The Monad.Reader Issue 23

by Maciej Piróg ⟨maciej.adam.pirog@gmail.com⟩
and Ilya Klyuchnikov ⟨ilya.klyuchnikov@gmail.com⟩
and Dimitur Krustev ⟨dkrustev@gmail.com⟩
and Henrik Bäärnhielm ⟨redstar@kth.se⟩
and Daniel Sundström ⟨daniel@monkeydancers.com⟩
and Mikael Vejdemo-Johansson ⟨mikael@johanssons.org⟩
and Alberto Gomez Corona ⟨agocorona@gmail.com⟩
and Neil Brown ⟨neil@twistedsquare.com⟩

April 23, 2014

Edward Z. Yang, editor.

Contents

Editorial

by Edward Z. Yang ⟨ezyang@cs.stanford.edu⟩

This issue's Monad Reader clocks a record page count, with over a hundred pages of Haskell goodness from nine authors. Given that there's so much; I thought this issue's editorial would be best devoted to giving teasers for each of the articles herewithin.

- Ever wanted to solve a programming interview question by writing a domain-specific language on the fly? In **FizzBuzz in Haskell**, Maciej takes a whimsical look at the classic interview question, dancing through syntax, semantics and algebraic transformations to arrive at a solution that would be called elegant by most people.
- Ilya and Dimitur write in with a somewhat apologetic but very interesting tutorial about **Supercompilation**, a whole-program transformation technique. It is a nice introduction to the subject, and comes with running code to boot.
- In a break from the usual uses of Haskell, Henrik, Daniel and Mikael describe **Haskell sound specification DSL** used in service of a live action role-playing game situated in the Battlestar Galactica universe.
- Alberto talks about his web framework, **MFlow**, which provides a continuation-like method of structuring web applications, without actually opening the can of worms that is explicitly serializing continuations.
- And last but not least, Neil offers a gentle introduction to **Practical Type System Benefits**, giving a taster of some of the more intermediate features of Haskell, including generic programming, parallelism, newtypes, quasiquoting and lightweight contracts.

Take a look!

FizzBuzz in Haskell by Embedding a Domain-Specific Language

by Maciej Piróg ⟨maciej.adam.pirog@gmail.com⟩

The FizzBuzz problem is simple but not trivial, which makes it a popular puzzle during job interviews for software developers. The conundrum lies in a peculiar but not unusual control-flow scenario: the default action is executed only if some previous actions were not executed. In this tutorial, we ask if we can accomplish this without having to check the conditions for the previous actions twice; in other words, if we can make the control flow follow the information flow without loosing modularity. The goal is to have the most beautiful code!

We deliver a rather non-standard, and a bit tongue-in-cheek solution. First, we design a drastically simple domain-specific language (DSL), which we call, after the three commands of the language, Skip-Halt-Print. For each natural number n, we devise a Skip-Halt-Print program that solves FizzBuzz for n. Then, we implement this in Haskell, and, through a couple of simple transformations, we obtain the final program. The corollary is a reminder of the importance of higher-order functions in every functional programmer's toolbox.

The FizzBuzz problem

FizzBuzz is a simple game for children, and therefore a really hard nut to crack for programmers and computer scientists. To quote the rules [1]:

> Players generally sit in a circle. The player designated to go first says the number '1', and each player thenceforth counts one number in turn. However, any number divisible by three is replaced by the word *fizz* and any divisible by five by the word *buzz*. Numbers divisible by both become *fizzbuzz*.

In this tutorial, we focus on a single step of the game, that is to convert a natural number n into *fizz*, *buzz*, *fizzbuzz*, or its string representation.

There are a lot of solutions floating around the Internet, but most of them are, from our point of view, unsatisfactory. Exhibit A:

```
fizzbuzz :: Int → String
fizzbuzz n =
  if n `mod` 3 ≡ 0 ∧ n `mod` 5 ≡ 0 then
    "fizzbuzz"
  else if n `mod` 3 ≡ 0 then
    "fizz"
  else if n `mod` 5 ≡ 0 then
    "buzz"
  else
    show n
```

Exhibit B:

```
fizzbuzz :: Int → String
fizzbuzz n =
  if n `mod` 3 ≡ 0
    then "fizz" ++ if n `mod` 5 ≡ 0
                     then "buzz"
                     else ""
    else if n `mod` 5 ≡ 0
           then "buzz"
           else show n
```

Though both programs are correct with respect to the specification, Exhibit A, in some cases, performs the `` `mod` 3 `` and `` `mod` 5 `` tests more than once, while Exhibit B disperses the *buzzing* code into more than one place in the program. Meanwhile, we want there to be at most one place that outputs *fizz*, *buzz*, or the string representation, and each test to be performed only once. Outputting *fizzbuzz* should be done by executing the (one and only) piece of code that outputs *fizz*, followed by the (one and only) piece that outputs *buzz*. That is because *fizzing* and *buzzing* are two separate activities – consider the FizzBuzzHissHowl problem, where *hiss* and *howl* are printed for multiples of 7 and 11 respectively. The program design of Exhibit A and B would lead to an explosion of code complexity.

We can examine the output itself, which gives us an opportunity for yet another unsatisfactory attempt, Exhibit C:

```
(◁) :: String → String → String
"" ◁ s = s
```

```
a  ◁ s = a
fizzbuzz :: Int → String
fizzbuzz n = ((  if n `mod` 3 ≡ 0 then "fizz" else "")
             ++ if n `mod` 5 ≡ 0 then "buzz" else "")
             ◁ show n
```

The problem with this solution is far more subtle: many might say this solution is simple and elegant. Be that as it may, we do not like the fact that the $\triangleleft$ operator has to check if its first argument is empty. After all, we have already checked the conditions ‘*mod*‘3 and ‘*mod*‘5, so the third test ($\triangleleft$’s pattern matching) seems redundant from the information-flow point of view (compare Exhibit B, which always performs only two tests).

So, is out there a program that reflects the information-flow structure as Exhibit B, but, at the same time, is as modular as Exhibit C? Let’s find out!

Skip-Halt-Print and contexts

If one feels overwhelmed by the number of (better or worse) possible ways to solve such a simple problem as FizzBuzz in Haskell, they can start with a simpler language. The one we propose is called Skip-Halt-Print, and it is very imperative.

A program in Skip-Halt-Print is a (possibly empty) list of commands, which are executed sequentially. There are only three different commands:

- Skip is an idle instruction; it does nothing at all;
- Halt stops the computation; the rest of the program is not executed;
- Print `s` prints out the string `s`.

More formally, the syntax is given by the following grammar, where c denotes commands, p denotes programs, and ϵ is the empty program:

$$c ::= \textsc{Skip} \mid \textsc{Halt} \mid \textsc{Print}\ \texttt{s}$$
$$p ::= c;\ p \mid \epsilon$$

For brevity, we will give `s` also as an integer literal with implicit conversion. For example, the command Print 42 is meant to print out the string of characters `42`.

The formal semantics can be given by a denotation function $[\![-]\!] : p \to \mathit{String}$, where *String* is the set of all strings of characters. In the following, $+\!\!+$ denotes concatenation and `""` denotes the empty string.

$$[\![\textsc{Skip};\ p]\!] = [\![p]\!]$$
$$[\![\textsc{Halt};\ p]\!] = [\![\epsilon]\!] = \texttt{""}$$
$$[\![\textsc{Print}\ \texttt{s};\ p]\!] = \texttt{s} +\!\!+ [\![p]\!]$$

For example:

$$[\![\textsc{Print}\ \texttt{"studio"};\ \textsc{Skip};\ \textsc{Print}\ 54]\!] = \texttt{studio54}$$
$$[\![\textsc{Print}\ \texttt{"nuts"};\ \textsc{Halt};\ \textsc{Print}\ \texttt{"and bolts"}]\!] = \texttt{nuts}$$

For every natural number n, we construct a Skip-Halt-Print program that solves FizzBuzz for n. The building blocks for this construction are called *contexts* – they are programs with holes (one hole per context). We denote contexts by putting angle brackets $\langle - \rangle$ around the programs, while holes are denoted by the $\bullet$ symbol. For example:

$$\langle \textsc{Print}\ \texttt{"keep"};\ \bullet;\ \textsc{Print}\ \texttt{"calm"} \rangle$$

A hole is a place in which we can stick another program and get a new program as a result. We denote this operation by juxtaposition:

$$\begin{aligned} &\langle \textsc{Print}\ \texttt{"keep"};\ \bullet;\ \textsc{Print}\ \texttt{"calm"} \rangle\ (\textsc{Print}\ \texttt{"nervous and never"}) \\ &= \textsc{Print}\ \texttt{"keep"};\ \textsc{Print}\ \texttt{"nervous and never"};\ \textsc{Print}\ \texttt{"calm"} \end{aligned}$$

Two contexts can be composed, and for that we use the $\circ$ symbol:

$$\begin{aligned} &\langle \textsc{Skip};\ \bullet;\ \textsc{Print}\ 0 \rangle \circ \langle \textsc{Halt};\ \bullet \rangle \\ &= \langle \textsc{Skip};\ \textsc{Halt};\ \bullet;\ \textsc{Print}\ 0 \rangle \end{aligned}$$

What does the FizzBuzz program do? Essentially, it prints out n, unless something else (like *fizzing* or *buzzing*) happens. This behaviour is captured by the following context:

$$\mathit{base}(n) = \langle \bullet;\ \textsc{Print}\ n \rangle$$

What about *fizzing*? If only n is divisible by 3, it prints out `fizz`, but it also needs to prevent the default action from happening by Halt-ing the computation. In between Print-ing and Halt-ing anything (like *buzzing*) can happen:

$$\mathit{fizz}(n) = \begin{cases} \langle \textsc{Print}\ \texttt{"fizz"};\ \bullet;\ \textsc{Halt} \rangle & \text{if } n \bmod 3 = 0 \\ \langle \bullet \rangle & \text{otherwise} \end{cases}$$

The context for *buzzing* is analogous:

$$\mathit{buzz}(n) = \begin{cases} \langle \textsc{Print}\ \texttt{"buzz"};\ \bullet;\ \textsc{Halt} \rangle & \text{if } n \bmod 5 = 0 \\ \langle \bullet \rangle & \text{otherwise} \end{cases}$$

The program that solves FizzBuzz for n, which we call $fb(n)$, is a composition of all these contexts, which we cork with Skip:

$$fb(n) = (base(n) \circ fizz(n) \circ buzz(n))\ \textsc{Skip}$$

Examples:

$$\begin{aligned} fb(1) &= \textsc{Skip};\ \textsc{Print}\ 1 \\ fb(3) &= \textsc{Print}\ \texttt{"fizz"};\ \textsc{Skip};\ \textsc{Halt};\ \textsc{Print}\ 3 \\ fb(5) &= \textsc{Print}\ \texttt{"buzz"};\ \textsc{Skip};\ \textsc{Halt};\ \textsc{Print}\ 5 \\ fb(15) &= \textsc{Print}\ \texttt{"fizz"};\ \textsc{Print}\ \texttt{"buzz"};\ \textsc{Skip};\ \textsc{Halt};\ \textsc{Halt};\ \textsc{Print}\ 15 \end{aligned}$$

Exercise 1. For any natural number n, give a Skip-Halt-Print program that solves the FizzBuzzHissHowl problem for n.

Exercise 2. What is the formal definition of the operations on contexts described above? Show that contexts with composition form a monoid; that is, $\circ$ is associative, $(f \circ g) \circ h = f \circ (g \circ h)$, and $\langle \bullet \rangle$ is its left and right unit, $\langle \bullet \rangle \circ f = f$ and $f \circ \langle \bullet \rangle = f$ respectively.

Haskell implementation

Now, to solve FizzBuzz in Haskell, we implement Skip-Halt-Print, both syntax and semantics, together with the language of contexts. For each n, we construct the right composition of contexts as described above and then execute the resulting program.

Then, we apply a series of algebraic transformations that simplify the code into our proposed solution. By "algebraic", we mean transformations that depend only on local properties of the components, without the actual understanding of the implemented algorithm. In other words, something that can be deduced solely from the shape of the code, like the *fold* pattern, and applied by simple equational calculation.

Direct definition

The commands of Skip-Halt-Print are implemented as a three-constructor data type *Cmd*, and the program is, of course, a list of commands. We call the $[\![-]\!]$ function *interp*.

```
data Cmd     = Skip | Halt | Print String
type Program = [Cmd]
```

```
interp :: Program → String
interp (Skip    : xs) = interp xs
interp (Halt    : xs) = ""
interp (Print s : xs) = s ++ interp xs
interp []             = ""
```

Contexts are more tricky. Instead of specifying their syntax and operations, we encode them as functions from programs to programs (this technique is sometimes called *higher-order abstract syntax*). In this case, sticking the program in a context becomes Haskell's function application, and the composition of contexts becomes simply Haskell's ∘. However, note that not every Haskell function of the type *Program* → *Program* is a valid context in the sense specified in the previous section.

```
type Cont = Program → Program

fizz, buzz, base :: Int → Cont
fizz  n | n `mod` 3 ≡ 0 = λx → [Print "fizz"] ++ x ++ [Halt]
        | otherwise     = id
buzz  n | n `mod` 5 ≡ 0 = λx → [Print "buzz"] ++ x ++ [Halt]
        | otherwise     = id
base  n                 = λx → x ++ [Print (show n)]

fb :: Int → Program
fb n = (base n ∘ fizz n ∘ buzz n) [Skip]

fizzbuzz :: Int → String
fizzbuzz n = interp (fb n)
```

Interpretation is a fold

To solve FizzBuzz for n, we first build a program (a datastructure) and then interpret it (by traversing the datastructure). This calls for some deforestation – the removal of the intermediate structures! First, we notice a known pattern here: *interp* is a fold. We can rewrite it as follows:

```
step :: Cmd → String → String
step Skip      t = t
step Halt      t = ""
step (Print s) t = s ++ t

interp = foldr step ""
```

Additionally, *foldr* has the following property (see Exercise 3):

$$\mathit{foldr}\ \mathit{step}\ \texttt{""}\ p = \mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{fmap}\ \mathit{step}\ p)\ \texttt{""}$$

So, instead of writing programs like

$$[\mathit{Skip}, \mathit{Halt}, \mathit{Print}\ \texttt{"c"}]$$

and interpreting them by folding with *step*, we can write programs like

$$[\mathit{step}\ \mathit{Skip}, \mathit{step}\ \mathit{Halt}, \mathit{step}\ (\mathit{Print}\ \texttt{"c"})]$$

and interpret them by folding $\circ$. Also, we can inline the definition of *step*:

$$\begin{aligned}&[\mathit{step}\ \mathit{Skip}, \mathit{step}\ \mathit{Halt}, \mathit{step}\ (\mathit{Print}\ \texttt{"c"})]\\ =\ &[\lambda t \to t, \lambda t \to \texttt{""}, \lambda t \to \texttt{"c"} +\!\!+\ t]\\ =\ &[\mathit{id}, \mathit{const}\ \texttt{""}, (\texttt{"c"} +\!\!+)]\end{aligned}$$

Why build and then interpret? We can manually deforest the situation by fusing the two: instead of

$$\mathit{foldr}\ (\circ)\ \mathit{id}\ [\mathit{id}, \mathit{const}, (\texttt{"c"} +\!\!+)]$$

we write

$$\mathit{id} \circ \mathit{const} \circ (\texttt{"c"} +\!\!+)$$

In summary, we can define the next version of Skip-Halt-Print commands as follows:

$$\begin{aligned}&\textbf{type}\ \mathit{Program} = \mathit{String} \to \mathit{String}\\ &\mathit{skip}, \mathit{halt} :: \mathit{Program}\\ &\mathit{skip}\ \ = \mathit{id}\\ &\mathit{halt}\ \ = \mathit{const}\ \texttt{""}\\ &\mathit{print}\ \ \ :: \mathit{String} \to \mathit{Program}\\ &\mathit{print} = (+\!\!+)\end{aligned}$$

Now, our programs look like this:

$$\mathit{print}\ \texttt{"hello"} \circ \mathit{skip} \circ \mathit{print}\ \texttt{"world"} \circ \mathit{halt}$$

To execute them, we apply them to an empty string, for example:

$$(\mathit{print}\ \texttt{"hello"} \circ \mathit{skip} \circ \mathit{print}\ \texttt{"world"} \circ \mathit{halt})\ \texttt{""} = \texttt{"helloworld"}$$

We need to accordingly adjust the bodies of our contexts:

```
type Cont = Program → Program
fizz, buzz, base :: Int → Cont
fizz  n | n `mod` 3 ≡ 0 = λx → print "fizz" ∘ x ∘ halt
        | otherwise     = id
buzz n  | n `mod` 5 ≡ 0 = λx → print "buzz" ∘ x ∘ halt
        | otherwise     = id
base n                  = λx → x ∘ print (show n)
fizzbuzz :: Int → String
fizzbuzz n = (base n ∘ fizz n ∘ buzz n) skip ""
```

Notice that $\circ$ is now overloaded: it composes both programs from commands (as in the bodies of functions *fizz*, *buzz*, and *base*) and contexts (as in the body of *fizzbuzz*).

Inlining

The truth is that we do not need to implement the entire Skip-Halt-Print language to solve FizzBuzz – our three contexts suffice. Thus, we inline the definitions of *base*, *skip*, *halt*, and *print* in *fizzbuzz*. We also put *fizz* and *buzz* as local definitions, so that we don't have to pass n around:

```
fizzbuzz :: Int → String
fizzbuzz n = (fizz ∘ buzz) id (show n)
where
  fizz  | n `mod` 3 ≡ 0 = λx → const ("fizz" ++ x "")
        | otherwise     = id
  buzz  | n `mod` 5 ≡ 0 = λx → const ("buzz" ++ x "")
        | otherwise     = id
```

Final polishing

As the last step, we abstract over the divisor and the printed message in *fizz* and *buzz*:

```
fizzbuzz :: Int → String
fizzbuzz n = (test 3 "fizz" ∘ test 5 "buzz") id (show n)
  where
    test d s x | n `mod` d ≡ 0 = const (s ++ x "")
               | otherwise     = x
```

What is going on in this program? The (higher-order) function *test* has the following, longish type:

$$test :: Int \to String \to (String \to String) \to String \to String$$

To understand its logic, it is convenient to name the last argument and rewrite the function to the following equivalent definition:

$$\begin{array}{lll} test\ d\ s\ x\ v \mid n\ `mod`\ d \equiv 0 & = & s +\!\!+ x\ \texttt{""} \\ \quad \mid otherwise & = & x\ v \end{array}$$

The argument $v :: String$ represents the default value of the function (originally set by the function *fizzbuzz* to the string representation of n), while $x :: String \to String$ represents a continuation – the rest of the computation parametrised by a new default value. If the modulus test fails, we change neither the continuation nor the default value. If the test succeeds, we print out the string s, but also change the default value to the empty string, so that the string representation of n is not printed out.

Exercises

Exercise 3. Prove that for $f :: t \to s \to s$ and $a :: s$, the following equality holds:

$$foldr\ f\ a\ xs = foldr\ (\circ)\ id\ (fmap\ f\ xs)\ a$$

Exercise 4. In the "Inlining" step, we silently performed some cleaning-up. In reality, a bald inlining of *base* and *skip* in *fizzbuzz* yields

$$((\lambda x \to x \circ (+\!\!+)\ (show\ n)) \circ fizz \circ buzz)\ id\ \texttt{""}$$

Show that it is equal to $(fizz \circ buzz)\ id\ (show\ n)$.

Exercise 5. Adjust the final solution to the FizzBuzzHissHowl problem. Do you have to go through the entire derivation once more, or is the final solution modular?

Summary

To solve a trivial problem, we went through a bit of a hassle: formal language design and semantics, embedded DSLs, interpreters, higher-order abstract syntax to implement contexts, algebra of programming in the form of reasoning about folds. One might also argue that the obtained solution is not too intuitive. Do we really need such heavy artillery to solve FizzBuzz?

Though this tutorial is not meant to be dead serious and is mostly a pretext for some fun with the functional programming technologies listed above – also, going through this derivation might be a risky move during a job interview –

there is a small point it wants to convey: **Functional programmers! Remember higher-order functions!** They are your tool to express programs with non-trivial structure, to follow the information-flow more closely, to dynamically build your programs in runtime. A harsh, cantankerous functional programming pedagogue might say that they are such a basic tool that the final FizzBuzz program shouldn't appear complicated at all (and could be easily written by hand) if one knows their paradigm.

Closing remarks

The Skip-Halt-Print language is based on Edsger W. Dijkstra's Skip-Abort, which can be found in his textbook *A Discipline of Programming* [2, Chapter 4] (I am grateful to Tomasz Wierzbicki for the reference.) However, one needs to be aware of a difference in semantics between HALT and ABORT: the former peacefully ends the computation (like `return` in *C*-like languages), while the latter atrociously breaks it (like Haskell's *error* function).

I would also like to thank Jeremy Gibbons for his comments. The first idea for this tutorial sparkled in my head after Laurence E. Day's Facebook post:

> I can write doctoral level Haskell without so much as missing a beat,
> but I'd have a genuinely hard time writing a FizzBuzz program in Java.

The main point of this tutorial is that FizzBuzz is **not at all trivial**, but, many thanks to higher-order functions in Haskell, solvable. I don't know about Java.

References

[1] Wikipedia. `http://en.wikipedia.org/wiki/Fizz_buzz`.

[2] Edsger W. Dijkstra. **A discipline of programming**. Prentice-Hall series in automatic computation, Prentice-Hall, Incorporated (1976).

Supercompilation: Ideas and Methods

by Ilya Klyuchnikov ⟨ilya.klyuchnikov@gmail.com⟩
and Dimitur Krustev ⟨dkrustev@gmail.com⟩

*Supercompilation (**supervised compilation**) is a program transformation technique based on the construction of a **full and self-contained model** of the program.*

*We illustrate the fundamental ideas and methods of supercompilation with a working supercompiler called **SC Mini** [**?**, 3] for a very simple purely functional language.*

The Essence of Supercompilation

Supercompilation was invented by V. F. Turchin in the Soviet Union during the 1970s. In his own words [1]:

> A program is seen as a machine. To make sense of it, one must observe its operation. So a supercompiler does not transform the program by steps; it controls and observes (SUPERvises) the machine, let us call it M_1, which is represented by the program. In observing the operation of M_1, the supercompiler COMPILES a program which describes the activities of M_1, but it makes shortcuts and whatever clever tricks it knows, in order to produce the same effect as M_1, but faster. The goal of the supercompiler is to make the definition of this program (machine) M_2 self-sufficient. When this is achieved, it outputs M_2 in some intermediate language L^{sup} and simply throws away the (unchanged) machine M_1...
>
> A supercompiler would run M_1 in a general form, with unknown values of variables, and create a graph of states and transitions between

> possible configurations of the computing system ... in terms of which the behavior of the system can be expressed. Thus the new program becomes a self-sufficient model of the old one.

Article Goals

Functional programming practitioners, who are aware of supercompilation, have various opinions. The most common reaction is one of mistrust or even full rejection, mostly because of the fact that there is no industrial-strength supercompiler yet. All existing supercompilers have an experimental status and, in most cases, are only successfully used by their own authors.

Supercompilation is, in a certain sense, a very general method, but, in practice, only specialized tools – re-using just some of the ideas of supercompilation – work well enough to be useful. This clashes with the expectation that a general tool should come out of a general idea. Another possible reason for the mistrust is the fact that supercompilation is still lacking a "killer application".

One common misconception persists: that supercompilation is only a tool for program optimization. Supercompilation is a **program transformation technique**. Program transformations can have different goals. One such goal can be optimization, another – program analysis. Supercompilation is equally applicable to both these goals. Most works dealing with supercompilation consider only the optimization aspect, but we should not be misled that optimization is the only useful application of supercompilation.

We must distinguish the notions of **supercompilation** and **a supercompiler**. Many articles describe various technical difficulties arising from the implementation of a specific supercompiler and ways to overcome these difficulties, while mostly leaving aside the underlying key ideas of supercompilation. In most cases, the parts of a given supercompiler interact in intricate ways, which may give the impression of a complicated monolithic construction. Or, as Simon Peyton-Jones put it in the interview [2]: "... To build a supercompiler, if you look at how it works, there are a number of things all wound together in one rather complicated ball of mud. I found it extremely difficult when I really wanted to understand it. I found it very difficult to understand the papers. ...".

