EXTENDED ABSTRACT

Step-by-step tutorial of good abstraction design

Philip K.F. Hölzenspies

Motivated by wanting to teach good abstraction skills in programming practice to moderately unseasoned programmers, I developed and taught a narrative to argue against ad hoc abstractions. The central idea to get across to students is, that the abstractions you introduce while programming should somehow come together in a coherent system of abstractions. Individual abstractions should be clearly defined in the same terminology as all other abstractions and their distinctions should be carefully thought through. Strongly typed functional programming languages help reason about abstractions in an illustrative way. Because we can reason about programs in these languages on both a value- and a type-level, students' reasoning can easily be driven towards abstractions in question-driven, highly interactive tutoring sessions. As a pleasant side-effect of this approach, students are shown how (propositional and constructive) logic, type systems, reasoning about programs and abstractions hang together. Monads just happen along the way.

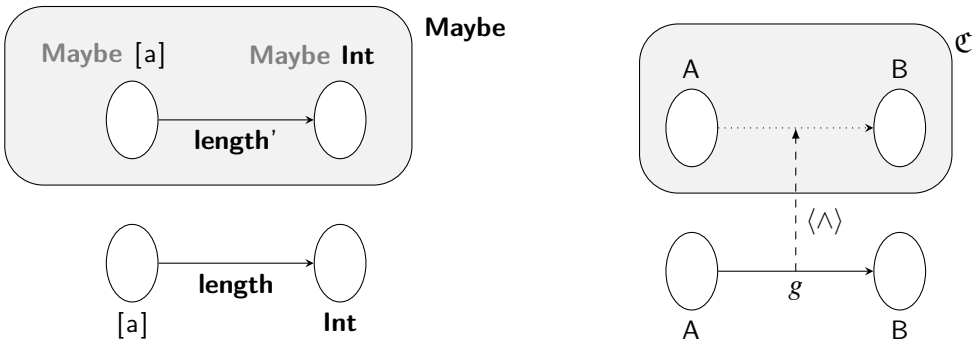
1 Introduction

The motivation for a tutorial in programming with abstractions was one that even relatively novice programmers recognise. While programming, a feeling of vague recognition often arises; the feeling that we have written something similar before. Even if we find our old code that came to mind, it turns out more often than not that, even though indeed very similar, the code we wrote in another context can not simply be reused in this new scenario, due to slightly different data-structures or other constraints. It is tempting to try and find abstractions that cover the cases that we have seen and maybe the cases that we expect in the (near) future. These, I refer to as *ad hoc abstractions*.

Ad hoc abstractions often prove to be quite weak, i.e. it does not require a large number of cases to find a case where they fail to capture the required behaviour. Instead, we would do better to think about the abstractions we want in the case at hand and then try to place them in the "world of abstractions" independent of the case from which they arose. In other words, in relation to the abstractions that we already have, they should fill a previously overlooked gap. Sometimes, such an examination of the world of abstractions shows us that the abstractions we already have suffice to express the type of behaviour we want. In this case, reasoning about abstractions structures our search for solutions. When existing abstractions can not express the behaviour we want, we can find new abstractions that are more likely to stand the test of time. So far, reasoning about a tutorial on programming with abstractions has itself turned out to be a rather abstract affair. The narrative with which to explain this process to students is discussed in more detail in section 2.

The group of students that followed the experimental version of this course was very diverse. It consisted of first- and second-year undergraduate students of computer science and mathematics/physics, and Ph.D. students in computer science. The common prerequisite knowledge assumed was the basic material discussed in the introduction to functional programming coursework [1, Ch. 1–7]. Some had not treated type systems or natural deduction in logic yet, whereas others had difficulty recalling specifics. Thus, the beginning of the course covered—in order of treatise—propositional logic, natural deduction, constructive logics, untyped lambda calculus, propositions as types and simply typed lambda calculus.

To appear in EPTCS.



(a) Specific case (b) Generalisation

Figure 1: Depiction of values outside of and in a context

Of course, the material covered was grossly incomplete and very much geared towards the final goal of having at least a passive understanding of type inference.

As anyone who has taught propositions as type knows, it is (i) woefully lacking good standard lecture material and (ii) a real eye-opener for many students. By treating all these topics in quick succession (one or two sessions on each topic, one session per week, with homework), students experienced a number of these eye-openers in terms of what these formalisms are used for and how they relate to each other. Having spent a final, summarising session on how all of these topics come together in (very simple) Haskell programs, the students had sufficient know-how to proceed to reason about programming abstractions outside of their intended use-cases.

