Paradise: A two-stage DSL embedded in Haskell

Lennart Augustsson Howard Mansell Ganesh Sittampalam
Global Modelling and Analytics Group, Credit Suisse
{lennart.augustsson,howard.mansell,ganesh.sittampalam}@credit-suisse.com

Abstract

We have implemented a two-stage language, *Paradise*, for building reusable components which are used to price financial products. Paradise is embedded in Haskell and makes heavy use of type-class based overloading, allowing the second stage to be compiled into a variety of backend platforms.

Paradise has enabled us to begin moving away from implementation directly in monolithic Excel spreadsheets and towards a more modular and retargetable approach.

Categories and Subject Descriptors D.2.13 [*Reusable Software*]: Domain engineering

General Terms Languages

1. Introduction

Our group within Credit Suisse, the *Global Modelling and Analytics Group* (GMAG), is responsible for developing the quantitative models that are used to value financial products across the Securities Trading division of the bank. Typically, these models are implemented in C++ for efficiency reasons. Their use falls into two broad categories: *valuation* and *pricing*. The former is concerned with tracking the value of transactions that are already on the bank's trading books, and is typically done in IT systems which calculate aggregate values and risk across an entire portfolio. The latter activity is carried out by sales and trading staff in a course of their day-to-day work; they need to be able to quickly value proposed transactions and they require user-friendly tools that integrate well with their normal workflow to do so.

Historically, such tools, known as *pricing models*, have been delivered by manually constructing Excel spreadsheets which call the C++ code packaged as Excel addins. From the point of view of our users, this is a convenient platform as they are typically already familiar with it, and it is easy to make minor local modifications, or to use empty parts of the worksheet as scratch space for their own calculations.

In some ways, Excel is also a convenient development platform for GMAG: it is its own integrated development environment, and there is no compile-run cycle to go through when making changes. Over a period of many years, GMAG has developed a large set of addins to augment the built-in functionality of Excel, as well as specialist financial calculations (*e.g.*, getting a discount curve), a variety of general-purpose data structures and routines have been added, including support for arrays, relational algebra, higher-order and anonymous functions, and combinators such as *map* and *fold*.

However, this approach also has significant pitfalls. Excel spreadsheets are essentially binaries and therefore cannot be placed under any effective version control. They also lack any reasonable notion of modularity, and thus re-use of pieces of a spreadsheet is accomplished with copy-and-paste, with all the obvious maintenance headaches that this entails.

We are therefore replacing this approach with a domain-specific language for developing pricing models, which we have named Paradise. It is embedded in Haskell, and makes heavy use of various features such as type classes. Using Paradise allows pricing models to be built up from reusable components that can be shared between many different models.

Paradise programs are not directly executed. Instead, they are compiled into an appropriate implementation platform in a manner similar to Pan (Elliott et al. 2003). Currently, our target platforms are Excel spreadsheets and .NET applications. In future, we may target yet more diverse platforms such as web browsers.

2. The Paradise Language

Paradise is actually two separate languages built around the common model-view-controller architectural pattern. The first language implements the model and is used to describe the calculations and dataflow of the pricing model. The second implements the view, describing the layout of the user interface. The controller is implemented once for each backend and does not need to be customised for individual pricing models.

The model and view are connected by defining a Haskell datatype which describes the data that could appear in the user interface.

Here is a simple example (real code often has 20-100 fields instead of three) of a Paradise program:

```
data Adder = Adder {
    x :: E Double,
    y :: E Double,
    z :: E Double }

adder = do
    x <- input 2
    y <- input 3
    z <- output (x+y)
    return Adder{..}

instance Viewable Adder where
    view Adder{..} = grid [[label "x",   view x],
                           [label "y",   view y],
                           [label "x+y", view z]]
```

Here, the datatype *Adder* defines a type of a component in which x, y and z are all values that could be presented to the user. The model *adder* constructs an *Adder* in which x and y are *input cells*, *i.e.*, something that the user is allowed to change, with initial values of 2 and 3 respectively, and z is an *output cell*, *i.e.*, something which is computed and presented to the user (but not directly changable), whose value tracks the sum of x and y. The construction must be done inside a state monad so that we can assign unique (internal) identifiers to different cells, as the user will expect to be able to interact with them separately.