The main goal of this article is to give a clear and concise illustration of the aforementioned Turchin's quote, describing the essence of supercompilation by using a minimalistic example supercompiler. The accent is on delineating and explaining the basic building blocks and showing how these blocks can be implemented and made to work together in a clear and simple fashion.

Organization of the article: A important companion to this article is the full source code of the toy supercompiler SC Mini, with detailed comments [**?**] (250

lines of main code + 200 lines of auxiliary utilities in Haskell). The text of the article itself can be seen as an introduction to studying these supercompiler sources. The article introduces and explains the main supercompiler terminology and ingredients, and we illustrate SC Mini's behavior on suitable examples. These examples are chosen to be neither too complicated, nor oversimplified. The reader is thus invited to carefully study all examples.

Things left out: Do not expect to find a detailed comparison of supercompilation to other methods of program optimization/transformation – it would require a survey article of a much larger scope. A comprehensive bibliography of supercompilation literature is similarly out of scope, although we do give many useful references where relevant.

Supercompilation by Example

> The advantages of every formal language are determined not only by its ease of use by humans, but also by the amenability of its texts to formal transformations.
>
> V.F. Turchin [4]

It would be nice if all computer-science articles were accompanied by some formal executable description (in the form of a program or a formal specification) in order to ensure reproducibility of results.

The SC Mini supercompiler is such an executable description of the fundamental methods of supercompilation, which are the main topic of this article. SC Mini is based on the supercompiler described in the groundbreaking MSc thesis of Morten H. Sørensen [5] (which remained only on paper).

No supercompilation description is complete without describing subtle fundamental notions such as program semantics, evaluation results, computation state. All these notions are formally represented in the sources of SC Mini, which helps avoid ambiguities.

SC Mini transforms programs written in a toy language called SLL (= Simple Lazy Language; corresponding roughly to Sørensen's M_0). Paraphrasing Turchin's quote from the beginning of the section, we argue that SLL's main advantages are:

1. A simple language definition, and
2. A simple definition of a supercompiler for SLL programs.

Despite the fact that SC Mini is a supercompiler for a specific language, the methods used in its construction are fairly general and appear with only small variations in most supercompilers.

P	::=	$d_1 \ldots d_n$	program
d	::=	$f(v_1, \ldots, v_n) = e;$	"indifferent" function
	|	$g(p_1, v_1, \ldots, v_n) = e_1;$	"curious" function
		$\ldots$	
		$g(p_m, v_1, \ldots, v_n) = e_m;$	
e	::=		expression
		v	variable
	|	$C(e_1, \ldots, e_n)$	constructor
	|	$f(e_1, \ldots, e_n)$	function call
p	::=	$C(v_1, \ldots, v_n)$	pattern

Figure 1: SLL abstract syntax

We shall see further through several examples that SC Mini can optimize programs (giving first a formal definition of a program optimizer), and it can also be used to automatically prove different program properties.

SLL Object Language

Defining a language entails defining its syntax and semantics. Syntax is typically defined using a context-free grammar. One possible way to define semantics is to implement an interpreter for the language. The following natural-language description is more succinctly and simply re-expressed – using Haskell – in SC Mini's sources.

Fig. 1 gives the abstract syntax of SLL. An SLL expression can be one of the following:

- variable,
- constructor, having SLL expressions as arguments, or a
- function call, having SLL expressions as arguments.

We assume further the same convention as in Haskell: constructor names start with an uppercase letter; variable names start with a lowercase letter.

An SLL expression consisting only of constructors is called a **value**. An SLL expression without any variables inside is called a **closed expression**. We define **configuration** as an alias of an SLL expression with free variables.

An SLL program consists of function definitions. We distinguish two kinds of functions – "indifferent" and "curious" (originally called by Sørensen in [5] f- and g-functions respectively). Curious functions replace case expressions (which are

```
add(Z(), y)  = y;
add(S(x), y) = S(add(x), y);

mult(Z(), y)  = Z();
mult(S(x), y) = add(y, mult(x, y));

sqr(x) = mult(x, x);

even(Z())  = True();
even(S(x)) = odd(x);

odd(Z())  = False();
odd(S(x)) = even(x);

add'(Z(), y)  = y;
add'(S(v), y) = add'(v, S(y));
```

Figure 2: `prog1`: Working with Peano numbers

always lifted to the top level). Indifferent functions just transfer their arguments to other functions or constructors; their definition is just a single expression. Curious functions perform a simple pattern match on their first argument; their definitions consist of many expressions – one per case.

SLL has no built-in data types (Booleans, numbers, etc.). SLL can be straightforwardly extended with Haskell-like data-type declarations and Hindley-Milner type inference, but we ignore issues of typing from now on, silently assuming that all programs under consideration would be well-typed under such a type system. We assume the following convention:

- the constructors `True()` and `False()` represent Boolean values,
- the value `Z()` represents 0, `S(Z())` - 1, `S(S(Z()))` - 2, etc. If the value `n` corresponds to the natural number n, then the value `S(n)` corresponds to the natural number $n + 1$ (the so-called Peano numbers).

Fig. 2 shows a program defining some functions for working with Peano numbers. It contains a single indifferent function – squaring (`sqr`). All the other functions are curious, as they do a pattern match on their first argument.

A **substitution** binds variables $v_1, v_2, \ldots, v_n$ to expressions $e_1, e_2, \ldots, e_n$ and is written as a list of pairs $\{v_1 := e_1, \ldots, v_n := e_n\}$. Application of a substitution to an expression e is defined in the usual way, and is denoted $e/\{v_1 := e, \ldots, v_n := e_n\}$.

Exercise 6. In SC Mini's sources substitution is literally represented as a list of pairs. Application of a substitution `s` to an expression `e` is denoted as `e // s`.

$$
\begin{array}{lll}
\mathcal{I}_p[\![e]\!] & \Rightarrow e & (I_1) \\
 & \text{if } e \text{ is a value} & \\
\mathcal{I}_p[\![C(e_1,\ldots,e_n)]\!] & \Rightarrow C(\mathcal{I}_p[\![e_1]\!],\ldots,\mathcal{I}_p[\![e_n]\!]) & (I_2) \\
 & & \\
\mathcal{I}_p[\![con\langle f(e_1,\ldots,e_n)\rangle]\!] & \Rightarrow \mathcal{I}_p[\![con\langle e/\{v_1:=e_1,\ldots,v_n:=e_n\}\rangle]\!] & (I_3) \\
 & \text{if } f(v_1,\ldots,v_n) \stackrel{\text{p}}{=} e & \\
\mathcal{I}_p[\![con\langle g(C(e_1,\ldots,e_m),e_{m+1},\ldots,e_n)\rangle]\!] & \Rightarrow \mathcal{I}_p[\![con\langle e/\{v_1:=e_1,\ldots,v_n:=e_n\}\rangle]\!] & (I_4) \\
 & \text{if } g(C(v_1,\ldots,v_m),v_{m+1},\ldots,v_n) \stackrel{\text{p}}{=} e &
\end{array}
$$

Figure 3: SLL: interpreter $\mathcal{I}_p$ for a program p

con	::=	$\langle\rangle \mid g(con,\ldots)$	context
red	::=	$f(e_1,\ldots,e_n) \mid g(C(e_1,\ldots,e_n),\ldots)$	redex

Figure 4: SLL: context and redex

Find `e`, `v1`, `e1`, `v2`, `e2`, such that
`e // [(v1, e1), (v2, e2)]` $\neq$ `e // [(v1, e1)] // [(v2, e2)]`.

Fig. 3 shows the formal reduction semantics of SLL expressions. The notation $e_1 \stackrel{\text{p}}{=} e_2$ means that the program p contains a definition $e_1 = e_2$. The semantics is defined through a rewriting SLL interpreter with normal-order reduction. The SLL interpreter $\mathcal{I}_p$ processing a program p evaluates, step by step, each closed SLL expression to an SLL value (or falls into an infinite loop, or terminates with an error message, but we shall often ignore the last case for simplicity). As far as reduction is concerned, we distinguish two kinds of closed expressions:

1. $e = C(e_1,\ldots,c_n)$ – a constructor is "pushed" outside, and we proceed to reduce its arguments.
2. $e \neq C(e_1,\ldots,c_n)$ – then we locate the leftmost reducible function call (**redex**, see Fig. 4) and unfold it according to the corresponding definition from the program. A function call is reducible if it is a call to 1) an indifferent function or 2) a curious function, where the first argument starts with a constructor.

The rules for evaluating SLL expressions are (as typical for functional languages) **compositional**, meaning that, if the expression $e_1/\{v := e_2\}$ is closed, then (taking laziness into account):

$$\mathcal{I}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{I}_p[\![e_1/\{v := \mathcal{I}_p[\![e_2]\!]\}]\!]$$

Remark that we can consider $\mathcal{I}_p$ as a machine, describing the program p.

Exercise 7. In SC Mini the interpreter is defined as a function `eval`. The notation $\mathcal{I}_p[\![e]\!]$ corresponds to the call `eval p e`. Prove that the evaluation of

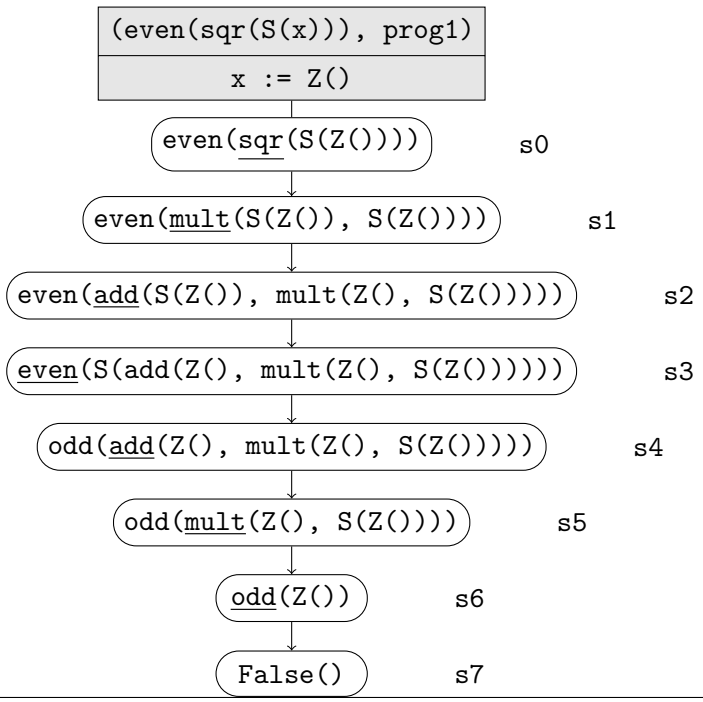


Figure 5: An example of a task evaluation

`eval p (e1 // [(v, e2)]) == eval p (e1 // [(v, eval p e2)])` cannot give `False`.

We call an **SLL-task** (or sometimes simply **task**) the pair (e, p) of a configuration e and a program p (recall that a configuration is simply an expression with free variables). The evaluation of a task (e, p) on arguments $args$ is defined as:

$$\mathcal{R}_{(e,p)}[\![args]\!] = \mathcal{I}_p[\![e/args]\!]$$

(In SC Mini the corresponding call is `sll_run (e, p) args`.)

Fig. 5 shows an example of evaluating the expression `even(sqr(S(Z())))` with the interpreter for the program `prog1`. The redexes are underlined. The root node represents the task with its arguments.

Next, we formally define what is an **optimizer** of SLL tasks. An SLL optimizer takes a task (e, p) as input and outputs another task (e', p'). Our first requirement is correctness: for all arguments $args$, the results of both tasks must be the same:

$$\mathcal{R}_{(e,p)}[\![args]\!] = \mathcal{R}_{(e',p')}[\![args]\!]$$

Our second requirement is optimization: the evaluation of the second task must require no more reduction steps than the evaluation of the first one. (The number

of steps of a reduction sequence can be computed by the function `sll_trace` in SC Mini's sources.)

An Overview of SC Mini

SC Mini transforms a task (e, p) into another task (e', p'), proceeding as follows:

1. It constructs a "machine" $\mathcal{M}_p$, which encodes the behavior of the interpreter $\mathcal{I}_p$ of a program p over a generalized input – configurations (instead of values);
2. It performs a finite (but large enough) number of executions of the machine $\mathcal{M}_p$, analyzing their results;
3. It builds a finite "graph of configurations", which fully describes the behavior of $\mathcal{M}_p$ during those executions; and
4. It simplifies the graph of configurations and builds a new task (e', p') from it.

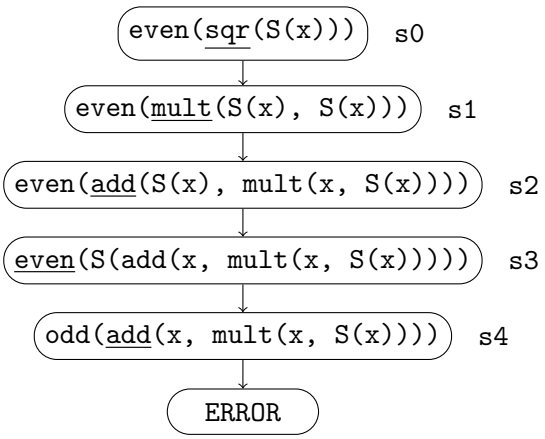
We now explain these steps in detail.

Driving

What would happen, if we tried to perform the following task on an empty list of arguments?

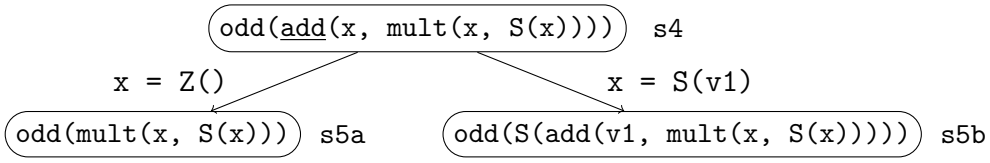
```
(even(sqr(S(x))), prog1)
```

The SLL interpreter is defined in such a way that it can actually evaluate (to some degree) expressions with free variables:

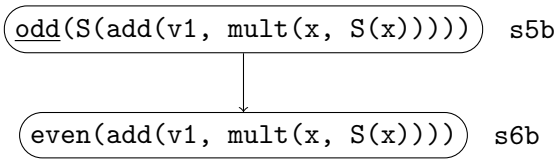


The interpreter does not expect a variable as a first argument of `add`. The presence of such a free variable `x`, however, does not stop it from performing some initial reduction steps, mostly thanks to the call-by-name reduction strategy.

Let us "extend" the interpretation process $\mathcal{I}$: instead of stopping with an error, let us consider the different ways in which evaluation might proceed. The definition of `add` in `prog1` provides 2 possibilities: either `x = Z()`, or `x = S(x1)`:



Now evaluation can proceed with the right subexpression, and so on:



The interpreter $\mathcal{I}_p$ is intended to reduce step-by-step closed expressions. The result of interpretation $\mathcal{I}_p[\![e]\!]$ is some SLL value (unless there is an infinite loop). We introduce a machine $\mathcal{M}_p$, which will compute over configurations. Each transition of this machine – $\mathcal{M}_p[\![c]\!]$ – models a step of the interpreter; it can produce one or several new configurations, labeled with the kind of step taken. We distinguish the following kinds of machine steps:

1. Transient step – the corresponding interpretation step does not depend on the value of any free variable in the configuration. Example: the move from state `s0` to state `s1`.
2. Stop – further modeling is not possible. This occurs when the expression to be reduced is a variable or a value.
3. Decomposition – some part of the result is already known. Example: in the expression `S(sqr(x))`, the outer constructor is clearly a part of the result. We can continue processing its subexpressions.
4. Case analysis – modeling cannot proceed unambiguously. We can, however, consider **all possible further steps of the interpreter, as defined in the program**. Example: the transitions from state `s4` to states `s5a` and `s5b`.

Such modeling – which is a key part of supercompilation – is called **driving**. The preceding paragraph informally defines one driving step.

Exercise 8. In the SC Mini sources building the machine $\mathcal{M}_p$ is performed by `driveMachine p`. Show that `driveMachine p` is powerful enough to evaluate closed expressions. In other words, define a function `f`, such that
`f (driveMachine p) e = eval p e`

Trees of Configurations

The **tree of configurations** for an SLL task (e, p) is built in the following way. We create a machine $\mathcal{M}_p$ modeling p. In the beginning, the tree starts as a single node, labeled with the start configuration e (the task goal). Then, for each tree leaf n, which contains a configuration c, we perform one machine transition $\mathcal{M}_p[\![c]\!]$ and attach new children nodes to n, each of them containing one of the configurations resulting from this transition. We repeat the process for each new leaf. The resulting tree is called "tree of configurations", as each node is labeled by a configuration. Note that the trees of configurations are sometimes called "process trees" in the literature.

Of course, the tree built in this way will be infinite in general. But we can always consider the tree only up to a given (finite) depth (in the previous subsection we have already seen an example – the top of the tree arising from the task `(even(sqr(S(x))), prog1)`).

Exercise 9. SC Mini builds the tree of configurations for a task (e, p) as follows:
`buildTree (driveMachine p) e`
Try to find a characterization of the class of tasks, which result in finite trees of configurations.

The notion of a tree of configurations is not strictly necessary for building a working supercompiler. Many existing supercompilers, especially those built in recent years, make no explicit use of either trees of configurations or graphs of configurations (which will be introduced shortly). Trees and graphs of configurations can be useful, however, both from theoretical and from educational point of view, as they open the way for defining a much more modular supercompiler.

Let's assume for a moment, that we can build (and somehow store) infinite trees of configurations. Then, if for a task (e, p), using a machine $\mathcal{M}_p$, we have built a tree of configurations t, we can throw away the program p and the machine $\mathcal{M}_p$, and only work with the tree t. This is possible, as the tree t fully describes the computation of the task (e, p) without any reference to the text of the program p.

Exercise 10. SC Mini sources contain an executable evidence for the last statement – a "tree interpreter" `intTree`, which can perform tasks using only the corresponding tree of configurations, and not the original program. If `t` is the tree of configurations for the task `(e, p)`, then `intTree t args` gives the result of computing the task over arguments `args`. Convince yourselves that
`eval (e, p) args == intTree (buildTree (driveMachine p) e) args`
cannot produce `False`.

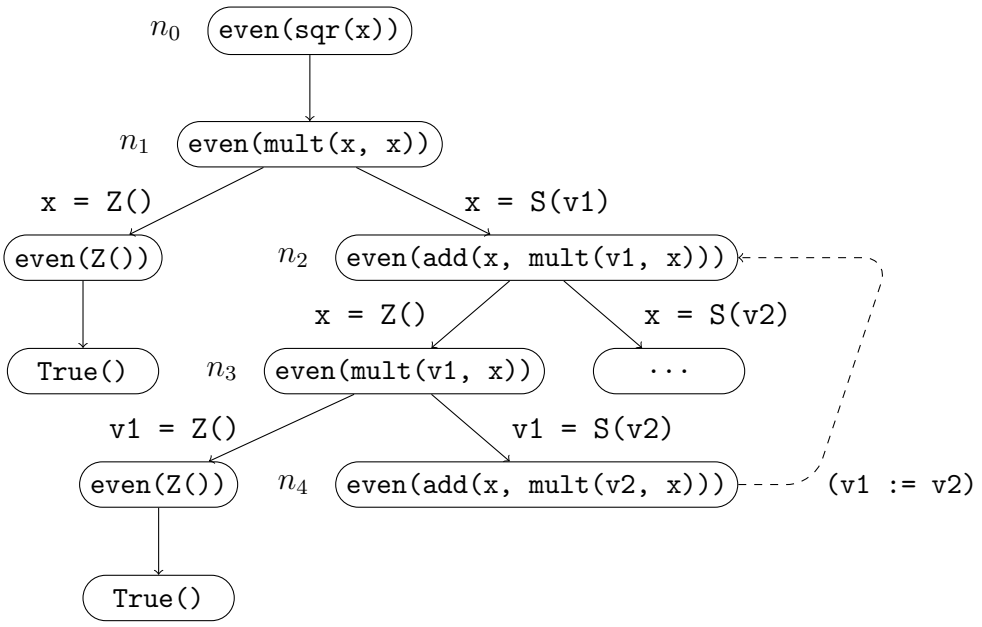
Folding

> Infinite driving graphs are useful in many ways, but to use a graph as an executable program it must be finite.
>
> V. F. Turchin [6]

We can evaluate `even(sqr(S(x)))` for each `x`, by building the tree of configurations and then using the "tree interpreter" `intTree`. The tree of configurations is, however, infinite. The goal of **folding** is to turn the infinite tree into a finite object, from which we could recover the original infinite tree, if needed.

How does folding work? Assume that, while building the tree of configurations, there is a path with nodes $n_0 \to \ldots \to n_i \to \ldots \to n_j \to \ldots$ Assume further that the configuration in node n_j is a renaming (differs only in the choice of variable names) of the configuration in another node n_i $(i < j)$. It is clear that we can reconstruct the subtree starting from node n_j by the subtree starting from node n_i if we systematically rename the variables in the corresponding configurations.

Consider the tree of configurations for the task `(even(sqr(x)), prog1)`:



The configuration in node n_4 is a renaming of the one in node n_2. The (infinite) subtrees starting from node n_2 and from n_4 differ only by the names of the variables in the corresponding nodes. So, we can simply memorize that the subtree from n_4 can be built by a given renaming from the subtree starting at n_2, without building explicitly this subtree. It is important to note that the reconstruction of

the subtree n_4 from the subtree n_2 does not require the original program (nor the machine $\mathcal{M}_p$).

We denote such situations with a special kind of edge, leading from the lower node to the upper one. As a result, the tree of configurations is no longer a tree but rather a **graph of configurations** (also sometimes called "process graph").

We are lucky with the tree of configurations for the task `(even(sqr(x)), prog1)` – it folds into a graph. We can see, that in such a way we can turn some infinite trees into finite graphs. The resulting graph will serve as a new self-contained representation of the given task. We shall denote the graph for the task t as $\mathcal{G}_t$.

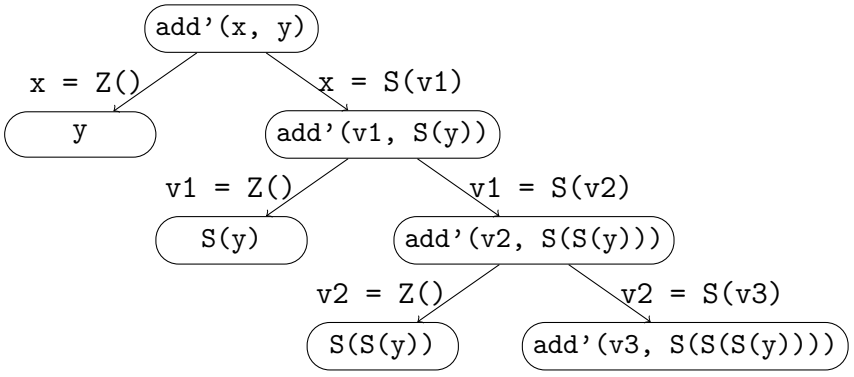
Exercise 11. Just some small modifications of the tree interpreter `intTree` are sufficient to make it work also with folded graphs of configurations. Take a look in SC Mini's sources to see how it is done.

The graph $\mathcal{G}_t$ for the task $t = (e, p)$ can be converted into a new task (e', p'). The resulting program `p'` is called **residual**, and we shall also call the corresponding task $(e',\ p')$ residual.

Exercise 12. SC Mini's sources contain a function `residuate`: `residuate g` transforms a graph of configurations `g` into a new task `(e', p')`.

Generalization: converting a tree into a foldable tree

We were lucky with the example in the previous subsection, as it was possible to fold the tree into a graph. It would be splendid if all trees of configurations were foldable! Unfortunately, this is not the case in general. The program `prog1` contains a function `add'`, which defines addition using an accumulating parameter. The construction of the tree of configurations for `(add'(x, y), prog1)` goes as follows:



...

This tree is not folding. Indeed, a configuration can only be a renaming of another if both have the same size. In this example, the path through the rightmost branches of the tree contains ever-growing configurations. However, if the size of configurations be bounded, the tree will always fold to a finite graph.

Exercise 13. Try to prove the last statement. *Hint:* To what kind of relation does renaming naturally give rise?

