Imposing a Memory Management Discipline on Software Deployment

Eelco Dolstra, Eelco Visser and Merijn de Jonge
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
{eelco, visser, mdejonge}@cs.uu.nl

March 25, 2004
The problem

Software deployment (the act of transferring software to another system) is surprisingly hard.

• Must ensure correctness.
  – Dependency information must be complete.
  – Component compatibility.
  – Atomicity of upgrades/downgrades.
  – Safe removal of unused components.
• Lot of effort.
  – Packaging is often (semi-)manual.
  – Source/binary distributions.
  – Must package each variant.
  – Don’t want to install all component separately.
  – Especially a problem with small-grained reuse (e.g., Strate-goXT).

• Should support multiple versions/variants.
  – Test a component before production use.
  – Multiple users.
Incomplete Dependencies

Eelco Dolstra

STC—25/3/2004
Interference

Upgrade of App2 (causes upgrade of LibB to LibB')

Removal of App3 (causes deletion of LibA)
The core problems

• Must prevent *unresolved component dependencies*.
  – A component should never refer to another component not present on the target system.
  – Hard to validate; how to detect use of undeclared dependencies?
  – Timeline issues: (related) dependencies at build and run time.
• Must prevent *component interference*.
  – Different versions/variants of a component (or completely unrelated components) should not interfere with each other.
  – Upgrades are usually *destructive*. E.g., only one /usr/bin/gcc.
Software deployment as a memory-management problem

- memory ⇔ disk
  - objects (values) ⇔ components
  - addresses ⇔ path names
- pointers are numbers ⇔ pointers are strings
- pointer dereference ⇔ I/O
- pointer arithmetic ⇔ string operations
- dangling pointer ⇔ reference to absent component
- object graph ⇔ dependency graph
- persistence/serialisation ⇔ deployment
Closures

• Correct deployment of component $c$ requires distributing the smallest set of components $C$ containing $c$ closed under the “has-a-pointer-to” relation.

• I.e., we have to discover the pointer graph.
Determining the pointer graph

• This is just what garbage collectors for programming languages have to do.

• GC requires a **pointer discipline**:
  – Ideally, entire memory layout is known, and no arbitrary pointer formation (e.g., integer ⇔ pointer casts).
  – But even C/C++ has rules: pointer arithmetic is not allowed to move a pointer out of the object it points to.
  – This is why **conservative GC** works: assume that everything that looks like a pointer *is* a pointer.

• But software components do not have any pointer discipline.
  – Any string can be a pointer.
  – Pointer arithmetic and dereferencing directories can produce pointers to any object in the file system.
A pointer discipline

Solution: *impose* a pointer discipline.

- Each component should include in its a path a unique identifying string.
  
  /nix/store/15373f8c93776a3a5f86fec65914e59d-subversion-0.37.0
  
  /nix/store/b70b48128d8d13725346684ea43963c4-strategoxt-0.9.3

- Then we can apply conservative GC techniques to determine the pointer graph.
## Scanning for pointers

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>080</td>
<td>00 80 04 08 34 41 01 00 34 41 01 00 05 00 00 00</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>090</td>
<td>00 10 00 00 01 00 00 00 34 41 01 00 34 d1 05 08</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>0a0</td>
<td>34 d1 05 08 b4 04 00 00 c4 04 00 00 06 00 00 00</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>0b0</td>
<td>00 10 00 00 02 00 00 00 7c 41 01 00 7c d1 05 08</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>0c0</td>
<td>7c d1 05 08 90 01 00 00 90 01 00 00 06 00 00 00</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>0d0</td>
<td>04 00 00 00 04 00 00 00 60 01 00 00 60 81 04 08</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>0e0</td>
<td>60 81 04 08 20 00 00 00 04 00 00 00 04 00 00 00</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>0f0</td>
<td>04 00 00 00 50 e5 74 64 20 41 01 00 20 c1 05 08</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>100</td>
<td>20 c1 05 08 14 00 00 00 14 00 00 00 04 00 00 00</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>110</td>
<td>04 00 00 00 2f 6e 69 78 2f 73 74 6f 72 65 2f 38</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>120</td>
<td>64 30 31 33 65 61 38 37 38 64 30 66 66 38 34 63</td>
<td>d013ea878d0ff84c</td>
</tr>
<tr>
<td>130</td>
<td>62 31 37 38 61 34 62 31 36 30 65 34 30 32 36 2d</td>
<td>b178a4b160e4026-</td>
</tr>
<tr>
<td>140</td>
<td>67 6c 69 62 63 2d 32 2e 33 2e 32 2f 6c 69 62 2f</td>
<td>glibc-2.3.2/lib/</td>
</tr>
<tr>
<td>150</td>
<td>6c 64 2d 6c 69 6e 75 78 2e 73 6f 2e 32 00 00 00</td>
<td>ld-linux.so.2...</td>
</tr>
<tr>
<td>160</td>
<td>04 00 00 00 10 00 00 00 01 00 00 00 47 4e 55 00</td>
<td>GNU.</td>
</tr>
<tr>
<td>170</td>
<td>00 00 00 00 02 00 00 00 02 00 00 00 05 00 00 00</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>180</td>
<td>83 00 00 00 bb 00 00 00 58 00 00 00 ab 00 00 00</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
<tr>
<td>190</td>
<td>ae 00 00 00 a1 00 00 00 00 00 00 00 00 00 00 00</td>
<td>4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A 4A</td>
</tr>
</tbody>
</table>

