Component Assemblies and Component Runtimes

Yu David Liu Ran Rinat Scott F. Smith
Department of Computer Science
The Johns Hopkins University
{yliu, rinat, scott}@cs.jhu.edu

ABSTRACT
We propose a component programming language that supports an integrated notion of both compile-time and runtime component. The centerpiece of this paper is the static, compile time notion of *assembly*, complementing our previous work on the dynamic, runtime notion of *cell*. An assembly is a declarative, stateless piece of code that facilitates code combination. It offers explicit typed interfaces to outsiders, called *linkers*, which can be used to link smaller assemblies into bigger, compound assemblies. Each assembly may in turn be *loaded* at run-time, producing a cell in the runtime environment. A cell is a dynamic, stateful component that interacts with other cells via explicit runtime interfaces. Thus, the static assemblies and the dynamic cells are fully integrated. Assemblies may also be *dynamically* linked into a running cell, thereby expanding its code base on runtime. We present the model and the concepts, and then go on to define a toy language supporting assemblies and cells. We precisely define the langauge via a formal operational semantics and a set of typing rules.

1. INTRODUCTION
Most computer scientists would quickly agree that "component-based software" is a good thing, but would then likely disagree about what is meant by the term "component-based software". Szyperski in his book [17] and language designs such as Jiazzi [11] emphasize the static, code assemblage aspects of components. Components in the "real world"—Corba, COM, and Java Beans—emphasize the dynamic, run-time services that components offer, as do some other research projects [10, 4, 2]. A main thesis of this paper is that a language supporting components should explictly address *both* the statics and the dynamics of components. The *statics* concern code and how code is assembled, and is a topic related, but not identical to, module systems. The *dynamics* concern the run-time interface offered by components for interaction with outsiders.

In this paper we propose the notion of an *assembly* to model the static, declarative aspects of components. This complements our previous work on the notion of a *cell* [15, 14], a dynamic, run-time component. We call them "assemblies" because their purpose is to assemble components. Each assembly is a stateless piece of code that when loaded produces a cell in the runtime environment. That each assembly is also a factory for an explicit, stateful, run-time entity—a cell—is an important difference with most module systems. These static assemblies fit Szyperski's concept of a component [17]: they are deployable, stateless binary objects with clear static interfaces for use by third parties. And, the dynamic cells have explicit run-time interfaces like COM, Corba and Java Bean components.

Assemblies are for cells what classes are for objects: they contain definitions for everything that makes up a cell—classes, connectors, services, and instance variables. In addition, they have module features: they have fixed *linkers* which may import and export classes from other assemblies. Assemblies are separately compilable; this is facilitated through *assembly signatures*, which independently specify class types. Assemblies also compose: they may be linked together to produce larger, *compound* assemblies. These module aspects of assemblies are somewhat similar to Units/Jiazzi [11, 6]. Assemblies also may be dynamically linked to a running cell, through an explicit linking interface.

Before going into the features of assemblies in more detail, we review our previous work on the cell component architecture.

1.1 Review of Cells
In [15, 14], we defined cells. These cells were, using a class/object analogy, a *prototype-based* notion of component in the spirit of Self [18]. Prototype-based cells model both code combination and run-time interaction. There is something to be said for a prototype-based component language, but we have concluded that a cleaner language design is possible if the language clearly separates the statics from the dynamics. Thus in this paper we delegate code combination to assemblies, and keep cells purely for dynamic interaction. We now review some key features of cells.

Cells are stateful containers of objects and code, which run on a Cell Virtual Machine (CVM). A given CVM may have multiple cells running on it. Cells have explicit interfaces for interaction: they provide typed *connectors* for forming persistent run-time connections with other cells. Connectors carry *plugins* and *plugouts*, via which operations may be invoked on connected cells. Standard client-server style component interfaces, *services*, are also provided on cells.

Connectors allow long-term peer-to-peer interactions to be explicitly expressed in the language; they are analogous to the cabling of hardware systems running between monitor–computer, CD player–receiver, or computer–power outlet. Services, on the other hand, are open interfaces offering services to anyone. A good analogy would be the volume control on the TV, or the keyboard used to type this sentence on. Continuing the cable analogy, cells are also designed to allow "hot-plugging", cables that may be connected and disconnected while the system is running. We believe that there is a fundamental reason to have both connector- and service-type interaction; see [15, 14].

Cells are referenced universally via unique *Cell IDentifiers*, CID's. A CID is a bit string which includes location and other information. Thus, holding only a CID, a user can interact with that cell, whatever network-accesible node it resides on. CID's are large and random enough to be unguessable, so they also serve as secure capabilities for accessing a cell. In this regard, they are similar to the object references of E [13].

Cells are fundamentally distributed in that they may be linked locally or across the network. They also may be dynamically loaded, unloaded, copied, and moved.

A cell is a closed universe of objects: no direct object references may leak to other cells. This is achieved by adopting a by-copy semantics for passing object parameters.

Cells share concepts with existing component run-times such as CORBA [7] and COM/DCOM: they are larger-grained than objects, have explicit run-time interfaces for interaction, and support both local and distributed service invocations. Cells also differ in several ways. Most significantly, they are designed to make long-term interactions explicit by forming persistent peer-to-peer connections with other cells, via the connectors mentioned above. Cells also "own" a collection of objects, and so are a more explicit part of the run-time state than CORBA/COM components. And, capability-based security is built into the design.

Several distributed system frameworks include explicit persistent connections that are something like our cell connectors [10, 4]. ArchJava [2] is a language extension to Java which also has a feature something like our connectors. ArchJava components have *ports*, which are similar to cell plugins and plugouts. Their "components" are however much more lightweight, they are just objects with connectors on them. In a recent paper they have broadened the notion of connector supported in ArchJava to include *e.g.* distributed connections [1].

We have developed a preliminary implementation of prototype-based cells on top of the Java JVM [8]. The implementation supports a cell file format, loading of cells from files, dynamic connecting and disconnecting on connectors, operation invocation between connected cells, and full distributed functionality, including formation of remote cell connections and remote service invocation.

2. INFORMAL OVERVIEW

In this section we provide an informal overview of the key concepts of assemblies and signatures, via a running example.

2.1 Defining and Composing Assemblies

Figure 1 shows an example assembly architecture, a peer-to-peer music sharing system similar to the now-defunct Napster. The figure shows several assemblies, and assembly linkers whereby one assembly can import code from another.

In this example, a central server keeps information on songs, which are themselves kept on other client computers. Users access the central server to learn of music held by other users, and then may download desired songs directly from them. In addition to the server assembly, the developers of this system provided end-programmers with a reusable, middleware client-side assembly. End-programmers use this middleware assembly to construct their own client-side application using the provided song and file transfer utilities. The end application is responsible for decoding and playing the mp3 files, constructing a pleasing GUI, etc.

In Fig. 1, MusicServer is the assembly for the central server, and MusicClient is the client-side assembly. These names stand for *assembly identifiers*, AID's. Each assembly has a unique AID, which is a long bit string that uniquely and globally identifies each assembly across the network, in a manner related to COM UUID's. As will be seen in the formal semantics below, the only "true" name of an assembly is its AID. This design builds code versioning fundamentally into the language, avoiding a large source of errors and ambiguities. In a real implementation, programmers would likely not specify AIDs directly, but instead use a naming service to retrieve assemblies by symbolic names.

Now, here is an excerpt from MusicServer's code:

```
 assembly MusicServer

{
  connector SongInfo   // For storing and retrieving song info
     plugouts
       addSong : (s : Song) :  void { addSong code ...   }
       getSong : (n : String) : Song {  getSong code ...  }
  ...
}
```

The MusicServer assembly contains code defining the template for a cell, as depicted in Fig. 1. When loaded, that cell will carry a connector named SongInfo with two *plugouts*: addSong, for registering a new song in the central database, and getSong, for retrieving songs information. A client's cell communicates with the server cell via a dual connector, i.e. one carrying the same operation names, but as *plugins*. A given client must be explicitly connected to a given server via the connector before the client can invoke the server operations. Since cell references are universal, such a client could be on any accessible network node.

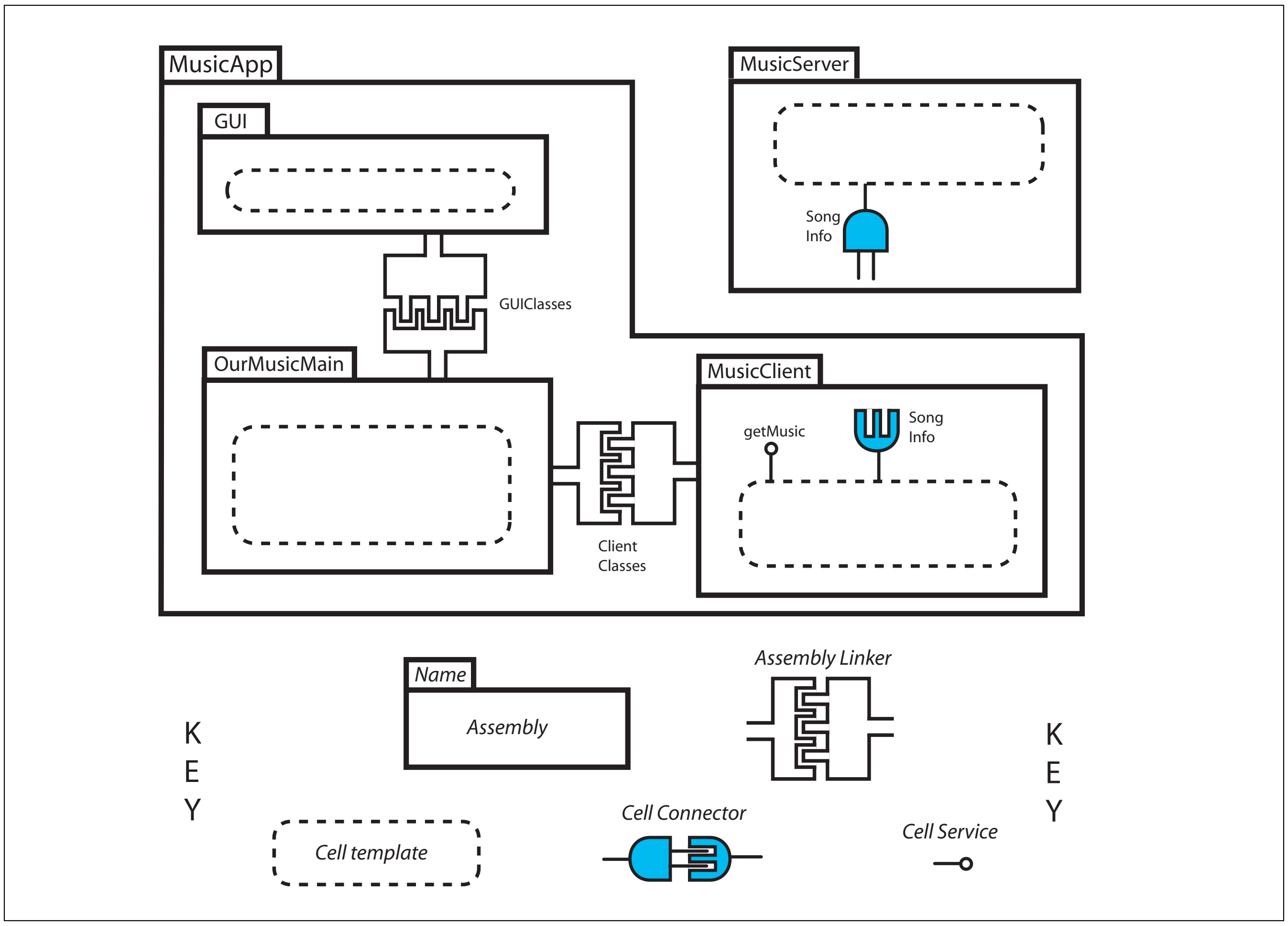

Figure 1: Music Sharing Example

The client cell is defined by a *compound assembly*, MusicApp, illustrating the concept of linking assemblies. It is defined as follows:

assembly MusicApp
  link OurMusicMain MusicClient GUI

i.e. MusicApp is the result of linking three smaller assemblies. Compound assemblies are the union of the classes, services and connectors of the component assemblies. The manner in which assembly linking unions the cell services and connectors is analogous to how a mixin of classes produces objects which have the union of the methods of the component classes. In case two or more of the linked assemblies carry an exporter, an importer, a connector, or a service by the same name, the link fails, unless they originate from the same assembly. This case is analogous to Eiffel's *repeated inheritance*. It allows two compound assemblies containing a common sub-assembly to link.

Assembly linking occurs through the *linkers* of each sub-assembly. A linker is either an *importer*, which lists classes being imported, or an *exporter*, which lists classes being exported. The matching of importers to exporters is made on the basis of identical linker names; and similarly, inside a linker, imports are matched to exports by name. Linking is a stateless operation that produces a new assembly from the component ones. This new compound assembly also has a unique AID, abbreviated MusicApp in this case.

Assembly MusicClient is provided by the middleware developers. It defines the dual connector needed to communicate with the server, a service for transferring music between clients, and classes for song manipulation and storage.

Here is an excerpt from MusicClient's code:

```
assembly MusicClient

  exporter ClientClasses  // to be exported to
                            assembly OurMusicMain
    class SongAction       // access to cell of MusicServer
    class Song             // Song details
    class Playable         // like Song plus mp3 file.
    class SongStore        // local songs store

{ // assembly body
  class SongAction
    {  inst ...   meth  ... }
 ... definitions of other classes, including Song, Playable, Song-
Store, PlayerGUI ...

  connector SongInfo   // For storing and retrieving song info
      plugins
        addSong : (s : Song) :  void { addSong code ...  }
        getSong : (n :  String) : Song {  getSong code ... }
 ...

   service transferSong (s : Song) : Playable { ... }
}
```

The MusicClient assembly has one linker, the exporter ClientClasses. It exports the classes an end-programmer will need to share music files. In this example, these classes are imported by the end-progammer's main-program assembly OurMusicMain, providing it with song manipulation and archiving functionality. The class SongAction provides access to the connector SongInfo. In particular, it is able to call the operations addSong and getSong. Song encapsulates all song information including a reference to the client cell where that song exists. Song does not contain the mp3 file itself. Playable is Song plus the mp3 file. SongStore is a song a archive class for local use by each client.

