SUGAR 3.0?

Or how I spent my winter vacation

David Bindel

UC Berkeley, EECS
Overview

- Literate programming
- C++ via C
- Core objects
- Using Lua
- Using Matlab
- Things done
- Things to do
Literate programming

- Conceived by Knuth
- Programs as *human-oriented* documents
- Use document preparation language (e.g. TeX) plus conventional language
- Famous example: Hanson’s lcc compiler
- Ultimate products: an executable and a book
The noweb system

- One of many literate programming implementations
- Extremely simple to use
- Agnostic to programming language used
- Supports \LaTeX, \TeX, HTML, and troff docs
- Has an Emacs mode
A \([\text{material}_t]\) structure represents a material (or process layer) object.

\begin{verbatim}
<<types>>=
<<method table>>
typedef struct {
    model_material_t* model;
    void* self;
} model_material_t;
@
\end{verbatim}
Literate example

The `[[model_material_t]]` structure holds the method tables for material models.

```c
<<method table>>=
typedef struct {
    <<material methods>>
} model_material_t;
@
```
The [[init]] method initializes local data used by a material object.

\[
\begin{align*}
\text{<<material methods>> = void* (*init)(struct mesh_t* mesh,} \\
\text{\quad const char* model);} \\
\end{align*}
\]
The [[destroy]] method frees local data, if it needs to be explicitly freed.

```c
<<material methods>>=
void* (*destroy)(void* self);
@
```
Literate example

The [[param]] method queries the material for a named parameter (e.g. [[Youngsmodulus]]).

<<method definitions>>=
struct mesh_param_t*
    (*param)(void* self,
        const char* name);
@

SUGAR 3.07 – p.9/24
Literate example

The \[[\text{output}]\] method generates an external representation (maybe text, maybe a data structure).

\begin{verbatim}
<<method definitions>>=
void (*output)(void* self,
   struct netout_t* name);
@
\end{verbatim}
Example reviewed

- Revised excerpt from *modelmgr.nw*
- Style is “C++-like C”: *this* pointer and vtable are explicit.
- One interface for several implementations
- Generated C code looks good
A mesh is a collection of nodes, elements, and materials.

- A node has (optional) name and position.
- Material object interface just reviewed.
- Element object mostly interacts with assembly routines.
- Model manager object retrieves models by name.
Assembly pattern

- Assembly object calls methods for each element
- Methods call assembly object method to make local contribution
Element methods

- The `init` and `destroy` methods construct and destroy model.
- The `vars` method gets global indices for local variables.
- The `displace` method contributes to boundary conditions.
- The `R` and `dR` methods contribute to the residual vector and tangent matrix.
- The `output` and `display` methods perform element output.
Assembly objects

- The *variable manager* figures out names of nodal variables and builds mappings from (node, name) pairs to global index
- The *assembler* objects build displacement boundary condition lists, system matrices, residual vectors
- The *mesh output and drawing* objects generate external representations of meshes.
Assembly objects

- Multiple implementations of many types
- Both sparse and dense matrix assemblers
- Output generates text or Matlab structure
- Can draw to X, Matlab, or file for Java reader
Mesh generation

- C mesh generation routines in mesh module
- Want to build a wrapper those interfaces
- New wrapper language of choice: Lua
- Could also build or add to mesh using Matlab, C, Smalltalk, ...
Lua language

- Designed at TecGRAF, PUC-Rio in 93
- Intended as an extension language
- Centered around a single data structure (tables, or associative arrays)
- Pascal-like syntax; functions are first-class; support for closures
- Has been used for finite element input
- Popular as a game scripting language
Why Lua?

- Simple – syntax fits on a single page
- Flexible – can add user types, functions; extend semantics with “tag methods”
- Portable – small code base in ANSI C
- Speed – uses stack-based bytecode
- User environment tightly controlled
- Documentation and code base maintained by someone else!
use("mumps.net")

A = node {name = "A"; 0, 0, 0}
b = node {name = "b"; 100u, 0, 0}

element{ A ; model="anchor",
material=p1, w=10u }

element{ A, b; model="beam2d",
material=p1, w=2u }

element{ b ; model="force",
x=30u, y=30u }
Matlab interfaces

- Single MEX file, multiple functions
  (handle = sugarmex('use', file))
- Allows sharing C data structures opaque to Matlab
- Handle system for adding opaque C types and objects to Matlab.
- Registration system to add new functions with minimal overhead
Front-end and back-end Matlab

- Front-end Matlab routines access high level functionality (e.g. load a netlist)
- Back-end Matlab routines provide functionality under a C interface (e.g. models in Matlab)
- Matlab models use callback interfaces to C objects to make local contribution
Things done

- Core architecture moved into C
- Several models, including Matlab
- Assembly interfaces (dense and sparse)
- Output routines (to text, Matlab structure, display, or Java file format)
- Mesh generation interface in Lua
- Front-ends in Lua and in Matlab
Things to do

- Move often-used model functions to C
- Build wrapper for old Matlab model interface
- Build wrapper to call FEAP models?
- Add dependence on dynamic variables
- Extend Lua mesh generator (e.g. transformation stack)
- User-level documentation
- Add sparse matrix support to Matlab front-end
- Put in all the old algorithms (C or Matlab)