We can thus apply a divide-and-conquer heuristics. We limit the size of configurations inside nodes by some constant `sizeBound`, and if the size of a configuration becomes larger than this constant, we split it into smaller parts which we can process **independently**.

Recall that the evaluation of closed SLL expressions is compositional. For a closed expression $e_1/\{v := e_2\}$ we have:

$$\mathcal{I}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{I}_p[\![e_1/\{v := \mathcal{I}_p[\![e_2]\!]\}]\!]$$

The same property of compositionality holds for evaluation of tree of configurations. (Look into SC Mini source code to see how `intTree` handles splitted configurations.)

SC Mini limits the size of configurations in the tree and builds a foldable tree in the following way:

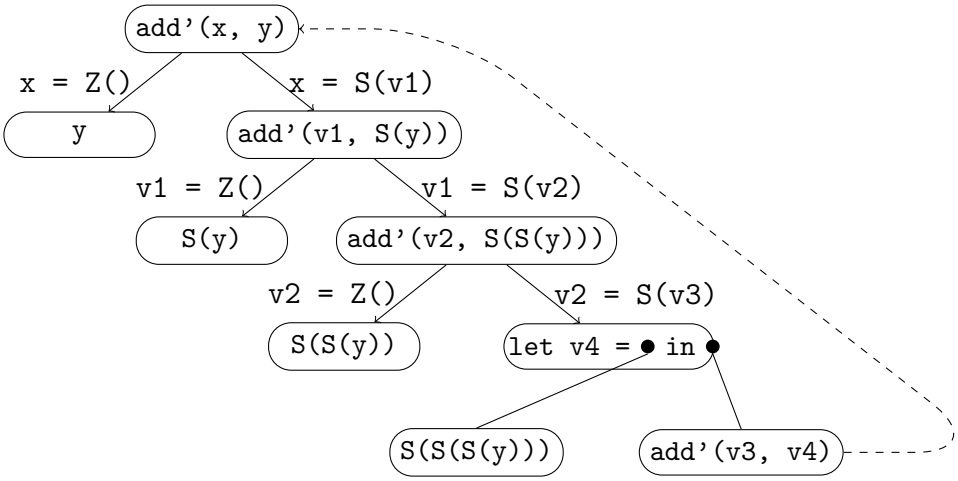
- if while building the tree a configuration e is encountered, which is a function call, and whose size is greater than `sizeBound`, then this configuration is splitted,
- otherwise, the construction of the tree is done in the standard way.

Splitting represents the configuration e as $e = e_1/(v := e_2)$, where e_2 is the largest subexpression of e, and e_1 – the original expression e, in which e_2 is replaced by a (fresh) variable v. Such configurations are encoded as `let`-expressions $let\ v = e_2\ in\ e_1$. Then we process the configurations e_1 and e_2 independently.

Exercise 14. We do not need to check the size of constructor nodes. Why?

The configuration e_1 is a generalization of e. And e is, in turn, a special instance of e_1. By a slight abuse of terminology, the process of transforming the tree of configurations into a foldable one – as described above – is also called **generalization**.

Now, if we limit the growth of configurations (by size), for any task we can build a (potentially infinite) tree of configurations, which will in all cases be folded into a **finite** graph of configurations. Returning to our current example, if we set `sizeBound` equal to 5, then as a result of generalizing the rightmost configuration we obtain:



Exercise 15. What properties should a task have, in order for the configuration tree to be foldable without using generalization?

Exercise 16. It is very easy to extend `intTree` with treatment of `let`-expressions. Check how it is done in the SC Mini sources.

The idea of generalizing configurations is one of the cornerstones of supercompilation. What we have described above is one of the possibly simplest and least sophisticated ways to perform generalization.

The supercompiler part responsible for deciding when to generalize is historically called the **whistle**. Whistles in most existing supercompilers are typically more complicated than just limiting configuration size.

The term "whistle", despite sounding a bit unscientific, is generally accepted and used in most descriptions of supercompilation. The whistle is a heuristic component, whose task is to signal the danger of possibly non-foldable (and thus infinite) fragments appearing in the tree of configurations. Of course, this task is an instance of the halting problem (if the object language is Turing-complete) and does not have an exact solution. Any whistle will necessarily give only approximative results, and even the best heuristics cannot guarantee the absence of blunders, such as whistling just a few steps before an actually foldable configuration. The only strict requirement is to always err on the side of caution and never let infinite tree paths slip through, as it would make the supercompiler itself loop.

Supercompiler prototype

In the SC Mini sources generalization and folding are implemented via following functions:

- `buildFTree` – it builds a foldable tree of configurations by limiting size of configurations (by means of splitting)

- `foldTree` – it takes a (possibly infinite) tree of configurations and folds it into a finite graph.

Let's create the following program transformer:

```
transform :: Task -> Task
transform (e, p) =
        residuate $ foldTree $ buildFTree (driveMachine p) e
```

A most interesting thing in this transformer happens there: the machine $\mathcal{M}_p$ is not only run – the result of its run is analyzed, and possibly a decision to generalize is taken.

We argue that this program transformer fulfills the requirements of the previously stated definition of "supercompiler". Although this transformer performs no interesting optimizations, it can serve as a foundation for a full-blown supercompiler. We shall use it as a baseline to which we shall compare the transformers that follow (deforestation and supercompilation).

Exercise 17. Try to show that for each task `(e, p)` and each substitution `s` the computation `sll_run (e', p') s` (where `(e', p')` is the corresponding residual task) requires exactly as many reduction steps as the computation `sll_run (e, p) s`. That means that the transformer `transform` does not degrade performance, but it does not improve it either.

Does this transformer modify the input program at all? Yes, and here is an example:

`(even(sqr(x)), prog1)` $\xrightarrow{transform}$ `(f1(x), prog1T)`

Where `prog1T =`

```
f1(x) = g2(x, x);
g2(Z(), x) = f3();
g2(S(v1), x) = g4(x, x, v1);
f3() = True();
g4(Z(), x, v1) = g5(v1, x);
g4(S(v2), x, v1) = f7(v2, v1, x);
g5(Z(), x) = f6();
g5(S(v2), x) = g4(x, x, v2);
f6() = True();
f7(v2, v1, x) = g8(v2, v1, x);
g8(Z(), v1, x) = g9(v1, x);
g8(S(v3), v1, x) = f16(v3, v1, x);
g9(Z(), x) = f10();
g9(S(v3), x) = g11(x, x, v3);
f10() = False();
g11(Z(), x, v3) = g9(v3, x);
```

```
g11(S(v4), x, v3) = f12(v4, v3, x);
f12(v4, v3, x) = g13(v4, v3, x);
g13(Z(), v3, x) = g14(v3, x);
g13(S(v5), v3, x) = f7(v5, v3, x);
g14(Z(), x) = f15();
g14(S(v5), x) = g4(x, x, v5);
f15() = True();
f16(v3, v1, x) = g17(v3, v1, x);
g17(Z(), v1, x) = g18(v1, x);
g17(S(v4), v1, x) = f7(v4, v1, x);
g18(Z(), x) = f19();
g18(S(v4), x) = g4(x, x, v4);
f19() = True();
```

We can easily detect at least one property of the transformer – the resulting program contains no nested function calls: it is "flat".

KMP Test

Consider the program in Fig. 6 (where function names are deliberately short, to keep the drawings of graphs of configurations small): the function `match(p, s)` checks if a string `p` is contained inside the string `s`. For simplicity we use a 2-letter alphabet – `'A'` and `'B'`. Strings are represented – as in Haskell – by lists of characters. We also use some list/string syntactic sugar from Haskell for readability, even if it is not actually implemented in SLL.

The function `match` is general, but inefficient. Consider the call `match("AAB", s)`, which checks if the substring (pattern) `"AAB"` appears inside the string `s`. Our program will perform this check in the following way: compare `'A'` with the first character of `s`, `'A'` – with the second one, `'B'` – with the third one. If any of these comparisons fails, we restart the comparison sequence after skipping the first character of `s`. This strategy is far from optimal, however. Let's assume that the string `s` starts with `"AAA..."`. The first 2 comparisons will succeed, the 3rd one will fail. It is inefficient to repeat the same sequence of comparisons on the tail `"AA..."`, as we already have enough information to know, that the first 2 comparisons of `"AAB"` with `"AA..."` will succeed. The deterministic finite automaton (DFA), built by the Knuth-Morris-Pratt algorithm [7], will consider each character of `s` at most once.

A simple way to estimate the power of an optimizing transformation is to see if it can produce – fully automatically – a well-known efficient algorithm from a "naïve", less-efficient one. In the case of supercompilation, it turns out that – starting from a naïve string-matching algorithm, and fixing the value of the substring we try to match – we can automatically obtain an efficient specialized matching program,

```
match(p, s) = m(p, s, p, s);

-- matching routine
-- current pattern is empty, so match succeeds
m("", ss, op, os) = True();
-- proceed to match first symbol of pattern
m(p:pp, ss, op, os) = x(ss, p, pp, op, os);

-- matching of the first symbol
-- current string is empty, so match fails
x("", p, pp, op, os) = False();
-- compare first symbol of pattern with first symbol of string
x(s:ss, p, pp, op, os) =
  if(eq(p, s), m(pp, ss, op, os), n(os, op));

-- failover
-- current string is empty, so match fails
n("", op) = False();
-- trying the rest of the string
n(s:ss, op) = m(op, ss, op, ss);

-- equality routines
eq('A', y) = eqA(y);  eqA('A') = True();
eqB('A') = False();
eq('B', y) = eqB(y);  eqA('B') = False();
eqB('B') = True();
-- if/else
if(True(), x, y) = x;
if(False(), x, y) = y;
```

Figure 6: `prog2`: find a substring inside a string

analogous in action to the well-known Knuth-Morris-Pratt algorithm (hence the name of the test).

Our original simple transformer `transform` is unable to obtain an efficient matching program. But in the next couple of sections, we will augment `transform` using two "tricks", which are key ingredients of "real" supercompilers, and the transformer `supercompile` produces a residual task exactly corresponding to this DFA.

This program is the so-called KMP-test, which is a classical example in the context of supercompilation, as it demonstrates its greater power compared to

similar program transformations like deforestation and partial evaluation[5, 8]. This test is not only a good demonstration of the combined effect of the two tricks mentioned above, but it also gives some measure of the extent of program transformations performed by supercompilation.

Our original simple transformer `transform` gives the following result:

`(match("AAB", s), prog2)` $\xrightarrow{transform}$ `(f1(s), prog2T)`

where prog2T is:

```
f1(s) = f2(s);
f2(s) = g3(s, s);
g3("", s) = False();
g3(v1:v2, s) = f4(v1, v2, s);
f4(v1, v2, s) = g5(v1, v2, s);
g5('A', v2, s) = f6(v2, s);
g5('B', v2, s) = f22(v2, s);
f6(v2, s) = f7(v2, s);
f7(v2, s) = g8(v2, s);
g8("", s) = False();
g8(v3:v4, s) = f9(v3, v4, s);
f9(v3, v4, s) = g10(v3, v4, s);
g10('A', v4, s) = f11(v4, s);
g10('B', v4, s) = f20(v4, s);
f11(v4, s) = f12(v4, s);
f12(v4, s) = g13(v4, s);
g13("", s) = False();
g13(v5:v6, s) = f14(v5, v6, s);
f14(v5, v6, s) = g15(v5, v6, s);
g15('A', v6, s) = f16(v6, s);
g15('B', v6, s) = f18(v6, s);
f16(v6, s) = g17(s);
g17("") = False();
g17(v7:v8) = f2(v8);
f18(v6, s) = f19(v6, s);
f19(v6, s) = True();
f20(v4, s) = g21(s);
g21("") = False();
g21(v5:v6) = f2(v6);
f22(v2, s) = g23(s);
g23("") = False();
g23(v3:v4) = f2(v4);
```

This long listing is reproduced here only to serve as a baseline for comparison. Note that the residual program contains no nested function calls, but each of the

characters of `s` may be checked multiple times, as in the input program.

Transient step removal

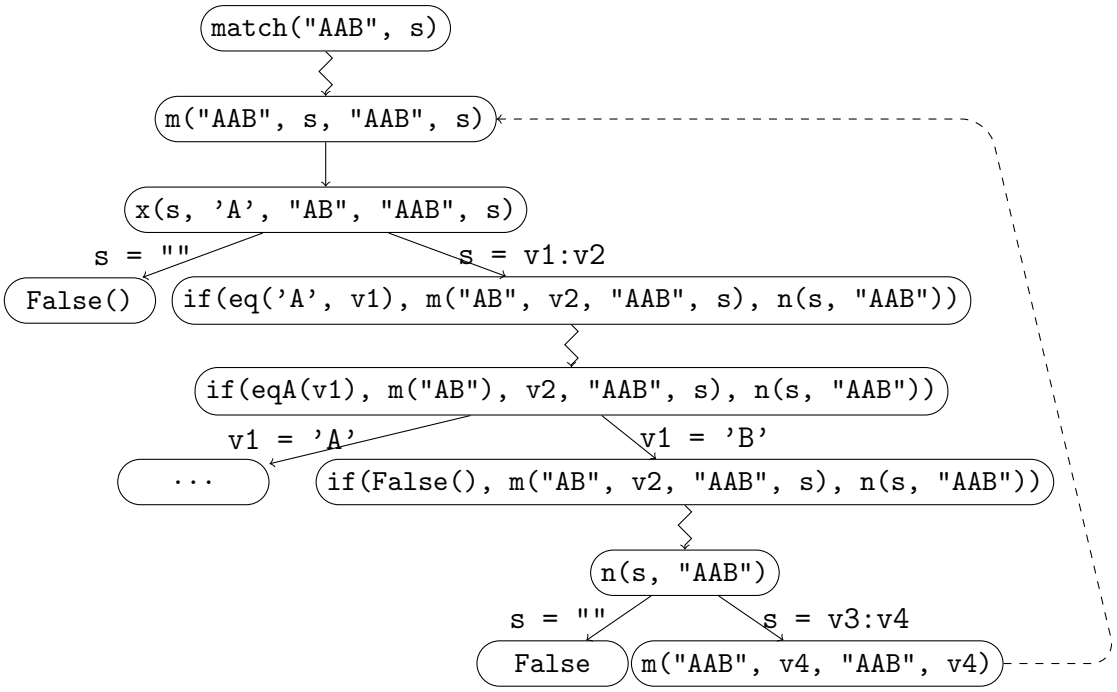
Very often graphs of configurations contain fragments of the following kind:

```
... c1 → c2 → c3 ...
```

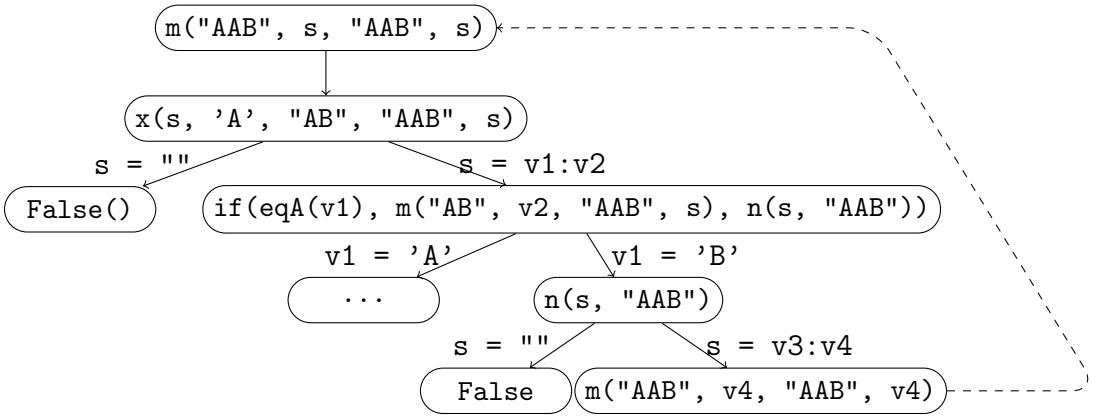
where the transition `c2 → c3` corresponds to a transient step (as defined in subsection "Driving"). We can always – thanks to the absence of side effects in SLL – replace such a fragment by:

```
... c1 → c3 ...
```

As an example, consider the top part of the graph of configurations for the KMP-test program, which is produced by `transform`:



The transient steps are displayed with zigzag arrows. If we remove them, the cleaned-up graph of configurations will look like this:



The SC Mini sources contain a transformer `deforest` – a modification of `transform` – which removes transient edges and the corresponding nodes from the graph of configurations.

If we pass the test program through `deforest`, we get:

`(match("AAB", s), prog2)` $\xRightarrow{deforest}$ `(f1(s), prog2D)`

where `prog2D =`

```
f1(s) = g2(s, s);
g2("", s) = False();
g2(v1:v2, s) = g3(v1, v2, s);
g3('A', v2, s) = g4(v2, s);
g3('B', v2, s) = g10(s);
g4("", s) = False();
g4(v3:v4, s) = g5(v3, v4, s);
g5('A', v4, s) = g6(v4, s);
g5('B', v4, s) = g9(s);
g6("", s) = False();
g6(v5:v6, s) = g7(v5, v6, s);
g7('A', v6, s) = g8(s);
g7('B', v6, s) = True();
g8("") = False();
g8(v7:v8) = f1(v8);
g9("") = False();
g9(v5:v6) = f1(v6);
g10("") = False();
g10(v3:v4) = f1(v4);
```

The deforested program contains less than half the number of functions (23 vs. 10) compared to the result of `transform`, as some intermediate functions were removed (roughly speaking, inlined). Although the residual program is smaller, it still contains the same inefficiency we discussed above.

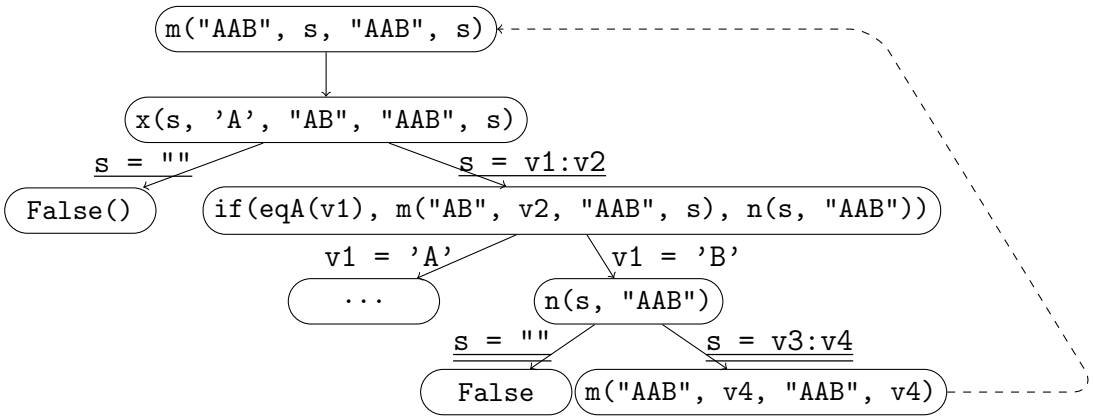
The removal of transient edges is a form of simplification of graphs of configurations. This simplification is **one of the two** main mechanisms for optimization, which are employed by supercompilation. It is most effective when used in conjunction with the second mechanism – information propagation – which we shall discuss next.

Exercise 18. It is also possible to remove transient steps from the tree of configurations (before folding it to a graph). What pitfalls might such an approach have?

There is a separate program transformation technique called **deforestation** [9, 10]. Its main goal is to reduce the creation of intermediate data structures – lists, trees (hence the name), etc. – which arise as a result of composing different functions in a program. Ferguson et al. [10] describe deforestation for a language very similar to SLL. Classical deforestation does not require generalization, as it considers only programs conforming to certain syntactic restrictions (which guarantee that the resulting tree is always foldable). A practical advantage of deforestation is that for many special cases it can be formulated as a simple set of rewrite rules (**shortcut deforestation**, or **shortcut fusion** – see for example [**?**]), which are easy to implement. A form of shortcut deforestation is implemented inside GHC, for example.

Information propagation

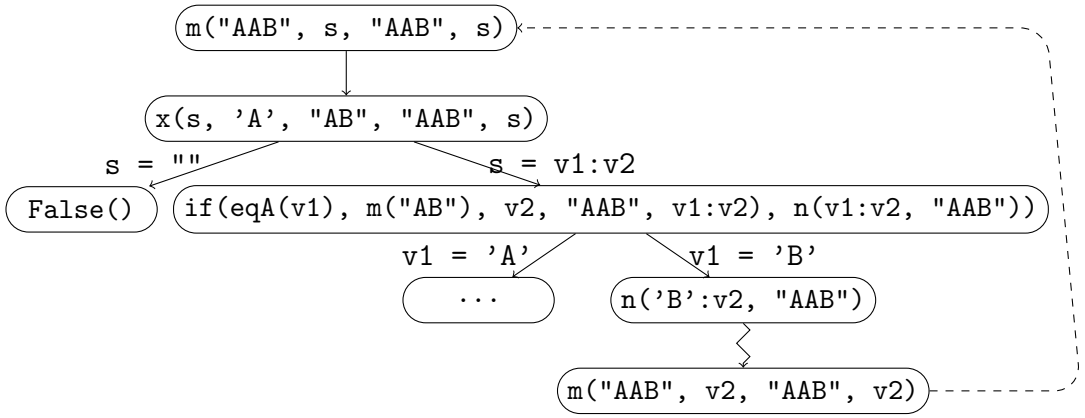
Here is the graph of configurations for the KMP test produced by `deforest`:



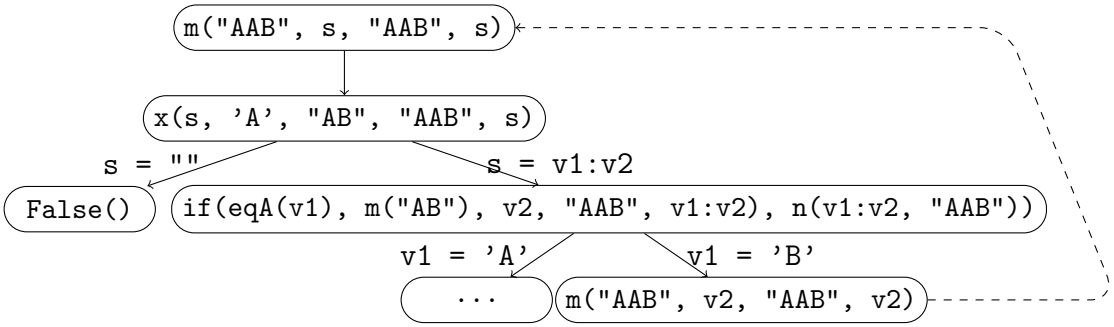
It contains a step which checks if the string `s` is empty or not (underlined). Below it, there is another step, which makes the same check (double-underlined), although we have assumed already that `s` is not empty (in the corresponding subtree after the first check). Such repeated checks are clearly redundant.

Supercompilers remove such redundancies as follows: each time case analysis is performed during driving, information about its outcome is propagated in each corresponding subtree.

Let's propagate the fact that `s = v1:v2` (and hence non-empty) in the corresponding subtree:



We have not only removed a superfluous test, but we have also unmasked a transient step, which can be further removed. As a result we obtain:



The transformer `supercompile` (inside SC Mini sources) implements such information propagation. In contrast to `deforest`, each time driving performs case analysis, its results are propagated in the corresponding new configurations. Recall that SLL contains just a single kind of test – matching the first argument of a curious function against a list of simple patterns. Hence, the only kind of information we can propagate is that a given configuration variable is equal to a given pattern. Such a (restricted) form of information propagation is called **positive information propagation**. Correspondingly, supercompilers propagating only positive informtion are called **positive supercompilers**. If, for example, SLL permitted also default clauses (in curious-function definitions), we could propagate also negative information – in the default clause – that the variable is not equal to any

of the patterns in the list. A supercompiler, which propagates both positive and negative information, is called **perfect** [**?**]. It is also possible to have intermediate situations, where only some forms of negative information are propagated.

Here is the final result of supercompiling our KMP test:

```
(match("AAB", s), prog2) ⟹ (f1(s), prog2S)
```

where `prog2S =`

```
f1(s) = g2(s);
g2("") = False();
g2(v1:v2) = g3(v1, v2);
g3('A', v2) = g4(v2);
g3('B', v2) = f1(v2);
g4("") = False();
g4(v3:v4) = g5(v3, v4);
g5('A', v4) = f6(v4);
g5('B', v4) = f1(v4);
f6(v4) = g7(v4);
g7("") = False();
g7(v5:v6) = g8(v5, v6);
g8('A', v6) = f6(v6);
g8('B', v6) = True();
```

We finally obtain the desired effect: each character of the string `s` is considered at most once.

