Documentation on a metaprogramming experiment with Hephaestus*†

Ralf Lämmel

December 6, 2014

Contents

1 Objective of the experiment

The objective was to understand whether a) Haskell metaprogramming techniques can be used to obtain instances of Hephaestus that cover selected variations of assets, e.g., use cases and business processes, and b) such a metaprogramming approach can be provided in a way that the Hephaestus product-line infrastructure is applied to itself.

2 Summary of the results

Subject to a number of simplifying assumptions, the experiment was successfully completed. One simplification concerns the assets in scope. We used mockups as opposed to the actual implementations in Hephaestus. If we were to use the actual implementations, the relevant modules may require some changes. Another simplification concerns the feature model of Hephaestus. We only consider the OR feature for the different assets.

*This experiment was initiated during Vander Alves' visit to Koblenz in August 2011.

†For internal use by the involved researchers only.

3 Testing the implementation

The experiment is provided by a self-contained directory *meta-hephaestus* that is added to a master of Hephaestus as obtained from git. The following cabal packages were required: *funsat-0.6.0* (as required by Hephaestus anyway) and *haskell-src-1.0.1.3* (needed for metaprogramming).

There are the following files and directories:

- File *Makefile*: build and test various products.
- Directory *doc*: this documentation.
- Directory *HplAssets*: the asset base of the Hephaestus product line.
- Directory *HplProducts*: products built with the Hephaestus product line.
- Directory *HplDrivers*: Main modules to initiate building or loading.

Just run "make test" to exercise all building and loading.

4 Description of the approach

The experiment focuses on showing that certain key types and functionality of any specific Hephaestus instance can be derived by metaprogramming. To this end, we consider an *empty product* (see directory *HplProducts*) from which to build "non-empty products". All products are expected to define *SPLModel*, *InstanceModel*, *ConfigurationKnowledge*, *TransformationModel*, and a transformation function *transform*.

All forms of assets are hence provided in a form that they can contribution to the above-mentioned entities; see directory *HplAssets*. We use metaprogramming operators that build the product's entities (say, types and functions) from the corresponding parts of the asset-related modules by adding parts incrementally to the empty product.

Because of the objective of self-application of Hephaestus, the above-mentioned transformational approach is actually packaged as another kind of asset: Hephaestus; see again directory *HplAssets*. That is, the construction of any specific Hephaestus instance is controlled by a feature configuration relative to Hephaestus' feature model and the corresponding configuration knowledge; see module *HplAssets.Hephaestus.MetaData*.

The required metaprogramming operations are of different complexity; see module *HplAssets.Hephaestus.MetaProgramming*. We begin with the simpler operations. There is an operation *setModuleName* to modify the module name so that the module of the empty product can be renamed into a module for the emerging product, say Hephaestus instance. There is an operation *addImportDecl* to add an import declaration to the main module of the emerging product; this operation is needed to incorporate any additional kind of asset. There is an operation *addField* to extend a given record type with a field; this

operation is needed in the extension of Hephaestus' data types *SPLModel* and *InstanceModel*. There is also an operation *addConstructor* to extend a given algebraic data type with a constructor declaration; this operation is needed for assembling Hephaestus' data type *TransformationModel* for transformations on possibly different forms of assets. (We note here that *addConstructor* is deliberately limited to only support the addition of a constructor with a single constructor component and to actually reuse that component's specified type for the constructor's name.)

The addition of fields and constructors is relatively straigthforward at the type level, but we need additional, non-trivial operations that transform functions that readily use the affected types. There is the operation *initializeField* which modifies all expressions for record construction for a given record type such that a given field is initialized by a constant (say, a function name with assumed zero arity). Other forms of reaction to added fields are conceivable, but the given form turned out to be sufficient in the experiment. There is also an operation *addUpdateCase* which extends function definitions by a case in reply to a previously added constructor. Different forms of adding cases are conceivable. The following, non-trivial form was required in the experiment.

Addition of a constructor is needed for *TransformationModel*, which in turn is to be used in a *transform* function that essentially interprets transformation models ('terms'). The type of the function is this:

```
transform :: TransformationModel
          -> SPLModel
          -> InstanceModel -> InstanceModel
```

The idea is here that the function perhaps case discrimination on the *first* argument and essentially delegates to a more specific transformation function that readily handles the given transformation on the further arguments at hand. An additional complication arises from the fact that the more specific transformation function would not be able to operate on the composed types *SPLModel* and *InstanceModel*. For example, consider a Hephaestus instance that should support use cases and business processes. Then, the following *transform* function has to be synthesized; see module *HplProducts.Test*:

```
transform (UseCaseTransformation x0) x1 x2
  = x2{ucm = transformUcm x0 (splUcm x1) (ucm x2)}
transform (BusinessProcessTransformation x0) x1 x2
  = x2{bpm = transformBpm x0 (splBpm x1) (bpm x2)}
```

Hence, the operation *addUpdateCase* must add cases such that indeed case discrimination is performed on the first argument, additional arguments are *unwrapped* when being passed to the function on the RHS, and the result obtained from the function on the RHS must be used to update the last argument.

5 Future work

- The experiment does not properly support existing forms of assets such as use cases, process models, etc. Additional complexities are likely to arise when moving from mockups to real code.
- Additional features such as import from and export to different formats or GUI support have not been considered. Additional complexities will arise. In particular, additional metaprogramming support may be required.
- The product derivation is naive in so far that relevant forms of assets are simply imported by irrelevant forms are not removed. That is, Hephaestus instances would contain much 'dead code'. This would be easily addressed by refining the metaprogramming approach to take care of the modules that need to be physically included into the final product. (Alternatively, a post-processing approach for dead-code elimination could be applied.)
- The metaprogramming operations are just implemented up to a level of proof of concept. For instance, they may have issues with variable capture.
- Static correctness of the Hephaestus product line has not been addressed. It would be possible though for a feature model such as Hephaestus' one to exhaustively construct all products and check them with the Haskell system.
- Alternative approaches may need to be investigated. For instance, we could consider pre-processing, CIDE, Feature House, or the use of type classes.