Finally, we instantiate the type class *Viewable* and the function *view* of type $Adder \rightarrow View$ to define the default view for any *Adder* component; all of x, y and z are presented to the user with appropriate labels. We could also have implemented other, non-default views by defining another function of type $Adder \rightarrow View$;

we might want alternate layouts or simplified versions that omitted certain cells.

One piece of syntax that may be unfamiliar in this Haskell code is *Adder*{..}. This provides a shorthand for the rather tedious process of explicitly listing all the fields in the *Adder* record; in this simple example this would just have been a matter of writing *Adder*$\{x = x, y = y, z = z\}$, but with bigger components with more descriptive field names, this quickly becomes a real nuisance. The {..} syntax can be used in both expression and pattern context: in expression context, it defines a record value by matching up the field names with the current set of variables in scope, and in pattern context, it brings all the fields into scope with their own names.

This extension to GHC was originally implemented by us for use in Paradise, but it has also been adopted by the GHC developers and will be available in GHC 6.10 using the *RecordWildCards* language extension.

The final point of note in this example is the types of x, y and z. Recall that Paradise programs are compiled to specific backends such as an Excel spreadsheet or a .NET application. This is achieved by making Paradise a *staged* language. In the first stage, a Paradise program is compiled and run as a normal Haskell program. The execution of this first stage generates the second stage, (*i.e.*, the spreadsheet or .NET application) that the end-user will actually interact with.

Since x and y are designated as input cells, their values will not be available until the second stage. So $x + y$ cannot simply be implemented as the normal Haskell $+$ operator since that is only available at the first stage. Instead, it has to become the $+$ of the target platform. We achieve this by taking advantage of the fact that $+$ in Haskell is actually a member of the *Num* type class, which allows us to substitute our own implementation if we also define our own datatype. This is where the E comes from in the types of x, y, and z; values of type $E\,a$ are actually abstract syntax trees that will represent computations that can be done in the second stage and produce a value of type a at that time. So x is a *Double*, but one that will only be available in the second stage. Although the actual Haskell *Double* type is not available at that point, some corresponding run-time type will be.

More complicated components can be constructed by composition of simpler ones. For example, we could plug the adder defined above into something bigger by defining one element to have type *Adder* and calling the *adder* function to construct that element.

3. Paradise and Haskell

This section describes in more detail how Paradise takes advantage of being embedded in Haskell.

When defining a typed domain specific language you have to define a type system for the DSL as well as the various language constructs. For an embedded DSL it is rare that the type system of the host language is an exact match for the type system of the embedded language. Using Haskell type classes it is possible to customize the embedded type system to a large degree. For example, what is the type of the *input* function in the example above? It takes an initial value and returns an input of the same type as the initial value in some monad (A). The initial value can itself be "dynamic", in which case the input value will track changes in this initial value until the user types something into the input cell itself, so the type is $E\,a \rightarrow A\,(E\,a)$. But as is typical for an embedded language this type is too general. It is not the case that you can have an input of any type $E\,a$; only certain types make sense. In the case of Excel the only types you can have in an input cell are *Double*, *String*, and *Bool*. So the actual typing is *input* :: $(\mathit{CellInput}\,a) \Rightarrow E\,a \rightarrow A\,(E\,a)$, where *CellInput* is a class that restricts a to the types that make sense.

3.1 Overloading

We have already mentioned Paradise's use of overloading. As well as overloading the numeric operators, Haskell allows us to overload numeric literals. Therefore, the literals 2 and 3 in our *adder* example also have E types, because the type of *input* forces them to. In addition, the expression $x + 1$ might be an expression that runs at either the first or the second stage depending on the type of x. Indeed, if x is suitably polymorphic, then $x + 1$ will be too.

We also wanted to be able to overload string literals, but in Haskell such literals are always of type *String*. We therefore extended GHC slightly to remedy this limitation. The extension, which has been released in GHC version 6.8 with the name *OverloadedStrings*, adds an *IsString* class and changes string literals to have type $\mathit{IsString}\,a \Rightarrow a$. The *IsString* class has a single method, $\mathit{fromString} :: \mathit{String} \rightarrow a$, which is analogous to the $\mathit{fromInteger} :: \mathit{Integer} \rightarrow a$ method of *Num* that underlies integral literal overloading. This extension is not only useful for DSELs, but also for libraries like ByteString which implement a more efficient string type where you still want to be able to use ordinary string literals.