Here is an excerpt from OurMusicMain:

```
 assembly OurMusicMain

   importer ClientClasses
     class SongAction
     class Song
     class Playable
     class SongStore

   importer GUIClasses
     class Window
     class Frame
     ...

{ // assembly body

 class PlayerEngine ...  this class actually plays a file

}
```

OurMusicMain contains the logic for the interaction with human users, the central song server, and other client cells. To do that, it needs to import music-related classes from MusicClient and GUI classes from GUI. This is facilitated by the importers ClientClasses and GUIClasses. As explained above, the class SongAction provides control over the connector SongInfo defined in MusicClient. OurMusicMain defines the class PlayerEngine, which actually decodes and plays mp3 files.

To complete the picture, the GUI library assembly looks like this:

```
 assembly GUI

   exporter GUIClasses
     class Window
     class Frame
     ...

{  class Window { ... }
   ... other GUI classes ... }
```

Figure 2 shows a dynamic, runtime view of the example. Assemblies are loaded to create cells; the expression syntax

load MusicApp

dynamically loads a given assembly, in this case the one with AID MusicApp. A loaded assembly is a cell, and a reference to that cell is the value returned by the **load**. In the example, there are two client cells, MusicApp1 and MusicApp2, loaded from MusicApp, and one server cell, MusicServer1, loaded from MusicServer.

Once loaded, cells can *connect* with one another to form persistent connections. The syntax for this operation is *e.g.*

connect MusicServer **at** SongInfo

Fig. 2 shows the music sharing example after each MusicApp client has connected to the server connector. These connections are persistent, so once the client is connected to a particular server, the client can invoke addSong() and getSong() directly. This example also shows how a conector plugout can be multiply plugged-in: both MusicApp1 an MusicApp2 can invoke MusicServer1's plugout operations directly. Connections can be broken by the syntax *e.g.*

disconnect at SongInfo

—if either client or server disconnects at this connector, the client will no longer be able to invoke addSong() or getSong() on the server. So, for example if the client-server network failed, the client could disconnect from its SongInfo server and reconnect to a different server.

The client cells can transfer songs directly between themselves using the cell service transferSong, as indicated by the dashed line in Fig. 2.

2.2 Dynamic Linking

There are two notions of linking defined for assemblies, one a compositional, static notion, **link**, and the other a dynamic, run-time notion, **dynlink**. Compositional linking, as described in the previous Section, is used to build applications: it creates a bigger binary out of smaller binaries. Compositional linking is purely at the assembly level—no cells are involved. *Dynamic linking* on the other hand may be used to extend the codebase of a given cell at runtime. That is, a dynamic link command

dynlink $AID_1 \ldots AID_n$

may be executed within any running cell. It links the assemblies $AID_1, \ldots AID_n$ to that cell, adding the classes in $AID_1 \ldots AID_n$ to its code base. No new *AID* or *CID* is created. As with static link, the matching of linkers is done by name. The linked assemblies also link *among themselves* on identically named linkers.

We illustrate dynamic linking by extending our running example. Consider the extension shown in Fig. 3. The assembly OurMusicMain now has an extra importer and an

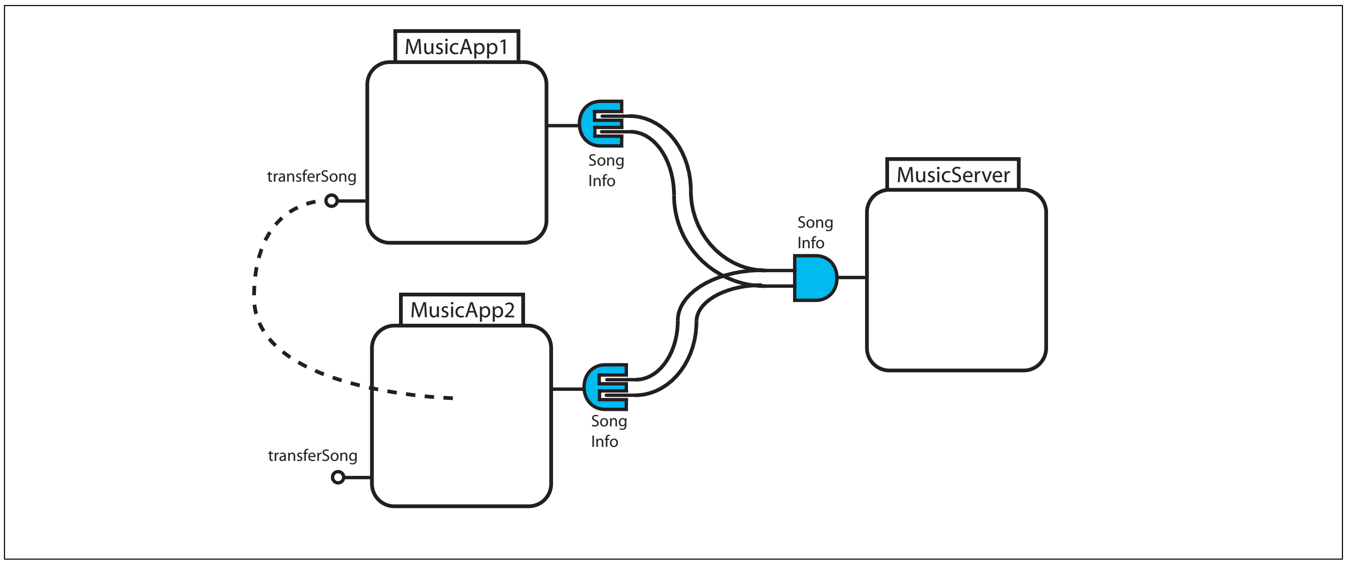

Figure 2: The Dynamics of Music Sharing

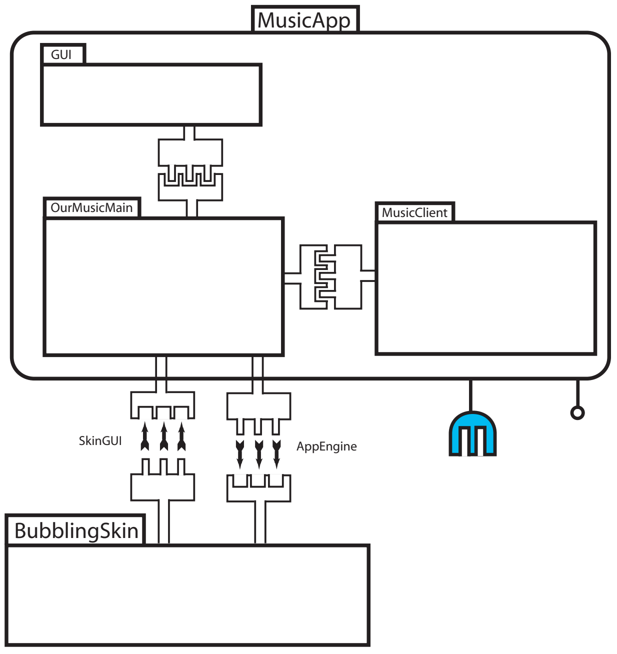

Figure 3: Dynamically linking an Assembly

extra exporter. This new pair of linkers allow it to exchange classes with an assembly defining a *skin*. The latter will provide a class for displaying the player with the skin.

Here is the code for the extended OurMusicMain.

```
 assembly OurMusicMain

   importer ClientClasses
    as before ...

   importer GUIClasses
     as before ...

   importer SkinGUI // new importer
     class PlayerGUI

   exporter AppEngine // new exporter
     class PlayerEngine


{ // assembly body

 class PlayerEngine  ...

}
```

OurMusicMain is designed to export its locally defined PlayerEngine class and to import back a PlayerGUI class, which will be a customized display, from the skin assembly.

The skin assembly, however, is not statically linked into the application. Instead, these linkers remain open so that different skins may be dynamically linked at runtime. One important aspect of dynamic linking in assemblies is it uses the *exact* same linkers also used for static linking, so applications which dynamically link clearly declare what their open channels of code import and export are. This is one of the novel aspects of assemblies.

Here is sample code for a particular skin assembly, BubblingSkin:

```
 assembly BubblingSkin  // A sample skin which
                           emits 3D bubbles

   exporter SkinGUI
     class PlayerGUI ...

   importer AppEngine
     class PlayerEngine ...

   importer GUIClasses
     class Window ...

{ // assembly body

 class PlayerGUI ...

}
```

Assembly BubblingSkin imports the class PlayerEngine and builds around it a GUI with bubbling graphics, class PlayerGUI, which it exports back to the main application.

The MusicApp application is defined as before:

```
 assembly MusicApp  link OurMusicMain MusicClient GUI
```

i.e., without a skin assembly. It acquires the linkers SkinGUI and AppEngine from OurMusicMain. After MusicApp is loaded, the resulting cell may dynamically link to BubblingSkin as shown in Fig. 3, via an execution sequence of

```
 load MusicApp
      .
      .
      .
 dynlink BubblingSkin
```

The appearance of the player can later be changed by linking in another skin:

```
 dynlink YellowSnakeSkin
```

This demonstrates an important property of dynamic linking: if any of the newly linked assemblies carry exporters that match importers bound by previous dynamic links, these importers are re-linked to the new exporters. So here, after linking YellowSnakeSkin, objects instantiated in MusicApp from PlayerGUI will be from the YellowSnakeSkin version of that class. Dynamic linking however is a purely *additive* operation in terms of code: the BubblingSkin version of the class is not explicitly unloaded, so there may be objects instantiated before the second link still using BubblingSkin code.

Dynamically linked assemblies cannot define connectors or services. Using the analogy of cells and objects, it would be like an object adding methods at run-time. Dynamically adding new connectors or services may in fact be a useful feature to eventually support, but it is fraught with design difficulties and we have left this feature out of our current design.

The information regarding a dynamic link is kept in the cell performing the link, in its *link table*. The assemblies themselves are not modified, they are stateless and immutable. This is a primary difference with static linking, were a new assembly with its own *AID* is the result of composition.

2.3 Type Signatures

Assembly Signatures

An assembly is a unit of compilation, and to compile an assembly, the types of all classes it references must thus be known. While some of them are local and hence fully defined in the assembly, others will be imported from other assemblies. The type of such these external classes is made available to the typechecker through *Assembly Signatures*.

An assembly signature is a collection of class signatures declared by the programmer; it is assigned a unique *Assembly Signature IDentifier*, an ASID. ASIDs along with their definitions are stored in a global repository, just like AIDs. Inside an assembly, a list of ASIDs may be specified after the **use** keyword. The typechecker must find any non-local class type among the **use**d assembly signatures.

We illustrate assembly signatures by elaborating on our running example. Consider for example the end-user's main assembly OurMusicMain. It has three importers, each of which specifies several classes. In order to actually use objects of these classes in the code, their types must be known. Therefore the assembly needs to specify assembly signatures in its **use** clause. A typical usage is there will be one signature listed for every assembly it intends to link with, statically or dynamically.

Here is the code for OurMusicMain:

```
 assembly OurMusicMain
   use MusicClientSig GUISig SkinSig

   importer ClientClasses
     class SongAction
     class Song
     class Playable
     class SongStore

   importer GUIClasses
     class Window
     class Frame
     ...

   importer PlayerGUIClass
     class PlayerGUI

   exporter PlayerEngineClass
     class PlayerEngine


{ // assembly body

... song =  new Song;   song.artist := "Sher"; ...
```

}

The assembly signatures MusicClientSig, GUISig, and SkinSig define the class types for the classes defined on the importers ClientClasses, GUIClasses, and PlayerGUIClasses respectively. These class types enable the typechecking of references to these classes in the body of the assembly, such as the sample line shown.

The assembly signature itself simply contains definition of class types. For example, MusicClientSig is:

```
assembly signature MusicClientSig
    use BasicClassesSig
    clsig SongAction {
       addSong : (s : Song) :  void
       getSong : (n : String) : Song
       connect : (c :  cellsig MusicServerCellSig) :  void
       disconnect : ( void):  void
    }
    clsig Song {
       title: String
       ...
    }
```

MusicClientSig is the ASID, which realistically would be a long bit string. Like AIDs, in a real implementation ASIDs would not be specified directly, but a name service would be used instead.

The type of class PlayerGUI, to be imported dynamically from a skin assembly, is defined via *one* assembly signature SkinSig, even though many skin assemblies are expected to be linked.

Assembly signatures may themselves refer to external classes in fields and method parameters. For that reason, they may also contain **use** clauses, listing other ASIDs, providing these external types. For example, the MusicClientSig uses a BasicClassesSig signature that defines the type of basic classes, such as String.

One of the main goals of the type system design is a "declare once" principle which means a given type will only be written once. This is the reason for managing assembly signatures in a global repository via unique ASIDs. After a signature has been registered with some ASID, it cannot be changed. A new signature must be defined if a change is made.

Typically, each assembly will have a corresponding signature that declares the types of classes defined in the assembly. So MusicServer has a corresponding signature MusicServerSig. But in general, assembly signatures are just a collection of class type definitions.

Cell Signatures

In MusicClientSig above, the parameter for the connect method in class SongAction is a *cell signature*. Cell signatures have unique identifiers, CSIDs, and are also managed by global repositories. They are used to declare the type of cell references that may be passed, and constitute a collection of typed connectors and services.

The cell signature MusicServerCellSig used above is:

```
cell signature MusicServerCellSig
  use BasicClassesSig
 connector SongInfo   // For storing and retrieving song info
     plugouts
       addSong : (s : Song) :  void
       getSong : (n : String) : Song
```

Like assembly signatures, cell signatures specify a list of **use**d ASIDs, which serve to resolve class types referred to by name. The global ID principles that apply to assembly signatures also apply to cell signatures: once they are loaded they cannot be changed.

3. THE LANGUAGE

In this section we present the language of assemblies and cells used to code the examples in the previous sections, and which we give a formal semantics to in subsequent sections. The language features represent a compromise between a language too simple to address the difficult issues, and too complex for formal treatment. Inheritance, exceptions, and distributed execution are some features left out for simplicity. Figure 4 shows the syntax for assemblies and expressions.

