PSciCo Coding Conventions

The PSciCo Project

July 30, 2001

Contents

1 Naming Conventions

1.1 Signature, Functor and Structure Names

We will adopt the standard conventions of full capitalization for signatures and initial capitalization for structures. Examples: `signature MATRIX`, `structure RealSparseMatrix`. Underscores should be used in signatures with multiple words: *e.g.*, `signature SIMPLICIAL_COMPLEX`.

We also adopt the policy of emphnot using the same name for a signature and functor or structure. Note: for the present purposes, `MATRIX` and `Matrix` are considered to be the *same* name.

1.2 Variable and Function Names

We will use the convention that variables start with a lower-case letter: `thisIsAVariableName`.

Names should be long rather than short. The motivation for this is that often when a short name is use, it makes a lot of sense at selection time but looses its meaning or becomes ambiguous later. An example was the use of quadElement in the finite-element code. This could either mean quadratic element or quadrilateral element. At the time it was introduced there were no quadratic elements so the name was "unique", but later as quadratic elements were added it became ambiguous.

1.3 Type Names

The advantage of using the type name `t` is that it makes it easy to take advantage of "accidental" name matching so that signature will match other signature definitions. For example, `ORDERED`, with `t` and `compare`, `EQUALITY`, with `t` and `equal`, `PRINTABLE`, with `t` and `toString`, and `READABLE`, with `t` and `fromString`.

However, people do not like the name `t` because it has no meaning. To satisfy this criticism we will give *both* a descriptive name and the name `t` wherever appropriate. The descriptive name and the conventional name are to be equated in the signature. For example, if you are writing a signature that supports standard operations such as `compare`, `equal`, and `toString`, the signature should include a type `t` which is equated to the descriptive name for that type. For example, in the matrix package there will be a declaration of the form `type matrix` and a declaration `type t = matrix`.

Another issue was having a long chain of signatures each of which is a "super-signature" of the previous one. An example of this is

```
monoid -> group -> rng -> ring ->
  divisionRing -> field -> number
```

For this case the main type should be called something like `element` so that each is a proper supersignature of the previous.

1.4 File Names

The prefix of the file name should be exactly the same as the name of the structure/functor/signature in the file (including capitalization). The suffix

should always be `.sml`. For example, the file `MATRIX.sml` contains the signature `MATRIX`, and the file `RealSparseMatrix.sml` contains the structure `RealSparseMatrix`.

Since we do not allow structures/functors and signatures to have the same name (where same includes different cases), this will not cause problems on file systems that are case insensitive.

We discourage the use of files with more than one signature, structure, and/or functor. In the case such files are used, the name should include a hyphen, *e.g.*, `Int-Structures.sml`.

1.5 Constructor Names

Constructor names are to be either infix or capitalized.

2 Documentation

2.1 Headers

Every file should start with a header that contains at least the following information. Ideally it should include more documentation about the contents of the file.

```
(*********************************************************
 ** ComplexNd.sml
 ** sml
 ** Umut A. Acar
 **
 ** An N-dimensional simplicial-complex implementation.
*********************************************************)
```

3 Style

3.1 Placement of small structures

There is often the issue of where to put the definition of structures that are directly generated from functors (*e.g.*, `IntQueue`). Each such definition is typically one or two lines of code.

In general such definitions should be placed in the directory with the definition of the base type, rather than the functor. For example `IntQueue` should be defined along with `Int`, not along with `Queue`. One possibility

is to have a subdirectory along with the definition of `Int` which contains a bunch of tiny files each with the definition of one structure (*e.g.*, `IntQueue`, `Int Table`,...)

3.2 Currying

There is no hard and fast way to decide whether (or how) a function should be curried. However, we would like some consistency about how things should be curried. In some cases where efficiency is critical it is worth defining to versions of a function, one curried and one not. An example of this is `sub`, which takes the ith element out of a vector (or other aggregate structure). From the programmer's point of view it makes most sense to curry this so the user can say `sub vec` and then apply this multiple times to integers to get the relevant elements out. However, this might mean a serious loss in efficiency. The solution would be to supply a second `sub'` that is not curried, and presumably more efficient.

3.3 Clausal function definitions

Should be avoided for all but small internal functions. For example, one should use

```
fun length l =
    case l
    of nil => 0
     | _::t => 1 + length t
```

and not

```
fun length nil = 0
  | length (_::t) = 1 + length t
```

especially for top level functions.

4 The compilation manager

We need to better study the new manager before addressing this.