Feng Shui for Standard ML Programmers

Stephen Gilmore

LFCS, Edinburgh

Abstract. *Feng Shui* is an ancient knowledge which provides instruction about the importance of order and harmony in living. Programmers working with sophisticated programming languages such as Standard ML want to achieve correctness (order) in programs. However, they also need to engineer trade-offs (harmony) between elegance and efficiency; time and space; generality and specialisation; encapsulation and re-use; and the imperative and applicative worlds. This paper teaches how the principles of Feng Shui can be utilised in the cost-effective creation of Standard ML programs which are both beautiful and correct.

Introduction

It may seem fanciful to think that the skills which are needed to produce a well-designed computer program could have elements in common with the abilities which are needed to create a harmonious dwelling place but no less a source than the *Definition of Standard ML* encourages us to think of programs as being like houses:

> ... a robust program written in an insecure language is like a house built upon sand. [1]

That the setting for the construction of the house is considered to be highly significant is also a cherised teaching in Feng Shui. In ancient China great care was taking in choosing a site to establish a village, a town, or a capital. A well chosen site could bring prosperity, whereas a poorly chosen site could being about famine or war. The duty of selecting a site fell to the most learned men, as programming language design historically fell to the most experienced computer scientists [2, 3].

Domestic metaphors abound in the literature on Standard ML. As another example, Mads Tofte writes this about the Standard ML modules system:

> ... a program which is written as one long list of core language declarations can easily end up looking rather like a shopping list where items have been added in the order they came to mind. [4]

So, where we have a well-designed language, it is like a good place to build a house and conduct domestic activities. Still, it might seem that the theory of Feng Shui cannot be applied to a mathematical process such as computer programming. The purely formalist approach to formal development of programs takes the view

↑ datatypes and abstypes
↑ higher-order functions
↑ opaque signatures `:>`
↑ fully functorial style
↑ exceptions
↑ polymorphism
↑ Basis library

↓ type abbreviations
↓ higher-order functors
↓ transparent signatures
↓ `open`
↓ `callcc`
↓ equality types
↓ `Unsafe.CInterface`

Fig. 1. Flux and polarity of Standard ML

that a program is methodically developed to agree with a specification so where is the opportunity for Feng Shui to assist (improve)? There are several questions which the formalist approach does not answer and cannot answer.

– How does *invention* play a part in the process? Even a routine programming task can require the creation of a new software component or assembly which needs to be shaped. The formalist approach does not help the programmer to think creatively.
– How does *intuition* play a part in the process? There are many possible paths through a software development process. Some of these will produce results more easily than others although it may be unclear at the outset which should be chosen. The formalist approach does not help the programmer to judge wisely.

Although the formalist approach can allow you to discover after the fact whether or not a programming process was a success—under a purely formalist notion of correctness—we all believe that luck and timing also play a part in determining success or failure in our endeavours in life. We may wish to develop our programs formally but we *also* will need good luck as well. Feng Shui teaches us that each house also has its own luck and timing.

Some parts of the programming process are not considered worthy of discussion by the formalists; for example formatting. Indentation of programs plays an important part in the work of the programmer: with improved formatting we can invite beauty to enter our programs. Standard ML takes this to another (meta)-level by being able to beautify programs [5] beautifully [6]. With Standard ML you can bring good luck to your programs by always putting a space before a `#` sign and always putting a space after an `=` sign. These methods are inherently lucky and you can bring more luck into your program by improving the indentation of your program between compilations by using the method of changes. This is similar to the idea that changing the furniture layout in the house can make it possible for the resident to attain good fortune. Flux and polarity of good and bad fortune in Standard ML is shown in Fig. 1.

Feng Shui for Types

Balance: The principle of balance is essential for Feng Shui. When searching for balance, it is possible to find the Feng Shui which you seek. Lucky shapes have meaning and should balance: in Standard ML type variables should never be lost. The following datatype declaration will be unlucky and should not be used.

```
datatype 'a unlucky = lost;
```

Similarly the following datatype will be unlucky. A type parameter is like a meal served in a house: two people should not eat just because one person can. This type is a greedy type and should not be used.

```
datatype 'a greedy = hungry | feed of ('a * 'a) greedy;
```

This program will show why the type is greedy. If we look at the hungry value after it was fed it is more hungry than it was before.

```
feed hungry; feed it; feed it; feed it; feed it; feed it;
val feed it = it; val feed it = it; val feed it = it;
val feed it = it; val feed it = it; val feed it = it;
```

Luck: Type errors can be caused by bad luck. If you have a type error during compilation of the form `cannot unify T with T` or if you see unpleasant type variables like `?{T}`, `?.T` or even `T(hidden)` then close your session; unplug your workstation and plug it in again in a different location (it need not be more than a few inches away). On returning to your Standard ML programming you can find that the type error has disappeared.

Clutter: Elimitating unnecessary clutter in your house can improve your luck. The same is true for Standard ML programs. Type identifiers can become cluttered with unnecessary primes. If you have written this:

```
type ''a set = ''a list;
```

or even this:

```
type '''a set = '''a list;
```

then you can improve your luck by eliminating the unnecessary primes:

```
type 'a set = 'a list;
```

An unneeded prime is like a cricket chirping at the fireplace.

Walls: Abstraction forms the walls in the house of programming. When you have a wall you should make sure that it is solid. Do not replicate your datatypes outside an abstraction. Oddly placed doors in a house can bring calamity.

```
abstype wall = solid
with
  datatype door = datatype wall
end;
```

Feng Shui for Run-time

Unclean: The run-time system is like the sewer which runs under the house. It serves a useful purpose but no-one can play there and expect to remain clean. Avoid the `Unsafe.CInterface`. Functions written in C absorb essential "sheng chi" (vital energy) leaving practically nothing for neighboring functions.

Unfaithful: Avoid terminating functions of type `'a->'b`. Such functions are unfaithful to the spirit of the Definition of Standard ML.

```
abstype key = safe of int
with
  fun keep x = safe x
end;
val x : int = MLWorks.Internal.Value.cast (keep 3);
```

Venerate all the corners of the Definition. Semanticists beware implementors.

Conclusions

It is often said in jest that belief in a programming language is like a religious faith. Be guided by the wisdom in that jest and always program in good faith. When programming receive blessings for good chi.

Blessings

This essay is dedicated to my beloved Jane who introduced me to the literature on Feng Shui. I wish good Yang chi to our daughter Martha, who always has lots.

References

1. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised 1997.* The MIT Press, 1997.
2. Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971.
3. C.A.R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford University, 1973.
4. Mads Tofte. Four lectures on Standard ML. Technical Report ECS-LFCS-89-73, Laboratory for Foundations of Computer Science, The University of Edinburgh, 1989.
5. Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.
6. Larry Paulson. *ML for the Working Programmer.* Cambridge University Press, second edition, 1996.