We use the notation $\overline{X}$ to represent a finite sequence of X's. An assembly is defined by the **assembly** keyword. It can be either atomic or compound. An atomic assembly specifies an AID (α), a list of **use**d assembly signatures ($\overline{\delta}$), a set of linkers, and a body $\{\overline{inst}\ \overline{srvdf}\ \overline{cntdf}\ \overline{clsdf}\ e\}$. The body specifies the elements that make up a cell: instance variables (fields), services, connectors, class definitions, and an initialization expression, in this order. Each linker is either an importer or an exporter. An importer specifies a list of classes to be imported, defined by the syntax **importer** *ln* **class** *clsn*. *ln* is the linker's name and *clsn* is a class name. Similarly for exporters.

A compound assembly also specifies an AID, a list of assemblies to be linked, and a list of linkers on the resulting compound. In the link, importers are matched to exporters by name. The linkers in the resulting compound are declared and must be chosen from the constituent linkers minus those importers consumed by the link.

A service definition, **service** *srvn* $\{\overline{opdf}\}$, specifies a name for the service (*srvn*) and a set of implemented operations ($\overline{opdf}$). A connector definition
connector *cntn* **plugin** $\overline{opsig}$ **plugout** $\overline{opdf}$ specifies a name for the connector, plugins, and plugouts. Plugins are just operation signatures whereas plugouts are implemented operations.

The syntax **connect** *e* **at** *cntn* is used to connect the current cell to the cell referenced by *e* at connector *cntn*. For **disconnect** one need only specify the connector. The syntax *cntn*−> *opn*(*e*) invokes plugin or plugout *opn* on connector

$$
\begin{array}{lll}
\mathit{asm} & ::= \ \textbf{assembly}\ \alpha\ \textbf{use}\ \overline{\sigma}\ \overline{\mathit{lnkr}}\ \{\overline{\mathit{inst}}\ \overline{\mathit{srvdf}}\ \overline{\mathit{cntdf}}\ \overline{\mathit{clsdf}}\ \ e\} & \textit{atomic assembly} \\
 & \quad \mid \textbf{assembly}\ \alpha\ \textbf{link}\ \overline{\alpha}\ \overline{\mathit{lnkr}} & \textit{compound assembly} \\
 & & \\
\mathit{srvdf} & ::= \ \textbf{service}\ \mathit{srvn}\ \{\overline{\mathit{opdf}}\} & \textit{service defn} \\
\mathit{cntdf} & ::= \ \textbf{connector}\ \mathit{cntn}\ \textbf{plugin}\ \overline{\mathit{opsig}}\ \textbf{plugout}\ \overline{\mathit{opdf}} & \textit{connector defn} \\
\mathit{clsdf} & ::= \ \textbf{class}\ \mathit{clsn}\ \{\textbf{inst}\ \overline{\mathit{inst}}\ \textbf{meth}\ \overline{\mathit{opdf}}\} & \textit{class defn} \\
\mathit{opdf} & ::= \ \mathit{opn}(x : n\tau) : n\tau\ \{e\} & \textit{operation defn} \\
 & & \\
x & & \textit{variable} \\
e & ::= \ \textbf{null} \mid i \mid b \mid x \mid \textbf{let}\ x = e\ \textbf{in}\ e & \textit{elementary expressions} \\
 & \quad \mid \textbf{this} \mid e.\mathit{opn}(e) \mid e.\mathit{instn} \mid e.\mathit{instn} := e \mid \textbf{new}\ \mathit{clsn} & \textit{object expressions} \\
 & \quad \mid \textbf{thiscell} \mid \mathit{instn} \mid \mathit{instn} := e \mid \ e..\mathit{srvn} {->} \mathit{opn}(e) \mid \mathit{cntn} {->} \mathit{opn}(e) & \textit{cell expressions} \\
 & \quad \mid \textbf{connect}\ e\ \textbf{at}\ \mathit{cntn} \mid \textbf{disconnect at}\ \mathit{cntn} & \textit{cell expressions} \\
 & \quad \mid \textbf{dynlink}\ \overline{\alpha} \mid \textbf{load}\ \alpha & \textit{assembly expressions} \\
\alpha, \sigma, \varsigma, \theta & & \textit{AID, ASID, CSID, CID}
\end{array}
$$

Figure 4: Abstract Syntax

cntn. It may be issued onyl from within the cell having this connector. The syntax $e..srvn{->}opn(e)$ is used to call the operation *opn* on the service *srvn* of cell e. It is used from *outside the cell*, hence the need to specify the cell reference e.

load α loads the assembly with AID α to create a cell in the runtime environment. **dynlink** $\overline{\alpha}$ dynamically links the assemblies $\overline{\alpha}$ into the current cell.

Dot notation $e.opn(e)$ etc. is used for object operations: method call, field access, and field update.

Figure 5 shows the type syntax.

clsn is a class name. It is used to denote both class types on linkers and object types in fields and operation parameters. **cellsig** ς, where ς is a CSID, is the type of a cell. It is used in instance variables an operation parameters. *srvsig* is a service signature—the type of a service. *cntsig* is a connector signature—the type of a connector.

Assembly signatures specify an ASID (σ), a set of **use**d ASIDs ($\overline{\sigma}$), and a list of class types ($\overline{csln}$). Cell signatures specify a CSID (ς), a set of **use**d assembly signatures ($\overline{\sigma}$), a list of service signatures ($\overline{srvsig}$), and a list of connector signatures ($\overline{cntsig}$).

4. THE DYNAMIC SEMANTICS

This section formally defines the dynamics of our language. We first give some basic mapping definitions we will use throughout the paper, and then define the structure of cell states. The bulk of the section defines operational semantics rules for the language.

4.1 Global Mapping Definitions

The unverse of computation is composed of assemblies, assembly signatures, cell signatures and the running cells. Each of these entities are identified by their AIDs, ASIDs, CSIDs and CIDs, respectively. The following mapping exists globally to map the *ID*s to the real entities:

$\mathbf{A} : \alpha \mapsto \langle asm, \sigma, \varsigma \rangle$
$\mathbf{AS} : \sigma \mapsto asig$
$\mathbf{CS} : \varsigma \mapsto csig$
$\mathbf{S} = \mathbf{AS} \cup \mathbf{CS}$

A maps an AID α to its assembly code *asm*, ASID σ of the assembly signature it implements, and CSID ς of the cell signature it implements; Note that each assembly signature or cell signature can be implemented by many different assemblies, but each assembly has a unique assembly signature and cell signature, which serves as the public information it communicates to others. **AS** maps an ASID σ to its assembly signature *asig*; **CS** maps an CSID ς to its cell signature *csig*.

Although this paper does not deal with security specifically, some general guidelines on protection of these mappings are followed when we define semantics and type systems for our language. **AS** and **CS** mappings are public: types are free. For **A**, mapping an AID to its ASID or CSID is also public, while mapping to its code *asm* may be restricted.

4.2 Cell State

Unlike assemblies, cells are run-time entities and are therefore have a state. The universe of cell states is represented by Σ, which is defined as a tuple:

$$\Sigma \overset{\text{def}}{=} \langle \Sigma_{\texttt{aid}}, \Sigma_{\texttt{lnk}}, \Sigma_{\texttt{cnt}}, \Sigma_{\texttt{obj}}, \Sigma_{\texttt{ins}} \rangle$$

For a specific cell with CID θ, its state $\Sigma(\theta)$ includes the following parts. Except the first one, all are defined as sets:

- $\Sigma_{\texttt{aid}}(\theta)$ is the AID of the assembly (atomic or compound) cell θ is loaded from.
- $\Sigma_{\texttt{lnk}}(\theta)$ is the link table of cell θ. Tuple $\langle \alpha_i, ln, \alpha_e, flag \rangle$ denotes the importer *ln* of α_i is currently linked to the exporter (by the same name) of α_e. If an importer is currently unmatched with any exporter, α_e is set to **null**; if an exporter is unmatched with any importer, α_i is set to **null**. Since our language supports both static linking and dynamic linking, *flag* differentiates its nature. If *flag* is set to `s`, the link is static; if set to `d`, the link is dynamic. Since dynamic linking is possible, this information must be maintined at run-time.
- $\Sigma_{\texttt{cnt}}(\theta)$ is the connection table of cell θ. Tuple $\langle \alpha, cntn_t, \theta \rangle$ denotes connector $cntn_t$ with code defined in α is cur-

$\tau ::= \textbf{nat} \mid \textbf{bool} \mid \mathit{clsn} \mid \langle \mathit{clsn}, \sigma \rangle \mid \textbf{cellsig}\ \varsigma$	*user type*
$\mathit{inst} ::= \mathit{instn} : n\tau$	*cell/class instance*
$\mathit{opsig} ::= \mathit{opn} : n\tau \to n\tau$	*operation sig*
$\mathit{srvsig} ::= \textbf{service}\ \mathit{srvn}\ \{\overline{\mathit{opsig}}\}$	*service sig*
$\mathit{cntsig} ::= \textbf{connector}\ \mathit{cntn}\ \{\textbf{plugin}\ \overline{\mathit{opsig}}\ \textbf{plugout}\ \overline{\mathit{opsig}}\}$	*connector sig*
$\mathit{lnkr} ::= \textbf{importer}\ \mathit{ln}\ \overline{\textbf{class}\ \mathit{clsn}} \mid \textbf{exporter}\ \mathit{ln}\ \overline{\textbf{class}\ \mathit{clsn}}$	*linkers*
$\mathit{asig} ::= \textbf{assembly signature}\ \sigma\ \textbf{use}\ \overline{\sigma}\ \overline{\mathit{lnkr}}\ \{\overline{\textbf{clsig}\ \mathit{clsn}\ \{\ \textbf{inst}\ \overline{\mathit{inst}}\ \textbf{meth}\ \overline{\mathit{opsig}}\}}\}$	*assembly sig*
$\mathit{csig} ::= \textbf{cell signature}\ \varsigma\ \textbf{use}\ \overline{\sigma}\ \{\overline{\mathit{srvsig}}\ \overline{\mathit{cntsig}}\}$	*cell sig*

Figure 5: Types

rently connected to cell θ. θ is set to **null** if the current connector is not connected with any other cell.

- $\Sigma_{\mathtt{obj}}(\theta)$ is the object store for cell θ. This is the place a cell stores its locally generated objects during execution. Tuple $\langle \alpha, ob_t, \langle \mathit{clsn}, \mathit{IS} \rangle \rangle$ denotes object ob_t has its code defined by class *clsn* from α, and has an object instance store *IS*. *IS* is a simple store mapping object instance variables to values.
- $\Sigma_{\mathtt{ins}}(\theta)$ is the store for cell instances. Tuple $\langle \alpha, \mathit{instn}_t, v \rangle$ denotes the instance variable instn_t defined in the name space of α has a value v.

The Link Table

Since a cell might be loaded from a compound assembly and/or it might be dynamically linked at run-time, its code may come from many different atomic assemblies. The link table defines how every building block—atomic assembly—is linked to one another to eventually form the complete codebase of the running cell. Because our language has a strong notion of codebase, the linking process preserves the identity information of all the linked parties.

Given an assembly, the inital link table, which reflects the static linking structure of assemblies, is now defined. Future dynamic linking will modify the table to reflect these new links. This is achieved by the following function *InitLT*:

$$\mathit{InitLT}(\alpha) \overset{\text{def}}{=} \begin{cases} \mathit{Imp}(\alpha) \cup \mathit{Exp}(\alpha) & \\ \qquad\qquad \alpha \text{ atomic} & \\ \mathit{Scrn}(\mathit{MrgLT}(\mathtt{s}, \{\mathit{InitLT}(\alpha_1), \ldots \mathit{InitLT}(\alpha_n)\}), \alpha) & \\ \qquad\qquad \alpha \text{ cmpd of } \alpha_1 \ldots \alpha_{\mathrm{n}} & \end{cases}$$

Before giving the formal definitions of the functions used in the above, we first give an overall explanation. This definition shows that the link table for an atomic assembly records only importer and exporter information. For an importer *iln*, tuple $\langle \alpha, \mathit{iln}, \textbf{null}, \mathtt{d} \rangle$ is entered into the table. For an exporter *eln*, tuple $\langle \textbf{null}, \mathit{eln}, \alpha, \mathtt{d} \rangle$ is entered. Note that these **null** importer and exporter entries in the link table stand for unlinked parties. It is therefore irrelevant whether d or s should be declared as a flag. We pick d as a convention. The definitions of *Imp* and *Exp* define this precisely:

$$\mathit{Imp}(\alpha) \overset{\text{def}}{=} \bigcup_{\mathit{iln} \in^{\mathtt{s}}_{\mathtt{ilnk}} \alpha} \{\langle \alpha, \mathit{iln}, \textbf{null}, \mathtt{d} \rangle\}$$

$$\mathit{Exp}(\alpha) \overset{\text{def}}{=} \bigcup_{\mathit{eln} \in^{\mathtt{s}}_{\mathtt{elnk}} \alpha} \{\langle \textbf{null}, \mathit{eln}, \alpha, \mathtt{d} \rangle\}$$

The link table for a compound assembly is the merge of all link tables of its subparts. During the merge, if one subpart has an importer and another subpart has an exporter and these two share the same name, a link is formed. The link table records this. Note that exporters do not get absorbed after being linked, they can be re-exported by the compound. Function $\mathit{MrgLT}(\mathit{flag}, \{\mathit{lt}_1, \ldots, \mathit{lt}_n\})$ merges link tables $\mathit{lt}_1, \ldots, \mathit{lt}_n$. *flag* indicates whether the merge is static or dynamic. The process of computing a link table of a compound is a static one, which can be realized by function *MrgLT* when *flag* is set to s:

$$\mathit{MrgLT}(\mathit{flag}, \{\mathit{lt}_1, \ldots, \mathit{lt}_n\}) \overset{\text{def}}{=} \bigcup_{t \in \{1..n\}} \mathit{lt}_t \ \cup\ \mathit{NewL} - \mathit{OldL}$$

where

$$\begin{aligned} \mathit{NewL}(\mathit{flag}) = \{\langle \alpha_u, \mathit{ln}, \alpha_v, \mathit{flag} \rangle \mid \quad & (\langle \alpha_u, \mathit{ln}, \alpha_w, \mathtt{d} \rangle \in \mathit{lt}_p) \wedge \\ & (\langle \textbf{null}, \mathit{ln}, \alpha_v, \mathtt{d} \rangle \in \mathit{lt}_q) \wedge \\ & ((\mathit{flag} = \mathtt{s}) \Longrightarrow (\alpha_w = \textbf{null}))\} \end{aligned}$$

$$\begin{aligned} \mathit{OldL}(\mathit{flag}) = \{\langle \alpha_u, \mathit{ln}, \alpha_w, \mathtt{d} \rangle \mid \quad & (\langle \alpha_u, \mathit{ln}, \alpha_w, \mathtt{d} \rangle \in \mathit{lt}_p) \wedge \\ & (\langle \textbf{null}, \mathit{ln}, \alpha_v, \mathtt{d} \rangle \in \mathit{lt}_q) \wedge \\ & ((\mathit{flag} = \mathtt{s}) \Longrightarrow (\alpha_w = \textbf{null}))\} \end{aligned}$$

for some $p,\ q \in \{1..n\},\ p \neq q$ and some w

Dynamic linking, illustrated in *MrgLT* when *flag* is set to d, is slightly different from its static counterpart. For static linking, to form a link an *unlinked* importer is assumed $((\mathit{flag} = \mathtt{s}) \Longrightarrow (\alpha_w = \textbf{null}))$. This constraint is loosened in dynamic linking, where we allow *relinking*: a formed dynamic link can be overrided and the importer can be relinked to a newly matched exportor.