Exercise 19. Prove the last statement.

The information propagation we have just described is the second optimization mechanism employed by supercompilation (the first was the removal of transient steps). This propagation has 3 important effects:

1. No variable is tested twice.
2. Avoiding repeated tests uncovers further possibilities for removing transient steps.
3. The graph of configurations is pruned of unreachable branches, which results in dead-code removal in the residual program.

In the example above, the graph built by `deforest` contains the following path:

```
{s = v1:v2}  → ... → {s = ""} →
```

It is clear that no actual computation can take this path. Deforestation, however, leaves it in the residual program; as a result the deforested program `prog2d` above contains a function `g10` whose first argument can never be an empty string:

```
g10("") = False();
g10(v3:v4) = f1(v4);
```

As a result of supercompiling the KMP test, we obtained a program which contains no dead code. Of course, this is not always possible, as dead-code removal is undecidable in general. In the case of supercompilation, dead code may still appear in the residual program if we have applied generalization.

Exercise 20. Most works discussing supercompilation interweave information propagation with other aspects of the transformation (for example, with transient-step removal). Check if this is the case as well inside SC Mini's sources.

The essence of SC Mini

This section turned out rather long, so let's take a brief look – from a slightly different angle – at what it covered.

The SC Mini supercompiler is an attempt to build a minimalistic supercompiler in Haskell. Its main goal is to delineate the main ingredients which are present in most supercompilers and show how they can be combined to work well together.

Let's review the gradual transition from the basic transformer `transform` (which neither improves, nor degrades performance), passing through `deforest`, and finally arriving at the ultimate transformer `supercompile`.
Starting from the baseline transformer `transform`...

```
transform :: Task -> Task
transform (e, p) =
    residuate $ foldTree $
        buildFTree (driveMachine p) e
```

...we add simplification of the graph of configurations by removing transient steps...

```
deforest :: Task -> Task
deforest (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (driveMachine p) e
```

...and we finish by adding information propagation:

```
supercompile :: Task -> Task
supercompile (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (addPropagation  $ driveMachine p) e
```

Thanks to propagating test outcomes in the corresponding consequent configurations and further simplifications of the graph, our minimalistic supercompiler managed to convert a naïve substring search algorithm into the well-known efficient KMP algorithm.

Not only optimization

SC Mini can also be used to check different program properties. Let's show that the operation `add` – defined in `prog1` – is associative, that is, for all arguments `args`:

```
sll_run (add(add(x,y),z), p1) == sll_run (add(x,add(y,z), p1)
```

We could of course prove this by induction (using the rules of SLL operational semantics). But we can also compare the residual programs produced by SC Mini in both cases – they turn out syntactically equivalent:

```
(add(add(x, y), z), p1) ⟹supercompile (g1(x, y, z), prog3S')
(add(x, add(y, z)), p1) ⟹supercompile (g1(x, y, z), prog3S')
```

where `prog3S' =`

```
g1(Z(), y, z) = g2(y, z);
g1(S(v1), y, z) = S(g1(v1, y, z));
g2(Z(), z) = z;
g2(S(v1), z) = S(g2(v1, z));
```

This is enough to prove associativity of `add` (assuming of course that SC Mini always preserves the semantics of the input program), as we have:

```
sll_run (add(add(x, y), z), p1) ==
  sll_run (g1(x, y, z), prog3S') ==
    sll_run (add(x, add(y, z)), p1)
```

Problems

The aim of this section is to outline what are the practical problems our simple supercompiler faces. These problems are typical – in varying degrees – for most existing supercompilers.

Result unpredictability

The KMP test we saw indeed shows supercompilation at its best. We were lucky that the whistle did not blow, allowing us to fold the tree of configurations to a graph without resorting to generalization. In many other cases, however, the size of configurations in the tree will continue to grow, and, sooner or later, we shall be forced to perform generalization in order to ensure building a finite graph of configurations.

Recall a task we have already seen:

```
(even(sqr(x)), prog1)
```

Deforestation manages to build a finite graph of configurations without applying generalization. This is not the case when we use supercompilation instead – generalization becomes necessary. This is one drawback of information propagation, as it typically increases the size of the new configuration. The appendix accompanying this article lists the results of deforesting and of supercompiling this task. It also gives a comparison of the speed of the corresponding residual programs. Based on these observations, we can draw the following conclusions:

1. The supercompiled program is much larger than the deforested one.
2. For small numbers `x` the supercompiled program works faster. Exactly the opposite is true for big numbers `x` however.

In a way, this example is an antithesis of the KMP test, as it displays the main weaknesses of the SC Mini supercompiler. (We shall call it anti-KMP-test for brevity. Such an example can be found for each of the other existing supercompilers.)

While missing optimization opportunities is a feature we can readily consider acceptable, the fact that the supercompiler can produce much larger residual program, compared to the input one, is a more critical issue. The danger of code explosion is an important problem for supercompilation in general. In the case of SC Mini, by limiting configuration size we indirectly limit the growth of graph width as well. There is no similar indirect limit to the growth of graph depth (apart for ensuring it is finite), and this can result in very large residual programs.

Exercise 21. Try to modify SC Mini, by imposing an explicit limit on graph depth as well. What changes in the examples we considered so far?

Exercise 22. Can we estimate the size of the residual program for a given input?

SC Mini is built in such a way as to guarantee that the residual program never takes more reduction steps than the input one. This is but one possible measure of efficiency. Another one is memory consumption, and here the story is not so rosy. Existing supercompilers can produce programs, which – although requiring less reduction steps – can sometimes consume considerably more memory than the original ones.

Scalability issues

The supercompiler imitates the behavior of the input program by taking into account the dynamic interactions of all its parts. This can result, unfortunately, in the size of the model (the graph of configurations), growing very superlinearly as a function of the size of the input program. For example, a new input program, larger by just 30%, can make the supercompiler work 10 times longer.

Exercise 23. Try to find such examples for SC Mini.

What is worse, the running time of the supercompiler usually depends superlinearly on the size of the graph of configurations.

Exercise 24. Show that this statement is true for SC Mini as well.

Supercompiling big programs can be a very unpredictable process: in the worst case, the supercompiler may take a lot of time, and it may produce a huge program, which works only slightly faster than the original.

Supercompilation performs a global optimization for a given entry point. It is an open question how to apply supercompilation methods to optimize libraries.

What about debugging?

Supercompilation – being a form of program transformation – converts the input program into a new one, which is then submitted to the usual programming language compiler or interpreter. What if we want to debug the resulting program? It appears possible in principle – in a way similar to standard compilers generating "debug" info – to generate some information about the correspondence of residual program lines to input program lines. This can be a tedious task, however, and authors of experimental supercompilers avoid it. We can hope that one day industrial-strength supercompilers will support this feature as well.

Details, details, details...

We saw only a very small, toy supercompiler – for a toy language – which has some obvious deficiencies. Building a supercompiler for a similar toy language, but not having the problems we mentioned, is already a complicated, PhD-level task. If we consider more realistic programming languages, we will quickly stumble upon a number of details, which must be taken care of.

Global state, side effects: SC Mini uses the compositionality of SLL semantics in an essential way in order to perform generalization:

$$\mathcal{I}_p[\![e_1/\{v := e_2\}]\!] = \mathcal{I}_p[\![e_1/\{v := \mathcal{I}_p[\![e_2]\!]\}]\!]$$

The expression e_2 may be evaluated outside of the context, where it appears in the input program.

If the language features global state and/or side effects (which is inevitable in one form or another for any practical programming language), then the notion of compositionality becomes more complicated, if it can be formulated at all. In order to take this into account, an additional analysis phase appears necessary. Things

can get even more complicated in the presence of concurrency. What makes the situation less bleak is that many modern programming languages follow the lead of Haskell and provide means to isolate pure functional code from side-effecting code. In such cases a simple solution is to supercompile only the purely functional parts of the program, leaving the side-effecting parts unmodified.

Taking strictness and laziness into account: SC Mini owes its simplicity, to a large extent, to the fact that we assumed a call-by-name semantics for SLL. Driving is simplest in the case of call-by-name evaluation. Call-by-name typically leads to bad performance, however, and most programming languages do not use it as the default evaluation mechanism. Call-by-value or call-by-need is used instead. While recent research has shown it is possible to build a supercompiler for a call-by-need [11, 12, 13] or a call-by-value [14, 15] language, the supercompilation process is typically much more involved, compared to the case of call-by-name.

Exercise 25. Write a call-by-value interpreter for SLL. Find a program, which behaves differently after supercompilation by SC Mini, when run by the new interpreter.

Exercise 26. Write a call-by-need interpreter for SLL. Compare its size to the call-by-name one. What differences in driving would you expect in the case of call-by-need?

Why is it worth it then?

The open problems of supercompilation are wide, the results of supercompilation difficult to rely on. Why is it, then, that interest in supercompilation has repeatedly resurfaced and grown in the several decades since its invention, with the latest wave of renewed interest starting 6-7 years ago [16, 17, 13, 15, 18, 19, 20]?

First of all, it turns out that many program optimization methods can be seen as special cases of supercompilation. This includes deforestation, partial evaluation, different kinds of fusion, inlining, defunctionalization, etc. (While this statement will be intuitively obvious to most researchers in the area of supercompilation, not all of these cases have been formally described in research papers.) It is most impressing when a (relatively) simple supercompiler can optimize some programs equally well (or better!) than some other complicated specialized tool.

A second important reason is the conceptual simplicity of supercompilation. Supercompilation is not tied to a single specific language, nor even to a family of languages, although most of the existing research has been done in the context of functional languages.

Another valuable feature of supercompilation is that it has applications beyond optimization. There are very successful attempts to apply it to program analysis,

for example. It is tempting to image a 2-in-1, or even 3-in-1 tool, which can cover program optimization, analysis, synthesis, etc.

Last but not least, supercompilation is itself just a specialized application of a very general philosophical principle, invented also by V. F. Turchin – the theory of "metasystem transitions". We shall speak a bit more about that in the next section.

A little history

Valentin Fyodorovich Turchin (1931–2010) – the creator of supercompilation – could be called a Programmer-Philosopher.

"Valentin Fedorovich Turchin, born in 1931, holds a doctor's degree in the physical and mathematical sciences. He worked in the Soviet science center in Obninsk, near Moscow, in the Physics and Energetics Institute and then later became a senior scientific researcher in the Institute of Applied Mathematics of the Academy of Sciences of the USSR. In this institute he specialized in information theory and the computer sciences. While working in these fields he developed a new computer language that was widely applied in the USSR, the "Refal" system. After 1973 he was the director of a laboratory in the Central Scientific-Research Institute for the Design of Automated Construction Systems. During his years of professional employment Dr. Turchin published over 65 works in his field. In sum, in the 1960s and early 1970s, Valentin Turchin was considered one of the leading computer specialists in the Soviet Union." (from L. R. Graham's foreword to "The phenomenon of science" by Turchin [21].)

"The intellectual pivot of the book is the concept of the metasystem transition – the transition from a cybernetic system to a metasystem, which includes a set of systems of the initial type organized and controlled in a definite manner. I first made this concept the basis of an analysis of the development of sign systems used by science. Then, however, it turned out that investigating the entire process of life's evolution on earth from this point of view permits the construction of a coherent picture governed by uniform laws. . . ." (from the introduction to "The phenomenon of science" by Turchin [21].)

In 1966, Turchin invented Refal (REcursive Functions Algorithmic Language) – a programming language that was quite different from most other existing ones. Refal was oriented towards describing and processing other languages. Even a brief description of Refal is beyond the scope of this article, let's just simply mention that it quickly gathered a small group of active supporters which met regularly in Moscow. The next 5–6 years were devoted to creating an efficient implementation of Refal – first an interpreter, and later a compiler as well. The compiler was itself written in Refal: it took Refal programs as input and generated assembly

programs as output. But nothing prevented the generation of Refal as output as well. This was how the idea of driving was born, which Turchin originally described as "equivalent transformations of Refal functions". In 1974, driving was described already in the context of the theory of metasystem transitions.

In 1977, Turchin was forced to leave the Soviet Union; in the same year an English translation of Turchin's "opus magnum" – the book "The phenomenon of science" – was published in the USA. This book was already finished in 1970 and ready for printing in 1973, but its publication in the Soviet Union was blocked for political reasons. Since then, Turchin lived and worked in New York, first in the Courant Institute, later in the City College. Starting from 1989, Turchin was again able to visit Russia, which he did regularly till his death in 2010.

We can distinguish – quite subjectively – 3 periods in the history of supercompilation.

- **1970s–1980s. Refal supercompilers.** Starting from 1979, Turchin published (with coauthors) several tens of articles on supercompilation and metacomputation. It is now obvious that a huge number of interesting ideas lies scattered inside these articles. The problem was that many concepts were given in only a semi-formal, fragmentary way, and only in terms of Refal: for Turchin, Refal and supercompilation were inseparable. Unfortunately, for most people at the time even the description of driving seemed too complicated and incomprehensible. And no article contained a full and self-contained description of the complete process of supercompilation. For these reasons, in spite of the large number of publications on the subject, till the early 1990s supercompilation remained understood and appreciated only by a small number of "initiates". Turchin's articles [4, 22, 1, 23, 24, 25, 26] are some of the milestones of this period.

- **1990s. Supercompilation of first-order functional languages.** Andrei Klimov's and Robert Glück's article "Occam's Razor in Metacompuation: the Notion of a Perfect Process Tree" [27] about the essence of driving was the first work aimed at understanding supercompilation as a general technique, independent of Refal. In 1994 Morten H. Sørensen made an important further step – in his MSc thesis, "Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation," [5] he reformulated the key ideas of supercompilation in the context of a simple first-order functional language (essentially the same as SLL). This was the first work describing the process of supercompilation in full. It was followed by a number of other articles [28, 29, 30, 31, 8, 32] explaining supercompilation and comparing it to other program-transformation methods.

- **2000s. Supercompilation of higher-order functional languages.** The latest wave of renewed interest in supercompilation started in the second half of the 2000s. Many new milestones were passed (supercompilers for call-by-need, call-by-value, etc.), but the single most important shift so far has been from first-order to higher-order functional languages. Regular international workshops on supercompilation are being held – META-2008 [33], META-2010 [34], META-2012 [35], META-2014 [36].

Current trends

This is only an introductory article on supercompilation, and a detailed survey of the current state of the field is beyond its scope. Instead, we list some existing supercompilers in the next section. Very briefly, one of the main goals in current research is to build a supercompiler which can transcend the experimental status and become practically useful for a larger audience. Another important trend is the application of the ideas of supercompilation in new contexts.

If you are curious to see a more detailed picture of the current state of supercompilation research, many recent PhD theses and other articles [5, 37, 38, 11, 39, 14, 40, 15, **?**] contain very good overview sections.

Existing supercompilers

We list here some existing supercompilers. The list is not exhaustive; it contains implementations which have interesting features, are publicly accessible, and which are either actively developed, or at least have been until recently. Note that all these implementations are considered experimental.

- **SCP4 [41].** SCP4 is the latest incarnation of the first ever supercompiler, which was developed for the Refal language. It supercompiles programs in a recent version of Refal, Refal-5. SCP4 utilizes some of the features of Refal, which make it particularly suited to supercompilation, like the associativity of sequence concatenation. It also features a number of extensions of the basic super-compilation method: recognition of constant functions, recognition of concatenation monomials, and collection and analysis of output formats. SCP4 is described in detail in [39] and in the monograph [42]. SCP4 can extend the program domain in the following sense: if the input program loops or stops with an error on certain inputs, it is sometimes possible for the residual program to successfully terminate on these inputs.

- **A (nameless) supercompiler for the TSG language [43].** TSG is a greatly simplified version of LISP, which is "flat" (no nested function calls are al-

lowed). Besides the experimental supercompiler for TSG described in [32], there exist implementations for a number of other methods, which have arisen in the context of supercompilation, and which, unfortunately, get less attention than they deserve. These methods include neighborhood analysis, neighborhood testing, inverse programming, and nonstandard semantics [28].

- **Jscp[44] [45].** A supercompiler for Java and the first supercompiler for an object-oriented, real-world language. One of the main conclusions of this experiment: supercompilation of non-functional languages is much more complicated. Jscp, unlike most other supercompilers in this list, is closed-source.

- **A supercompiler for Timber [46].** Timber is a pure object-oriented language with call-by-value semantics mostly inspired by Haskell. Its supercompiler treats the purely functional subset of the language. The main goal of this project is to achieve similar optimizations when supercompiling a call-by-value language – compared to a call-by-name language – while fully preserving the semantics of the original program. Good descriptions exist in [14, 15].

- **Supero [47].** A supercompiler for a subset of Haskell. Supero is the first supercompiler to treat a call-by-need language, which was actually the main goal of the project [11, 12].

- **SPSC [48].** SPSC is another toy supercompiler for SLL. The main goal was to implement positive supercompilation, as described (but not implemented) by Sørensen [31, 8]. SPSC is described in [49].

- **HOSC [50].** HOSC is a supercompiler for a (call-by-name) Haskell subset. Unlike many other projects, where the main goal is program optimization, here the main interest lies in program analysis by supercompilation. Besides standard supercompilation, HOSC can also perform "two-level" supercompilation, based on discovering and applying "improvement" lemmas [40, 51].

- **Optimusprime [52].** Optimizes functional programs, which are then run on a specialized FPGA-build processor (Reduceron) [19].

- **CHSC [53].** Another supercompiler for a Haskell subset [13]. The main goal of this project was to include supercompilation as an optimization pass inside GHC. An important technical difference is that CHSC does not perform lambda-lifting as a preprocessing step, unlike most other supercompilers for higher-order languages.

- **Distillation [54].** Most supercompilers use configurations based on expressions with free variables. Distillation uses configurations, each of which in

turn is similar to a configuration graph. In other words, the nodes of distillation's configuration graphs contain nested graphs; and folding and generalization must be defined over graphs. Such complications are the price for obtaining more powerful optimizations, compared to standard supercompilation [55, 56].

- **MRSC [57]** A toolkit for building "multi-result" supercompilers. Multi-result supercompilation [58] is another generalization of the standard supercompilation technique, where the supercompiler is permitted to return multiple residual programs (all correct, of course). While this may seem useless at first glance, it turns our that this extension opens the way for a much more flexible and modular description of supercompilation, and in the same time permits some optimizations, which are out of reach for the standard methods.

- **A Coq framework for building formally verified supercompilers [59]** A framework in Coq for building modular supercompilers by just defining the basic ingredients (configurations, driving, folding, etc.), and plugging them into a prefabricated generic supercompiler [60]. The organization of the supercompiler is very similar to the one we have just described, and there is a complete example supercompiler for a language very close to SLL. The main advantage of the framework is that it facilitates formally establishing the correctness of each new supercompiler, by just having to prove some simple properties concerning the basic building blocks.

- **A supercompiler for Erlang [61]** A recent first step towards a practical supercompiler for another popular language, Erlang.

- **TT-Lite [62]** A supercompiler for a version of Martin-Löf's Type Theory (MLTT). TT-Lite is the first supercompiler for a terminating (non-Turing-complete) language with dependent types [63]. Another interesting feature is the generation of "certificates", containing formal, automatically verifiable evidence that the residual program is equivalent to the input one in each specific instance. The certificates themselves are encoded as MLTT terms.

Instead of a conclusion

The main driving force leading to this article was the following idea: using Haskell to describe – in a clear and modular way – the main ingredients of a minimalistic supercompiler and show how they fit together. As a result we arrived at the following definition:

```
supercompile :: Task -> Task
```

```
supercompile (e, p) =
        residuate $ simplify $ foldTree $
                buildFTree (addPropagation $ driveMachine p) e
```

The text of the article is, in fact, just an attempt to describe each of the parts of this definition and to show what effects they can achieve together.

We hope the reader will take some time to study SC Mini's sources, where comments describe some interesting technical details of the implementation.

Where can we go from here?

If you are interested in learning more about supercompilation, here is a short list of possible next steps:

1. The articles [8, 31] are still very good and readable introductions to the standard techniques used in supercompilation, including some not covered here, such as using homeomorphic embedding as a whistle and using "most specific generalization" as a generalization algorithm.
2. the Google group "Supercompilation and Related Techniques" [64] contains the most up-to-date discussions and announcements concerning supercompilation.

References

[1] V. F. Turchin. The concept of a supercompiler. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, 8(3):pages 292–325 (1986).

[2] Simon L. Peyton Jones. Interview. **Practice of functional programming**, (6) (2010). (In Russian) `http://fprog.ru/2010/issue6/interview-simon-peyton-jones/`.

[3] `https://github.com/ilya-klyuchnikov/sc-mini`.

[4] Turchin V.F. Equivalent transformations of refal programs. Trudy CNIPIASS 6, CNIPIASS (1974).

[5] M. H. Sørensen. **Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation**. Master's thesis, Københavns Universitet, Datalogisk Institut (1994).

[6] V. F. Turchin. Program transformation by supercompilation. In **Programs as Data Objects**, volume 217 of **LNCS**, pages 257–281. Springer (1986).

[7] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. **SIAM Journal on Computing**, 6:page 323 (1977).

[8] M. H. Sørensen and R. Glück. Introduction to supercompilation. In **Partial Evaluation. Practice and Theory**, volume 1706 of **LNCS**, pages 246–270 (1998).

[9] P. Wadler. Deforestation: Transforming programs to eliminate trees. In **ESOP '88**, volume 300 of **LNCS**, pages 344–358. Springer (1988).

[10] A.B. Ferguson and P. Wadler. When Will Deforestation Stop? In **1988 Glasgow Workshop on Functional Programming** (1988).

[11] N. Mitchell. **Transformation and Analysis of Functional Programs**. Ph.D. thesis, University of York (2008).

[12] N. Mitchell. Rethinking supercompilation. In **ICFP 2010** (2010).

[13] Max Bolingbroke and Simon L. Peyton Jones. Supercompilation by evaluation. In **Haskell 2010 Symposium** (2010).

[14] P.A. Jonsson. **Positive supercompilation for a higher-order call-by-value language**. Licentiate thesis, Luleå University of Technology (2008).

[15] P.A. Jonsson. **Positive Supercompilation for Higher-Order Languages**. Ph.D. thesis, Luleå University of Technology (2011).

[16] Alexei Lisitsa and Andrei Nemytykh. Towards verification via supercompilation. In **COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference**, pages 9–10. IEEE Computer Society, Washington, DC, USA (2005).

[17] N. Mitchell and C. Runciman. A supercompiler for core haskell. In **Implementation and Application of Functional Languages**, volume 5083 of **Lecture Notes In Computer Science**, pages 147–164. Springer-Verlag, Berlin, Heidelberg (2008).

[18] Ilya Klyuchnikov and Sergei Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In **Perspectives of Systems Informatics**, volume 5947 of **LNCS**, pages 193–205 (2010).

[19] Jason S. Reich, Matthew Naylor, and Colin Runciman. Supercompilation and the Reduceron. In **Proceedings of the Second International Workshop on Metacomputation in Russia** (2010).

[20] Max Bolingbroke and Simon L. Peyton Jones. Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation. In **ICFP 2011** (2011).

[21] V. F. Turchin. **The phenomenon of science. A cybernetic approach to human evolution**. Columbia University Press, New York (1977). `http://pespmc1.vub.ac.be/POSBOOK.html`.

[22] V. F. Turchin. **The Language Refal: The Theory of Compilation and Metasystem Analysis**. Department of Computer Science, Courant Institute of Mathematical Sciences, New York University (1980).

[23] V. F. Turchin. The algorithm of generalization in the supercompiler. In **Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop** (1988).

[24] V. F. Turchin. Program transformation with metasystem transitions. **Journal of Functional Programming**, 3(03):pages 283–313 (1993).

[25] V. F. Turchin. Supercompilation: Techniques and results. In **Perspectives of System Informatics**, volume 1181 of **LNCS**. Springer (1996).

[26] V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In **Partial Evaluation**, volume 1110 of **Lecture Notes in Computer Science**, pages 481–509. Springer (1996).

[27] Robert Glück and A.V. Klimov. Occam's razor in metacompuation: the notion of a perfect process tree. In **WSA '93: Proceedings of the Third International Workshop on Static Analysis**, pages 112–123. Springer-Verlag, London, UK (1993).