Eelco Dolstra

*STC—25/3/2004*
Risks

• Like all conservative GC approaches, there is a risk of *pointer hiding*.
  – Compressed executables.
  – UTF-16 encoded paths.

• Hasn’t happened yet, though.
Persistence

• The unique strings should be cryptographic hashes of all inputs involved in building the component.
• This prevents address collisions in the target address space (i.e., path name collisions in the target file system).
Nix expressions

Component description in a pure functional language.

```
{stdenv, fetchurl, aterm, sdf}:

derivation {
    name = "strategoxt-0.9.3";
    system = stdenv.system;
    builder = ./builder.sh;
    src = fetchurl {
        url = ftp://.../strategoxt-0.9.3.tar.gz;
        md5 = "3425e7ae896426481bd258817737e3d6";
    };
    inherit stdenv, aterm, sdf;
}
```
Nix expressions (2)

Build script:

```sh
#!/ .../bin/sh

buildinputs="$aterm $sdf"
. $stdenv/setup || exit 1

tar zxf $src || exit 1
cd stratego* || exit 1
./configure --prefix=$out --with-aterm=$aterm \
    --with-sdf=$sdf || exit 1
make || exit 1
make install || exit 1
```
Nix expressions (3)

Composition: (all-packages.nix)

```
rec {
    strategoxt = (import ..//development/compilers/strategoxt) {
        inherit fetchurl stdenv aterm;
        sdf = sdf2;
    };
    aterm = (import ..//development/libraries/aterm) {
        inherit fetchurl stdenv;
    };
    sdf2 = (import ..//development/tools/parsing/sdf2) {
        inherit fetchurl stdenv aterm getopt;
    };
    stdenv = ...;
    ...
}
```
Nix expressions (4)

Consistency between components / variation points:

```nix
{ localServer ? false, httpServer ? false
, sslSupport ? false, swigBindings ? false
, stdenv, fetchurl
, openssl ? null, httpd ? null, db4 ? null, swig ? null
}:

assert expat != null;
assert localServer -> db4 != null;
assert httpServer -> httpd != null && httpd.expat == expat;
assert sslSupport -> openssl != null &&
  (httpServer -> httpd.openssl == openssl);
assert swigBindings -> swig != null && swig.pythonSupport;

derivation {
  name = "subversion-0.37.0";
  ...
}
```
User operations

To build and install StrategoXT:

$ nix-env -if ../../../all-packages.nix strategoxt

When a new version comes along:

$ nix-env -uf ../../../all-packages.nix strategoxt

If it doesn’t work:

$ nix-env --rollback

Delete unused components:

$ nix-collect-garbage
Transparent binary deployment

On the producer side:

$ nix-push $(nix-instantiate .../all-packages.nix) \ http://server/cache

On the client side:

$ nix-pull http://server/cache

Installation will now reuse pre-built components, *iff* they are exactly the same.
FAQ

“How to handle security patches (e.g., in the C library)? There you do want destructive updates.”

• No you don’t. How to roll-back if the patch breaks stuff?
• Just deploy new Nix expressions; to the extent that there is sharing with old ones, no rebuilds / redownloads are necessary.
• In the case of dynamic libraries, wrapper packages can be used to prevent a mass rebuild.

• A nice aspect of Nix is that we can generically find all components impacted by some source file.
Future work

- Build management ⇒ unify the whole build and deployment problem into a single formalism.
- Shared Nix stores ⇒ thanks to cryptographic hashes we can allow closures to be shared among different users without fear of trojans etc.
- (G)UI for selecting variants; interesting satisfiability problem.
- Should address component state.
Related work

• Deployment / package managers: RPM, Gentoo, etc.
  – Unsafe — incomplete deployment, not atomic.
  – Little or no support for variability.

• Build managers: Make.
  – Unsafe.

• Better build managers: Vesta, ClearCase.
  – Do not do deployment.
  – Cannot handle retained dependencies.
  – Not portable; rely on virtual file system.

• Autoconf.
  – Automatic adaptation to target environment is dangerous; shouldn’t be done by end-user.
Conclusion

- Concurrent installation of multiple versions and variants.
- Atomic upgrades and downgrades.
- Multiple user environments.
- Safe dependencies.
- Complete deployment.
- Transparent source and binary deployment.
- Safe garbage collection.
- Portability.
More information

- Website: http://www.cs.uu.nl/groups/ST/Trace/Nix.