In our language, a compound assembly also explicitly declares its importers and exporters. This declarative design is good from a software engineering point of view, and the explicit declaration can also be used to limit exporters. *MrgLT* "by default" does not consume exporters and a screening function *Scrn* is defined to deal with the issue of limiting exporters:

$$\mathit{Scrn}(\mathit{lt}, \alpha) \overset{\text{def}}{=} \mathit{lt} - \bigcup_{\mathit{ln} \notin^{\mathtt{s}}_{\mathtt{elnk}} \alpha} \{\langle \textbf{null}, \mathit{ln}, \alpha_e \rangle\}$$

The Connection Table and Cell Instance Store

For a cell whose code comes from multiple atomic assemblies, its connection table must contain the connector state of each and every subpart atomic assembly. The connection table is indexed by both the connector name and the AID of the atomic assembly defining the connector. When a cell is loaded, the initial connection table can be computed by the following simple function *InitCT*:

$$\mathit{InitCT}(\alpha) \overset{\text{def}}{=} \begin{cases} \bigcup_{\mathit{cntn} \in^{\mathtt{s}}_{\mathtt{cnt}} \alpha} \{\langle \alpha, \mathit{cntn}, \mathbf{null} \rangle\} & \alpha \text{ atomic} \\ \mathit{InitCT}(\alpha_1) \cup \cdots \cup \mathit{InitCT}(\alpha_n) & \alpha \text{ cmpd of } \alpha_1 \ldots \alpha_\mathrm{n} \end{cases}$$

Likewise, the function to compute the intial instance store when a cell is loaded from an assembly α is *InitIT*:

$$\mathit{InitIT}(\alpha) \overset{\text{def}}{=} \begin{cases} \bigcup_{\mathit{instn} \in^{\mathtt{s}}_{\mathtt{cellinst}} \alpha} \{\langle \alpha, \mathit{instn}, \mathbf{null} \rangle\} & \alpha \text{ atomic} \\ \mathit{InitIT}(\alpha_1) \cup \cdots \cup \mathit{InitIT}(\alpha_n) & \alpha \text{ cmpd of } \alpha_1 \ldots \alpha_\mathrm{n} \end{cases}$$

4.3 Operational Semantics

The Operational sematics of our language is given in Fig. 6, via a small step evaluation relation. Each rule takes the following form:

$$\mathbf{A} \vdash \langle e, \alpha, \theta, \Sigma \rangle \Longrightarrow \langle e', \alpha', \theta', \Sigma' \rangle$$

where α is the *assembly context* and θ the *cell context* of execution. The assembly context is the assembly in which code fragment e appears. The cell context keeps track of the cell in which e is executing.

Our language is rooted in a distributed environment, where cells and assemblies can be located on different locations. By nature distributed computation is multi-threaded. In this presentation, however, our focus is not on distribution or threads, and therefore we assume a single-threaded model to avoid distraction. In a single-threaded model we must context-switch when execution moves from one cell or assembly context to another; we have added a **to** expression, described shortly, to explicitly force a context switch.

We extend the original user level expressions with new syntax needed to implement the operational semantics. Values v and evaluation contexts E are also defined:

$$\begin{array}{lcl} e & ::= & \cdots \mid ob \mid \theta \mid \mathbf{to}(\alpha, \theta, e) \\ v & ::= & i \mid b \mid ob \mid \theta \mid \mathbf{null} \end{array}$$

$$\begin{array}{lcl} E & ::= & [\,] \mid \; E.\mathit{opn}(e) \mid \; v.\mathit{opn}(E) \\ & & \mid \; \mathit{instn} := E \mid \; E.\mathit{instn} \mid \; E.\mathit{instn} := e \mid \; v.\mathit{instn} := E \\ & & \mid \; \mathbf{let}\ x = E\ \mathbf{in}\ e \mid \; \mathbf{to}(\alpha, v, E) \\ & & \mid \; E..\mathit{srvn}\text{->}\,\mathit{opn}(e) \mid \; v..\mathit{srvn}\text{->}\,\mathit{opn}(E) \\ & & \mid \; \mathbf{connect}\ E\ \mathbf{at}\ \mathit{cntn} \mid \; \mathit{cntn}\text{->}\,\mathit{opn}(E) \end{array}$$

Expression $\mathbf{to}(\alpha, \theta, e)$ means that after evaluating E, the assembly context and cell context should be shifted to α and θ, respectively. The **to** expression is used to switch from executing code in one cell to executing code in another, *e.g.* when one cell invokes a service on another.

Some simple containment relations are used in the semantics rules. $\mathit{clsn} \in^{\mathtt{s}}_{\mathtt{clsdf}} \alpha$ denotes class *clsn* is locally defined in assembly identified by α; $\alpha_1 \in^{\mathtt{s}}_{\mathtt{compose}} \alpha_2$ denotes α_1 is a subpart (by one or more level composition) of α_2; All the other notations with symbol $\in^{\mathtt{s}}$ share the same pattern, and are defined in Fig. 10. $a \in^{\mathtt{s}}_{\mathtt{b}} c$ denotes a is declared as a $\mathtt{b}$ in c. "$||$" is an operator used to update cell states defined as follows:

$$\begin{array}{lcl} S||\langle x, y, \mathit{newV} \rangle & \overset{\text{def}}{=} & (S - \langle x, y, \mathit{oldV} \rangle) \cup \{\langle x, y, \mathit{newV} \rangle\} \\ & & \langle x, y, \mathit{oldV} \rangle \in S \\ S||\langle x, \mathit{newV} \rangle & \overset{\text{def}}{=} & (S - \langle x, \mathit{oldV} \rangle) \cup \{\langle x, \mathit{newV} \rangle\} \\ & & \langle x, \mathit{oldV} \rangle \in S \end{array}$$

Description of Semantics

The **load** rule prepares the initial link table, connection table and instance store, and the initialization code of the loaded cell is executed. Because of the compound nature of assemblies, cell loading triggers the execution of the initialization code of all atomic assemblies forming the loaded compound. The process will be multi-threaded in a full implementation, but we encode them in a single-threaded model here for simplicity. $\mathit{Init}(\alpha, \theta)$ is a function linearizing the expressions that will be evaluated when α is initialized in the context of cell θ:

$$\mathit{Init}(\alpha, \theta) \overset{\text{def}}{=} \begin{cases} \mathbf{to}(\alpha, \theta_s, e) & \alpha \text{ atomic, with init code } e \\ \mathit{Init}(\alpha_1, \theta); \ldots; \mathit{Init}(\alpha_n, \theta) & \alpha \text{ cmpd of } \alpha_1 \ldots \alpha_\mathrm{n} \end{cases}$$

where $e_1; e_2 \overset{\text{def}}{=} (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)$ and x does not occur free in e_2.

The **dynlink** rule dynamically merges the table of all assemblies in the parameter list with the table of the current cell. All the assemblies dynamically linked will be initialized. **new** rule creates an object, but since the class can also be imported, the link table may be needed to retrieve a non-local class. **connect** and **disconnect** add and remove the asociated connections from the connection table.

Invoking an operation on a connector can mean different things. If the operation is a plugout, it is a local call. If the operation is a plugin instead, the connection table needs to be used to locate the plugout operation the connected cell.

Notice we have different ways of handling parameters depending on inter-cell communication and intra-cell communication. For intra-cell communication, all parameters are passed as it is, including the objects being passed as references. For the former case, if the passed parameter is an object reference, the deep copy is passed. This is reflected in the **invoke srv op** and **invoke cnt plugin** rules. Deep copy is defined by function $\mathit{DeepCopy}(\theta_{src}, \theta_{dst}, v, \Sigma)$:

$$\mathit{DeepCopy}(\theta_{src}, \theta_{dst}, v, \Sigma) \overset{\text{def}}{=} \begin{cases} \langle v, \Sigma \rangle & \text{if } v \text{ is an integer, boolean, CID, or AID} \\ \langle v', \Sigma' \rangle & \text{if } v \text{ is an object reference} \end{cases}$$

v' is a fresh object reference held by θ_{dst}
$\Sigma' = \Sigma^{n+1}$ except $\Sigma'_{\mathtt{obj}}(\theta_{dst}) = \Sigma^{n+1}_{\mathtt{obj}}(\theta_{dst})||\langle \alpha, v', \langle \mathit{clsn}, \mathit{IS}' \rangle \rangle$
$\langle v'_i, \Sigma^{i+1} \rangle = \mathit{DeepCopy}(\theta_{src}, \theta_{dst}, v_i, \Sigma^i)$ for each $i \in [1..n]$
$\mathit{IS}' = \{\langle \mathit{instn}_1, v'_1 \rangle \ldots \langle \mathit{instn}_n, v'_n \rangle\}$
$\langle \alpha, v, \langle \mathit{clsn}, \mathit{IS} \rangle \rangle \in \Sigma_{\mathtt{obj}}(\theta_{src})$
$\mathit{IS} = \{\langle \mathit{instn}_1, v_1 \rangle \ldots \langle \mathit{instn}_n, v_n \rangle\}, \ \Sigma^0 = \Sigma$

(load)

$\mathbf{A} \vdash \langle E\llbracket\, \mathbf{load}\ \alpha_1 \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, \mathit{Init}(\alpha_1,\theta_1); \mathbf{to}(\alpha_0,\theta_0,\theta_1) \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma'\rangle$

θ_1 is a fresh CID, $\Sigma' = \Sigma$ except $\Sigma'(\theta_1) = \langle \alpha_1, \mathit{InitLT}(\alpha_1), \mathit{InitCT}(\alpha_1), \phi, \mathit{InitIT}(\alpha_1)\rangle$

(dynlink)

$\mathbf{A} \vdash \langle E\llbracket\, \mathbf{dynlink}\ \alpha_1, \ldots \alpha_n \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, \mathit{Init}(\alpha_1,\theta_1); \ldots \mathit{Init}(\alpha_n,\theta_1); \mathbf{to}(\alpha_0,\theta_0,\theta_0) \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma'\rangle$

$\Sigma' = \Sigma$ except $\Sigma'_{\texttt{lnk}}(\theta_0) = \mathit{MrgLT}(\mathtt{d}, \{\Sigma_{\texttt{lnk}}(\theta_0), \mathit{InitLT}(\alpha_1), \ldots \mathit{InitLT}(\alpha_n)\})$

$\Sigma'_{\texttt{ins}}(\theta_0) = \Sigma_{\texttt{inst}}(\theta_0) \cup \mathit{InitIT}(\alpha_1) \cdots \cup \mathit{InitIT}(\alpha_n)$

(new)

$\mathbf{A} \vdash \langle E\llbracket\, \mathbf{new}\ \mathit{clsn} \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, \mathit{ob} \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma'\rangle$

$\Sigma' = \Sigma$ except $\Sigma'_{\texttt{obj}}(\theta_0) = \Sigma_{\texttt{obj}}(\theta_0) || \langle \alpha, \mathit{ob}, \langle \mathit{clsn}, \mathit{IS}\rangle\rangle$, $\quad \mathit{IS} = \bigcup_{\mathit{instn} \in^{\texttt{s}}_{\texttt{clsinst}} \langle \mathit{clsn}, \alpha\rangle} \{\langle \mathit{instn}, \mathbf{null}\rangle\}$

ob is a fresh object reference, $\alpha = \begin{cases} \alpha_0 & \mathit{clsn} \in^{\texttt{s}}_{\texttt{clsdf}} \alpha_0 \\ \alpha_1 & \mathit{clsn} \in^{\texttt{s}}_{\texttt{clsilnkr}} \langle \mathit{ln}, \alpha_0\rangle,\ \Sigma_{\texttt{lnk}}(\theta_0, \alpha_0, \mathit{ln}) = \alpha_1 \text{ for some } \mathit{ln} \end{cases}$

(connect)

$\mathbf{A} \vdash \langle E\llbracket\, \mathbf{connect}\ \theta_1\ \mathbf{at}\ \mathit{cntn} \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, \theta_0 \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma'\rangle$

$\Sigma' = \Sigma$ except $\Sigma'_{\texttt{cnt}}(\theta_0) = \Sigma_{\texttt{cnt}}(\theta_0) || \langle \alpha_0, \mathit{cntn}, \theta_1\rangle$, $\quad \Sigma'_{\texttt{cnt}}(\theta_1) = \Sigma_{\texttt{cnt}}(\theta_1) || \langle \alpha_1, \mathit{cntn}, \theta_0\rangle$

$\mathit{cntn} \in^{\texttt{s}}_{\texttt{cnt}} \alpha_1,\ \alpha_1 \in^{\texttt{s}}_{\texttt{compose}} \Sigma_{\texttt{aid}}(\theta_1)$

(disconnect)

$\mathbf{A} \vdash \langle E\llbracket\, \mathbf{disconnect\ at}\ \mathit{cntn} \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, \theta_0 \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma'\rangle$

$\langle \alpha_0, \mathit{cntn}, \theta_1\rangle \in \Sigma_{\texttt{cnt}}(\theta_0),\ \langle \alpha_1, \mathit{cntn}, \theta_0\rangle \in \Sigma_{\texttt{cnt}}(\theta_1)$