[28] Abramov, S. M. **Metavychisleniya i ih primenenie (Metacomputation and its applications)**. Nauka (1995). In Russian.

[29] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd (editor), **Logic Programming: The 1995 International Symposium**, pages 465–479 (1995).

[30] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In **Selected Papers From the International Seminar on Partial Evaluation**, volume 1110 of **LNCS**, pages 137–160 (1996).

[31] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. **Journal of Functional Programming**, 6(6):pages 811–838 (1996).

[32] Abramov, S. M. and Parmyonova, L. V. **Metavychisleniya i ih primenenie. Superkompiliacia (Metacomputation and its applications. Supercompilation, in Russian)**. Program Systems Institute of the RAS (2006). In Russian.

[33] Program Systems Institute. **First International Workshop on Metacomputation in Russia** (July 2–5 2008). `http://meta2008.pereslavl.ru/`.

[34] Program Systems Institute. **Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia** (July 1-5 2010). `http://meta2010.pereslavl.ru/`.

[35] Program Systems Institute. **Third International Valentin Turchin Workshop on Metacomputation** (July 5-9 2012). http://meta2012.pereslavl.ru/.

[36] Program Systems Institute. **Fourth International Valentin Turchin Workshop on Metacomputation** (2014). http://meta2014.pereslavl.ru/.

[37] Jens Peter Secher. **Perfect Supercompilation**. Master's thesis, Department of Computer Science, University of Copenhagen (1999).

[38] J.P. Secher. **Driving-Based program transformation in theory and practice**. Ph.D. thesis, Department of Computer Science, Copenhagen University (2002).

[39] Nemytykh A.P. **Specialization of functional programs using supercompilation**. Phd thesis, Program Systems Institute of the RAS (2008). In Russian.

[40] Klyuchnikov I.G. **Inferring and proving properties of functional programs by means of supercompilation**. Phd thesis, M.V.Keldysh Institute of Applied Mathematics of the RAS (2010). In Russian, http://pat.keldysh.ru/~ilya/klyuchnikov-phd.pdf.

[41] http://www.botik.ru/pub/local/scp/refal5/refal5.html.

[42] Nemytykh A.P. **Supercompiler SCP4. General structure**. LKI, Moscow (2007).

[43] http://www.botik.ru/~abram/mca/.

[44] A.V. Klimov. An approach to supercompilation for object-oriented languages: the java supercompiler case study. In **First International Workshop on Metacomputation in Russia** (2008).

[45] http://supercompilers.ru.

[46] http://timber-lang.org/.

[47] http://community.haskell.org/~ndm/supero/.

[48] http://code.google.com/p/spsc/.

[49] Ilya Klyuchnikov and Sergei Romanenko. SPSC: a Simple Supercompiler in Scala. In **PU'09 (International Workshop on Program Understanding)** (2009). http://spsc.googlecode.com/files/Klyuchnikov__Romanenko__SPSC_a_Simple_Supercompiler_in_Scala.pdf.

[50] http://code.google.com/p/hosc/.

[51] Ilya Klyuchnikov. Towards effective two-level supercompilation. Preprint 81, Keldysh Institute of Applied Mathematics (2010). http://library.keldysh.ru/preprint.asp?id=2010-81&lg=e.

[52] http://hackage.haskell.org/package/optimusprime.

[53] https://github.com/batterseapower/chsc.

[54] https://github.com/distillation/distill.

[55] G. W. Hamilton. Distillation: extracting the essence of programs. In **Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation**, pages 61–70. ACM Press New York, NY, USA (2007).

[56] G. W. Hamilton. A graph-based definition of distillation. In **Second International Workshop on Metacomputation in Russia** (2010).

[57] https://github.com/ilya-klyuchnikov/mrsc.

[58] I. G. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute (2011). http://library.keldysh.ru/preprint.asp?id=2011-77&lg=e.

[59] https://sites.google.com/site/dkrustev/Home/publications.

[60] Dimitur Krustev. Towards a framework for building formally verified supercompilers in Coq. In Hans-Wolfgang Loidl and Ricardo Peña (editors), **Trends in Functional Programming**, volume 7829 of **Lecture Notes in Computer Science**, pages 133–148. Springer Berlin Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-40447-4_9.

[61] https://weinholt.se/.

[62] https://github.com/ilya-klyuchnikov/ttlite.

[63] Ilya Klyuchnikov and Sergei Romanenko. TT Lite: a supercompiler for Martin-Löf's type theory. Preprint, KIAM, Moscow (2013). http://library.keldysh.ru//preprint.asp?lg=e&id=2013-73.

[64] http://groups.google.com/forum/#!forum/supercompilation-and-related-techniques.

A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP

by Henrik Bäärnhielm ⟨redstar@kth.se⟩
and Daniel Sundström ⟨daniel@monkeydancers.com⟩
and Mikael Vejdemo-Johansson ⟨mikael@johanssons.org⟩

In March 2013, the battleship Småland in the Gothenburg harbor was transformed into a spaceship, the Monitor Celestra of the Battlestar Galactica fleet, for a three day Live Action Roleplaying Game. To support the feeling of total immersion into the game world, we built extensive technological support for the game experience, including a sound system with a programmable control layer written in Haskell.

In this article we will describe this control layer and our experiences building and using it for this game project.

The Monitor Celestra is a game that transformed a World War II era battleship into a space ship, inviting the participants of the game to enter the Battlestar Galactica [1] universe and forge their own path into the future. The game, a Nordic Style Live Action Roleplaying Game (LARP) assigns roles to the participants – as crew, passengers and refugees on board the space ship – and act out the resulting story within a framework of plots, story lines and clues. The medium differs from theatre plays in the lack of a script and from improvised theatre in the lack of an audience – to consume the medium is to participate in it.

Nordic Style LARPs are characterized, among other things, by a strong focus on immersion. The goal is for participants to feel embedded in the game world without the distraction of the real world disturbing the fiction. In recent years, more and more games have been organized that include technological support to improve the feeling of immersion into the game world.

For **The Monitor Celestra**, full immersion was implemented using a number of different techniques: the signage of the ship was replaced with signs that were

in the Battlestar Galactica aesthetic, the ship had custom-built control panels controlled a space dynamics simulator, and the game master team had the ability to trigger a large array of scripted, partially scripted or improvised events that launch sounds and influence the ship control panels to convey the story.

An important part of this immersion was 15 pairs of loudspeakers, each pair hooked up to a Raspberry Pi [2] and connected to a wired ethernet network. These loudspeakers played localized sound effects as well as an ambient sound backdrop through the entire game, drowning out the sounds of inner city Gothenburg and supporting the game experience by confirming consequences of player actions through mediated responses.

Background

LARP is a form of game where participants receive roles and then proceed to enact their roles within a framework of plots, story lines and clues. Traditionally, LARPs have been focused on telling stories in a fantasy/medieval setting, but the form has seen a wider spread of genres over years. LARP has developed into various design schools and style – mostly based on geographical distribution. Thus, there are Nordic LARPs, American LARPs and Russian LARPs, to name a few [3]. Each school of design has its own theories on what constitutes a good LARP and what goals one must strive to achieve with the game design, participation and outcome of the game.

In the same tradition as table-top role playing games and improvisation theatre, LARP is played out by players in real time in the same general physical area. Players wear costumes and make props to better simulate their respective characters. In general, participants receive instructions beforehand regarding specific game rules, other characters, shared background information, etc. Organizers define social mechanics and hard game rules for resolving combat, conflicts or sex.

Nordic LARP

The Monitor Celestra belonged to the Nordic LARP school of design. This school could be argued to emphasize a few different points as being major when creating or participating in the experience:

- The game should represent the world as faithfully and completely as possible (games which fulfill this criterion are commonly referred to as 360° games.) Everything players see and/or experience represents itself within the game – a gun is a gun, a knife is a knife.
- Participants should put emotional engagement and experience above achieving in-game goals. Powerful emotional impacts should take precedence when

deciding how to move the game forward.

- The players should control the outcome of the game. While parts of the game might be scripted, the main outcome of the game should be left to the players as far as possible.

Due to the nature of the Nordic school of design, games designed within that school tend to focus on moral choices and complex social interactions rather than traditional fantasy stories. There have been games depicting everything from Danish hobos on the road (**The White Road** [4]) and students investigating possessed ladies (**Prosopopeia** [5, 6]), to 1950s families hiding from nuclear war (**Ground Zero** [7]) and terminal cancer patients (**Luminescence** [7]).

LARP vs. Similar activities

What separates LARP from other similar activities such as tabletop roleplaying games, computer games and improvised theatre is still an open question. Researchers [8, 9, 10] have pointed to a few main points;

- Emotional engagements increased radically when players are physically engaged in the activity [9].
- Moral choices and consequences from player actions have larger impact when acted out by other physical players [9].
- Players allow themselves more freedom when acting within a ludic circle,[1] thus allowing for a wider range of emotions, experiences and shared actions [8, 11].
- Reflections on the complexity of social and/or humanistic behavior or models yield a higher level of understanding [10]. Research has shown that when trying to understand complex models of human behaviour, both in extreme situations such as emergencies and in everyday contexts - LARP tend to work very well as tool for exploring motivations, social roles and social systems based on incitements

It has been humorously suggested that LARPing may be seen as the "extreme" sport of interactive experiences due to the efforts involved with staging and participating in games, the impact on participants and the cumulative effects of player actions and organizer design.

Similar Events

There have been other efforts to use technology to enhance LARP experiences within the game[2] as a key part of staging the experience, mainly within the Nordic

[1] *ludic circle*: a shared space of playfulness, from *ludic*: playful

[2] As apart from using techonology for support functions, such as economy, participant management etc.

school of LARP design. Historically, these experiments have been conducted by a small number of game designers and writers. Experiences where technology were used in a similar capacity as on **The Monitor Celestra** includes:

- **Prosopopeia: Bardo 1** A technology-enhanced LARP played in the autumn of 2005. Players used technology concealed in an old reel-recorder to communicate with dead spirits.
- **Prosopopeia: Momentum** An continuation to the Prosopopeia project, **Momentum** played in the autumn of 2006. Players used a wide variety of technology, including hardware built using custom constructed components to explore and wage battles against forces on the other side of death.
- **Felsäkert Läge** A technology-enhanced LARP in the autumn of 2010 situated in a future, post-apocalyptic world, staged in an abandoned mining town in the north of Sweden. Technology was used to simulated surviving technology from before the catastrophe, telling stories and providing quests, items and treasures for the players to find and use in their own game.

An overview of similar experiments (and others) can by found in [7].

Immersive Soundscapes

Earlier efforts on immersive soundscapes include various implementations in Python, support in most holistic game engines and similar. However, the main difference between these implementations and the system built for **The Monitor Celestra** is the spatial scale. Usually, these soundscapes are meant to be enjoyed by a single person in headphones or a handful of persons in a single or a few rooms.

The difference in scale between these implementations and the one described in this paper makes comparisons hard to do. Most issues met by the usual implementations were either deemed irrelevant (such as super-focused spatial placement in the scene – a rough idea was more than enough for a battleship) or too specific (solutions for physical placing loudspeakers in a square room).

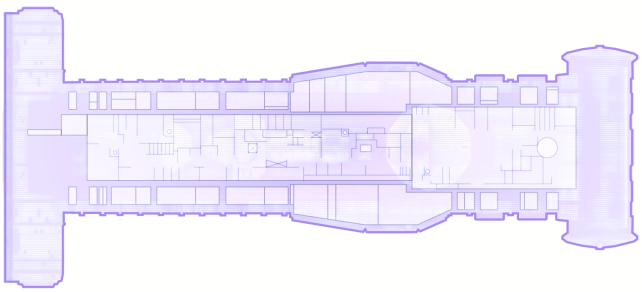


Figure 1: Floor plan of The Monitor Celestra

Structure of The Monitor Celestra

Every LARP differs in structure and design according to the design goals, creators and aims with the project. The Monitor Celestra had three runs with 150 participants each, split over three weekends. The runs were separated from each other and played out the same story with different participants (even though some participants bought tickets for all the runs).

Every run was split into four game sessions of 4–6 hours, with breaks between for sleep and briefings. Food was distributed and eaten while in play. Even though the battleship Småland was equipped with bunks for its crew, fire regulations prohibited sleeping on board the ship. Hence, the participants slept at a nearby hostel between game days.

The story design started out in the setting of Battlestar Galactica: a fleet of refugee spaceships carrying most of the remainder of humanity flee from robot aggressors. In the game, The Celestra is separated from the remaining fleet by accident, and the players have to decide on their actions, while facing various threats: robots hunting them down, colonies with radical political views and demands, and internal conflicts on board the ship. Each of the games ended in a different way, ranging from a valiant effort to save the human race by nuclear suicide taking out a threat before it could reach the rest of the fleet to setting course for deep space and forging a new path for the isolated ship.

Game control system design

The technology team for **The Monitor Celestra** settled at a relatively early stage on an overall design for the game support technology; the choices made included:

- Wired ethernet communication for everything, split into three different parallel networks: sound, game system, and player hackable. Due to the nature of the play area, parallel network backbones were installed for maximum durability and error management.
- Communication between game components over `AMQP` from a RabbitMQ running on a local game server on the main network. Game logic, world engine and game rules were written in Ruby.
- Game consoles and GUIs written in HTML5 Canvas rendered on Chromium running in fullscreen.
- Game consoles able to be used interchangeably with everything being designed for plug-and-play.
- No authentication required within the game network. All published messages can be trusted on a system level.
- A single game component responsible for game world, time progression and

synchronization; in this case, a rack server running latest stable Ubuntu LTS.
- Connection to external network and Internet through a single point-of-entry, the main game server.

These choices were made based on experiences from earlier projects staged in similar contexts where the main body of the work was done by unpaid volounteers in uncontrolled and rough environments.

Sound system design

The original specifications for the Celestra sound system were as follows:

- Each section/room of the ship should have an independent sound producer, so that each room could play different sounds simultaneously. Each sound producer should be able to mix several sounds.
- It should be possible to trigger sound effects at specific locations, both automatically via the game control system and manually by game masters. For example, a torpedo sound in the torpedo room is played when a torpedo is launched (triggered by a player), and a warning sound is played in every room when the hull is compromised.
- To maintain immersion, the sound effects must have low latency.
- It should be possible to trigger sounds at certain future times. For instance, it should be possible to specify that a torpedo hatch closing sound is played exactly one second after the torpedo launch sound.

The rest of the game control system was designed to run on a relatively small number of servers in a control room. One option for the sound system would be to have a sound server with one sound card for each section, connected to pairs of speakers located in each section. However, it was decided that a better solution would be to have actual computers at each section, with their own sound cards, connected to a central computer via wired Ethernet, which would already existed in most rooms. This setup would be easier to build in terms of hardware and would be easier to program for. The computers located next to the speakers in each room then had to be inconspicuous and inexpensive, since quite a large number would be needed. This led us to go for the Raspberry Pi (RP) setup.

The choice of RP added constraints. RP has limited RAM and CPU power; hence, we decided to avoid adding dependencies to the game back-ends on every RP. While most of the game control systems were written in Ruby and communicated over `AMQP`, we decided to implement a special purpose sound daemon in C++, which would run on each RP and communicate with the central sound server directly via sockets. This would enable us to achieve lower latency and not drain resources from the RP.

Sound daemon

All required sounds were stored as sound files on every RP, effectively creating a global sound list $\mathcal{S}$. The sound daemon should then accept a few simple commands on the socket:

- Play sound k from $\mathcal{S}$ at absolute time t, as sound with ID n.
- Stop sound with ID n, at absolute time t.
- Change volume to v on sound with ID n, at absolute time t.

The idea here was that by combining these commands, it would be possible to realise all the required sound system features. Each RP was time synced to the rest of the network using NTP [12], hence the sound commands could have absolute timestamps with high accuracy. To achieve the possibility of playing multiple sounds at a single RP, the sound daemon communicated with PulseAudio [13], creating one stream per sound being played. To achieve low latency and to simplify the implementation, the daemon read in $\mathcal{S}$ to RAM upon start-up, rather than reading a file upon each play command. This added the constraint that the total uncompressed size of the sounds had to fit into RAM of the RP.

Ambient and story sounds

It turned out that apart from the sound effect type of sounds that the sound daemon was created for, the game required two other types of sounds.

- Ambient sounds, which should be looped in the background in a section. For instance, in the reactor room there should be a constant sound from the reactor.
- Story sounds, which were played at specific events in the game. These were typically very long compared to ordinary sound effects and were triggered explicitly by the game masters.

These types of sounds did not quite fit into the sound daemon. The ambient sounds required looping and needed to be cross faded into themselves to avoid clicks; the story sounds were too large to fit into RAM. Moreover, the low latency requirement for these sounds were much less strict.

We therefore decided to use MPD [14] for these sounds. MPD reads files from disk, hence the sound size would not matter. It also has built-in cross fade and loop support and can output to PulseAudio, so it was compatible with the sound daemon. Each RP was running an MPD daemon, and the sound server communicated with it using a standard MPD client library.

Sound daemon controller

On the sound server, a small piece of controller software was running, which listened to AMQP sound commands, transformed them into sound packets for the daemon or to commands to MPD and sent them off to an RP. It was also responsible for transforming node numbers to IP addresses of RPs.

Like most of the game control software, this controller was written in Ruby. The original idea was that this would be the entry point of the sound system. However, it turned out that its simple interface, while in principle sufficient for all sound purposes, was too low-level for creating the sound scenes that the sound designers wanted to build for the game: crossfades, seamless loops and other sound design features would require a higher abstraction than the play/loop/stop/change volume setup for this interface

Thus, we were led to introduce the intermediate layer in Haskell, described in the following sections. From now on, we will be referring to the sound server described here as the **low-level system**, and to the Haskell layer as the **high-level system**. The Haskell layer works with two separate interface layers; a **low-level interface** facing the low-level system, and a **high-level interface** facing the game master and sound designer teams.

Requirements on the sound scene DSL

The work on building a dedicated domain specific language (DSL) for describing sound scenes started at a relatively late stage of the project, when most other platform decisions had already been made. The need for an intermediate layer emerged from the simultaneous need for low-level simple sound execution protocols that could run on a Raspberry Pi platform and for abstract sound descriptions with a low technical usage barrier to enable the game fiction design and sound design crews to interact with the system.

This setting produced a number of requirements that we tried to adapt to during the project:

Simple output Output from the DSL to the lower level systems needed to have a simple structure, preferably consisting of discrete orders to Play, Loop, Stop or Adjust sounds at sound nodes (both addressed by integers).

Accessible input The DSL needed an easy to read and easy to write format that the artistic side of the project could use to specify sound scenes.

Fast development The DSL needed to be developed during 4 months of volunteered free time, during a time span that included several major conference deadlines. This limited the amount of programmer manpower available.

System compatibility The DSL needed to react to `AMQP` messages, store state in `Redis`, and format its output in a way easy to parse by the receiving low-level and high-level interfaces.

Due to earlier experiences with Haskell, we chose GHC and the Haskell Platform as a host system for the DSL development. An early decision was to use the automatic parser generation in the standardized Haskell `Read` and `Show` classes; later on, we used the `Generics` extension to automatically generate JSON parsers and formatters in order to plug into the existing `AMQP` message passing architecture. This reduced large amounts of DSL design to designing appropriate Haskell types: writing native Haskell code with custom-built data types that represented the domain allowed for large freedom in defining our own semantics while retaining the full power of Haskell packages and programming styles.

Design of the sound scene DSL

From the overall design decisions for the entire sound system and game system, some parts of the design of the sound scene DSL subsystem were clear: we had to interact with a `Redis` database and with an `AMQP` communication system, and we needed to send JSON packages according to a fixed format in order to control the lower level system.

On top of this, we had requirements for swift development, ease of use, and minimizing the amount of extra parsers to write. These criteria were central in selecting Haskell as a platform for the tool: out of the platforms that members of the workgroup were familiar with, Haskell was far more capable of quick development and quick automatic generation of parsers and serializers than all the alternatives.

Our system ended up depending on a family of Haskell packages that cover many of the interoperability requirements. In total, we relied on `aeson` [15], `amqp` [16], `attoparsec` [17], `base` [18], `bytestring` [19], `containers` [20], `ghc-prim` [18], `hedis` [21], `mtl` [22], `regex-posix` [23], `text` [24], and `unordered-containers` [25], for all our library needs.

From these preconditions, we decided that the best way to construct a DSL would be to encode all important information in terms of specific Haskell datatypes, so that Haskell methods for generating parsers and serializers could be used. Additionally, we needed a hierarchy of types bridging the gap between the human-readable game master facing DSL and the machine facing (already defined) JSON protocol.

The resulting hierarchy of datatypes that we decided on was:

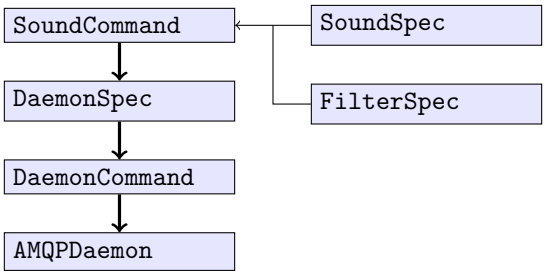


Figure 2: Dependency and compilation path hierarchy for the sound system.

where we had separate compilation steps to transform a `SoundCommand` into a `DaemonSpec`, a `DaemonSpec` into a `DaemonCommand` and a `DaemonCommand` into an `AMQPDaemon`.

These types all had different roles:

`SoundCommand` encoded all orders the system expected from game masters and sound designers. In order to more easily design datatypes, we separated out the descriptions of a sound scene and of a reaction trigger from the command type into the subordinate types `SoundSpec` and `FilterSpec`.

`DaemonSpec` encoded, abstractly, the order types the lower level system accepts.

`DaemonCommand` encoded in full detail a single order for the lower level system.

`AMQPDaemon` was a datatype explicitly constructed to serialize through `Aeson` into a JSON package that the lower level system could parse.

Here, the `SoundCommand` type encoded all orders that the game masters and sound designers wanted to be able to give to the system, while `DaemonSpec` abstractly encoded all command types the lower level system accepted.

All datatypes derived `Read`, `Show` and `Generic`, which allowed us to automate parsing both from a Read-Evaluate-Print-Loop as well as create automatic JSON parsers and encoders from `Aeson`.

SoundCommand

Our separation of the sound scene description from the Haskell layer control commands and the automated reactive triggers builded on their separation into different datatypes. The family of datatypes we designed closely mirrored the task

division for the system. At the top-most abstraction level, there was a data type `SoundCommand` enumerating the various commands that can be given to the system.

```
data SoundCommand =
    Define Id SoundSpec |
    Commit |
    Restore |
    Diagnostic |
    ReadState |
    ReadSounds |
    Compile SoundSpec |
    ReadOut SoundSpec |
    Execute Id SoundSpec |
    Trigger Id FilterSpec SoundCommand |
    SoundCommand :++ SoundCommand |
    Declare Id SoundCommand |
    Call Id |
    Delete Id |
    Nop
    deriving (Eq, Show, Read, Generic)
```

There were commands for interacting with the database state storage: `Commit` and `Restore`; commands for debugging and analyzing what a particular sound scene description was interpreted to: `Diagnostic`, `ReadState`, `ReadOut`; commands for naming and recalling both sound scenes and entire commands: `Define`, `Declare`, `Execute`, `Call`, `Delete`; and commands for triggering sound scenes either through events by `Trigger` or through direct command by `Execute`. Finally, there was `ReadSounds` that reported available sound scenes up to a top level UI layer, and `:++` for chaining commands together as well as a `Nop` that could finish an automatically generated chain of commands for programmatic creation of triggers and action chains.

The definition used the two types `SoundSpec` and `FilterSpec` to parametrize its entries. These were given by

```
data SoundSpec =
    Play File Segment Time Loudness |
    Loop File Segment Time Loudness |
    RadialDecay File Segment Time Loudness |
    SoundSpec :+ SoundSpec |
    Use Id |
    Stop Int Time |
```

```
    Fade Int Time Time Loudness |
    StopId Id Time |
    FadeId Id Time Time Loudness |
    NopS
    deriving (Eq, Show, Read, Generic)
```

and

```
data FilterSpec =
    FilterSpec :& FilterSpec |
    FilterSpec :| FilterSpec |
    MatchAll [FilterSpec] |
    MatchAny [FilterSpec] |
    MatchEvent String |
    MatchSender String |
    MatchKeyValue String String
    deriving (Eq, Show, Read, Generic, Ord)
```

The sound scene specification allowed for playing a sound by name or by index, either once or on an infinite loop, and for stopping and changing volume. These were the operations understood by the low level system as well.