2 The Narrative

From their software engineering courses, students know about the virtues of reusable code. Therefore, they are presented with this scenario: Suppose you already have a library of functions on "ordinary values." You can do arithmetic with numbers, test whether a character is lower or upper case, find the length of a list, etc. Now, suppose these values are placed in a *context*, for example, a context expressing whether or not an error has occurred. Thus, instead of ordinary values, your values are either "an error has occurred" or "all is well and 'this' is your ordinary value." You must now redefine all your functions to work on error-or-value values, instead of directly on values. In other words, you must give equivalent function definitions that know about error annotation. Finally, suppose that you create different contexts, e.g. non-deterministic values, values dependent on a state (with simple parsers as an example), etc. You must now redefine all your functions for all your contexts.

These example contexts are made concrete by giving Haskell definitions for them. Since the choices of type definitions and type classes (and instances of the former in the latter) in Haskell also take into account efficiency and the wider programming practice, the examples provided are not always compatible with definitions in Haskell's prelude. If some of these examples look as if containing errors, this is why. So for the examples mentioned above, the students are presented with these definitions:

```
data Maybe  a = Nothing | Just a    -- Error contexts
data NonDet a = ND [a]              -- Non-determinism
data Parser a = Parser (String → [(a, String)])
```

Next, students are presented with the graphic depicted in figure 1(a). This illustration serves as an example to introduce this graphical notation and makes the relation between 'ordinary values' and values in a context more obvious. As an exercise, the students must implement **length**' and a few other examples on the concrete contexts they have seen as examples. This both gives them a better understanding of the intended semantics of these contexts and lets them recall the aforementioned feeling of writing the same code in a different way over and over again. After these exercises, the illustration is generalised (as shown in figure 1(b)), using A and B to represent concrete types (i.e. non-variable types in the language sense, but representative of any type the student may want to envision) and $\mathfrak{C}$ to represent a similarly concrete context. The types in the context are no longer explicitly labelled as $\mathfrak{C}$ A, because their position inside the shaded context makes this sufficiently clear. The function inside the context is now drawn dotted, because it is never explicitly defined, but rather the result of a projection of the 'normal' function g into the context. The operator providing this projection is drawn dashed. All operators that implement abstractions are drawn dashed in this notation.

2.1 Abstractions

At this point, we begin discussing abstractions as things not tied to the specific cases that we have seen thus far. Having implemented a few versions of previously existing functions for different contexts, students are now already under the impression that there exists a generalisation to apply a function on regular values to those in context. However, this generalisation only generalises over the functions, not over the contexts, as the students have discovered in implementing the save function on different contexts and different functions on the same context. Thus, it is a property of the context whether or not 'normal' functions can be applied in context. This is where abstractions are introduced as type classes for contexts.

The Haskell prelude has a lot of historically motivated choices that I did not consider consistent from a pedagogical point of view. Since the students may well be using real Haskell libraries in their near future, I opted to not use Haskell class names and vary the notation somewhat. Inspired by the applicative notation[3], all operators that implement an abstraction on a context are written as an operator between angled brackets. Furthermore, only one (dashed) arrow is added in the diagram per abstraction. In other words, the type classes introduced to implement abstractions always contain precisely one operator or function. Students are advised that this is not necessarily a good strategy for efficient implementations, but merely a way to reduce complexity for the tutorial examples.