Our language also supports functions which will run at the second stage. This facility means that our own versions of standard higher-order combinators such as *map*, *fold*, etc., are also available at the second stage. (However, Haskell lists are not available then. Instead we provide an array type that fulfils a similar purpose, but is strict.) Of course, we need a way for users to actually write such functions in their code. The approach we have taken is based on higher-order abstract syntax (Pfenning and Elliott 1988; Kamin 1996); the users write normal Haskell functions whose argument and return types are E types. Since the E type is actually implemented at the first stage by an abstract syntax tree, we can instantiate the arguments with fresh variables. Doing this gives us a representation of the result in terms of those variables, and thus an explicit representation of the function. This technique relies on the E type being abstract, so that users cannot write functions which actually inspect its representation (in any case, such "functions" would not be implementable at the second stage).

3.2 Phantom Types

Paradise uses *phantom types* (Leijen and Meijer 1999) to implement the E type. Although the types that we expose to users are safe in the sense that they cannot use an $E\,\mathit{Int}$ where an $E\,\mathit{String}$ is expected, internally the type $E\,a$ is implemented by the unparameterised type, *Exp*. It is therefore incumbent on us, as the developers of Paradise, to make sure that when we expose operations to the user, we do so with the correct types.

An alternative approach would have been to use *GADTs* (Peyton Jones et al. 2006) to implement the E type. This would have allowed us to maintain type safety in our internal implementation as well. We chose not to do this partly because GADTs are relatively new and have yet to be standardised, and partly because it would have prevented us, as the Paradise developers, from taking various implementation shortcuts that, while somewhat unsafe, were very convenient. For example, the notion of "type" in Excel is much more limited than in Paradise, and there are therefore various coercions in Paradise that are no-ops in Excel. Late in the backend, we want to drop these coercions, but if we were using GADTs we would have to use a different representation when we did so. The unsafeness is not exposed to users unless we make a mistake.

3.3 GHC extensions

We make significant use of GHC extensions to the Haskell 98 language. In most cases this is simply for convenience of implementation and is not crucial to the design of the Paradise language; for example, we go beyond the normal Haskell 98 rules for class contexts, and we make heavy use of "convenience" features such as scoped type variables, pattern guards, and empty data declarations. We also use rank-2 types, multi-parameter type classes with functional dependencies, and GADTs.

By default, Paradise programs make use of the two extensions that we added ourselves and have already mentioned (overloaded strings and record wildcards), as well as the extended defaulting rules provided by GHC. We also use the *Data.Typeable* type class so that the implementation can use runtime typecasts in one particular location, and we use a GHC extension to allow users to automatically derive instances of this class when required. An alternative to *Data.Typeable* would have been to have used the type-safe references provided by the *ST* monad, but using these references would have involved "infecting" large chunks of user code with the phantom type parameter *s* that *ST* requires.

In some cases our users have also found it convenient to make use of Template Haskell (Sheard and Peyton Jones 2002) to simplify the production of boilerplate code, although where possible we try to provide abstractions that avoid the need for Template Haskell, which we tend to regard as a "last resort."

4. Discussion

4.1 Overloading

Without type classes, Paradise would be an unwieldy language—for example, we would need four different syntactic forms for literals and operators over each of the numeric types we support (*Int* and *Double* at the first stage, and their *E* variants at the second stage)—and it is doubtful that it would have been designed in the same way.

However, the overloading is still more limited than we would like, even after our addition of overloaded string literals. For example, we cannot overload the **if** . . . **then** . . . **else** operator, nor can we overload pattern-matching (nor allow the user access to algebraic datatypes in any other way). The same applies to list syntax (both literals and comprehensions). Our use of **do**-notation is also limited by the difficulty of using *restricted monads* in Haskell.

Of course, the full power of Haskell is available at the first stage, which can be used in constructing the *structure* of the component. The restrictions are only on the second stage, where the actual computations done by the component occur.

Even with numeric overloading, we run into some problems with the standard Haskell type class hierarchy. This makes *Eq* a superclass of *Num*, *i.e.*, all numbers are expected to have an equality operator $(==)$ that returns *Bool*. Unfortunately, we cannot provide a correct implementation of such an operator for a stage 2 type as their equality simply cannot be decided at Haskell runtime. In theory, any library function that is overloaded on *Num* might make use of $(==)$, but luckily, in practice we have found that very few do.