$\Sigma' = \Sigma$ except $\Sigma'_{\texttt{cnt}}(\theta_0) = \Sigma_{\texttt{cnt}}(\theta_0) || \langle \alpha_0, \mathit{cntn}, \mathbf{null}\rangle$, $\quad \Sigma'_{\texttt{cnt}}(\theta_1) = \Sigma_{\texttt{cnt}}(\theta_1) || \langle \alpha_1, \mathit{cntn}, \mathbf{null}\rangle$

(invoke srv op)

$\mathbf{A} \vdash \langle E\llbracket\, \theta_1..\mathit{srvn}\text{->}\,\mathit{opn}(v) \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, \mathbf{to}(\alpha_0,\theta_0, e[\theta_1/\mathbf{thiscell}, v'/x]) \,\rrbracket,\ \alpha_1,\ \theta_1,\ \Sigma'\rangle$

$\mathit{srvn} \in^{\texttt{s}}_{\texttt{srv}} \alpha_1,\ \mathit{opn} \in^{\texttt{s}}_{\texttt{srvop}} \langle \mathit{srvn}, \alpha_1\rangle,\ e$ is body of $\mathit{opn},\ \alpha_1 \in^{\texttt{s}}_{\texttt{compose}} \Sigma_{\texttt{aid}}(\theta_1)$

$\langle v', \Sigma'\rangle = \mathit{DeepCopy}(\theta_0, \theta_1, v, \Sigma)$

(invoke class op)

$\mathbf{A} \vdash \langle E\llbracket\, \mathit{ob}.\mathit{opn}(v) \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, e[\theta_0/\mathbf{thiscell}, \mathit{ob}/\mathbf{this}, v/x] \,\rrbracket,\ \alpha_1,\ \theta_0,\ \Sigma\rangle$

$\langle \alpha_1, \mathit{ob}, \langle \mathit{clsn}, \mathit{IS}\rangle\rangle \in \Sigma_{\texttt{obj}}(\theta_0)$, $\quad e$ is body of opn of clsn of α_1

(invoke cnt plugin)

$\mathbf{A} \vdash \langle E\llbracket\, \mathit{cntn}\text{->}\,\mathit{opn}(v) \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, \mathbf{to}(\alpha_0,\theta_0, e_1[\theta_0/\mathbf{thiscell}, v'/x]) \,\rrbracket,\ \alpha_1, \theta_1,\ \Sigma'\rangle$

$\mathit{opn} \in^{\texttt{s}}_{\texttt{plugin}} \langle \mathit{cntn}, \alpha_0\rangle$, $\quad \langle \alpha_0, \mathit{cntn}, \theta_1\rangle \in \Sigma_{\texttt{cnt}}(\theta_0)$, $\quad \alpha_1 \in^{\texttt{s}}_{\texttt{compose}} \Sigma_{\texttt{aid}}(\theta_1)$

$\mathit{opn} \in^{\texttt{s}}_{\texttt{plugout}} \langle \mathit{cntn}, \alpha_1\rangle$, $\quad e_1$ is body of opn, $\langle v', \Sigma'\rangle = \mathit{DeepCopy}(\theta_0, \theta_1, v, \Sigma)$

(invoke cnt plugout)

$\mathbf{A} \vdash \langle E\llbracket\, \mathit{cntn}\text{->}\,\mathit{opn}(v) \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, e_0[\theta_1/\mathbf{thiscell}, v/x] \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \qquad \mathit{opn} \in^{\texttt{s}}_{\texttt{plugout}} \langle \mathit{cntn}, \alpha_0\rangle$

(get cell instn)

$\mathbf{A} \vdash \langle E\llbracket\, \mathit{instn} \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, v \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \qquad \langle \alpha_0, \mathit{instn}, v\rangle \in \Sigma_{\texttt{ins}}(\theta_0)$

(set cell instn)

$\mathbf{A} \vdash \langle E\llbracket\, \mathit{instn} := v \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, v \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma'\rangle$

$\Sigma' = \Sigma$ except $\Sigma'_{\texttt{ins}}(\theta_0) = \Sigma_{\texttt{ins}}(\theta_0) || \langle \alpha_0, \mathit{instn}, v\rangle$

(get obj instn)

$\mathbf{A} \vdash \langle E\llbracket\, \mathit{ob}.\mathit{instn} \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, v \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \qquad \langle \alpha_1, \mathit{ob}, \langle \mathit{clsn}, \mathit{IS}\rangle\rangle \in \Sigma_{\texttt{obj}}(\theta_0),\ \langle \mathit{instn}, v\rangle \in \mathit{IS}$

(set obj instn)

$\mathbf{A} \vdash \langle E\llbracket\, \mathit{ob}.\mathit{instn} := v \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, v \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma'\rangle$

$\langle \alpha_0, \mathit{ob}, \langle \mathit{clsn}, \mathit{IS}\rangle\rangle \in \Sigma_{\texttt{obj}}(\theta_0),\ \mathit{IS}' = \mathit{IS} || \langle \mathit{instn}, v\rangle,\ \Sigma'_{\texttt{obj}}(\theta_0) = \Sigma_{\texttt{obj}}(\theta_0) || \langle \alpha_0, \mathit{ob}, \langle \mathit{clsn}, \mathit{IS}'\rangle\rangle$

(let)

$\mathbf{A} \vdash \langle E\llbracket\, \mathbf{let}\ x = v\ \mathbf{in}\ e \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, e[v/x] \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle$

(to)

$\mathbf{A} \vdash \langle E\llbracket\, \mathbf{to}(\alpha_1, \theta_1, v) \,\rrbracket,\ \alpha_0,\ \theta_0,\ \Sigma\rangle \Longrightarrow \langle E\llbracket\, v \,\rrbracket,\ \alpha_1,\ \theta_1,\ \Sigma\rangle$

Figure 6: Small Step Semantics

5. THE FORMAL TYPE SYSTEM

In this Section we define the well-typed assemblies, classes and expressions. First a closure conversion process is defined which resolves type name references, and then the type rules are given.

5.1 Extended Types

The user-level types presented in Fig. 5 include class/object types *clsn*, object closure types $\langle clsn, \sigma\rangle$, and cell signatures **cellsig** ς. All of these types share a common trait, they contain type references that need resolving: either class name references, ASID references, or both.

In order to define the well-typed programs, we need to resolve these type references. It is not possible to eagerly resolve all references because there may be recursive references and thus an eager approach would yield infinitely large types. We take a two-pronged approach here: we define a pre-processing step on types which disambiguates class/object types *clsn* by converting them to the object closure types $\langle clsn, \sigma\rangle$ indicating the assembly signature σ in which the class type was originally defined. This process we term *closure conversion*; it is described in Section 5.2.

Secondly, in type derivations we may lazily replace these object closure types with their one-level expansions, by defining type equivalences between the closure types and their expansions. For this process to work, we need to add the one-level expansions to the type grammar:

$$\begin{array}{l} e\tau ::= \tau \mid cellt \mid clsobt \\ cellt ::= \textbf{cellsig}\ \{\ \overline{srvsig}\ \overline{cntsig}\ \overline{inst}\} \\ clsobt ::= clsn\ \{\ \textbf{inst}\ \overline{inst}\ \textbf{meth}\ \overline{opsig}\} \end{array}$$

cellt is a full cell type, and *clsobt* is a full class/object type. (Note that since in our type system full object types and class types do not appear in the same context, we give them the same form for simplicity.)

The type eqivalences which relate full object types to object closure types, and full cell types to **cellsig** ς types are the subtyping rules (**obj eq**) and (**cellsig eq**) of Figure 9, which we repeat here:

$$(\textbf{obj eq})\ \frac{\mathbf{S} = \mathbf{AS} \cup \mathbf{CS}}{\mathbf{S} \vdash_{sub}\ \langle clsn, \sigma\rangle \equiv ObjectT(\mathbf{AS}(\sigma), clsn)}$$

$$(\textbf{cellsig eq})\ \frac{\mathbf{S} = \mathbf{AS} \cup \mathbf{CS}}{\mathbf{S} \vdash_{sub}\ \textbf{cellsig}\ \varsigma \equiv CellT(\mathbf{CS}(\varsigma))}$$

$\mathbf{S} \vdash_{eq} \tau \equiv \tau'$ denotes two types τ and τ' are equal under assembly signatures $\mathbf{S}$. *ObjectT* and *CellT* are the obvious functions to extract the relevant types from the signatures:

$$\begin{array}{l} ObjectT(asig, clsn) \overset{\text{def}}{=}\ clsn\ \{\ \textbf{inst}\ \overline{inst}\ \textbf{meth}\ \overline{opsig}\} \\ \quad asig =_s \textbf{assembly signature}\ \sigma\ \{\overline{ocsig}\} \\ \quad clsn\ \{\ \textbf{inst}\ \overline{inst}\ \textbf{meth}\ \overline{opsig}\} \text{ is an element in } \overline{ocsig} \end{array}$$

$$\begin{array}{l} CellT(csig) \overset{\text{def}}{=}\ \textbf{cellsig}\ \{\ \overline{srvsig}\ \overline{cntsig}\ \} \\ \quad csig = \textbf{cell signature}\ \varsigma\ \{\overline{srvsig}\ \overline{cntsig}\} \end{array}$$

5.2 Closure Conversion

In this section we define how every object type of the form *clsn* can be converted to an object closure type of the form $\langle clsn, \sigma\rangle$, resolving the type of the class to its defining signature σ. This type resolution algorithm proceeds with respect to the assembly, assembly signature, or cell signature where *clsn* appears. The basic idea is to see if the class is declared locally, and if not, search through the ASID's listed in the **use** clause to find an assembly signature giving the type information. After this stage, every class (or object type) occurence will be explicitly bound to its defining signature. The **use** clauses may then be ignored after closure conversion since they have served their purpose.

Assembly signature closure conversion $\Downarrow_{\texttt{as}}$ is defined as follows:

$$\begin{array}{l} \textbf{signature}\ \sigma_0\ \textbf{use}\ \overline{\sigma}\ \overline{lnkr}\ \{\overline{clst}\} \Downarrow_{\texttt{as}} \\ \textbf{signature}\ \sigma_0\ \overline{lnkr'}\ \{\overline{clst'}\} \end{array}$$

where $lnkr'$, $clst'$ are *lnkr*, *clst* with each class name occurence *clsn* replaced by $\langle clsn, \sigma\rangle$, for σ the first ASID in the sequence $\overline{\sigma}$ satisfying $clsn \in^{\texttt{t}}_{\texttt{clsig}} \mathbf{AS}(\sigma)$. Here "occurence" means *clsn* is used as an object type, or is used as an import or export declaration in a linker.

Cell signature closure conversion $\Downarrow_{\texttt{cs}}$ is defined as follows:

$$\mathbf{AS} \vdash \textbf{signature}\ \varsigma\ \textbf{use}\ \overline{\sigma}\ \{cellt\} \Downarrow_{\texttt{cs}} \textbf{signature}\ \varsigma\ \{cellt'\}$$

where $cellt'$ is *cellt* with each class name occurence *clsn* replaced by $\langle clsn, \sigma\rangle$, for σ is the first ASID in the sequence $\sigma_1, \ldots, \sigma_n$ satisfying $clsn \in^{\texttt{t}}_{\texttt{clsig}} \mathbf{AS}(\sigma)$.

Atomic assembly closure conversion $\Downarrow_{\texttt{a}}$ is defined as follows:

$$\begin{array}{l} \mathbf{AS}, \sigma_0 \vdash \textbf{assembly}\ \alpha\ \textbf{use}\ \overline{\sigma}\ \overline{lnkr}\ \{\overline{inst}\ \overline{srvdf}\ \overline{cntdf}\ \overline{clsdf}\ e\} \\ \quad \Downarrow_{\texttt{a}} \textbf{assembly}\ \alpha\ \overline{lnkr'}\ \{\overline{inst'}\ \overline{srvdf'}\ \overline{cntdf'}\ \overline{clsdf'}\ e'\} \end{array}$$

where $lnkr'$, $inst'$, $srvdf'$, $cntdf'$, $clsdf'$, e' are the same as their original counterparts except that every class name occurence *clsn* is replaced with $\langle clsn, \sigma\rangle$, where σ is the first ASID in sequence $\overline{\sigma}$ satisfying $clsn \in^{\texttt{t}}_{\texttt{clsig}} \mathbf{AS}(\sigma)$. Here occurence means *clsn* is used as an object type, or **class** *clsn* is used as an import or export declaration in a linker, or **new** *clsn* expression is used. Compound assemblies closure convert to themselves.

Globally, the closure conversion of the whole universe of assembly signatures, cell signatures and assemblies is the closure of all components in the universe, $\Downarrow_{\texttt{uconv}}$:

$$\frac{\begin{array}{c} \mathbf{AS} = \bigcup_i \{\sigma_i \mapsto asig_i\}, \mathbf{AS}' = \bigcup_i \{\sigma_i \mapsto asig'_i\} \\ asig_i \Downarrow_{\texttt{as}} asig'_i \text{ for each } i \\ \mathbf{CS} = \bigcup_j \{\varsigma_j \mapsto csig_j\}, \mathbf{CS}' = \bigcup_j \{\varsigma_j \mapsto csig'_j\} \\ \mathbf{AS}' \vdash_{cs} csig_j \Downarrow_{\texttt{cs}} csig'_j \text{ for each } j \\ \mathbf{A} = \bigcup_k \{\alpha_k \mapsto \langle asm_k, \sigma_k, \varsigma_k\rangle\} \\ \mathbf{A}' = \bigcup_k \{\alpha'_k \mapsto \langle asm'_k, \sigma'_k, \varsigma'_k\rangle\} \\ \mathbf{AS}', \sigma_k \vdash_a asm_k \Downarrow_{\texttt{a}} asm'_k \text{ for each } k \end{array}}{\langle \mathbf{AS}, \mathbf{CS}, \mathbf{A}\rangle \Downarrow_{\texttt{uconv}} \langle \mathbf{AS}', \mathbf{CS}', \mathbf{A}'\rangle}$$

The closure conversion process necessitates some small changes to the syntax used in the typing rules. **new** *clsn* becomes **new** $\langle clsn, \sigma\rangle$. And, classes **class** *clsn* appearing in linkers become **class** $\langle clsn, \sigma\rangle$.