In addition to these, the Haskell layer automatically generated packages to smoothly fade between volume settings (for a spatially distributed decaying sound scape) and for saving and recalling sound scenes. After an initial attempt to design the fades, we ended up building in support for remembering the last set volume for a sound, so that fades could be given by a target loudness rather than by start and stop loudness settings.

The inclusion of `Nop` and `NopS` helped us automate generation of lists of actions; together with the concatenation constructions given by `:++` and `:+`, there was a full monoidal structure on both these datatypes, enabling easy generation of composite commands from lists of command parameters.

Since the entire system actively listened to the `AMQP` traffic of the entire game system, it was easy to include a reactive component: using a simple regular expressions-based recognition engine, we were able to write simple rules that would when matched trigger arbitrary pre-constructed `SoundCommand` actions. The rules were encoded using the type `FilterSpec` and their corresponding actions were encoded with the `Trigger` constructor of `SoundCommand`. All `AMQP` messages in the game system contained a sender, an event key and some collection of key-value pairs, all of which could be regular expression matched with the rules specified as `FilterSpec` entities.

DaemonSpec

The `DaemonSpec` type encoded the abstract payload of a single instruction to the low-level system. An element of type `DaemonSpec` encoded all the descriptive information needed for a low-level system command, without containing transient information required for emitting any particular command package. In particular, there was a serial ID number assigned to low-level command packages to allow later commands to modify a running sound. These ID numbers were not added in the `DaemonSpec` representation, but rather in the next lower representation.

```
data DaemonSpec =
    DaemonPlay Int Segment Time Loudness |
    DaemonLoop Int Segment Time Loudness |
    DaemonStop Int Time |
    DaemonSet Int Time Loudness
              deriving (Eq, Show)
```

DaemonCommand

The `DaemonCommand` encoded a message that could be sent to the lower-level system. In particular, the command encoded a sequential id-number used for later modifications of looping sounds and set up a datatype for easy parsing for the receiving system.

```
data DaemonCommand = DaemonCommand {
      node :: Int,
      dcid :: Int,
      sound :: Int,
      time :: Int,
      volume_left :: Float,
      volume_right :: Float,
      command :: Int
    } deriving (Eq, Show, Generic)
```

AMQPDaemon

The type `AMQPDaemon` really only existed in order to wrap a `DaemonCommand` item for serialization with `Aeson` and transport in an `AMQP` package. The type is defined as:

```
data AMQPDaemon = AMQPDaemon {
      devent :: String,
```

```
      dsender :: String,
      dcmd :: DaemonCommand
    } deriving (Eq, Show, Generic)
```

with custom JSON instances created by

```
instance FromJSON AMQPDaemon where
    parseJSON (Object v) = AMQPDaemon <$>
                           v .: "event" <*>
                           v .: "sender" <*>
                           v .: "data"
    parseJSON _ = mzero

instance ToJSON AMQPDaemon where
    toJSON ad = object ["event" .= devent ad,
                        "sender" .= dsender ad,
                        "data" .= dcmd ad]
```

These were the only parser and encoder instances we wrote ourselves for this project.

Persistent state

There was a number of pieces of information the system needed access to, with various levels of persistence. We designed a tiered state type consisting of a serialisable section and a collection of transient state properties, described by

```
data SST = SST {
      serst :: SerST,
      dbconn :: R.Connection,
      achan :: A.Channel,
      cmdid :: Int
    }
```

Here, we encoded instance-specific connection data for the `Redis` database in `dbconn`, instance-specific connection data for the `AMQP` communication channel in `achan`, and an instance counter for sequential command ids in `cmdid`. The rest of the state was stored in the `serst` (**ser**ializiable **st**ate) field, which was saved to the database in order to persist settings between runs.

The serializable state in turn was given by

```
data SerST = SerST {
      soundscapes :: M.HashMap String SoundSpec,
```

```
    commands :: M.HashMap String SoundCommand,
    triggers :: [(Id,(FilterSpec, SoundCommand))],
    loops :: [(Int, (Segment, Int, Loudness))],
    tags :: M.HashMap String [Int]
  } deriving (Show, Generic)
```

where we make extensive use of the strict hashmap implementation from the `unordered-containers` package.

Here, `soundscapes` saved all named sound scape descriptions; `commands` saved all named sound system commands; `triggers` saved trigger definitions and is iterated through whenever a package showed up that the system might react to; `loops` saved currently playing loops and their most recently known loudness; and `tags` saved a lookup table from human readable names to command ids.

In addition to these, there was a pair of hardcoded lists defined in the source code itself: `playable` and `loopable`. These two lists contained the names used throughout the sound system to refer to all playable sounds, in an order kept synchronized with playlists on both the lower level system daemon and on the MPD instances. From these were also derived two hashmaps `playDict` and `loopDict` to enable faster index lookups given the names.

Sound scape design daemon

The library described above was then used by a daemon which ran throughout the game on one of the game control servers and reacted dynamically to instructions arriving by `AMQP`. This daemon stored the state in an `IORef` and used the callback structures in the `amqp` package to listen for and react to `AMQP` messages. The entire logic of the server was encapsulated in this callback and the functions it called.

The `callback` function parsed out the payload of the received `AMQP` package and checked whether it matched a regular expression. If so, it parsed the contained command and acted on it; if not, it ran the package through all defined patterns in `triggers` and ran the associated action for each matching pattern. This linear lookup may have been slower than more complex solutions, but it had the benefit of being easy and reliable to design and was probably fast enough for this application. In the end, as we will describe later, there were some latency issues with the system as a whole. Our diagnostics of the trigger list handling were inconclusive, but did not seem to generate the observed latencies.

The function `action` executed a `SoundCommand` and carried the entire logic of the system. This is where the data type was translated into actual reactions. Most of the implementations were straightforward: read the current state from the `IORef` variable, extract relevant parts of the state, and then either assemble a package for `AMQP` or for `Redis` and send it out, or modify the running state according to the

received instructions. By far the most involved of these was the implementation of the `Execute` command, responsible for sending out low-level instructions. This command constructed `DaemonSpec` descriptions, assigned sequential command id numbers, packed the result into `AMQP` packages, and – depending on the exact type of each command – modified the state to remember the details of the sent commands for later recall when constructing fades or stops.

Large swathes of the daemon code were reused to construct a command line interface that generated `AMQP` packages for controlling the system, allowing for an accessible debugging and programming interface.

Usage examples

In this section, we will give a few example programs written in the DSL, to give a flair for the kind of scene description language that emerged from our design choices.

There were different types of sounds specified by game masters and sound designers, based on the planned usage and design requirements of the sound. There were ambient sound loops used to simulate states such as running the ships main reactor or the sounds of massive computer banks running in the sensor processing room. There were feedback sounds triggered in response to participant actions, such as loading a torpedo or triggering a sensor sweep. These sounds mainly functioned as both confirmation to the participant that the system has registered the event and to communicate the occurrence of an event to participants in other parts of the ship. Apart from these sounds, the game masters had access to a wide variety of sounds and noises with no predetermined meaning in order to simulate everything from debris hitting the outer hull to onboard fighting, as well as prerecorded prologues used to segue players into the game at the start of each game session.

Some sounds needed to run through an Attack-Decay-Sustain-Release sequence, composed of multiple sound files and an arbitrarily long sustain-phase. Other sounds needed to trigger with fixed time intervals.

All sounds needed to be played on a set of speakers – but not all sounds were to be played on all speakers when played. For example, sounds of airlock doors opening were played in the section containing the doors and sections immediately adjacent to it.

We built a setup script to handle all the sound setup in a repeatable and restartable way. The script had a `main` consisting of:

```
main = mapM_ (putStrLn . show) $ <soundlist>
```

where the list of sounds was constructed in bits and pieces. We will explain and give examples for each section of the sound list here.

We also defined and used a single value for simultaneously addressing all sound sites (except the engine room sub-woofer)

```
global = [1..11] :: [Segment]
```

as well as a utility command to help string together sound declarations using the `SoundSpec` level concatenation operator:

```
concatSS = foldl (:+) NopS
```

Noises and game start sounds

The very easiest sounds to program for our DSL were the sounds that just needed to sound somewhere once. These consisted of a single sound file which was run all the way through.

All of these sounds were global. A typical sound definition for an environmental noise would look like the noise of a distant explosion.

```
Trigger "explosion distant 5"
 (MatchEvent "sound.trigger.explosion.distant.5")
 (Execute "explo distant 5"
  (concatSS
   (map
    (\i -> Play "Random/explosions/explo_distant_med_05" i 0 (100,100))
    global)))
```

Listing 1: Reactive event definition to play the sound of a distant explosion.

This sound would be triggered by the game masters by pushing a button in their sound control console that generated a `sound.trigger.explosion.distant.5` event on the joint `AMQP` communication bus.

The four different game starting sound sequences – with mood-setting music and a spoken recap – were triggered in the same way.

The same structure was used for game end sequences: depending on which of the six pre-written game ending sounds was deemed appropriate, a different trigger was sent over `AMQP`. This then phased over to an improvised game master epilogue spoken over the ship's PA system.

```
Trigger "celestra act 3 trigger"
 (MatchEvent "sound.trigger.act3")
 (Execute "celestra act3"
  (concatSS
   (map
    (\i -> Play "Story/Celestra_Akt3" i 0 (100,100))
    global)))
```

Listing 2: Reactive event definition to start the game session starting sound sequences.

Participant-triggered noises

Several of the participant actions were supposed to trigger noises as well. Whenever one of the players initiated a sensor scan or a torpedo loading procedure, sounds were triggered that corresponded to that soundscape. The only fundamental difference between these and the game-mastering sounds described above was that the triggers used were game system events. As an example, we can consider the active scanning sensor system.

```
Trigger "dradispingtrigger"
 (MatchEvent "space.sensor.dradis.ping")
 (Execute "dradisping"
  (concatSS
   (map
    (\i -> Play "112_active_dradis" i 1000 (80,80) :+
           Play "330_activating_dradis" i 0 (100,100))
    global))),
```

Listing 3: The event definition for the sound system to react to a participant-triggered active scan ping

As a player successfully initiates an active scan, the sensor control console sends a `space.sensor.dradis.ping` message over `AMQP`. This is picked up both by the game simulation system, that reacts to the use of the active sensor in the simulation, and by the sound system that launches two sounds to be played on the entire ship. First (with time-delay 0), the sound `330_activating_dradis`: a synthetic voice announcing the activation of the dradis system, and then, a second later (time-delay 1000ms), the actual ping sound `112_active_dradis`.

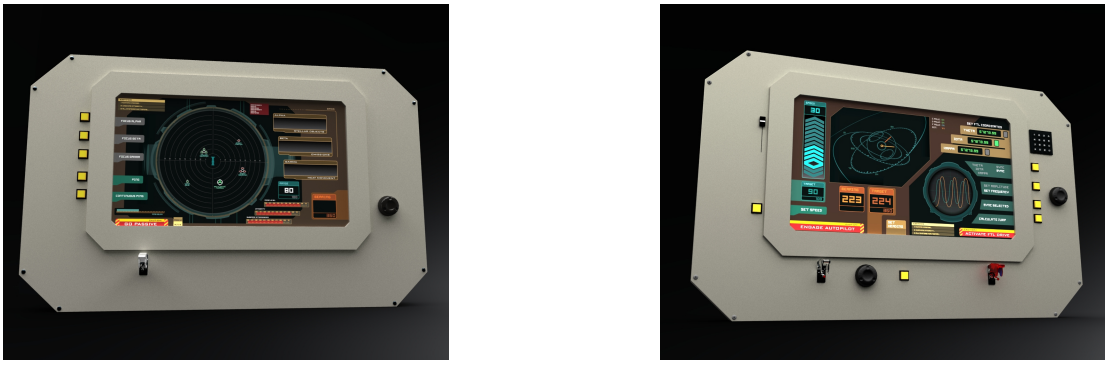


Figure 3: The control panels for the active scanner and the FTL drive.

Ambient loops

The game was designed with three ambient noise loops: one global, with the general soundscape of a space ship in action, and two localized loops with reactor sounds and the command bridge sounds. All these needed to be started as each game session started up, and we built a single trigger to create all of them:

The bridge was controlled by stations 8–11, while the large sub-woofer system for the deafening noise of the reactor engines was sound station 12.

In addition to these loops, one more looping sound was included. If a general alarm was sounded for some reason, the blaring siren sound sat in a loop construct.

Attack-decay-sustain-release

A large family of the sounds used in the soundscape went through a sequence of sound files, with fades between them. First, one sound would start the sequence. Next, a loopable sound would keep the sequence running – preferably of a length that could be determined by the game masters at will. Finally, a sound would finish the noise sequence, often indicating success or failure of the corresponding player action.

These were common enough and repetitive enough that we ended up building a specific function to generate them. In the `SoundSpec.hs` source file, we defined a function `attackLoopDecay` that generated a sequence of chained sound commands.

First, we execute the sound `atF` at loudness `atL` and at all the locations in `locs`. Next, we launch the loop after a delay of `t` and at loudness 0. We launch our cross-fade sequence at `t+100`ms going from the attack sound to the loop sound. Finally, we generate a family of triggers for the possible phasing out options. Each such trigger has a numbered identifier, and contains both the required action for

```
Trigger "startup loops trigger"
 (MatchEvent "sound.trigger.startup")
 (Execute "bridge ambience loop"
   (concatSS
    (map
     (\i -> Loop "136_bridgeambience" i 0 (10,10))
     [8,9,10,11])) :++
  Execute "reactor ambience loop"
   (Loop "124_reactor_run" 12 0 (60,60)) :++
  Execute "sound of celestra loop"
   (concatSS
    (map
     (\i -> Loop "103_soundofcelestra" i 0 (20,20))
     global))),
```

Listing 4: Reactive trigger that launches the ambient sound loops at the start of the game.

```
Trigger "alarm trigger"
 (MatchEvent "sound.trigger.alarm")
 (Execute "alarm"
  (concatSS
   (map
    (\i -> Loop "137_generalalarm" i 0 (50,50))
    global))),
```

Listing 5: Reactive trigger that launches the general alarm siren sound as a loop.

```
attackLoopDecay ::
  Id -> -- ^ Slug to generate ids for this command set
  Id -> -- ^ atF: attack sound
  Loudness -> -- ^ atL: attack loudness
  Int -> -- ^ t: time before fadeover
  Id -> -- ^ lpF: loop sound
  Loudness -> -- ^ lpL: loop loudness
  [Int] -> -- ^ locs: location list
  [(FilterSpec, SoundCommand)] -> -- ^ opts: Options for phasing out
  SoundCommand
attackLoopDecay slug atF atL t lpF lpL locs opts =
 Execute (slug ++ "attack")
         (foldl (:+) NopS (map (\i -> Play atF i 0 atL) locs)) :++
 Execute (slug ++ "loop")
         (foldl (:+) NopS (map (\i -> Play lpF i t (0,0)) locs)) :++
 Execute (slug ++ "xfade")
         ((FadeId (slug ++ "attack") t (t+100) (0,0)) :+
          (FadeId (slug ++ "loop") t (t+100) lpL)) :++
 (foldl (:++) Nop
  (map
   (\ (i,(fs,sc)) ->
     Trigger (slug++"trigger"++(show i))
             fs
             (sc :++
              foldl (:++)
                    (Execute (slug++"stop")
                             (StopId (slug ++ "loop") 0))
                    (map
                      (\j -> Delete (slug ++ "trigger" ++ show j))
                      ([1..length opts]))
   ))
  (zip [1..] opts)))
```

Listing 6: The `attackLoopDecay` utility function

the phasing out option in `sc`, as well as commands to stop playing the loop and to clean out all the alternative phase out options.

With this utility function in place, we are able to construct sound sequences for potentially failing player-initiated events – such as using the Faster-Than-Light jump drive.

```
Trigger "ftl jump trigger"
 (MatchEvent "space.console.ftl_jump_started")
 ((Execute "ftl jump voice"
   (concatSS
    (map
     (\i -> Play "314_initiating_FTL" i 0 (80,80))
     global))) :++
  (attackLoopDecay "ftl jump "
     "117_FTLspinningup" (80,80) 150
     "123_FTLspinupcomplete" (80,80)
     global
     [(MatchEvent "space.console.ftl_jump_completed",
       Execute "ftl jump completed"
        (concatSS
         (map
          (\i -> Play "119_FTLjump" i 0 (80,80) :+
                 Play "318_FTLcomplete" i 9000 (100,100))
          global))),
      (MatchEvent "space.console.ftl_jump_failed",
       Execute "ftl jump failed"
        (concatSS
         (map
          (\i -> Play "118_FTLfail" i 0 (80,80) :+
                 Play "317_FTLmalfunction" i 4000 (100,100))
          global)))]))
```

Listing 7: Reactive trigger event to launch the FTL jump sound sequence, with two options for sequence finish: one for success, one for failure.

When the FTL command console initiates a jump sequence, the `AMQP` message `space.console.ftl_jump_started'` is issued. This launches a sound system reaction that first triggers the synthetic voice announcing ship-wide that FTL has been initiated, by playing `314_initiating_FTL` everywhere.

Next, we call the `attackLoopDecay` construct to generate the sounds for spinning

up and running the FTL engine. The related sounds are named `ftl jump attack`, `ftl jump loop`, `ftl jump xfade`, `ftl jump trigger 1` and `ftl jump trigger 2`.

The two triggerable end sounds are triggered with the game master generated `space.console.ftl_jump_completed` and `space.console.ftl_jump_failed` messages respectively, and play a sound and a synthetic voice message corresponding to the result.

Countdown

Never actually used in game, we also prepared a way to generate a synthetic voice counting down. We had pre-recorded numbers that could be concatenated, but needed to string them together programmatically.

To do this, we adapted the game engine to send out timing signals over `AMQP` every second, every 10 seconds, every 30 seconds and every minute. With these, we could construct functions that generated triggers listening to these timing signals and generating appropriate sound commands for the countdown sequence. They were all variants of the same fundamental structure, illustrated in Listing 8

```
minutes n =
 Declare
  ("countdown " ++ show n ++ "m0s")
  (Trigger "countdown"
   (MatchEvent "sound.heartbeat.minute")
   (Execute "countdown"
    (concatSS
     (map
      (\i ->
       Play "Countdown/T minus" i 0 (100,100) :+
       Play ("Countdown/" ++ show n) i 1000 (100,100) :+
       Play "Countdown/Minutes and counting" i 2000 (100,100)) global)) :++
    Call ("countdown " ++ show (n-1) ++ "m0s")))
```

Listing 8: Function to parametrically define a trigger event to launch a minute-by-minute count down sound sequence.

Calling `minutes 3` would store a `SoundCommand` named `countdown 3m0s` that generated a trigger waiting for the next `sound.heartbeat.minute` event and then playing a synthetic voice saying *Countdown T minus three minutes and counting*. Finally, the event would look up and call the stored `SoundCommand` named

`countdown 2m0s`.

For the last 15 minutes, the countdown would drop down to counting every 30 seconds; for the last minute, it would count every 10 seconds, for the last 10 seconds, it would count every second.

Access to these chains of callable commands was given to the game masters by generating triggers through versions of the trigger given in Listing 9.

```
minuteTrigger n =
 Trigger
  ("countdown trigger " ++ show n ++ "m0s")
  (MatchEvent ("sound.trigger.countdown." ++ show n ++ "m0s"))
  (Call ("countdown " ++ show n ++ "m0s"))
```

Listing 9: Game master callable triggers to launch the countdown sequence from a parametrizable starting time.

When an `AMQP` message of the form `sound.trigger.countdown.25m0s` arrives, the trigger generated by a call to `minuteTrigger 25` is activated. This trigger looks up and calls the stored `SoundCommand` named `countdown 25m0s`, which hooks into the chains of `SoundCommand`s described above.

Experiences

The experiences gathered in this project can be organized into the various stages of the project.

Rapid development and package designs

It turns out that the `amqp` package expects lazy bytestrings and the `hedis` package expects strict bytestrings. The `bytestring` package installed had no automatic way to convert between these – and it took a while in the project to figure out how to make all systems interoperate.

Several features of the resulting sound scape daemon were direct consequences of package and platform choices. The automatically generated serialization and parsing capabilities, both in the `Show`/`Read` dyad and in the `Generic` generation of JSON and bytestring serializers all mean that parsing and serialization worked automatically and immediately. Using the callback structure of the `amqp` library also meant that a reactive server loop was provided essentially for free.

All these features contributed to a rapid development with large amounts of functionality arriving early in the process.

Interfacing to other component platforms

Since the entire project was communicating by `AMQP` with JSON-encoded packages already, the use of these standards made interoperability easy. One of the most difficult parts of this side of the development process was in inspecting emitted JSON packages to figure out the particular encoding that the `Generic`-generated JSON parser worked with, so that appropriate and parseable structures could be emitted from the Ruby control interfaces.

Once the reactive interfaces through the `amqp` module were in place, it quickly became clear to us that the easiest way to provide user interfaces to the daemon would be through a Ruby web application front end that emitted packages crafted to trigger specific rules in our trigger system.

Interfacing to creative design teams

One original expectation that did not work out as expected was to produce a language for other users to work with; as often is the case with large scale projects relying on voluntary work and expansive creative vision, work on the project progressed beyond the launch of the first performance. The sound design teams worked on sound design far beyond the stage where they would have needed to familiarize themselves with syntax and functionality of the sound specification system to use it themselves.

Instead, as a compromise, we requested all relevant information to encode the designed soundscapes; and iterated through requests for more information and educated guesses until soundscapes were defined and could be tested. Here, the choice of embedding the language within Haskell paid off; the sound specification programmer was able to use many Haskell-specific idioms and structures to produce long sequences of parametrized and repetitive sound scape definitions. These were in the end stored as commands stored in `triggers`, listening to specific triggering packages, often creating new triggers for later reactions. This enabled constructions where a player action (pressing the Load Torpedos button) would start some sounds (a loading torpedo noise), but then wait for game master confirmations before playing further sounds (a loading-finished noise and a robotic voice confirmation). The later sounds would be temporarily installed trigger definitions waiting for a confirmation package and including cleanup code that erased all the temporary trigger conditions after sound execution, as described in the examples above.

Latency in performance

There were latency issues in the performance use. We had, in the end, 320 trigger expressions, each carrying up to 300 sound play events in short sequences. At

the first performance, the on-site crew noticed latencies reaching up towards 8–10 seconds between triggering action and sound execution. Before we were able to isolate the cause of these issues, and while we were still comparing complexity properties of the chosen containers – Haskell lists and strict hashmaps depending on whether inspection of all members would be a common operation or not – the performance issues decreased to within a few seconds. Given the acoustic layout of the ship, this was deemed sufficient for further performances.

It should be noted that the high-latency results required some dramaturgic edits; certain sound effects were not used because they were dependent on low latency, and thus infeasible even with the reduced performance issues. It was decided between the first and the second performance to run with the system as built and not try for specific further optimizations.

Hardware Issues

An issue arouse based on the hardware. Once the system was properly deployed and the project entered the testing and verification stage, it was discovered that before playing sounds, the system emitted a few seconds of pops and noise regardless of the sound played. After researching the issue, this was proven to be a known issue with the RP platform, stemming from the way power supply is routed to the sound chips on the PCB. Known workarounds existed which were applied [**?**]. These resolved enough of these issues for the solution to be acceptable.

Experiences on-site

The system was installed in the war ship two days before start of the game. Installation was done in three steps; preparing clients, server-side, and on-site configuration. Since circumstances at the location were more spartan than other, off-site locations available to the team – most preparation was done in-office and then transported to the actual game location.

Client preparation consisted of pre-loading SD cards for the Raspberry Pis with client daemons, the pre-configured sound library and connection bridges. A disk image was prepared and then duplicated onto the memory cards. This saved time at installation time, but proved to be a bad choice on-site. Certain updates could be performed over the network – but due to issues with the disk image duplication the memory cards had to be reprogrammed. This could only be done with the physical cards in hand and since the Raspberry Pis were distributed over a large area and, in many cases, in locations hard or cumbersome to reach – this proved to be a larger job than anticipated. Apart from this, pre-loading the memory cards worked very well.

