Bootstrap + Dynamic Libraries

Creating minimal images from scratch
Pre compiled libraries of Smalltalk code

Juan Vuletich, February 2024

Hello all. My name is Juan Vuletich. I am the leader of the Cuis Smalltalk project, and I will tell you today about Cuis Bootstrap and Cuis Dynamic Libraries.
I want to start by thanking you all for being here today, and the UK Smalltalk Users Group for hosting us.
This presentation today is about a major project in an ongoing effort to make Cuis meet the needs of commercial projects, like those developed at LabWare. So, I want to thank LabWare for sponsoring this project.
This presentation includes some material that was shown by Felipe Zak and me at the Smalltalks conference in Quilmes, Argentina in November 2023.

Objectives

Build Small Images from Scratch

- Not tied to a specific VM or image format
- Builder code is separated and independent of code in the new image
- Code to be included is explicitly specified
- Code to be included is not required to be a subset of an existing image

Pre-compiled Libraries of Smalltalk Code

- Class shapes in libraries are independent of the image that loads them
- They are also independent of other libraries loaded before or after this one
- Loading libraries require very little from the image loading them
- They offer quick load times
- No Compiler required to install, distribution of library source code is not required

These are some of the objectives we set for this project.

We want to build Smalltalk images from scratch.
We want to start with an explicit list of what should be included in the new image, ideally a set of source code files. The code in the image being built needs to be independent of the code in the builder image. There is no “mother" image or such. And the builder must be easy to extend to new image formats.

With respect to libraries, we want binary code libraries that don’t require source code and won’t use a compiler. We also want to be able to load libraries that define a possibly different class shape for classes that already exist in the loading image, or in another library that was previously loaded.
Library loading should almost never fail. And we want to be able to load tons of libraries quickly.

A problem. In the Builder Image:
(Unless we add serious extra complexity)

- Classes are deeply integrated into the system
- There can only be one class (and class shape) for each class name
- CompiledMethods refer to globals in this image
- There can only be one method for each class name + selector
- Existing instances of Hashed Collections are built for the currently assigned #identityHash value and bit size
- Existing instances of any collection follow the class shape and ivar contents that make sense in the Builder image

We want our new image and compiled libraries to include Classes and Methods. And also some Collections. If we simply load them into the Builder image, there are several possible problems.

- Our classes and methods may clash with stuff already in the image, having possible different class shapes and method code.
- All references are valid in this image, and most likely not in the new image.
- Class shapes for basic objects such as MethodDictionaries, SystemDictionary or even String could be different.
- The values and even bit size of #identityHash could be very different.

For this reasons, including in the builder image the code and state that is intended for the new image, could mean having to carefully untangle and transform these objects as they are serialized into a new image. Similar problems happen with libraries. Code performing this tasks is deeply involved both with the current and the new image format, shapes of basic classes, and details of hashing. So, this code tends to be complex and hard to evolve. So we did something different.

Strategy
In the Builder Image

- Don’t include target image classes
- Don’t include target image methods
- Instead build a separate model for them:
    - ForeignObject
    - ClassSpec
    - MethodSpec

The strategy chosen was, instead of loading the code in the Builder Image, build a separate Smalltalk model for it. These ClassSpecs are not classes. They can not create instances. These MethodSpecs are not CompiledMethods. They can’t be run. But they can be materialized into actual classes and methods. For other objects to be included in the new image, instead of, for example, creating a new SystemDictionary or many MethodDictionaries, we create instances of a ForeignObject hierarchy. These, together with an appropriate ImageBuilder class, that is specific to one Image Format, can be serialized to save a new image file.
These ClassSpecs and MethodSpecs are also materialized when loading libraries.

Design of Bootstrap 1

Model for all the objects to be included in the new image

Build an object graph, having one ForeignObject for each object to be included in the new image

- Kinds and formats of objects not limited to those in builder image
- Not limited to a pure bootstrap from sources
- Can also be used to build content-rich images

In order to build a Smalltalk model of everything that will be part of our new image, we need to describe the graph of objects it will include. For each of these objects we need to store some information that is usually hidden by the VM in a running system, but that is essential if we are to save a new Smalltalk image that can later run.

