedings of the*

18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, pages 241–254, 2003.

[DEW99] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A fragment calculus towards a model of separate compilation, linking and binary compatibility. In *Logic in Computer Science*, pages 147–156, 1999.

[DLE03] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible models for dynamic linking. In *12th European Symposium on Programming*, 2003.

[DS96] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, volume 31(6), pages 262–273, 1996.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[HL02] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, pages 6–20, 2002.

[HMN01] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 36–44, 1998.

[LS04] Yu David Liu and Scott F. Smith. Modules With Interfaces for Dynamic Linking and Communication (long version), `http://www.cs.jhu.edu/~scott/pll/assemblage/asm.pdf`. Technical report, Baltimore, Maryland, March 2004.

[Mac84] D. MacQueen. Modules for Standard ML. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 409–423, 1984.

[MFH01] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of OOPSLA*, October 2001.

[RS02] Ran Rinat and Scott Smith. Modular internet programming with cells. In *Proceedings of the Sixteenth ECOOP*, June 2002.

[SPW03] Nigamanth Sridhar, Scott M. Pike, and Bruce W. Weide. Dynamic module replacement in distributed protocols. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, May 2003.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[WV00] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Programming Languages and Systems, 9th European Symp. Programming*, volume 1782, 2000.