There are a couple of different ways that this problem could be solved. Firstly, the *Eq* superclass constraint could be removed. This would be slightly less convenient for "normal" users of Haskell, since they would no longer be able to rely on $(==)$ existing for all numbers, but it would mean that the constraint would become explicit in the signature of any functions that really did need this operator, avoiding the risk of runtime misbehaviour. Removing the *Eq* (and *Show*) superclass constraint for *Num* would also make sense from a theoretical point of view. There are numeric types, *e.g.*, constructive real numbers, for which there is no decidable equality, so having an *Eq* superclass makes no sense.

An alternative would be to introduce boolean overloading into the standard library, so that all functions like $(==)$ were overloaded on the return type as well as on the argument types, and constructs such as **if** . . . **then** . . . **else** were also overloaded. This would be ideal for us since even functions that required equality, would then work "out of the box" in our setting. However the extra overloading would also cause inconvenience for normal users because more types would become ambiguous and require a type signature to fix them. It would also require *Eq* to become a multi-parameter type class with one parameter for the type being compared and one for the result, as well as introducing a type class for boolean types.

The Haskell numerical class hierarchy has other problems similar to *Eq*, *e.g.*, *toInteger* in *Integral* and and *properFraction* in *RealFrac*.

We have opted for using our own multiparameter *Eq* class, but otherwise use the standard prelude. GHC allows you to replace the prelude, but we decided that maintaining our own prelude was too much of a burden.

4.2 Naming

One issue with our approach is that the variables in a Paradise program are all native Haskell variables, and thus their names are only available to GHC and not to our generator that produces the second stage program. This can make debugging the second stage something of a challenge. This problem has been ameliorated by using a preprocessor that inserts calls to an annotation function, making the names available to our generator as well. An alternative would have been to use Template Haskell, but this would have meant wrapping large blocks of user code with quotations, which we were reluctant to do.

4.3 Performance

Generally speaking, the performance of GHC-compiled code has been adequate for our needs, although we have had to take some care in a few parts of our implementation to ensure that our data structures "stream", *i.e.*, that the computations are lazy enough that we do not need to construct all of a data structure before we can start writing it to file. Without this, it is possible for the first stage to exhaust all available memory, and we observed this with some large models. Note that in general, the two-level nature of Paradise makes us somewhat decoupled from actual Haskell performance; only developers using Paradise run Haskell code, whereas the end users run what the Haskell code generated.

One significant issue occurs if we write something like **let** $z = x + y$, where x and y have E types, and then use z in multiple places. In the Haskell world, this produces a structure in which z is shared, and on our target platforms we can implement such sharing by defining a temporary. However, since sharing in Haskell is not *observable* (and not even guaranteed by the language standard, though all implementations do it), we cannot easily convert from one to the other. This problem was also encountered in Lava (Bjesse et al. 1998), and two solutions were proposed: one was to use a monad to regain sharing and the other was to introduce impure observable sharing. In our setting, since we construct components inside a state monad in any case, we simply introduce a combinator inside the monad that allows users to specify explicit sharing, and recommend that they use that combinator in place of **let**.

Any sharing that is lost by using **let** will be recovered by the Paradise implementation by doing common subexpressions elimination; the drawback of using this approach alone is that because we cannot observe the sharing, the intermediate data structures can get huge before CSE is able to collapse them.

4.4 Recruitment

One consideration when embarking on a functional programming project is whether you expect to be able to hire functional programmers of a high standard. Clearly, the number of functional programmers looking for commercial work is quite small and so one might expect good ones to be hard to find. Our experience has been very positive in this regard. Our willingness to embrace academic research and the potential to work on interesting problems has attracted some extremely capable Haskell programmers. We have also found that the "average" Haskell programmer we interview tends to be of a much higher standard than the "average" C++ programmer.

4.5 Organisational Acceptance

Most organisations are fairly conservative in adopting technologies that are new to them, and usually want to see widespread use (or anticipated use) of that technology in other, similar, organisations. So one might find it surprising that we embarked on a Haskell project when (to our knowledge at the time) no-one else in the finance sector was doing so. Our initial adoption of functional programming happened almost by accident. We have a long history of using Microsoft Excel spreadsheets to "glue together" code written as C++ addin functions. This approach has served us quite well in some regards, particularly in terms of being able to build new pricing models with very quick turnaround. It is, however, rather weak when considered in the context of what can be done in a programming language. In particular, change control, static type checking, abstraction and reuse are almost completely lacking.