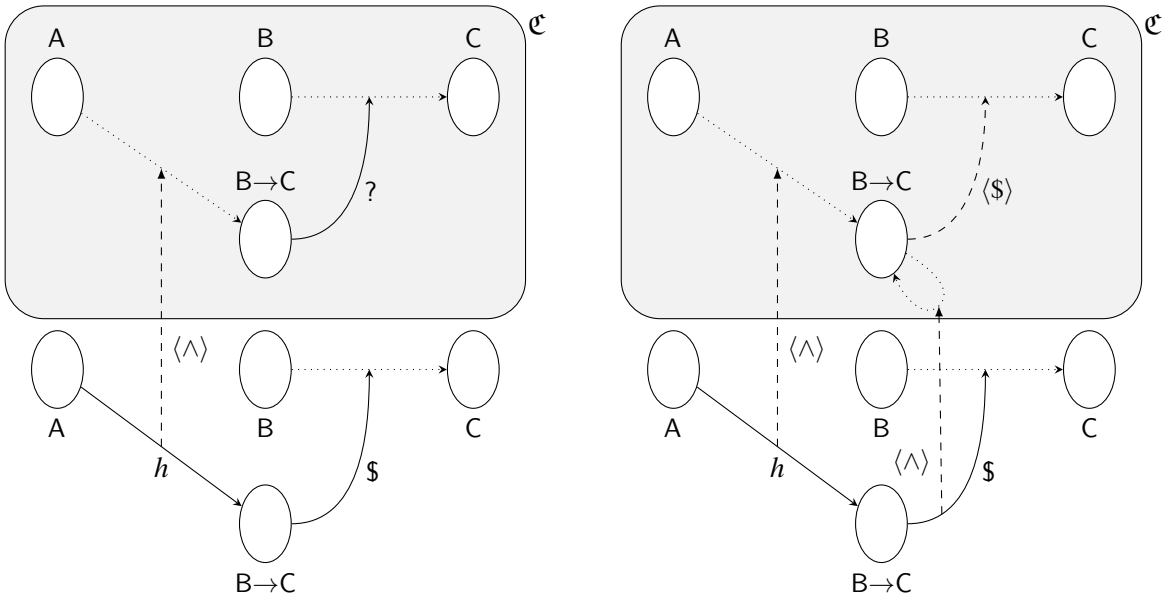
Having discussed types extensively in previous sessions, students can now give the types for the type class that implements the operator depicted in figure 1(b):

```
class Into c where (⟨∧⟩) :: (α → β) → c α → c β
```

The students must now give instances of this type class for all type definitions discussed so far. In the small tutorial group in which I tried this method of explanation, it was quite productive to come up with these instances with the entire group. To this end, I only guided the discussion by asking questions (e.g. What type does your solution have?) and engaging with the people that were not yet very sure. The group as a whole produced instances on the board (the entire tutorial was done on board and paper; without computers).

2.2 Breaking abstractions to find new abstractions

All examples up to this point have dealt with unary functions. The graphical notation used initially seems to pose a purely denotational problem. Students accept without problems that the type of a binary function



(a) Binary function conundrum

(b) New abstraction for application

Figure 2: Breaking the abstraction with binary functions

applied to one function is again a function, i.e.

$$\frac{\Gamma \vdash h : \mathsf{A} \to \mathsf{B} \to \mathsf{C} \qquad \Gamma \vdash x : \mathsf{A}}{\Gamma \vdash h\ x : \mathsf{B} \to \mathsf{C}}\ \text{Application}$$

Having done basic functional programming, the students should have no problem with the first-class citizenship of functions. The only problem with the graphical notation is that a function is simultaniously an object in a set of functions and a mapping between two other sets. I have chosen to use Haskell's explicit function application operator ($) to denote the mapping from an object in the set B→C to the mapping between the sets B and C, as shown in figure 2(a). Students were then asked how to apply the function h in the figure to two arguments. Many students came up with the solution to project $ itself into the context $\mathfrak{C}$ using $\langle\wedge\rangle$. The next question I asked of the group was to type check this option. Although by no means easy at this stage, most students do construct this inference:

$$\frac{\Gamma \vdash \$: (\overbrace{\alpha \to \beta}^{\varphi}) \to \overbrace{\alpha \to \beta}^{\psi} \qquad \Gamma \vdash \langle\wedge\rangle : (\varphi \to \psi) \to \mathfrak{C}\ \varphi \to \mathfrak{C}\ \psi}{\Gamma \vdash ((\$)\langle\wedge\rangle) : \mathfrak{C}\ (\alpha \to \beta) \to \mathfrak{C}\ (\alpha \to \beta)}\ \text{Application}$$

and from the figure, they come up with the desired type, viz.

$$\Gamma \vdash ? : \mathfrak{C}\ (\alpha \to \beta) \to \mathfrak{C}\ \alpha \to \mathfrak{C}\ \beta$$

Although students see these two types do not match, asking them to draw their inferred type in figure 2(a) can lead to confusion. Drawing it for them (figure 2(b)) does clarify and it makes abundantly clear that a new abstraction is required, which the students themselves can both specify and implement for the type definitions already given. Unfortunately, $\langle\$\rangle$ is used in the applicative notation [3] for functors, which correspond with our abstraction `Into`. However, because of the correspondence with the $ on ordinary values, I have decided to use $\langle\$\rangle$ for application in a context nonetheless.

```
class Apply c where (<$>) :: c (α → β) → c α → c β
```

2.3 Getting into context and further context operations

Having discussed how to apply functions on values in context, it is now high time to discuss how these contexts come about in the first place. First, we treat specific functions that introduce specific contexts. These examples use the semantics of the corresponding contexts directly, e.g.

```
invert x  = if x == 0 then Nothing else 1 / x :: Maybe Float
pickOneOf = ND                                :: [a] → NonDet a
readChar  = Parser \(c:cs) → [(c,cs)]         :: Parser Char
```