5.3 The Type System

Fig. 7 shows the top level typing rules; Fig. 8 shows the typing rules for expressions. Fig. 9 shows the subtyping rules.

Some notations with symbol $\in^{\mathtt{t}}$ are used in rules, which are of a common pattern: $a \in^{\mathtt{t}}_{\mathtt{b}} c$ denotes a is declared as a $\mathtt{b}$ in type c. These relations are defined in Figure 10.

Global Coherence Rule

(**global coherence**) of Fig. 7 ensures all assemblies in the universe $\mathbf{A}$ are well-typed in the presence of a universe of signatures $\mathbf{S}$, and thus the whole universe typechecks. For each $(\alpha \mapsto \langle asm, \sigma, \varsigma\rangle) \in \mathbf{A}$, the rule verifies that assembly *asm* implements assembly signature σ and cell signature ς. The Global coherence rule closure converts all items in the universe, and then ensures each (closure-converted) assembly in the universe typechecks.

Atomic Assembly Type Rule

The type of an atomic assembly is a pair, composed of its assembly signature and its cell signature.

For an atomic assembly to typecheck, each operation that defines services and connector plugouts, each locally defined class, and the initialization code must typecheck. The types of **thiscell** and **this** need to be handled carefully.

Atomic assemblies must also satisfy a few constraints to be well-formed. The most important is:

$$\begin{aligned}&\mathit{ClassUseValid}(\alpha) \overset{\text{def}}{=} \text{ for each } clsn \text{ some } \sigma, ln\\ &\quad clsn \in^{\mathtt{s}}_{\mathtt{needed}} \alpha \Rightarrow clsn \in^{\mathtt{s}}_{\mathtt{clsdf}} \alpha \text{ or } \langle clsn, \sigma\rangle \in^{\mathtt{t}}_{\mathtt{clsimp}} \langle ln, \sigma\rangle\end{aligned}$$

This asserts that any class name used in assembly α to declare object types, or appearing in a **new** expression, must be either defined locally, or imported via an importer. This constraint ensures every variable declared with an object type, or every **new** expression can get obtain its code at run-time.

Other constraints include:

$$\begin{aligned}&\mathit{AtomExpValid}(\alpha) \overset{\text{def}}{=} \text{ for each } clsn \text{ some } \sigma, ln\\ &\quad \langle clsn, \sigma\rangle \in^{\mathtt{t}}_{\mathtt{clsexp}} \langle ln, \sigma\rangle \Rightarrow clsn \in^{\mathtt{s}}_{\mathtt{clsdf}} \alpha\end{aligned}$$

$$\begin{aligned}&\mathit{AtomImpValid}(\alpha) \overset{\text{def}}{=} \text{ for each } clsn \text{ some } \sigma, ln\\ &\quad \langle clsn, \sigma\rangle \in^{\mathtt{t}}_{\mathtt{clsimp}} \langle ln, \sigma\rangle \Rightarrow clsn \notin^{\mathtt{s}}_{\mathtt{clsdf}} \alpha\end{aligned}$$

$$\begin{aligned}&\mathit{DistinctImp}(\alpha) \overset{\text{def}}{=} \text{ for each } clsn \text{ some } \sigma_1, \sigma_2, ln_1, ln_2\\ &\quad \langle clsn, \sigma_1\rangle \in^{\mathtt{t}}_{\mathtt{clsimp}} \langle ln_1, \sigma\rangle, \langle clsn, \sigma_2\rangle \in^{\mathtt{t}}_{\mathtt{clsimp}} \langle ln_2, \sigma\rangle \Rightarrow\\ &\qquad\qquad ln_1 = ln_2\end{aligned}$$

$\mathit{AtomExpValid}(\alpha)$ asserts all exported classes must be defined locally; $\mathit{AtomImpValid}(\alpha)$ enforces the fact that imported classes cannot be defined locally; and, $\mathit{DistinctImp}(\alpha)$ asserts that no single class can be imported on more than one importer.

Compound Assembly Type Rule

The type of a compound assembly is also a pair, composed of its assembly signature and its cell signature. The assembly signature must be the merge of the assembly signatures of its components. The merged assembly signature must bear the linker information of the compound, and have a list of class/object type declarations combined from all type declarations in component assemblies. Similarly, the cell signature must be the combination of the lists of service signatures and connector signatures of the subpart assemblies. Function $\mathit{MergeAS}(\overline{lnkr_k}, \sigma, \{\sigma_1, \ldots \sigma_n\})$ produces a new assembly signature with identity σ by combining assembly signatures identified by $\sigma_1, \ldots \sigma_n$, with $\overline{lnkr_k}$ defining linkers of the new assembly signature. Function $\mathit{MergeCS}(\varsigma, \{\varsigma_1, \ldots \varsigma_n\})$ produces a new cell signature identified by $CSID$ by combining cell signatures identified by $\varsigma_1, \ldots \varsigma_n$. We omit their formal definitions since they are purely syntactical.

To typecheck a compound assembly α, a set of constraints must to be satisfied. First, certain linker import and export conflicts must be avoided.

$$\begin{aligned}&\mathit{NoImpConflict}(\alpha_1, \ldots \alpha_n) \overset{\text{def}}{=}\\ &\quad ln \in^{\mathtt{t}}_{\mathtt{ilnk}} \sigma_i, ln \in^{\mathtt{t}}_{\mathtt{ilnk}} \sigma_j, i, j \in [1..n], i \neq j \Rightarrow\\ &\qquad |\mathit{Orig}(\{\alpha_1, \ldots \alpha_n\}, ln)| = 1\end{aligned}$$

$$\begin{aligned}&\mathit{NoExpConflict}(\alpha_1, \ldots \alpha_n) \overset{\text{def}}{=}\\ &\quad ln \in^{\mathtt{t}}_{\mathtt{elnk}} \sigma_i, ln \in^{\mathtt{t}}_{\mathtt{elnk}} \sigma_j, i, j \in [1..n], i \neq j \Rightarrow\\ &\qquad |\mathit{Orig}(\{\alpha_1, \ldots \alpha_n\}, ln)| = 1\end{aligned}$$

Each linker (importer or exporter) defined on a compound assembly must have been originally defined on one of its component atomic assemblies. *NoImpConflict* generally requires none of the linked assemblies to share a linker name, *but* linkers of the same name imported in two different component assemblies of a compound are allowed if they ultimately originated from the same atomic assembly. This check resembles how Eiffel [12] allows a class to multiply inherit from two classes, each of which had inherited from the same class A: only one copy of A is meant. In assemblies this case arises if two assemblies each are compounds including the same assembly α, and those two assemblies are then compounded. *NoExpConflict* is similar. Finding the atomic assembly where the linker was initially defined is achieved by function $\mathit{Orig}(\mathcal{S}, ln)$. It extracts all atomic assemblies occuring either directly or inside a compound in an assembly in set $\mathcal{S}$ that define *ln*; if there is only one such atomic assembly then the definition is unique. Its definition is as follows given $\langle asm_i, \sigma_i, \varsigma_i\rangle = \mathbf{A}(\alpha_i)$ for each i:

$$\mathit{Orig}(\mathcal{S}, ln) \overset{\text{def}}{=} \begin{cases} \phi & \mathcal{S} = \{\alpha\}, \alpha \text{ atomic}, ln \notin^{\mathtt{t}}_{\mathtt{lnk}} \sigma\\ \{\sigma\} & \mathcal{S} = \{\alpha\}, \alpha \text{ atomic}, ln \in^{\mathtt{t}}_{\mathtt{lnk}} \sigma\\ \bigcup_i \mathit{Orig}(\{\alpha_i\}, ln) & \mathcal{S} = \{\alpha\}, \alpha \text{ compound of } \alpha_1 \ldots \alpha_{\mathrm{n}}\\ \bigcup_i \mathit{Orig}(\{\alpha_i\}, ln) & \mathcal{S} = \{\alpha_1 \ldots \alpha_n\}\end{cases}$$

In exactly the same style, *NoSrvConflict*, *NoCntConflict* and *NoClsConflict* can be defined. They claim no two subparts of

$$(\textbf{global coherence})\quad \frac{\begin{array}{l}\mathbf{S} = \mathbf{AS} \cup \mathbf{CS}, \quad \langle \mathbf{AS}, \mathbf{CS}, \mathbf{A} \rangle \Downarrow_{\texttt{uconv}} \langle \mathbf{AS}', \mathbf{CS}', \mathbf{A}' \rangle, \quad \mathbf{S}' = \mathbf{AS}' \cup \mathbf{CS}' \\ \mathbf{S}', \mathbf{A}', \alpha \vdash_a asm : \langle \mathbf{AS}'(\sigma), \mathbf{CS}'(\varsigma) \rangle \text{ for each } (\alpha \mapsto \langle asm, \sigma, \varsigma \rangle) \in \mathbf{A}'\end{array}}{\mathbf{S}, \mathbf{A} \vdash_g \mathbf{OK}}$$

$$(\textbf{atomic asm})\quad \frac{\begin{array}{l} \mathit{ClassUseValid}(\alpha), \quad \mathit{AtomExpValid}(\alpha), \quad \mathit{AtomImpValid}(\alpha), \quad \mathit{DistinctImp}(\alpha) \\ \mathit{cellt} = \textbf{cellsig } \{\overline{\textbf{service } \mathit{srvn}_s \ \{\overline{\mathit{opn}_{s_w} : \tau_{s_w} \rightarrow \tau'_{s_w}}\}} \\ \qquad \overline{\textbf{connector } \mathit{cntn}_c \textbf{ plugin } \overline{\mathit{opn}_{c_u} : \tau_{c_u} \rightarrow \tau'_{c_u}} \textbf{ plugout } \overline{\mathit{opn}_{c_v} : \tau_{c_v} \rightarrow \tau'_{c_v}}} \ \overline{\mathit{instn}_i : \tau_i}\} \\ \mathbf{S}, \mathbf{A}, \alpha, \{x_{s_w} : \tau_{s_w}, \textbf{thiscell} : \mathit{cellt}\} \vdash e_{s_w} : \tau'_{s_w} \text{ for each } s_w \\ \mathbf{S}, \mathbf{A}, \alpha, \{x_{c_v} : \tau_{c_v}, \textbf{thiscell} : \mathit{cellt}\} \vdash e_{c_v} : \tau'_{c_v} \text{ for each } c_v \\ \mathbf{S}, \mathbf{A}, \alpha, \Gamma \cup \{\textbf{thiscell} : \mathit{cellt}\} \vdash_c \mathit{clsdf}_r : \mathit{clst}_r \text{ for each } r \\ \mathbf{S}, \mathbf{A}, \alpha, \{\textbf{thiscell} : \mathit{cellt}\} \vdash \ e : \tau \\ \langle asm, \sigma, \varsigma \rangle = \mathbf{A}(\alpha) \\ \mathit{asig} =_s \textbf{signature } \sigma \ \overline{\mathit{lnkr}_m} \ \{\overline{\mathit{clst}_r}\} \\ \mathit{csig} =_s \textbf{signature } \varsigma \ \{\overline{\textbf{service } \mathit{srvn}_s \ \{\overline{\mathit{opn}_{s_w} : \tau_{s_w} \rightarrow \tau'_{s_w}}\}} \\ \qquad \overline{\textbf{connector } \mathit{cntn}_c \textbf{ plugin } \overline{\mathit{opn}_{c_u} : \tau_{c_u} \rightarrow \tau'_{c_u}} \textbf{ plugout } \overline{\mathit{opn}_{c_v} : \tau_{c_v} \rightarrow \tau'_{c_v}}}\} \end{array}}{\begin{array}{l} \mathbf{S}, \mathbf{A}, \alpha \vdash_a \textbf{assembly } \alpha \ \overline{\mathit{lnkr}_m} \\ \qquad \{\overline{\textbf{service } \mathit{srvn}_s \ \{\overline{\mathit{opn}_{s_w}(x_{s_w} : \tau_{s_w}) : \tau'_{s_w} \ \{e_{s_w}\}}\}} \\ \qquad \overline{\textbf{connector } \mathit{cntn}_c \textbf{ plugin } \overline{\mathit{opn}_{c_u} : \tau_{c_u} \rightarrow \tau'_{c_u}} \textbf{ plugout } \overline{\mathit{opn}_{c_j}(x_{c_v} : \tau_{c_v}) : \tau'_{c_v} \ \{e_{c_v}\}}} \\ \qquad \overline{\mathit{instn}_n : \tau_n} \quad \overline{\mathit{clsdf}_r}\} \quad e\} : \ \langle \mathit{asig}, \mathit{csig} \rangle \end{array}}$$

$$(\textbf{cmpnd asm})\quad \frac{\begin{array}{l} \mathit{NoImpConflict}(\alpha_1, \ldots \alpha_n), \quad \mathit{NoExpConflict}(\alpha_1, \ldots \alpha_n), \quad \mathit{LinkTypeMatched}(\alpha_1, \ldots \alpha_n), \\ \mathit{NoSrvConflict}(\alpha_1, \ldots \alpha_n), \quad \mathit{NoCntConflict}(\alpha_1, \ldots \alpha_n), \quad \mathit{NoClsConflict}(\alpha_1, \ldots \alpha_n) \\ \mathit{CompImpValid}(\alpha), \quad \mathit{CompExpValid}(\alpha), \\ \langle asm, \sigma, \varsigma \rangle = \mathbf{A}(\alpha), \quad \langle asm_i, \sigma_i, \varsigma_i \rangle = \mathbf{A}(\alpha_i) \text{ for each } i \end{array}}{\mathbf{S}, \mathbf{A}, \alpha \vdash_a \textbf{assembly } \alpha \textbf{ link } \overline{\alpha_i} \ \overline{\mathit{lnkr}_k} : \ \langle \mathit{MergeAS}(\overline{\mathit{lnkr}_k}, \sigma, \{\sigma_1, \ldots \sigma_n\}), \mathit{MergeCS}(\varsigma, \{\varsigma_1, \ldots \varsigma_n\}) \rangle}$$

$$(\textbf{class})\quad \frac{\mathit{obt} = \mathit{clsn} \ \{ \textbf{ inst } \overline{\mathit{instn}_i : \tau_i} \textbf{ meth } \overline{\mathit{opn}_j \ \tau_j \rightarrow \tau'_j}\}, \quad S, \mathbf{A}, \alpha, \Gamma \cup \{x : \tau_j, \textbf{this} : \mathit{obt}\} \vdash e : \tau'_j \text{ for each } j}{\begin{array}{l}\mathbf{S}, \mathbf{A}, \alpha, \Gamma \vdash_c \textbf{class } \mathit{clsn} \ \{ \textbf{ inst } \overline{\mathit{instn}_i : \tau_i} \textbf{ meth } \overline{\mathit{opn}_j(x : \ \tau_j) : \tau'_j \ \{e\}}\} : \\ \qquad \mathit{clsn} \ \{ \textbf{ inst } \overline{\mathit{instn}_i : \tau_i} \textbf{ meth } \overline{\mathit{opn}_j \ \tau_j \rightarrow \tau'_j}\}\end{array}}$$

Figure 7: Top-level Typing Rules

the compound assembly can have overlapped services, connectors and class signatures, unless they ultimately come from the same atomic assembly.