For this, we create a ForeignObject for each object that will be created in the new image, and an object graph is built with them.

There are many kinds of ForeignObjects. This reflects the fact that some objects have instance variables, while others have indexed slots, like Arrays. Still other objects are made of bytes or other numeric values. And yet others have their value encoded in pointers to it, like SmallIntegers, and they hold no state at all.

Design of Bootstrap 2

ForeignObjects for regular objects

```
ForeignObject (representedObject nameOfClassToInstantiate)
        AllocatedForeignObject (oop identityHash nextInList)
                IvarsForeignObject "For regular objects"
                IndexedOopsForeignObject "For instances, Arrays"
                Indexed8BitForeignObject "For ByteStrings and ByteArrays"
                BoxedFloat64ForeignObject
                LargeIntegerForeignObject
        ImmediateForeignObject
                SmallIntegerForeignObject
                SmallFloat64ForeignObject
                ImmediateCharacterForeignObject
```

These are the ForeignObject classes used to model regular Smalltalk objects. Each of them knows how to create a real object of its kind when serializing the object file. As you can see, AllocatedObjects hold the pointer in memory where their real object will reside, and the identityHash it will use. ImmediateForeignObject has neither, because immediate objects don't use memory as they are encoded in the very pointers referencing them.

This means that creating these ForeignObjects and creating the object graph is also allocating memory in the new image. The moment that memory is allocated and pointers are assigned to, is when we build this graph.

Also, these ForeignObjects form a linked list, in the order as they were allocated, that is also the order in which they will be stored in the memory of the new image, and in the image file.

Design of Bootstrap 3

ForeignObjects for Smalltalk Code

ForeignObject (representedObject nameOfClassToInstantiate)

AllocatedForeignObject (oop identityHash nextInList)

ClassForeignObject (nameForeignObject metaclassForeignObject methodDictForeignObject instanceVariablesForeignObject subclassesForeignObject ...)

CompiledMethodForeignObject (methodClassForeignObject)

MetaclassForeignObject (thisClassForeignObject methodDictForeignObject instanceVariablesForeignObject)

MethodDictionaryForeignObject

MethodContextForeignObject

These are the ForeignObjects used to create Smalltalk code entities in the new image. The represented object (that is held in the ‘representedObject’ instance variable) will not be a class or a method, but instances of new classes called ClassSpec and MethodSpec. Still, these ForeignObjects will create the real classes and methods in the new image.

The most important ones are those for Classes, Metaclasses and CompiledMethods. Also we build ForeignObjects for MethodDictionaries. These ForeignObjects form the metamodel of the new Smalltalk image. The object graphs they form is not unlike that of existing Classes, Metaclasses, MethodDictionaries and CompiledMethods. But they form a completely separated object graph.

Besides as they will be used for serializing them into a new image file, and not for running them in the current image and VM, the services they provide are completely different. These are not classes and methods!

Design of Bootstrap 4

Model for Smalltalk Code elements

Smalltalk code model: ClassSpec and MethodSpec

- not Class/Metaclass and CompiledMethod!
- Independent of existing classes in the host system
- Also used for Dynamic Libraries

We want to build a new Smalltalk image that includes essentially Smalltalk code. We could think that our existing Smalltalk image already includes a model for Smalltalk code, right? That is true, but we don't need or want to be able to run this new code in our old image. Bootstrap includes a new object model for Smalltalk code that is completely separated and independent from the existing classes and methods of the image that runs the Bootstrap tool. The model for the code in the new image is not a set of Classes and CompiledMethods, but a set of ClassSpecs and MethodSpecs. These include all the required information to be later materialized as Classes and Methods into a running system, but they are not part of the host image metamodel.