Server-side installation and configuration was done via `cabal` on the main game server. Haskell code was kept up-to-date via a Git repo and regularly synchronized from an online repository. Installation on server went as expected and was very easy to get set up.

On-site configuration was designed to be done in three steps; physical installation, soundscape configuration and verification. Physical installation was cumbersome but not beyond expectations. Soundscape configuration worked almost as expected. The `Commit` command built into the system proved invaluable since it allowed for one part of team working remotely to write an import-definition of the soundscape then send it to the on-site team, who imported it into the server and then made local backups and exports.

Conclusion

Haskell as a platform, coupled with programming techniques connected to a functional reactive programming style, datatype construction for domain modeling and a package collection that neatly covered all required technological dependencies all combined to a successful project where the sound specification system produced significant value to the game experience and to the immersion into the game world for the participants. Features of Haskell as a platform and of the packages used produced a more capable deliverable than the original specification had expected.

We are not releasing the source code for this system, however, we encourage anyone interested in further details or derivative projects to get in touch with any of the authors.

References

[1] Glen Larson. Battlestar galactica. Television (1978, 1980, 2003, 2004–2009, 2010, 2013).

[2] Raspberry Pi foundation. Raspberry pi (2012). `http://www.raspberrypi.org`. `http://www.raspberrypi.org`.

[3] Thomas Duus Henriksen, Christian Bierlich, Kasper Friis Hansen, and Valdemar Kølle (editors). **Think Larp – Academic Writings from KP2011**. Rollespilsakademiet (2011).

[4] Bjarke Pedersen and Lars Munck. Walking the white road: A trip into the hobo dream. In Markus Montola and Jaakko Stenros (editors), **Playground Worlds. Creating and Evaluating Experiences of Role-Playing Games**, pages 102 – 109. Ropecon ry (2004).

[5] Staffan Jonsson, Markus Montola, Annika Waern, and Martin Ericsson. Prosopopeia: experiences from a pervasive larp. In **Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology**, page 23. ACM (2006).

[6] Markus Montola and Staffan Jonsson. Prosopopeia. playing on the edge of reality. **2006). Role, Play, Art. Collected Experiences of Role-Playing**, pages 85–99 (2006).

[7] Jaakko Stenros and Markus Montola (editors). **Nordic Larp**. Fëa Livia (2010).

[8] Markus Montola. **On the Edge of the Magic Circle. Understanding Role-Playing and Pervasive Games.** Ph.D. thesis, University of Tampere (2012).

[9] Mike Pohjola. The manifesto of the Turku school. `http://mikepohjola.com/turku/manifesto.html`. 3rd Edition, `http://mikepohjola.com/turku/manifesto.html`.

[10] Thomas Henriksen. On the transmutation of educational role-play: A critical reframing to the role-play in order to meet the educational demands. In Markus Montola and Jaakko Stenros (editors), **Beyond Role and Play – Tools Toys and Theory for Harnessing the Imagination**, pages 107 – 130. Ropecon ry (2004).

[11] Jaakko Stenros. In defence of a magic circle: The social and mental boundaries of play. In **Proceedings of the DiGRA Nordic 2012 Conference: Local and Global – Games in Culture and Society** (2012).

[12] Network Time Foundation. Network time protocol. `http://www.ntp.org`. `http://www.ntp.org`.

[13] PulseAudio. Pulseaudio. `http://www.pulseaudio.org`. `http://www.pulseaudio.org`.

[14] Music Player Daemon. Music player daemon. `http://musicpd.org`. `http://musicpd.org`.

[15] Bryan O'Sullivan. `Aeson`. Hackage (2011).

[16] Holger Reinhardt. `AMQP`. Hackage (2012).

[17] Bryan O'Sullivan. `attoparsec`. Hackage (2013).

[18] The Haskell community. The haskell platform (2012).

[19] Don Stewart, Duncan Coutts, David Roundy, Jasper Van der Jeugt, and Simon Meier. `bytestring`. Hackage (2012).

[20] Johan Tibell. `containers`. Hackage (2012).

[21] Falko Peters. `hedis`. Hackage (2011).

[22] Andy Gill and Edward Kmett. `mtl`. Hackage (2012).

[23] Christopher Kuklewicz. `regex-posix`. Hackage (2010).

[24] Bryan O'Sullivan and Tom Harper. `text`. Hackage (2013).

[25] Johan Tibell and Edward Z. Yang. `unordered-containers`. Hackage (2013).

MFlow, a continuation-based web framework without continuations

by Alberto Gomez Corona ⟨agocorona@gmail.com⟩

Most of the problems which complicate the coding and debugging of web applications come from the same properties which make web applications scalable and flexible: HTTP is stateless and HTML is untyped. Statelessness gives freedom to the user, but it implies an inversion of control, where the HTTP requests invoke different request handlers that manage a global state and generate individual web pages.

Dijkstra's famous admonitions about goto statements apply to request handling mechanisms that manage these state transitions. Handlers do not share variable scopes, so, like in the case of the goto, code relies on global variables or on global session variables. There is no top-level structure in the code to make the main sequence evident. For the maintainer, it is very difficult to know what any piece of code is trying to achieve. Even worse, there is no inherent fixed sequence and the execution may be broken by out-of-order requests that do not match with the state of the application, forcing developers to pollute code with checking conditions and error messages.

Continuation-based frameworks solve these problems since they maintain execution state snapshots that match with each possible request. But continuations are memory hungry, and they cannot be easily serialized and shared. This precludes horizontal scalability and failover. As a result, these web frameworks have not been accepted by the mainstream, despite their advantages.

In this article, I will show how an alternative monadic structure for thread state serialization and backtracking can solve these problems. MFlow [1] is a Haskell web framework that automatically handles the back button and other out-of-order requests. Unlike continuation-based frameworks, each GET page is addressable by a REST URL, and the architecture is scalable since the state is composed of serializable events. It is based on enhanced formlets that permit the creation of fully self-contained, composable components called widgets with high level combinators, allowing for the creation of dynamic applications with no explicit use of JavaScript.

Motivation: State management

Many web applications require the same sequence of steps to execute in different contexts. Often, these sequences are merely components of a larger task that the user is trying to accomplish. Such a reusable sequence is called a flow. User registration, log in, checking in for a flight, applying for a loan, shopping cart checkout, or even adding a confirmation step to a form: these are all examples of a flow. In many of these cases, the website needs to store state, and this state sometimes must extend across browser sessions. Storing all the state in the browser can have security issues, and the state is limited to a single web browser. To deal with this problem, web frameworks include server session state that every HTTP event handler can access and modify. The program becomes a state transition machine: such state transitions are hard to code and maintain.

The *Spring Web Flow*, a flow oriented framework on top of the Java *Spring* framework enumerates the following problems when developing web applications [2]:

- Visualizing the flow is very difficult.
- The application has a lot of code accessing the session state.
- Enforcing controlled navigation is important but not possible.
- Proper browser back button support seems unattainable.
- Browser and server get out of sync with "Back" button use.
- Multiple browser tabs cause concurrency issues with HTTP session data.

The ideal solution would be to codify the application flow in the most natural and maintainable way possible. It would be necessary to preserve, in the code, the sequencing whenever the sequence exist, in the same way as a console application. However, each individual page should be addressable by means of an URL.

There is an apparent incompatibility between the state transition machine model, enforced by the web architecture, and the sequential description proposed above. However, the request handlers triggered by each request can be considered as a continuation in the flow, with the state generated by all the previous events. It is possible to invert back the inversion of control [3] so that the program can express the flow directly in the form of a sequence which contains inputs and outputs from/to the Web browser, just like a console application.

This transformation does not change the way the web server and the program interact: the server invokes request handlers, but the handlers are sequenced inside a procedure, so that they share variable scopes. Since the state is in the form of ordinary language variables, the maintainer can observe the sequence of state changes, and a statically typed compiler can verify the consistency of the state transitions. Each request goes to the handler that has the appropriate state to handle the request. This is the basic design of continuation-based frameworks such are ocsigen (OCaml) [4] or Seaside (Smalltalk) [5].

However, serialization and sharing of continuation state across computer nodes

is necessary if we want state persistence and horizontal scalability.

The input-output load involved in state sharing is the reason why REST recommends to use as little state as possible. This is hard to do: even if the language has serializable closures, a continuation includes program code which is linked with many libraries which may render the serialization huge; additionally, it may be linked to non serializable objects of the operating system [6].

The problem of the size and portability of serialized continuations is still unsolved, although there have been some advances on this area in the Scala language [7]. Another option is to store the portable state in the client [8]. This solution constrains the state to a single browser session.

Another way to deal with these problems is to redirect every user to an unique server. In long living flows, especially when it exceeds the mean time between failures, this can end up in loss of state. each page in a flow supposes the serialization of the closure that handles the request of the page, so that the problem of state management multiplies by the number of steps. Maintaining only the most recent states impairs the navigability with the back button. These problems have maintained the continuation-based frameworks out of mainstream use, despite their huge advantages.

MFlow state management

In the Haskell language, it is not possible, to date, to serialize a dynamic closure. Cloud Haskell [**?**] communicate pointers to closures whose addresses are know at compilation time. That method is not feasible for the dynamic closures generated by continuations, that include the state of the computation.

However, it is possible to create an execution log and recover the execution state by running the program taking as inputs the logged events when the procedure is re-started. The log will contain the intermediate results of each step in the computation, though not the computation itself. The machinery for log creation and recovery can be hidden in a monad.

This is the purpose of the Workflow monad, defined in the workflow package [10]. The effect of logging and recovery are managed by the `step` call. The monad is essentially a state transformer which stores intermediate values in persistent storage.

The pseudocode of the lift operation (`step`) that would do logging and recovery would be:

```
import Control.Monad.State
```

```
type Workflow = StateT

step:: m a -> Workflow m a
step mx = do
    st <- get
    if thereRemainResultsInLog st
        then getResultfromLog st
        else do
            r <- mx
            storeInLog r st
            return r
```

Since the state must be written to (read from) permanent storage, results must be serializable, so `step` is not defined for all types, but for types with Serialize instances. The real signature is:

```
step:: (MonadIO m, Serialize a, Typeable a) => m a -> Workflow m a
```

To make the logging and recovery more efficient, the workflow monad caches the read and write operations. `Serialize`, from the RefSerialize [11] package, allows incremental serializations of the modifications of a data structure in the log, rather than logging the entire structure in each step. This is very useful when the flow manages containers or large user-defined structures.

Let's see how to use the workflow monad. A computation in the IO monad, for example:

```
main = do
    n  <-  ask "give me the first number"
    n' <-  ask "give me the second number"
    print $ n + n'

ask s = do
    putStrlLn s
    r <- getLine
    return $ read r
```

Can be lifted to the workflow monad:

```
include Control.Workflow
include Control.Monad.IO.Class(liftIO)

main = exec1 "wfname" $ do
    n  <- step $ ask "give me the first number"
    n' <- step $ ask "give me the second number"
    liftIO . print $ n + n'
```

The lifted computation includes effects of logging and recovery. If we interrupt the program before answering the second question, upon restart the computation will not ask the first question. Instead, it will retrieve the answer from the log and immediately ask the second question. The variables and the execution state then will be the same than when the program was interrupted: the state has been recovered from the log.

If we define `ask` to handle HTTP requests and responses instead of console input/output, a procedure of this kind can express a web flow. In particular, the log can be stored in the browser by means of hidden form variables. If the process is finished after each response, that would make the server applications stateless and, hence, very scalable and fullly REST compliant, since each page keeps its own log state. That is the solution adopted by Peter Thiemann in WASH [12], an excellent Web framework mistreated by its syntax. But client stored state is not appropriate in many cases, for the functional and security reasons above mentioned. Moreover, a re-spawn of the process and a replay of the log are necessary on every request.

MFlow takes a different approach. In this case the server stay running waiting for the next request until a timeout is reached, instead of being stopped after each request. The non-blocking nature of the Haskell threads make the context switch for each request as efficient as event handlers.

After the timeout the process dies. When a new request arrive, the process restart using the log to recover his state. Unfortunately, there is an added problem on this new approach: the server process must synchronize in some way when the process is running and receives out of order requests instead of the next request in the flow sequence, e.g., when the user has pressed the back button.

Synchronization problem

To solve the back button problem, a form of backtracking is necessary, so that when the browser send a request from of a previous page in the navigation, the procedure can go back until a previous page handler in the flow match with the

data sent. From this moment on, the flow proceed normally. This is the purpose of the fail-back monad:

```
data FailBack a = BackPoint a | NoBack a | GoBack

newtype BackT m a = BackT { runBackT :: m (FailBack a ) }


instance Monad m => Monad (BackT  m) where
    fail  _ = BackT $ return GoBack
    return x = BackT . return $ NoBack x
    x >>= f  = BackT $ loop
    where
    loop = do
        v <- runBackT x
        case v of
            NoBack y  -> runBackT (f y)
            BackPoint y  -> do
                z <- runBackT (f y)
                case z of
                  GoBack  -> loop
                  other -> return other
            GoBack -> return  GoBack


instance MonadTrans BackT where
    lift f = BackT $ f >>= \x -> return $ NoBack x

breturn = flowM . BackT . return . BackPoint
```

This monad executes as an Identity monad as long as it is handling `NoBack` results. But if `GoBack` is returned by some statement, then, if the previous one returned a `BackPoint`, it will be re-executed again.

If the previous statement was not a `BackPoint`, then this previous statement will return `GoBack` as well, so the computation will proceed further back until another previous `BackPoint` is found.

Each `BackPoint` in the backtrack is re-evaluated. If it returns `NoBack` or `BackPoint`, the process will resume forward execution. If it returns `GoBack`, it continues backtracking.

This console application is the same above program lifted to the fail-back monad:

```
main = runBackT $ do
    lift (print "will return here at most") >> breturn ()
    n  <- lift  $ ask "give me the first number"
    n'  <- lift  $ ask "give me the second number"
    lift $ print  $  n + n'

    where
    ask s = do
        putStrLn s
        s <- getLine
        if s == "back" then  fail "" else breturn $ read s
```

Here, `ask` is designed to return to the previous line when the string "back" is entered. In the last line, `ask` either fails and initiates a backtracking to the previous ask or returns with `breturn` and becomes a back point that will be called again in case of backtracking.

If `ask` is redefined to accept formlets [13] descriptions instead of text, to send/receive HTTP requests and responses, then the fail-back monad can handle gracefully the back button: it is only necessary to verify the match of the parameters with what the formlet expects. If there is no match, it triggers backtracking until some previous ask match. This solves the problem of the back button.

All that concerning the back button, but that does not solve the general problem of tracking from a branch in the navigation tree to another. I will show below how backtracking together with a careful design of link syntax make it possible to address any page in the flow *forward* by means of a REST URL.

The `FlowM` monad is essentially the same failback transformer stacked above a State transformer that carries out the formlet state. It may be stacked on top of the (`Workflow IO`) monad or the `IO` monad, depending on whether or not persistent state is desired:

```
newtype FlowM v m a = FlowM {runFlowM :: FlowMM v m a}
  deriving (Monad, MonadIO, MonadState (MFlowState v))

type WState view m = StateT (MFlowState view) m
type FlowMM view m = BackT (WState view m)
```

User Interface

The rendering and validation in MFlow are inspired by the formlets [13] concept. The View data carries out a rendering format `v` for an underlying monad `m`, which is `IO` always, except in the case of cached widgets.

As in the case of the `FlowM` monad, the View data carries the formlet state information.

The formlet information is stored in a `FormElm` data structure, which has the generated rendering as well as the result of the match of the request parameters with the formlet in a Maybe value:

```
newtype View v m a = View { runView :: WState v m (FormElm v a) }

data FormElm v a = FormElm [v] (Maybe a)
```

The `v` in these definitions is the particular rendering. A widget must have a rendering format that must be an instance of the `FormInput` class.

```
class (Monoid view,Typeable view)  => FormInput view where

    toByteString ::  view -> B.ByteString
    toHttpData ::  view -> HttpData
    fromStr ::  String -> view
    fromStrNoEncode ::  String -> view
    ftag ::  String -> view  -> view
    inred  ::  view -> view
    flink ::  String -> view -> view
    flink1::  String -> view
    flink1 verb = flink verb (fromStr  verb)
    finput ::  Name -> Type -> Value -> Checked -> OnClick -> view
    ftextarea ::  String -> T.Text -> view
    fselect ::  String -> view -> view
    foption ::  String -> view -> Bool -> view
    foption1 ::  String -> Bool -> view
    foption1  val msel = foption val (fromStr val) msel
    formAction  ::  String -> view -> view
    attrs ::  view -> Attribs -> view
```

This class describes how to create form elements, links and error messages in that particular format. Using the class methods, the widgets are not tied to a particular rendering. In most cases, blaze-html rendering is used, but there are also bindings for xhtml and HSP. It is possible to create widgets that can use any rendering, thanks to the `FormInput` abstraction. The widgets in `MFlow.Forms.Widgets` module are defined in this way.

The pseudocode of an `Int` input box would be:

```
getInt :: (FormInput view, MonadIO m) => Maybe Int -> View view m Int
getInt = View $ do
    parm <-  genNewId
    mr <- lookupParam parm
    return \$ FormElm [finput "text" parm "" False Nothing ] $  mr
```

`ask` gets a widget, displays it, gets the response from the user and returns it under the FlowM monad:

```
ask ::  (FormInput view) => View view IO a -> FlowM view IO a

page = ask
```

`page` is a synonym of `ask`

The formlets can be combined with applicative operators to create more complex widgets. A page is a widget. For this purpose, `View` has an applicative instance:

```
instance (Functor m, Monad m) => Applicative (View view m) where
  pure a  = View $  return (FormElm [] $ Just a)
  View f <*> View g = View $
                  f >>= \(FormElm form1 k) ->
                  g >>= \(FormElm form2 x) ->
                  return $ FormElm (form1 ++ form2) (k <*> x)
```

This is a complete program that uses the FlowM monad and widgets combined with applicative operators:

```
{-# LANGUAGE OverloadedStrings #-}
import MFlow.Wai.Blaze.Html.All
import Control.Applicative
```

```
main = runNavigation "sum" . step $ do
      setHeader html . body
      (n1,n2) <- page $ (,) <$> getInt Nothing <++ br
                            <*> getInt Nothing <++ br
                            <** submitButton "send"

      page $ p << (n1+n2) ++> wlink () "click to repeat"
```

`runNavigation` executes the FlowM computation in a endless loop.

```
runNavigation ::  String -> FlowM Html (Workflow IO) () -> IO ()
```

It expects persistent navigation, but the example is transient, so we apply a single `step` to the transient computation. The first `page` rendering is an applicative combination of `getInt` formlets to create a 2-tuple.

There are line breaks (`br`) added using blaze-html formatting. They are added with the operator:

```
(++>)    :: v -> View v m a -> View v m a
```

... that add formatting to a widget.

This operator:

```
(<<) :: (v -> v) -> v -> v
```

... encloses formatting within a tag (in this case the blaze-html tag "b"). Since ■ has less precedence that `++>`, the composition does not need parentheses.

The second `page` presents an HTML paragraph with the sum of the numbers. The `wlink` widget renders an HTML link, with a URL that invokes the flow again; the page will return `wlink` value.

```
wlink ::  (Typeable a, Show a, MonadIO m,  FormInput view)
        => a -> view -> View  view m a
```

Because `wlink` is in the last FlowM statement, `runNavigation` will restart the computation again.

Routing

Note that `wlink` can also consume inmediately an element of a RESTful path, if it is available, without displaying anything. When a widget returns a match, `page` does not send a page response. Instead, it returns the result to the flow immediately. This is key for addressing any page in the flow in a REST URL:

```
data Option = Option1 | Option2

main = runNavigation "verb" .  step  $  do
    r <- page   $ h3 << "Alternative operator  used here"
              ++> wlink Option1 << b << "Choose Option1" <++ br
              <|> wlink Option2 << b << "Choose Option2"

    case r of
      Option1 -> proc1
      Option2 -> proc2
```

In this example, URLs which start with `http://host/verb/option1/...` will go straight to `proc1`: the menu will not be presented. As you would expect, URLs which start with `http://host/verb/option2/..` will go straight to proc2. The URL `http://host/verb` will go to the initial menu.

Thus, it is possible to express REST routes as flows using `wlink`. The code above has three implicit routes.

This is combined with the backtracking mechanism to make any GET page in the flow addressable. For example, if in the above example the flow is running the `proc1` branch, when the user enters the URL of the second option, the backtracking mechanism will backtrack to `/verb`, after which the second wlink will match the `/option2` segment of the URL, so `proc2` will be executed.

Single page applications

As seen above, the applicative instance permits the combination of widgets using applicative operators. But `View` also has an Monad instance:

```
instance (Monad m) => Monad (View view m) where
    View x >>= f = View $ do
                   FormElm form1 mk <-  x
                   case mk of
                     Just k  -> do
                         FormElm form2 mk <-  runView $ f k
```

```
                        return $ FormElm (form1 ++ form2) mk
                    Nothing ->
                        return $ FormElm form1 Nothing

    return = View .  return . FormElm  [] . Just
```

This monad instance is somewhat remarkable. In this monadic instance, the second widget consumes the output of the previous statement, provided that the latter validates. But if the previous widget return `Nothing`, then the computation is interrupted and the rendering becomes what was produced until that moment.

`page` in this case will present the rendering again and again until the monadic computation finish with a valid output that will be returned to the flow. Since each iteration read the input of the previous one and each widget can modify its presentation depending on the previous widget output, sophisticated input forms and presentations are possible:

```
data Client = Person{pname, paddress  :: String, age :: Int}
            | Company{cname,caddress :: String, numWorkers :: Int}

main = runNavigation "input" . step $ do
    page $ do
        type <- getRadio[\n -> n ++> setRadioActive v n
                        | v <-["person","company"]
        case type of
          "person" -> Person
                        <$> "name: "  ++> getString Nothing
                        <*> "address " ++> getString Nothing
                        <*> "age "    ++> getInt Nothing
          "company" -> Company
                        <$>"name: "    ++> getString Nothing
                        <*> "address " ++> getString Nothing
                        <*> "workers " ++> getInt Nothing
    ...
```

Here, two options in a radio button are presented. Because it is a monad, the second statement can use the `type` to present the appropriate applicative form.

In the first iteration, only the radio buttons are presented. When the user chooses one of the options, the appropriate formulary appears below the radio buttons. When the user completes the formulary, `page` will return the `Client` data to the flow. If the user changes the option in the radio button, the form will change accordingly.

To avoid sending the whole page at each iteration, there are some page modifiers that refresh only the elements that change using AJAX. Push is also possible using long polling. The execution of the server code is the same but only the rendering of the widget activated by the request is refreshed in the page.

Note that the single page example above can be transformed into a two pages example by translating the View monad to the FlowM monad with only a few changes.

If in the bind operation of the View monad we discard the rendering of the first formlet, the effect is that the rendering of the second formlet is substituted for the first when the former is validated. That is how MFlow implements the callback mechanism of the Seaside framework [**?**]

Using these and other combinators it is possible to create different kind of applications using basically the same above elements in a pure Haskell EDSL [15].

Web services

A widget in MFlow is essentially a request parser and a writer of responses. A page is the composition of this kind of elements.

That parser-writer combination of the View monad can be used as a general request-response mechanism. Adding a simple set of combinators it is possible to create web services in an idiomatic way without the need of special constructions.

The three services implement a sum and a product of two integers.

First, a REST web service, where the parameters are in a REST path.

```
main = runNavigation "apirest" . step $ ask $ do
         op <- getRestParam
         term1 <- getRestParam
         term2 <- getRestParam
         case (op, term1,term2) of
           (Just  "sum", Just x, Just y) ->  wrender (x + y :: Int) **> stop
           (Just "prod", Just x, Just y) ->  wrender (x * y) **> stop
           _  ->do -- blaze Html
                     h1 << "ERROR. API usage:"
                     h3 << "http://server/api/sum/[Int]/[Int]"
                     h3 << "http://server/api/prod/[Int]/[Int]"
                ++> stop

stop = empty
```

`getRestParam` read the next REST segment in the path if there is any.

`stop` is a synomym of of the `Applicative` class `empty` method. By definition, the computation will stop, `ask` will fail and the Flow will not navigate forward. The content of the writer (the one generated by `wrender`) is sent as response. If the parameters are not the expected ones, the a HTML message is sent.