Another important constraint to satisfy is *LinkTypeMatched*, which makes sure parties matched by name during linking have compatible types, in particular export types must be subtypes of import types. Given an assembly α composed of $\alpha_1, \ldots \alpha_n$ and $\langle asm_i, \sigma_i, \varsigma_i \rangle = \mathbf{A}(\alpha_i)$ for $i \in [1..n]$, the constraint is defined as:

$$\begin{array}{l} \mathit{LinkTypeMatched}(\alpha_1, \ldots \alpha_n) \ \stackrel{\text{def}}{=} \\ \quad \mathit{ln} \in^{\mathtt{t}}_{\mathtt{ilnk}} \sigma_i, \mathit{ln} \in^{\mathtt{t}}_{\mathtt{elnk}} \sigma_j, i, j \in [1..n], i \neq j \\ \quad \{\sigma_{ig}\} = \mathit{Orig}(\{\alpha_1, \ldots \alpha_n\}, \mathit{ln}) \\ \quad \{\sigma_{jg}\} = \mathit{Orig}(\{\alpha_1, \ldots \alpha_n\}, \mathit{ln}) \\ \quad \langle \mathit{clsn}, \sigma \rangle \in^{\mathtt{t}}_{\mathtt{clsimp}} \langle \mathit{ln}, \sigma_{ig} \rangle, \quad \langle \mathit{clsn}, \sigma' \rangle \in^{\mathtt{t}}_{\mathtt{clsexp}} \langle \mathit{ln}, \sigma_{jg} \rangle \Rightarrow \\ \qquad \mathbf{S}, \phi \vdash_{sub} \langle \mathit{clsn}, \sigma' \rangle <: \langle \mathit{clsn}, \sigma \rangle \end{array}$$

Note that after closure conversion, every type declared in a linker is also assoicated with an ASID, which represents its type information. This piece of type information is used to realize atomic assembly separate compilation, and the goal of *LinkTypeMatched* is to preserve type soundness: the type provided for separate compilation really matches the type the import/export entity is expected to have during linking. Also note that a match in our case does not require two types are exactly the same; we only need the export to have a type which is a subtype of the corresponding import.

Other constraints include: *CompImpValid* asserts every importer of the subpart of the compound is either matched with an exporter from another subpart or redirected to be the importer of the compound; and, *CompExpValid* asserts each exporter of a compound assembly needs to be an exporter of one of its subparts. Their definitions are:

$$\begin{array}{l} \mathit{CompImpValid}(\alpha) \ \stackrel{\text{def}}{=} \\ \quad \mathit{ln} \in^{\mathtt{t}}_{\mathtt{ilnk}} \sigma_i, i, j \in [1..n], i \neq j \Rightarrow \mathit{ln} \in^{\mathtt{t}}_{\mathtt{elnk}} \sigma_j \text{ or } \mathit{ln} \in^{\mathtt{t}}_{\mathtt{ilnk}} \sigma \end{array}$$

$$\mathit{CompExpValid}(\alpha) \ \stackrel{\text{def}}{=} \ \mathit{ln} \in^{\mathtt{t}}_{\mathtt{elnk}} \sigma, i \in [1..n] \Rightarrow \mathit{ln} \in^{\mathtt{t}}_{\mathtt{elnk}} \sigma_i$$

Expression Type Rules

Fig. 8 gives the type rules for expressions. Assertion $\mathbf{S}, \mathbf{A}, \alpha, \Gamma \vdash e : \tau$ means, given $\mathbf{S}, \mathbf{A}$ and local type environment Γ, ex-

$$\textbf{(int const)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash c \text{ is an integer}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash c : \mathbf{int}}$$

$$\textbf{(bool const)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash c \text{ is an boolean}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash c : \mathbf{bool}}$$

$$\textbf{(variable)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \Gamma(x) = \tau}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash x : \tau}$$

$$\textbf{(new object)}\ \frac{}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathbf{new}\ \langle \mathit{clsn}, \sigma\rangle : \langle \mathit{clsn}, \sigma\rangle}$$

$$\textbf{(get obj instn)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e : \mathit{obt}, \quad (\mathit{instn} : \tau) \in^{\mathtt{t}}_{\mathtt{objinst}} \mathit{obt}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e\mathbf{.}\mathit{instn} : \tau}$$

$$\textbf{(set obj instn)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_1 : \mathit{obt}, \quad \mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_2 : \tau, \quad (\mathit{instn} : \tau) \in^{\mathtt{t}}_{\mathtt{objinst}} \mathit{obt}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_1\mathbf{.}\mathit{instn} := e_2 : \tau}$$

$$\textbf{(invoke obj op)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_1 : \mathit{obt}, \quad (\mathit{opn} : \tau' \to \tau) \in^{\mathtt{t}}_{\mathtt{objop}} \mathit{obt}, \quad \mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_2 : \tau'}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_1\mathbf{.}\mathit{opn}(e_2) : \tau}$$

$$\textbf{(get cell instn)}\ \frac{\{\mathbf{thiscell} : \mathit{cellt}\} \in \Gamma, \quad (\mathit{instn} : \tau)\ \in^{\mathtt{t}}_{\mathtt{cellinst}}\ \mathit{cellt}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathit{instn} : \tau}$$

$$\textbf{(set cell instn)}\ \frac{\{\mathbf{thiscell} : \mathit{cellt}\} \in \Gamma, \quad \mathbf{S},\mathbf{A},\alpha \vdash e : \tau, \quad (\mathit{instn} : \tau) \in^{\mathtt{t}}_{\mathtt{cellinst}} \mathit{cellt}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathit{instn} := e : \tau}$$

$$\textbf{(invoke cnt op)}\ \frac{\begin{array}{l}\{\mathbf{thiscell} : \mathit{cellt}\} \in \Gamma, \quad \mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e : \tau' \\ (\mathit{opn} : \tau' \to \tau) \in^{\mathtt{t}}_{\mathtt{plugin}} \langle \mathit{cntn}, \mathit{cellt}\rangle \text{ or } (\mathit{opn} : \tau' \to \tau) \in^{\mathtt{t}}_{\mathtt{plugout}} \langle \mathit{cntn}, \mathit{cellt}\rangle\end{array}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathit{cntn}\text{->}\,\mathit{opn}(e) : \tau}$$

$$\textbf{(invoke srv op)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_1 : \mathit{cellt}, \quad \mathit{srvn} \in^{\mathtt{t}}_{\mathtt{srv}} \mathit{cellt}, \quad (\mathit{opn} : \tau' \to \tau) \in^{\mathtt{t}}_{\mathtt{srvop}} \langle \mathit{srvn}, \mathit{cellt}\rangle, \quad \mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_2 : \tau'}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_1\mathbf{..}\mathit{srvn}\text{->}\,\mathit{opn}(e_2) : \tau}$$

$$\textbf{(connect)}\ \frac{\begin{array}{l}\{\mathbf{thiscell} : \mathit{cellt}_1\} \in \Gamma, \quad \mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e : \mathit{cellt}_2, \quad \mathit{cntn} \in^{\mathtt{t}}_{\mathtt{cnt}} \mathit{cellt} \\ \mathbf{S},\mathbf{A},\alpha,\Gamma \vdash (\mathit{opsig} \in^{\mathtt{t}}_{\mathtt{plugout}} \langle \mathit{cntn}, \mathit{cellt}_1\rangle) \Rightarrow\ (\mathit{opsig} \in^{\mathtt{t}}_{\mathtt{plugin}} \langle \mathit{cntn}, \mathit{cellt}_2\rangle) \text{ for each } \mathit{opsig} \\ \mathbf{S},\mathbf{A},\alpha,\Gamma \vdash (\mathit{opsig} \in^{\mathtt{t}}_{\mathtt{plugin}} \langle \mathit{cntn}, \mathit{cellt}_1\rangle) \Rightarrow\ (\mathit{opsig} \in^{\mathtt{t}}_{\mathtt{plugout}} \langle \mathit{cntn}, \mathit{cellt}_2\rangle) \text{ for each } \mathit{opsig}\end{array}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathbf{connect}\ e\ \mathbf{at}\ \mathit{cntn} : \mathit{cellt}_1}$$

$$\textbf{(disconnect)}\ \frac{\{\mathbf{thiscell} : \mathit{cellt}\} \in \Gamma, \quad \mathit{cntn} \in^{\mathtt{t}}_{\mathtt{cnt}} \mathit{cellt}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathbf{disconnect\ at}\ \mathit{cntn} : \mathit{cellt}}$$

$$\textbf{(dynlink)}\ \frac{\begin{array}{l}\mathit{NoImpConflict}(\alpha, \alpha_1, \ldots \alpha_n), \quad \mathit{NoExpConflict}(\alpha, \alpha_1, \ldots \alpha_n), \quad \mathit{LinkTypeMatched}(\alpha, \alpha_1, \ldots \alpha_n) \\ \mathit{NoClsConflict}(\alpha, \alpha_1, \ldots \alpha_n), \quad \mathit{NoSrv}(\alpha_i), \quad \mathit{NoCnt}(\alpha_i) \text{ for each } i \in [1..n], \quad \langle \mathit{asm}, \sigma, \varsigma\rangle = \mathbf{A}(\alpha)\end{array}}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathbf{dynlink}\ \alpha_1, \ldots \alpha_n : \mathbf{cellsig}\ \varsigma}$$

$$\textbf{(let)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e_1 : \tau', \quad \mathbf{S},\mathbf{A},\alpha,\Gamma \cup \{x : \tau'\} \vdash e_2 : \tau}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\textbf{(load)}\ \frac{\langle \mathit{asm}', \sigma', \varsigma'\rangle = \mathbf{A}(\alpha')}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash \mathbf{load}\ \alpha' : \mathbf{cellsig}\ \varsigma'}$$

$$\textbf{(sub)}\ \frac{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e : \tau, \quad \mathbf{S},\{\} \vdash_{sub} \tau <: \tau'}{\mathbf{S},\mathbf{A},\alpha,\Gamma \vdash e : \tau'}$$

Figure 8: Expression Type Rules

pression e found in assembly α has type τ. Of these rules, (**connect**) checks to make sure *thiscell* has matched plugins and plugouts with its communicating counterpart. (**dynlink**) resembles the static compound checking rule, where conflicts among imports, exports and class signatures are checked, the type matching is also ensured. Unlike static compound checking, we disallow the dynamically linked assemblies have services and connectors, which can be represented by two simple constraints:

$$\begin{aligned} \mathit{NoSrv}(\alpha) &\overset{\text{def}}{=} \mathit{srvn} \notin^{\mathtt{t}}_{\mathtt{srv}} \varsigma \text{ for any } \mathit{srvn} \\ \mathit{NoCnt}(\alpha) &\overset{\text{def}}{=} \mathit{cntn} \notin^{\mathtt{t}}_{\mathtt{cnt}} \varsigma \text{ for any } \mathit{cntn} \end{aligned}$$

From inspection of the rules it can be seen that atomic assemblies may be typechecked (and thus compiled) seperately: the general **A** is only used in the (**dynlink**) rule, and all constraints associated with **dynlink** are functions solely depending on the universe of assembly signatures, not the universe of assemblies.

Subtype and Type Equivalence Rules

The subtyping rules are define in Fig. 9. We allow subtyping on cell types, full object types and object closure types. Subtyping for cell types are purely extensional, while objects are subtypes if they are structurally subtypes and share the same name.

Object and cell types may be recursive or mutually recursive. Our subtyping system can handle recursive types correctly in the standard style [3]; C is the set of subtyping assumptions.

We have two rules to handle cell type subtyping. The complexity arises partly because the type of **thiscell** is handled uniquely in our type system, which allows internal instances to be part of the type.

6. CONCLUSIONS

The main contribution to this paper is the development of a component programming langauge which has integrated notions of both run-time (cell) and compile-time (assembly) component. In this regard we know of no other peer research. There is a substantial body of research in module systems, which we now briefly compare assemblies to.

6.1 Related Module Systems

Our notion of module and linking starts from the classical one found in languages such as Modula-2, where there is a notion of module linking defined, and imports and exports are matched. Cardelli [5] first formalized this linking process. Units and Jiazzi [6, 11] extend this core to include other features. The module aspects of assemblies are most closely related to these latter systems. Abstractions for components have been studied in the framework of a formal calculus in [16]. Standard ML modules [9] take a different approach, modelling linking as function (functor) application. The topic of contrasting ML-style modules with the more standard style is a fairly involved topic and enough of a tangent that we will not pursue it here. It should be said that ML modules, in the form of structures, do have an explicit run-time presence, but it is relatively weak compared to a cell.

So, we will focus our comparisons on classic module systems and Units/Jiazzi. Many of our design constraints are ones found in classic module systems. We guarantee one assembly need not be recompiled if an assembly it is linked with, but not its signature, changes. We support recursive references between assemblies. Like Units and Jiazzi, we support a compositional notion of linking, *i.e.* linking is a compositional operator and partial linking is possible, producing an assembly which can be further linked. One of the main differences between assemblies and these existing systems is our use of explicit interfaces for dynamic linking: the same assembly linkers may be used for either static or dynamic links. Units do support dynamic linking through an interface, but the running code does not itself present an interface to the dynamically-linked code, only vice-versa. Our dynamic link is more symmetric: a running program must still present a linker to which the new assembly is dynamically linked.

We aim to more directly model the software lifecycle in the language, further "typechecking" programmers into good practice: *AID*, *ASID*, and *CSID* identifiers give universal names to immutable code objects.