Next, we discuss how to insert ordinary values into contexts or how to generalise the expressions of error occurring, which both require new abstractions

```
class Point   c   where point :: α → c α
class Failure c e where fail  :: e → c α
```

Failure is the first new abstraction that does not apply to all the contexts discussed. Although some students will immediately try to 'fix' the definition of these contexts. This is a good point to discuss how not all abstractions are universally applicable to all contexts. By using a multi-parameter type class for failure, which parametrises over the representation of errors, it can be further illustrated to students that some contexts that capture failure have no way to express errors explicitly (e.g. **Maybe**), whereas others do (e.g. **Either** e). This is revisited in section .

In further discussion of examples, new abstractions arise in a fashion similar to that discussed in section 2.2. Most notably, when trying to (non-deterministically) pick a number between one and a non-deterministic number, nested contexts occur naturally, i.e.

```
pickOneOf <∧> ((\x → [1..x]) <∧> pickOneOf [10..20])
    :: NonDet (NonDet Int)
```

This inspires yet-anther-abstraction, which students do not perceive as something special:

```
class Join c where join :: c (c α) → c α
```

If students are now presented with any common example from monad tutorials, they examine the abstractions they have to determine whether a new abstraction is needed for the bind operation. After drawing the system of abstractions (figure 3) and (graphically) determining the type of the bind operator, they simply deduce the implementation as being a combination of $\langle\wedge\rangle$ and **join**.

From the definition of Join, students recognise that contexts are themselves manipulable objects, i.e. they are first-class to the language. Introducing them to other abstractions, such as error recovery ($\langle|\rangle$) and state aggregation (empty and $\langle+\rangle$, effectively giving rise to Monoid) has now become a natural process.

2.4 Combination of different contexts

The last stage in the tutorial is the discussion of the combination of different contexts into new contexts. To this end, students are presented with *context transformers*[1] as "contexts with a context-shaped hole in them." In terms of presentation, very little new material is required. The major contribution of this final topic comes from exercises. By doing a lot of exercises about transformers, in the terminology as used above, students can put their newly acquired insights to the test.

[1] Similar to monad transformers, but not requiring the monad constraint.

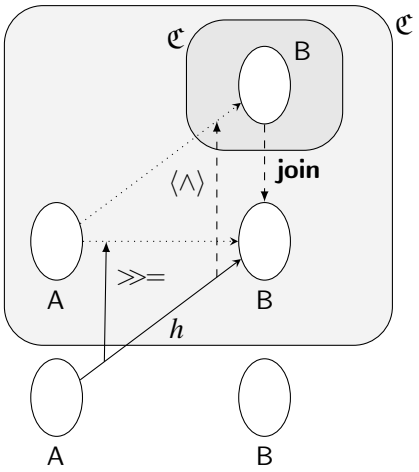


Figure 3: Constructing "bind" as combination of existing abstractions

3 Organisation of the tutorial

The material presented to students is *very* abstract. One student described the course as an abstraction over abstractions. The only way to make the material concrete for students is through a large number of exercises, all dealing with similar topics.

To keep a reasonable pace, only one abstraction was discussed per session. During the sessions, students were engaged directly by asking them questions about types and semantics. Every session, one or two instances for the newly presented type class was discussed and the remaining instances were homework for the next session. Sometimes, homework included more theoretical questions, like "does NonDet allow for the expression of errors and, if so, what is the trade-off to use this expression?"

After the tutorial, students had sufficient insight to be referred to further reading, such as Brent Yorgey's Typeclassopedia [4] and the semigroupoids package [2], both of which inspired the above narritive greatly.

Acknowledgements

I would like to thank Edwin Brady for extensive discussions on types and abstractions before many tutorial sessions. Of course, I would also like to acknowledge the students that took part in the tutorial for no credits or reward, other than wanting to learn.

References

[1] Graham Hutton (2007): *Programming in Haskell*. Cambridge University Press.

[2] Edward A. Kmett (2012): *Semigroupoids package*. Available at `http://hackage.haskell.org/package/semigroupoids`.

[3] Conor McBride & Ross Paterson (2008): *Applicative programming with effects*. *J. Funct. Program.* 18(1), pp. 1–13. Available at `http://dx.doi.org/10.1017/S0956796807006326`.

[4] Brent Yorgey (2012): *Typeclassopedia*. Available at `http://www.haskell.org/haskellwiki/Typeclassopedia`.