Some example of invocations:

```
> curl "http://mflowdemo.herokuapp.com/apirest/prod/4/3"
12

> curl "http://mflowdemo.herokuapp.com/apirest/sum/5/7"
12
```

The second example read key-value parameters for the numbers instead of REST parameters:

```
main = runNavigation "apikw" . step . ask $ do
         op <- getRestParam
         term1 <- getKeyValueParam "t1"
         term2 <- getKeyValueParam "t2"
         case (op, term1, term2) of
           (Just  "sum", Just x, Just y) -> wrender (x + y :: Int) **> stop
           (Just "prod", Just x, Just y) -> wrender (x * y) **> stop
           _ -> do -- blaze-html
                     h1 << "ERROR. API usage:"
                     h3 << "http://server/api/sum?t1=[Int]&t2=[Int]"
                     h3 << "http://server/api/prod?t1=[Int]&t2=[Int]"
                ++> stop
```

Some example invocation:

```
> curl "http://mflowdemo.herokuapp.com/apikv/prod?t1=4&t2=3"
12

> curl "http://mflowdemo.herokuapp.com/apikv/sum?t1=4&t2=3"
7
```

The third service uses key-value parameters as well, but there are defined parsec-like combinators that are Widgets as well:

```
main  = runNavigation "apiparser" . step . asks $
            do rest "sum" ; disp $ (+) <$> wint "t1" <*> wint "t2"
```

```
        <|> do rest "prod"; disp $ (*) <$> wint "t1" <*> wint "t2"
        <?> do  -- blaze Html
               h1 << "ERROR. API usage:"
               h3 << "http://server/api/sum?t1=[Int]&t2=[Int]"
               h3 << "http://server/api/prod?t1=[Int]&t2=[Int]"
    where
    asks w = ask $ w >> stop
```

Some example invocation:

```
> curl "http://mflowdemo.herokuapp.com/apiparser/prod?t1=4&t2=3"
12

> curl "http://mflowdemo.herokuapp.com/apiparser/sum?t1=4&t2=3"
7
```

The combinators `rest` and `wint` are defined below, from `getRestParam` and `getKeyValueParam`. They are part of the `MFlow.Forms.WebApi` module

`rest` verify that the next rest parameter is an expected value. `wint` read a parameter of type Int which has a given key.

The operator `<?>` present a blaze-html message when the parser does not match, like in a parser combinator.

`disp` writes the result of a computation.

It uses the same applicative, alternative and monadic combinators defined for any other widget in MFlow.

```
stop = empty

restp = View $ do
   mr <- getRestParam
   return $ FormElm [] mr

rest v = do
   r <- restp
   if r == v
     then return v
     -- restore the rest index and fail
     else modify (\s -> s{mfPIndex = mfPIndex s-1}) >> stop

wparam par = View $ do
   mr <- getKeyValueParam par
```

```
   return $ FormElm [] mr

disp :: Show a => View Html IO a -> View Html IO ()
disp w = View $ do
   elm@(FormElm f mx) <- runView w
   case mx of
     Nothing -> return $ FormElm f Nothing
     justx@(Just x) -> return $ FormElm (f++[fromStr $ show x])
                               $ return ()

infixl 3 <?>
(<?>) w v = View $ do
  r@(FormElm f mx) <- runView w
  case mx of
    Nothing -> runView $ v ++> stop
    Just _ -> return r

wint p = wparam p :: View Html IO Int
```

Scalability

Because the logs grow by small update events, it is easy for two or more servers to synchronize state by interchanging `step` events instead of entire states. The small size of the step events makes MFlow state easier to synchronize and architecture independent, without the problems of continuation-based frameworks.

Execution traces

Trace logging by means of hand-made statements, although tedious and cumbersome, is a traditional way of tracking errors in web applications in production environments.

MFlow produces automatic execution traces under the `FlowM v IO` monad. These logs are produced at failure time. This log is created by the same backtracking mechanism of the fail-back monad.

Other exception-treatment monads like Error, Maybe or the Exception monad fail back to the calling method. The failback monad instead returns to the previous statement, so a trace rather than a call stack will be produced. This is a great improvement, especially for web applications where it is necessary to extract the

maximum amount of information from each failure in the exploitation environment.

The trace mechanism uses the monadloc[16] package. `withLoc` is a method of the MonadLoc class, so it can be redefined for the particular needs of each monad. In this case, the instance inserts an exception handler that adds the line of error to the trace and initiates a backtracking. The backtracking proceeds back to the beginning, following the execution history in reverse order. This is the pseudocode of the instance:

```
instance  MonadLoc (FlowM v IO)  where
    withLoc loc f = do
          r <- compute f `catch` (\e ->do
                    insert (show e) in the list
                    return GoBack)  -- backtrack
          if  trace going
              prepend location (loc) info to the list
              return GoBack  -- continue backtracking
            else
              return r  -- normal return
```

The monadloc-pp preprocessor inserts a `withLoc` call behind each line of code with information about the module and line number within the first parameter.

When the backtracking is complete, the scheduler detects the trace at the root of the execution and prints it in the console.

To see an example of trace log, see [17]. In the case of persistent flows, the FlowM v (Workflow IO) monad generates its own log at execution time, which is readable and can be inspected in case of error.

Conclusions

MFlow demonstrates the power of monadic computations for creating web flows and web navigation in a concise, intuitive and maintainable way, while reducing drastically the plumbing and the error ratio in web programming. The navigation is verifiable at compile time, and this facilitates the testing and maintenance, while the scalability and navigability is not compromised.

Future work

One important task in the future is to develop a synchronization mechanism for MFlow servers to realize the theoretical scalability and failover of the architecture,

using Cloud Haskel [18].

The routing example shows how the FlowM machinery can work as an event scheduler without inversion of control. As such, it can be used in very different scenarios.

The supervisor package [19] contains a enhanced version of the fail-back monad applicable to any context. It and can perform specific actions when the computation goes forward and backward thanks to a programmer-defined instance. It will be used by MFlow in the future to substitute the less general fail-back monad.

Event scheduling without inversion of control is ideal for process interaction in cloud environments [20]. Optional backtracking and persistence can make it more expressive and convenient.

Persistent flows can be used in enterprise integration scenarios [21] with long-running transactions where backtracking can be used to perform rollbacks.

References

[1] `http://www.haskell.org/package/MFlow`.

[2] Spring web flow. `http://projects.spring.io/spring-webflow`.

[3] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. **SIGPLAN Not**, 38:page 2003 (2001).

[4] `http://ocsigen.org`.

[5] `http://en.wikipedia.org/wiki/Seaside_(software)`.

[6] Continuation Fest 2008 Tokyo, Japan April 13, 2008. **Clicking on Delimited Continuations** (2008). `http://okmij.org/ftp/Computation/Fest2008-talk-notes.pdf`.

[7] Ian Clarke. Swarm, a new approach to distributed computation. video. `http://tech.slashdot.org/story/09/10/11/1738234/Swarm-mdash-a-New-Approach-To-Distributed-Computation?from=rss`.

[8] Jay A. McCarthy. Automatically restful web applications: marking modular serializable continuations. **SIGPLAN Not.**, 44(9):pages 299–310 (August 2009). `http://doi.acm.org/10.1145/1631687.1596594`.

[9] The distributed-static package. `http://hackage.haskell.org/package/distributed-static/docs/Control-Distributed-Static.html`.

[10] `http://www.haskell.org/package/Workflow`.

[11] The refserialize package. `http://www.haskell.org/package/RefSerialize`.

[12] http://www.informatik.uni-freiburg.de/~thiemann/WASH.

[13] http://www.haskell.org/package/formlets.

[14] L Renggli S Ducasse, A Lienhard. Seaside: A flexible environment for building dynamic web applications. Technical report. http://citeseer.uark.edu:8080/citeseerx/showciting;jsessionid=00305F8E94C1AA992461768584AA0B7E?cid=9279754.

[15] Mflow as a dsl for web applications. https://www.fpcomplete.com/user/agocorona/MFlowDSL.

[16] http://hackage.haskell.org/package/monadloc.

[17] Demo of error traces in mflow. http://mflowdemo.herokuapp.com/noscript/errortraces/trace.

[18] Cloud haskell. http://www.haskell.org/haskellwiki/Cloud_Haskell.

[19] The supervisor package. https://hackage.haskell.org/package/supervisor.

[20] Martin Odersky Philipp Haller. Event-based programming without inversion of control. http://lampwww.epfl.ch/~odersky/papers/jmlc06.pdf.

[21] Alberto G. Corona. How haskell can solve the integration problem. https://www.fpcomplete.com/business/blog/guest-post-solve-integration-problem.

Practical Type System Benefits

by Neil Brown ⟨neil@twistedsquare.com⟩

One of Haskell's key selling points is its type system. This article provides several practical examples of how the type system helped while writing some data analysis software. The article will touch on monads, concurrency, invariants, generic programming, and quasi-quoters.

The Project

As part of a research project, we are collecting a large volume of Java source code which we want to analyse[1]. At the time of writing, we have 50 gigabytes of source code snapshots from six months' worth of recording. To perform the analysis, we have a machine with two six-core hyperthreaded Xeon processors and 32 gigabytes of RAM. I might term this "medium data": small enough to fit on one machine, but big enough that you can make multiple cups of tea while the analysis is running.

I felt that for writing this analysis, Haskell was a good choice. This was mainly because of its easy support for generic programming and parallel processing, as will be explained in the course of this article. I thought my experiences may be of interest to others, highlighting practical uses of Haskell's powerful type system in my project. The article assumes general familiarity with Haskell, and is aimed at someone who has learned a little but wants to see what the advantages are of learning more.

[1]For more details, see "Blackbox: A Large Scale Repository of Novice Programmers' Activity" by Brown et al. (2014), available via `http://twistedsquare.com/publications.html`

Generic Programming

One of the main reasons that I felt Haskell would give me an advantage for performing source code analysis was its easy support for generic programming[2]. For example, one small analysis I wanted to perform was to look for **if** statements in Java like this:

```
if (x < 5);
{
    x = 5;
}
```

Note the crucial semi-colon after the if condition; a semi-colon is a valid statement in Java, so the semi-colon becomes the if body. Thus the if statement is pointless, and the body is executed regardless of whether x is less than five. Effectively, the compiler sees this version:

```
if (x < 5)
    ; // if has empty body

// This is now a totally separate block of code that is executed next:
{
    x = 5;
}
```

This is not a compile-time or run-time error, but without an else clause, I think it can always be termed a programmer mistake. The first thing to do in looking for this mistake was to parse the source into an Abstract Syntax Tree (AST) – for this, I used (a slightly augmented version of) the `language-java` library from Hackage. The relevant portion of the AST data type is this[3]:

```
data Stmt = Empty | IfThen Exp Stmt | IfThenElse Exp Stmt Stmt | ...
```

Exp is the data type for expressions, while Stmt is the data type for statements. You can see that Stmt has a constructor for the empty statement (i.e., the solitary semi-colon). So my empty statement will be a value that looks like IfThen something Empty. That's easy to pattern-match for, given a specific Stmt. Here's the code:

```
emptyIf :: Stmt -> Bool
```

[2]Generic programming refers to programming that works with multiple types. Although they share the same name and concept, Java's generics are a fairly restrictive form of generic programming. An example in Haskell might be a generic fold, which can fold the contents of a heterogenous container together.

[3]To be clear: the triple dots are omitted code, not Haskell syntax!

```
emptyIf (IfThen _ Empty) = True
emptyIf _ = False
```

However, a full Java AST has the type CompilationUnit and can have Stmt values in all sorts of places. For example, Java's support for anonymous inner classes means that statements can be inside method declarations that are inside a class definition in an expression which itself could be in another inner class. This is where generic programming enters the picture.

Generic programming can be used to automatically find all values of one type (e.g. Stmt) inside a complex structure containing many different types (e.g. CompilationUnit). The `listifyDepth :: (b -> Bool) -> a -> [b]` function finds a list of all values of type b (Stmt) within the given value of type a (CompilationUnit) that satisfy the given function. This particular function is implemented in the Alloy generic programming library, but the exact same function exists in Scrap Your Boilerplate (named `listify`), and similar functions are available in other generic programming libraries.

With this function, augmenting our program to look across the whole CompilationUnit structure is very easy:

```
hasEmptyIf :: CompilationUnit -> Bool
hasEmptyIf = not . null . listifyDepth emptyIf

-- OR:

findEmptyIfs :: CompilationUnit -> [Stmt]
findEmptyIfs = listifyDepth emptyIf
```

Generic programming is particularly easy in the Haskell language, and in general, dealing with tree structures like ASTs is much easier in Haskell than in other languages like Java.

Concurrency and Parallelism

Being a pure functional language, Haskell has great support for functional parallelism: evaluating several pure functions in parallel. However, it also has good support for concurrency: running several imperative threads of control alongside each other. In our application, the analysis is almost always in a map-reduce style:

1. Read some initial state from the database to determine our list of tasks.
2. Run a large batch of tasks that each fetch data from the database (or some files with cached intermediate state) and then process the data.
3. Collate the results together.

The middle step is amenable to being run across many processor cores, but since it requires imperative access to the database or file system, we must use imperative forms of concurrency (i.e. forkIO). The classic problem with parallel programming is synchronising and coordinating access to mutable variables. In the default pure functional style in Haskell, this is not an issue. However, there are still problem with synchronising concurrent access to *external resources*, such as the database connection or files. It is easy, at first, to accidentally write code like this:

```
do conn <- newConnection
   mapM_ forkIO [doTask i conn | i <- [0..9]]
```

This will unsafely use a single connection in parallel from many tasks, but will produce no compile-time error. In a language like C, where a thread is run from a different top-level function, the values shared between threads are more explicit, and so resource- sharing mistakes can sometimes be easier to pick up. However, Haskell makes it so easy to write inline parallelism that it can actually be less obvious in Haskell that values like our connection handle are being improperly shared between threads.

I found that the best way to avoid this problem was to build the connection into a monad, using a reader monad (called DB). (A reader monad encapsulates some read-only state, which you can read or use via a monadic action.) This both avoided passing the connection parameter around (it could just be accessed from the monad), and meant that I could build a parallel function that would automatically open a new database connection for each new thread. The parallel function had a simple API:

```
parallel :: [DB a] -> DB [a]
```

This API neatly hid the complexity of connection management. The function started up a bunch of new worker threads (one per core) and opened a new connection for each thread. Then each thread picked worker tasks from a list and executed them, before returning the results. With lists of tasks being 10,000s, opening a database connection per worker thread (of which there were 24) was much lower overhead than opening one per task. I have since generalised the functionality and released it as part of the `parallel-tasks` library on Hackage: `http://hackage.haskell.org/package/parallel-tasks`.

Models

The database that our application is reading from was originally created by a Ruby on Rails application. Rails uses a relational database as a persistence layer to store objects that it calls “models”. Each model has a column for each of its fields. The primary key, “id”, is an auto-incrementing integer, and thus foreign

keys are also integers. Our user and session models might be stored in a database table something like this:

User	
Field Name	Field Type
id	64-bit integer
uuid	string
created_at	datetime

Session	
Field Name	Field Type
id	64-bit integer
user_id	64-bit integer
created_at	datetime

A naive approach to dealing with this data from Haskell is to declare an ID type alias:

```
type Id = Int64
```

You might get these from a query from the `mysql-simple` library:

```
do (sessionId :: Id, userId :: Id) <- query conn "select s.id, u.id
    from sessions as s, users as u inner join on s.user_id = u.id"
```

You might then have utility functions like these:

```
getNewUsersSince :: Date -> DB [Id]

getUserInfo :: Id -> DB User

getSessionListForUser :: Id -> DB [Id]

getSessionInfo :: Id -> DB Session
```

With all these `Id` fields flying around, it is only a matter of time before you do something like this:

```
do us <- getNewUsersSince d
   mapM (\u -> do ss <- getSessionListForUser
                  mapM getSessionInfo us {- whoops! -}) us
```

Instead of passing a list of session IDs to `mapM getSessionInfo`, we passed user IDs. But as they all have the same type, so this type checks. To make matters worse, the database is fine with this too: it won't even produce an error, just wrong results. This small example is slightly contrived; but these things do happen, especially if the function took a list of user IDs and a list of session IDs – it's easy to get the parameter order wrong.

The solution is to give each ID its own type, to prevent users from mixing them up. A first attempt might look like this:

```
newtype UserId = UserId Int64
newtype SessionId = SessionId  Int64

data User = User { userId  :: UserId,
                   userUuid :: String,
                   userCreatedAt :: Datetime }
data Session = Session { sessionId  :: SessionId,
                         sessionCreatedAt  :: Datetime }

getNewUsersSince :: Date -> DB [UserId]

getUserInfo  :: UserId  -> DB User

getSessionListForUser  :: UserId  -> DB [SessionId]

getSessionInfo  :: SessionId  -> DB Session
```

The behaviour of these functions is now more apparent from their type, and there is greater type-safety, with less possibility to mix up a UserId and SessionId. We now turn to look at how these types are used in the context of database queries. One way to use these types is to apply them directly after the database query:

```
do (rawSessionId, rawUserId) <- query conn "select s.id, u.id
     from sessions as s, users as u inner join on s.user_id = u.id"
   let sessionId = SessionId rawSessionId
       userId = UserId rawUserId

-- OR:

do (sessionId, userId) <- (SessionId *** UserId) <$> query conn "select
     s.id, u.id from sessions as s, users as u inner join on s.user_id = u.id"
```

A better solution is available if you know what the query function does. It has a selection of type-classes like Result which support conversion from the database wire protocol into recognisable Haskell types (Int32, String, etc). We don't actually need to know what exactly they do or how they work – using the GHC GeneralizedNewtypeDeriving language feature we can just tell the compiler to port them from Int64 to our newtype wrapper of Int64:

```
newtype UserId = UserId Int64 deriving (..., Result, ...)
newtype SessionId = SessionId Int64 deriving (..., Result, ...)
```

This allows us to extract these types directly from the database query without wrapping them ourselves:

```
do (sessionId :: SessionId, userId :: UserId) <- query conn "select
    s.id, u.id from sessions as s, users as u inner join on s.user_id = u.id"
```

(You can omit those type declarations if type inference can infer them from future statements.)

Increasing Type Safety

We have already prevented some mistakes by preventing SessionIds from being mixed up with UserIds. However, there is still potential for mistakes in our program around the MySQL queries. A MySQL query is one of those instances where you are dealing with external data, and just as when you read from a file or a socket, there is the possibility to make a mismatch with the schema or protocol. The error cannot be picked up by the compiler (unless you go to greater lengths by converting the schema/protocol into a type or structure that the compiler understands). For example, here's a possible error:

```
do (sessionId :: SessionId, userId :: UserId) <- query conn "select
    u.id, s.id from sessions as s, users as u inner join on s.user_id = u.id"
```

Do you notice the problem? The query is asking for the user ID and then a session ID (in that specific order), but the code labels the results as being a session ID and then a user ID. We are back to the previous problem: a mixup in parameters that cannot be detected by the compiler. In general, I find that several problems in Haskell can be solved by involving the type system further[4]. What I really want to do here is add a type annotation *into the MySQL queries themselves*. We can do this using quasi-quoters – I will briefly explain what quasi-quoters are, and then go on to show how I use them with MySQL queries.

Interlude: quasi-quoters

A quasi-quoter is a pre-compile step that is a bit like a pre-processor; it takes a String input and, *at compile-time*, parses it and spits out source code which is then fed into the full compilation. To illustrate this, consider the case when you want to print out some text, some of which comes from a string, some from an integer, and some from the program itself. In C, I would probably use printf for this purpose:

```
char name[256];
```

[4]The trick is to involve the type system in ways that provide a big benefit for the cost, but not going so far that the type system gets in the way of code that you know – but can't easily prove to the compiler – is safe

```
int yearOfBirth;
// ... ask user for name and year of birth
printf("Hi s, you will reach d years old this year",
          name, currentYear - yearOfBirth);
```

In Haskell, you might write it like this:

```
do name <- ...
   yearOfBirth <- ...
   putStrLn ("Hi " ++ name ++ ", you will reach " ++
     show (currentYear - yearOfBirth) ++ " years old this year")
```

Haskell code that is uglier than its C equivalent? Dang. A simple quasi-quoter allows you to write a string with embedded code (a la Perl, Ruby and many others). The substitution is done at compile-time (not at run-time using eval or similar), so errors in the code in the string cause compile-time errors. Using the `here` package from Hackage, you can write:

```
do name <- ...
   yearOfBirth <- ...
   putStrLn [i|Hi ${name}, you will reach ${currentYear - yearOfBirth}
     years old this year|]
```

The quasiquoting syntax consists of a square bracket, the name of the quasi-quoter function, a pipe, and then the string to be fed to the quasi-quoting function at compile-time, which is terminated by a pipe and square bracket. I'm surprised there aren't more quasi-quoters around (e.g., one for processing regexes at compile-time).

Back to our code

Let's now return to our problem of adding type annotations to MySQL queries. My quasi-quoter, qquery, takes a query String like this, with type annotations inside it:

```
[qquery| select s.id{SessionId}, u.id{UserId} from sessions as s, users as u
   inner join on s.user_id = u.id|]
```

It then turns it into the same String with the annotations removed and placed into the type of the query, i.e.:

```
(mkQuery "select s.id, u.id from sessions as s, users as u inner join
  on s.user_id = u.id" :: Query (SessionId, UserId))
```

Thus, if we now re-write our existing code:

```
do (userId, sessionId) <- query conn [qquery| select s.id{SessionId},
    u.id{UserId} from sessions as s, users as u
    inner join on s.user_id = u.id |]
```

The types are now annotated *exactly where they are used* inside the query string. The above code will now give a compile-time error because of the mismatch in the ordering between user ID and session ID. I have packaged this type-annotation quasi-quoter into a library on Hackage (`http://hackage.haskell.org/package/mysql-simple-quasi`), so you can use it if you wish.

As a general observation: these benefits do not come completely for free (the quasiquoter had to be implemented), but once you have the extra capability to add more type information, I believe it pays off time and time again in program clarity and the ability to avoid mistakes as early as possible (compile-time). Since I have a growing ecosystem of a core central library and lots of quickly hacked together analysis programs to explore data, the benefits of extra safety in the core libraries keep paying off as I use them in my small other programs.

Types as Light-Weight Contracts

After all that database talk, I will end with a simpler example of using types to your advantage. Often, you will know a property or pre-requisite of a value, e.g. that a particular integer is (or should be) non-negative. In my program I pull out sorted lists from the database (or sort them myself). Later, I want to merge these lists into a single sorted list. If I am playing it safe, I have to write the merge functions like so:

```
mergeLists :: [a] -> [a] -> [a]
mergeLists xs ys = mergeSorted (sort xs) (sort ys)
```

If the list is already sorted, as is usually the case in my program (because I use "ORDER BY" in the database query), I am unnecessarily attempting to sort the lists again. (This is not just slow because it makes an extra pass of the list, but it also forces evaluation of the list, which may interfere with laziness.) Instead, I can annotate that the lists are sorted by making a new type for this:

```
newtype SortedList a = Sorted [a] deriving (Eq, Ord, ...)
```

The type is exported opaquely: outside the module, the only way to create a sorted list is via the exported `sortList` function, or a few other generators:

```
sortList :: [a] -> SortedList a
sortList = Sorted . sort

enumFromToSorted :: a -> a -> SortedList a
```

```
enumFromToSorted x y = Sorted (enumFromTo x y)
```

I can then write an $O(n)$ function to merge two sorted lists and be sure that I will never accidentally call it on an unsorted list:

```
mergeSorted :: SortedList a -> SortedList a -> SortedList a
```

This technique can also be used in other languages (e.g., defining a new collection class for sorted lists). But Haskell's newtype wrappers make it very easy to add a new type, with all the features of the old type (by deriving type-classes automatically), and some more features added or removed[5]

Conclusion

Often, when people talk about the advantages of a particular programming language, they talk in generalities. Pure functions make your code better; garbage collection lets you write less code; monads are really useful. If you know nothing or little of the language, these general benefits are often hard to understand. I hope that in this article I have given some small, understandable, concrete examples of how some of Haskell's type features can be useful in a real programming project – without turning into one of those infamously baffling monad tutorials.

[5] You can see more discussion, and a more complicated example of this for array index checking, at Oleg's site: `http://okmij.org/ftp/Haskell/types.html#branding`