This means that the classes and methods that will be included in the new Smalltalk image we are building don't need to resemble those in our existing system. We could even model new kinds of classes, not only those supported by our host VM. For instance, it is trivial to model immediate Floats or Characters in a system that doesn't support them. This feature could have made the migration of Squeak and Cuis, to Closures and Spur much easier and less hacky in the past, if we had something like this.

Additionally, references from these objects are controlled carefully, so there's no risk of unwanted objects leaking into the new image.

Design of Bootstrap 5

Hierarchy of ImageBuilders

- ImageBuilder
  - SpurImageBuilder
    - Spur32ImageBuilder
    - Spur64ImageBuilder
  - V3ImageBuilder (deprecated when Cuis abandoned the V3 format)

In order to turn this object graph into a new SmalltalkImage comprised of the real objects we want, we need an ImageBuilder. The main purpose of an ImageBuilder is to traverse the collection of ForeignObjects, and for each of them write the bits for the object header, and then write the outward pointers or literal bytes for the object contents.

Format specific ImageBuilders also know how to build format specific data structures, the image file header and other details.

Currently we have ImageBuilders for 32 bit and 64 bit Spur image formates.

Some Criteria

For images Purely Bootstrap from Sources

If we want complete freedom on the contents and language semantics for our new image, we

- Can't run new code in the builder image
- Can't create object state for the new image in the builder image

The new images could be very different from our existing image. Not only it will include a different set of classes and methods, but it could even run on a VM with different execution semantics. This means we can not run any code that belongs in the new image until it is built, saved to disk, and run by itself on its appropriate VM.

This means that we can not run any of that code in our tool. We may not even have execution semantics for doing so.

It also means that we can't create arbitrary objects for the new image in our build tool, because the build tool runs in the "old" image, and doesn’t include those classes. Only objects whose shape and semantics is known to the ImageBuilder in use can be created in the Builder image.

Bootstrap from Sources

Currently only creates Code Entities

- Classes and Metaclasses
- CompiledMethods
- MethodDictionaries
- SystemDictionary
- Basic Literals

These are the only objects that a bootstrapped image will usually include, except for a few special objects known to the ImageBuilder, like Smalltalk, SymbolTable, Processor, etc.

Because of all this, by default, ImageBuilder tool will only create a few specific kinds of objects. These are objects for code: Classes, Metaclasses, CompiledMethods, MethodDictionaries, SystemDictionary and basic Literals.

In short, our bootstrapped images will only include Smalltalk code.

The only other objects we are creating are basic literals, and a few special objects, like the Smalltalk SystemDictionary, a SymbolTable, the Processor, and very few others, that are required by the VM to run, and that the ImageBuilder knows how to build.

Creation of Additional Globals

And Class Variable Initialization

- Are created by code that exists only in the new image
- That runs in the new image
- Only when the new image runs on its own

Any other object, for example, additional globals and class variables, must be created in the new image, by code included as part of the new image, and only when the new image is run by itself.

This may seem a bit restrictive, but it is needed if we want the new image to be fully specified by source code.

In any case, the ImageBuilders can be extended with new specially prebuilt objects if really needed.

Design of Dynamic Libraries 1

Model for all the objects to be loaded in the runtime image

A Dynamic Library is just a collection of

- ClassSpecs
  - They specify the type and shape of the class
  - Included ClassSpecs point to the parent ClassSpec
  - For each ClassSpec, include all the superclasses up to the hierarchy root
- MethodSpecs
  - They include method bytecode
  - For every MethodSpec, the defining ClassSpec must also be included

Both ClassSpecs and MethodSpecs are specific to one image format

What about Libraries?

Dynamic Libraries reuse much of the model built for Bootstrap.

A Dynamic Library is essentially a collection of ClassSpecs and MethodSpecs. Still, they are specific to an ImageFormat, because the image loading the library is not required to include the Compiler, or to know how to compute the internal state for new classes.