In 2005, we set about trying to enhance the spreadsheet environment with various tools to address these deficiencies, and since the spreadsheet is essentially a pure functional programming environment, we naturally looked to functional programming languages for inspiration. This led to the integration of higher-order functions into the Excel environment, but through fairly unnatural techniques (since they had to be implemented as Excel addins). At the same time, we recognised our lack of expertise in the field so we chose to hire some functional programmers. At this point, we were not expecting to use any particular functional language, but we wanted to bring the right perspective to the problem.

Our new hires certainly enabled us to make the spreadsheet programming environment better, but also helped highlight how it was still fundamentally lacking. We would never get static type checking, simple change control, and lightweight abstraction in a spreadsheet environment. A simple proof-of-concept of what would become Paradise convinced us to proceed with the project. The prototype enabled us to generate spreadsheets from clear, concise Haskell code, and held the promise of targeting other UI platforms in the future. This coincided with an increased focus on building easier-to-use pricing models from our sales and trading department, which led us to sponsor the project.

4.6 User Feedback

Some comments from modellers using Paradise:

> In my experience, the static type checking along with stage-1 runtime checks mean that, most of the time, if the model compiles, it works. However, this is hampered by the lack of type safety in Excel and our COM-based analytics. In particular, the computer-generated models aggressively expose quirks, inconsistencies, and edge cases in our analytics which would normally be hand-wired into spreadsheets. These are bugs we have either learned to avoid or have never discovered.

> Positive experience:
>
> 1. Code reusability, templating possibility, static type-checking, and source code control are great features to have, compared to the traditional way of developing Excel models.
> 2. I also like the fact that we can exploit Haskell's power during compile time to easily generate large but structured components.
> 3. Common subexpression elimination is very nice.
> 4. Overall, learning and using Paradise/Haskell have been a very enjoyable experience to me.
>
> Things that perhaps could still be improved:
>
> 1. Doing any change to the model (if done through Paradise) will always require an extensive test, even if the change is a minor one.
> 2. The problem above is made severe by the fact that Paradise and the libraries developed by modellers still undergo a lot of changes.

4.7 Impact

Over the space of a year and a half, we developed the core of Paradise and successfully completed some challenging pilot projects. This involved getting modellers in our group to work in Haskell. Most modellers have fairly limited programming experience and usually no knowledge of functional programming. While we experienced a fair degree of scepticism, generally we have found modellers to be quite accepting of Haskell, and developing in Haskell itself has not presented any significant issues. Our main issues have revolved around two areas—firstly, getting the combinators in our Domain Specific Language right, and secondly, being too optimistic about what can realistically be implemented in Excel with acceptable performance.

The combinators have evolved considerably, and continue to do so, which is something that must be expected with any Domain Specific Language. Dealing with the limitations of Excel has been a lot harder. When one adopts a generative approach it is tempting to generate arbitrarily complex code, but we have to exercise care to ensure that we generate Excel constructs that perform well, and sometimes we have to sacrifice generality for performance.

We now have a reasonable number of pricing models implemented using Paradise, and there is a large degree of re-use between them. When one wants to build a new pricing model which has many similar features to existing models, we can usually do these pretty quickly, since we can re-use code. However, when we want to make a very quick, simple change to an existing model, the turnaround time is longer than using Excel directly. This is probably the key trade-off between a dynamic environment like Excel and a more traditional programming environment.

5. Acknowledgements

We would like to thank the anonymous referees for their valuable feedback, Jessica Augustsson for proofreading, and Mat Peacock and Stephen Prajna for their user feedback.

References

P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. ICFP '98*, pages 174–184. ACM Press, 1998.

C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

S. Kamin. Standard ML as a meta-programming language. Technical report, University of Illinois at Urbana-Champaign, 1996.

D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proc. 2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, 1999.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. ICFP '06*, pages 50–61. ACM Press, 2006.

F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. PLDI '88*, pages 199–208. ACM Press, 1988.

T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *Proc. Haskell Workshop '02*, pages 1–16. ACM Press, 2002.