Bootstrapping a Smalltalk
G. Casaccio, S. Ducasse, L. Fabresse, J-B. Arnaud, B. van Ryseghem
Presented by: M. Denker
What is Bootstrapping?

A process that builds the minimal infrastructure of a language that is reusable to define this language itself.
Example: Bootstrapping a language X

Language X

- Compiler
- Tools
- Loader
Example: Bootstrapping a language X

How to create/write all of this for language X?
Example: Bootstrapping a language X
Example: Bootstrapping a language X

Language X
Resilient
≠

Language Y
Java
Example: Bootstrapping a language X

Language X
Squeak/Pharo

Language Y
Slang

Diagram:
- Compiler
- Tools
- Loader
- Description
- Compiler or Loader

Example: Bootstrapping a language X Compiler or Loader
Why Bootstrapping?

- Agile and explicit process
- Explicit malleability and evolution support
- Warranty of initial state
- Minimal self reference
Existing approaches

Bootstrap Approaches

Execution-based (tracing)
- OnAbsence
  - Spoon
- OnPresence
  - Chacharas

Static-based (generation)
- Dumping
  - MicroSqueak
  - Hazelnut
- Recreating
  - CorGen
  - GNU ST
Existing approaches

Bootstrap Approaches

Execution-based (tracing)
- OnAbsence
  - Spoon
- OnPresence
  - Chacharas

Static-based (generation)
- Dumping
  - MicroSqueak
  - Hazelnut
- Recreating
  - CorGen
  - GNU ST
Spoon

- Client / Server approach
- Client side starts as minimal
- On each slot access if the target object is missing, fetch and copy it from the server
Chacharas

Same approach as Spoon but:

- Analyze a server side execution

- All reached objects are copied on the client
1 - A kernel is loaded from files into a namespace

- Create stub objects
- Fix references of classes and metaclasses
- Fix references of compiled method
- Save it
Hazelnut

Same approach as MicroSqueak but:

- does not rely on a specific list of class that are manually edited

- takes a list of classes as input and recursively copy classes into a new namespace
**Discussion**

**Execution-based approaches:**
- difficult to control which objects will be selected for the bootstrapped image
- reflection breaks tracing
- not suitable for interactive programs
+ suitable to compute the minimal runtime required by a program

**Static-based approaches:**
+ easier to control the result of a static generation
+ suitable for deep changes in a system (new object format, ...)
- hard to specify / write / maintain
Our approach

Bootstrap Approaches

Execution-based (tracing)
- OnAbsence
  - Spoon
- OnPresence
  - Chacharas

Static-based (generation)
- Dumping
  - MicroSqueak
  - Hazelnut
- Recreating
  - CorGen
  - GNU ST
CorGen

1. Creation of the stub objects for literal objects: nil, true, false, characters
2. Definition of classes and metaclasses
3. Method compilation
4. Creation of process and special object array
5. Image serialization.

Bootstrap>>bootstrap [ self
  instantiateSmalltalkObjects;
  importClassesFromSources;
  processClasses;
  setupSmalltalkObjects;
  saveImage]
Stubs Creation

```
Bootstrap>>instantiateSmalltalkObjects [ self
    instantiateNilGst;
    instantiateTrueGst;
    instantiateFalseGst;
    instantiateCharactersTable; "build all the characters"
    instantiateEnvironment "create System Dictionary"]
```
Classes/Metaclasses creation

Bootstrap>>processClasses [ "fill the class stubs with real classes" self  "create classes and add them to System Dictionary" createClasses;  "compile and install CompileMethods" compileMethods ]
Compile Methods

- Methods either taken from the model or the source files

- Use a compiler parametrized by an environment and a symbol table

- Deep changes may be applied (change bytecode set, other optimization, ...)

Initializing the System

Bootstrap>>setupSmalltalkObjects [  
  self setupCharacter; "insert references to the Character table"
  setupSymbol; "insert references to the Symbol table"
  setupProcessor "create Processor and install it"
]

Bootstrap>>setupProcessor [ | processorGst |
  processGst := self createProcess.
  processorGst := GstProcessorScheduler new.
  processorGst scheduler: nilGst;
  processes: self buildProcessList;
  activeProcess: processGst;
  idleTasks: nilGst. ]
Initializing the System

Bootstrap>>createProcess [  
  | processGst |  
  (processGst := GstProcess new)  
    nextLink: nilGst;  
    suspendedContext: self createInitContext;  
    priority: 4;  
    myList: nilGst;  
    name: GstString new;  
    interrupts: nilGst;  
    interruptLock: nilGst ]
Saving the Image

- Comply with image file format:
  - Image header
  - Special object array

- Serialize objects according to their shape (CompiledMethods, ...)

- Avoid object duplication during serializing
The Result

A 54 classes Smalltalk Kernel:

Kernel (15 classes): Behavior, BlockClosure, BlockContext, Boolean, Class, ClassDescription, ContextPart, False, Metaclass, MethodContext, MethodInfo, Object, ProcessorScheduler, True, UndefinedObject.


Related Work

- Lisp: using image such Smalltalk, the bootstrap is done by migrate the current image, using a cross-compiler include in the host image.

- Ruby: the kernel is load and initialize by the VM some low level initialization is done in C, all the other is done by ruby processing. After all the module is load separately by the Virtual Machine.

- Python: the Python virtual machine is initialized. Some classes stubs are created and initialized in the virtual machine. (close of the ruby bootstrap)
Conclusion and Future Work

Pros and Cons:
- Execution-based bootstrapping (tracing) such as Hazelnut
- Static-based bootstrapping (declarative) such as CoreGen

Future work: Boostrapping Pharo
- use an execution-based approach as an intermediate solution
- reach a static-based bootstrap that can easily be maintained and co-evolve with the system
THANKS

Bootstrapping a Smalltalk
G. Casaccio, S. Ducasse, L. Fabresse, J-B. Arnaud, B. van Ryseghem

Presented by: M. Denker