Release Management for Stratego/XT

with Nix

Eelco Dolstra
Center for Software Technology
Utrecht University
eelco@cs.uu.nl

March 4, 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.
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.
• However, 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 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>Addr</th>
<th>Hex</th>
<th>ASCII</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........</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...4...</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>4................</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>........</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></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></td>
</tr>
<tr>
<td>0e0</td>
<td>60 81 04 08 20 00 00 00 20 00 00 00 04 00 00 00</td>
<td></td>
</tr>
<tr>
<td>0f0</td>
<td>60 81 04 08 20 00 00 00 20 00 00 00 04 00 00 00</td>
<td></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></td>
</tr>
<tr>
<td>110</td>
<td>04 00 00 00 2f 6e 69 78 2f 73 74 6f 72 65 2f 62 31 78 61 64 69 66 66 6f 72 65 2f 62 6c 69 62 63 2d 32 2e 33 2e 32 2f 6c 69 62 2f 6c 6d 2d 6c 69 75 78 2e 73 6f 2e 32 00 00 00</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>64 30 31 33 65 61 38 37 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></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></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></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></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></td>
</tr>
<tr>
<td>190</td>
<td>ae 00 00 00 a1 00 00 00 00 00 00 00 6c 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>
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:

```bash
#!/.../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
```
Composition: (all-packages.nix)

```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 = ...;
    ...;
}
```
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.
Implementation

- All components are stored in a *store* (e.g., `/nix/store`).
- Creation of components within the store described using *store expressions*.
- Store expressions describe a component build (a *derivation*) or the result thereof (a *closure*).
- Nix expressions are translated into store expressions. The path of the component is a cryptographic hash of *all* inputs into the build process. This ensures that no collisions occur between components.
StrategoXT release management

Not Nix’s core competency, but:

- Useful for implementing a automated release system.
  - Build source distributions.
  - Build RPMs.
  - Generate release web pages.
  - ...

Eelco Dolstra
StrategoXT release management (2)

Advantages of using Nix:

- Dependency management.
- Sharing and storage management.
- Multi-platform builds are easier (we think!).
- Specifications are nice and short.

Disadvantages:

- Building non-Nix packages (e.g., RPMs) is messy.
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.