6.2 Future Topics

There are several features we have left out of this presentation to keep the details down. There is no class inheritance, and no notion of module-level field/method hiding in classes, analogous to the package-protected fields and methods of Java. Currently there is no general type parametricity; classes and their types can be imported, but from a fixed source only. We expect it will not be hard to add parametric class imports to assembly linkers, and hope to be able to squeeze them into the final version.

There are several other topics specific to our language design which we plan on tackling. Our signatures are designed to be inferred, generally preventing duplication of information in the assembly and signature definitions. But, we present no inference algorithm here. We plan on focusing more explicitly on the compilation process: build scripts (makefiles) should be more tightly integrated into the language, and we plan on incorporating build targets explicitly into assembly definitions: assemblies should know everything it takes to get themselves up and running.

7. REFERENCES

[1] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *ECOOP*, 2003. to appear.

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *ICSE 2002*, 2002.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[4] Luc Bellissard, Slim Ben Atallah, Fabienne Boyer, and Michel Riveill. Distributed application configuration. In *International Conference on Distributed Computing Systems*, pages 579–585, 1996.

$$\textbf{(id)}\ \frac{}{\mathbf{S}, C \vdash_{sub} \tau \equiv \tau}$$

$$\textbf{(eq − sub)}\frac{\mathbf{S}, C \vdash_{sub} \tau_a <: \tau_b, \quad \tau_b <: \tau_a}{\mathbf{S}, C \vdash_{sub} \tau_a \equiv \tau_b} \qquad \textbf{(sub − eq)}\frac{\mathbf{S}, C \vdash_{sub} \tau_a \equiv \tau_b}{\mathbf{S}, C \vdash_{sub} \tau_a <: \tau_b, \quad \tau_b <: \tau_a}$$

$$\textbf{(trans)}\frac{\mathbf{S}, C \vdash_{sub} \tau_a <: \tau_b, \quad \tau_b <: \tau_c}{\mathbf{S}, C \vdash_{sub} \tau_a <: \tau_c} \qquad \textbf{(constraint)}\frac{C \Rightarrow \tau_a <: \tau_b}{\mathbf{S}, C \vdash_{sub} \tau_a <: \tau_b}$$

$$\textbf{(obj eq)}\frac{\mathbf{S} = \mathbf{AS} \cup \mathbf{CS}}{\mathbf{S}, C \vdash_{sub}\ \langle \mathit{clsn}, \sigma\rangle \equiv \mathit{ObjectT}(\mathbf{AS}(\sigma), \mathit{clsn})} \qquad \textbf{(cellsig eq)}\frac{\mathbf{S} = \mathbf{AS} \cup \mathbf{CS}}{\mathbf{S}, C \vdash_{sub}\ \textbf{cellsig}\ \varsigma \equiv \mathit{CellT}(\mathbf{CS}(\varsigma))}$$

$$\textbf{(object)}\ \frac{\begin{array}{l}\mathbf{S}, C \vdash_{sub} \mathit{clsn_1}\ \{\ \textbf{inst}\ \overline{\mathit{instn}_i : \tau_i}\ \textbf{meth}\ \overline{\mathit{opn}_j : \tau_j \to \tau'_j}\} \equiv \langle \mathit{clsn_1}, \sigma_1\rangle \\ \mathbf{S}, C \vdash_{sub} \mathit{clsn_2}\ \{\ \textbf{inst}\ \overline{\mathit{instn}_m : \tau_m}\ \textbf{meth}\ \overline{\mathit{opn}_n : \tau_n \to \tau'_n}\} \equiv \langle \mathit{clsn_2}, \sigma_2\rangle \\ \mathit{clsn_1} = \mathit{clsn_2}, \quad C' = C \cup \{\langle \mathit{clsn_1}, \sigma_1\rangle <: \langle \mathit{clsn_2}, \sigma_2\rangle\} \\ (\mathit{instn}_i = \mathit{instn}_m, \quad \mathbf{S}, C' \vdash_{sub} \tau_i \equiv \tau_m) \text{ for each } m \text{ some } i \\ (\mathit{opn}_j = \mathit{opn}_n, \quad \mathbf{S}, C' \vdash_{sub} \tau_j <: \tau_n, \quad \tau'_n <: \tau'_j) \text{ for each } n \text{ some } j\end{array}}{\mathbf{S}, C \vdash_{sub} \langle \mathit{clsn_1}, \sigma_1\rangle <: \langle \mathit{clsn_2}, \sigma_2\rangle}$$

$$\textbf{(cell 1)}\ \frac{\mathbf{S} = \mathbf{AS} \cup \mathbf{CS}, \quad C' = C \cup \{\textbf{cellsig}\ \varsigma_1 <: \textbf{cellsig}\ \varsigma_2\}, \quad \mathbf{S}, C' \vdash_{sub} \mathit{CellT}(\mathbf{CS}(\varsigma_1)) <: \mathit{CellT}(\mathbf{CS}(\varsigma_2))}{\mathbf{S}, C \vdash_{sub} \textbf{cellsig}\ \varsigma_1 <: \textbf{cellsig}\ \varsigma_2}$$

$$\textbf{(cell 2)}\ \frac{\begin{array}{l}(\mathit{srvn}_s = \mathit{srvn}_r,\ (\mathit{opn}_{s_w} = \mathit{opn}_{r_x},\ \mathbf{S}, C \vdash_{sub} \tau_{s_w} <: \tau_{r_x}, \tau'_{r_x} <: \tau'_{s_w}) \text{ for each } r_x \text{ some } s_w) \text{ for each } r \text{ some } s \\ (\mathit{cntn}_c = \mathit{cntn}_e,\ (\mathit{opn}_{c_u} = \mathit{opn}_{e_y},\ \mathbf{S}, C \vdash_{sub} \tau_{c_u} <: \tau_{e_y}, \tau'_{e_y} <: \tau'_{c_u}) \text{ for each } e_y \text{ some } c_u, \\ \qquad\qquad (\mathit{opn}_{c_v} = \mathit{opn}_{e_z},\ \mathbf{S}, C \vdash_{sub} \tau_{c_v} <: \tau_{e_z}, \tau'_{e_z} <: \tau'_{c_v}) \text{ for each } e_z \text{ some } c_v) \text{ for each } e \text{ some } c\end{array}}{\begin{array}{l}\mathbf{S}, C \vdash_{sub} \textbf{cellsig}\ \{\overline{\textbf{service}\ \mathit{srvn}_s\ \{\overline{\mathit{opn}_{s_w} : \tau_{s_w} \to \tau'_{s_w}}\}} \\ \qquad\qquad \overline{\textbf{connector}\ \mathit{cntn}_c\ \textbf{plugin}\ \overline{\mathit{opn}_{c_u} : \tau_{c_u} \to \tau'_{c_u}}\ \textbf{plugout}\ \overline{\mathit{opn}_{c_v} : \tau_{c_v} \to \tau'_{c_v}}}\ \overline{\mathit{instn}_i : \tau_i}\ \} <: \\ \qquad \textbf{cellsig}\ \{\overline{\textbf{service}\ \mathit{srvn}_r\ \{\overline{\mathit{opn}_{r_x} : \tau_{r_x} \to \tau'_{r_x}}\}} \\ \qquad\qquad \overline{\textbf{connector}\ \mathit{cntn}_e\ \textbf{plugin}\ \overline{\mathit{opn}_{e_y} : \tau_{e_y} \to \tau'_{e_y}}\ \textbf{plugout}\ \overline{\mathit{opn}_{e_z} : \tau_{e_z} \to \tau'_{e_z}}}\ \overline{\mathit{instn}_j : \tau_j}\ \}\end{array}}$$

Figure 9: Subtype and Type Equivalence Rules

[5] 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.

[6] 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.

[7] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, revision 2.5 edition, September 2001.

[8] Xiaoqi Lu. Report on the cell prototype project. JHU PhD qualifier Project Report, January 2003.

[9] David MacQueen. An implementation of standard ml modules. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, 1988.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin. `citeseer.nj.nec.com/magee95specifying.html`.

[11] Sean McDirmid, Matthew Flatt, and Wilson Hsieh. Jiazzi: New age components for old fashioned java. In *OOPSLA 2001*, 2001.

[12] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[13] Mark Miller. The E programming language. `http://www.erights.org`.

[14] Ran Rinat and Scott Smith. The cell project: Component technology for the internet. In *First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, October 2001.

[15] Ran Rinat and Scott Smith. Modular internet programming with cells. In *ECOOP 2002*, Lecture Notes in Computer Science. Springer Verlag, 2002. `http://www.cs.jhu.edu/hog/cells`.

[16] Joao Costa Seco and Luis Caires. A basic model of typed components. In *ECOOP*, pages 108–128, 2000.

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

[18] D. Ungar and R. Smith. Self: the power of simplicity. In *OOPSLA 1987*, pages 227–241, 1987.

(these definitions are implicitly parameterized by **AS** and **A**)

$\mathit{ln} \in^{\mathtt{s}}_{\mathtt{ilnk}} \alpha$	Linker with name *ln* is declared as an importer in assembly α
$\mathit{ln} \in^{\mathtt{s}}_{\mathtt{elnk}} \alpha$	Linker with name *ln* is declared as an exporter in assembly α
$\mathit{clsn} \in^{\mathtt{s}}_{\mathtt{clsilnkr}} \langle \mathit{ln}, \alpha \rangle$	Class *clsn* is in importer *ln* of assembly α
$\mathit{cntn} \in^{\mathtt{s}}_{\mathtt{cnt}} \alpha$	Connector with a name *cntn* is declared in assembly α
$\mathit{srvn} \in^{\mathtt{s}}_{\mathtt{srv}} \alpha$	Service with a name *srvn* is declared in assembly α
$\mathit{opn} \in^{\mathtt{s}}_{\mathtt{srvop}} \langle \mathit{srvn}, \alpha \rangle$	Operation with name *opn* is declared in a service *srvn* of assembly α
$\mathit{opn} \in^{\mathtt{s}}_{\mathtt{plugin}} \langle \mathit{cntn}, \alpha \rangle$	Operation with name *opn* is declared as a plugin in connector *cntn* of assembly α
$\mathit{opn} \in^{\mathtt{s}}_{\mathtt{plugout}} \langle \mathit{cntn}, \alpha \rangle$	Operation with name *opn* is declared as a plugout in connector *cntn* of assembly α
$(\mathit{instn}) \in^{\mathtt{s}}_{\mathtt{cellinst}} \alpha$	Instance variable *instn* is declared as a cell instance in assembly α
$(\mathit{instn}) \in^{\mathtt{s}}_{\mathtt{clsinst}} \langle \mathit{clsn}, \alpha \rangle$	Instance variable *instn* is declared in class *clsn* of assembly α
$\mathit{clsn} \in^{\mathtt{s}}_{\mathtt{needed}} \alpha$	Class *clsn* is used as an object type or appears in **new** expression
$\mathit{clsn} \in^{\mathtt{s}}_{\mathtt{clsdf}} \alpha$	Class with name *clsn* is defined in assembly α
$\alpha_1 \in^{\mathtt{s}}_{\mathtt{compose}} \alpha_2$	Assembly α_1 is a subpart of α_2

$\mathit{ln} \in^{\mathtt{t}}_{\mathtt{ilnk}} \sigma$	Linker with name *ln* is declared as an importer in $\mathbf{AS}(\sigma)$
$\mathit{ln} \in^{\mathtt{t}}_{\mathtt{elnk}} \sigma$	Linker with name *ln* is declared as an exporter in $\mathbf{AS}(\sigma)$
$\mathit{ln} \in^{\mathtt{t}}_{\mathtt{lnk}} \sigma$	$\mathit{ln} \in^{\mathtt{s}}_{\mathtt{ilnk}} \sigma$ or $\mathit{ln} \in^{\mathtt{s}}_{\mathtt{elnk}} \sigma$
$\langle \mathit{clsn}, \sigma' \rangle \in^{\mathtt{t}}_{\mathtt{clsimp}} \langle \mathit{ln}, \sigma \rangle$	Class *clsn* with sig declared in σ' is in importer *ln* of $\mathbf{AS}(\sigma)$
$\langle \mathit{clsn}, \sigma' \rangle \in^{\mathtt{t}}_{\mathtt{clsexp}} \langle \mathit{ln}, \sigma \rangle$	Class *clsn* with sig declared in σ' is in exporter *ln* of $\mathbf{AS}(\sigma)$
$\mathit{clsn} \in^{\mathtt{t}}_{\mathtt{clsig}} \sigma$	Class sig with name *clsn* is declared in assembly sig σ
$\mathit{cntn} \in^{\mathtt{t}}_{\mathtt{cnt}} \mathit{cellt}$	Declaration of connector *cntn* appears in cell sig *cellt*
$\mathit{srvn} \in^{\mathtt{t}}_{\mathtt{srv}} \mathit{cellt}$	Declaration of service *srvn* appears in cell sig *cellt*
$\mathit{opsig} \in^{\mathtt{t}}_{\mathtt{objop}} \mathit{obt}$	Operation sig *opsig* is declared in object sig *obt*
$\mathit{opsig} \in^{\mathtt{t}}_{\mathtt{plugin}} \langle \mathit{cntn}, \mathit{cellt} \rangle$	Operation sig *opsig* is declared as a plugin in declaration of connector *cntn* of cell sig *cellt*
$\mathit{opsig} \in^{\mathtt{t}}_{\mathtt{plugout}} \langle \mathit{cntn}, \mathit{cellt} \rangle$	Operation sig *opsig* is declared as a plugout in declaration of connector *cntn* of cell sig *cellt*
$\mathit{opsig} \in^{\mathtt{t}}_{\mathtt{srvop}} \langle \mathit{srvn}, \mathit{cellt} \rangle$	Operation sig *opsig* is declared in declaration of service *srvn* of cell sig *cellt*
$(\mathit{instn} : \tau) \in^{\mathtt{t}}_{\mathtt{objinst}} \mathit{obt}$	Instance variable *instn* with a type τ is declared as an object instance in object sig *obt*
$(\mathit{instn} : \tau) \in^{\mathtt{t}}_{\mathtt{cellinst}} \mathit{cellt}$	Instance variable *instn* with a type τ is declared as a cell instance in cell sig *cellt.*

Figure 10: Simple Containment Relations