Each ClassSpec specifies the shape of a class, as known by the Builder image, and as assumed when building the bytecode for the MethodSpecs. It is important to include the parent class path up to the hierarchy root, because this is the way to completely specify the shape of the instances, and the bytecodes for accessing instance variables.

Design of Dynamic Libraries 2

File Format for code libraries

A Dynamic Library file is the serialization of a Dynamic Library object.

Currently we are using ReferenceStream. A different serializer could be adopted.

There is no special file format for Dynamic Libraries. Library files are just the ReferenceStream serialization of the Dynamic Library object we just described.

Design of Dynamic Libraries 3
Loading Dynamic Libraries

To load a Dynamic Library

- The file is loaded and deserialized
- Validations are done
- ClassSpecs for classes not in the image are created as new classes
- ClassSpecs for classes already in the image are merged (new instance variables added to existing classes at the end)
- Old CompiledMethods affected by class reshaping have instance variable accesses fixed
- MethodSpecs are materialized into new CompiledMethods
- New CompiledMethod affected by class reshaping have instance variable accesses fixed
- New CompiledMethods are installed
- Class initialization is performed

Loading a Dynamic Library involves several steps. After loading the file and deserializing the objects, we need to run some validations. For instance, we require that every ClassSpec included in the library that also exists in the system, has the same parent. This applies recursively up to the hierarchy root. But the shape of the classes are not required to be the same. A ClassSpec can add new instance variables, and is not required to include all the instance variables in the existing class. Both adding ivars, and ignoring existing variables is OK.
If a library adds some instance variable to a class, all instance variables in subclasses are moved to the right, as new variables are added before them. Any existing method that accesses them needs will be adjusted for this.
Additionally, if the image loading a library includes instance variables that the library is unaware of, any MethodSpec accessing an instance variable that is now at a different position will also be fixed.
After this, class initialization is performed, and we are done loading the library.

Current State
Of Bootstrap

- Only the existing Cuis Compiler is used
- Only the Cuis bytecode set is implemented
- Only the Spur image format is implemented

The Bootstrap framework is designed with all the ideas stated before and is hopefully easy to extend, but so far we have only been using the existing Cuis compiler, and that means that right now we can not change language semantics. Enabling that will need pluggable compilers that generate different formats of CompiledMethods or enough information to materialize these CompiledMethods.

Right now we are only using the regular Cuis Compiler, and the standard V3 bytecode set, and we only have builders for the Spur image formats. The knowledge of the details of the Spur image formats is well localized in the design. Supporting additional image formats is mostly writing a new builder for them.

Also, as of today, ClassSpecs and ImageSpecs are created from existing classes and methods in the system. Complete separation of the contents of the new image in its own Source Code files requires additional work.

The system is designed to enable all that, and this is the first iteration.

Future Work
For Bootstrap

- Implement a straight SourceCode->MethodSpec Compiler
- Implement and use a pluggable Compiler architecture
- Additional ways to specify code to be included, for instance by including Code Packages or pre-loaded Code Libraries.
- Implement additional ImageBuilders
- New image formats with new capabilities could require new ForeignObjects

So, Future Work includes actually writing the missing parts. They include a separate Smalltalk Compiler that can compile SourceCode straight into ClassSpecs and MethodSpecs, using any desired bytecode set.

Other ways to specify code to be included are also possible. For instance, a list of Code Packages or Code Libraries to be pre-loaded in the bootstrapped image.

It also includes writing ImageBuilders for supporting additional image formats and VMs, and most likely new kinds ForeignObject to create whatever new kinds of objects those new VMs will support.

Future Work

For Compiled Libraries

- Integration with the existing Code Packages in Cuis
- Optionally including source code

Compiled Libraries are now usable as a means to deploy parts of a stand alone application written in Cuis.

But there is significant conceptual overlap with Code Packages, so integration with them makes sense. As part of this, Compiled Libraries could also include optional Source Code, for use by developers.

Demo time!

Yep. Let see it in action!

Thank you!

That’s all. It is my hope that you find Bootstrap both interesting and useful.

Thanks for your time.