Free Theorems Simply, via Dinaturality

Janis Voigtländer

University of Duisburg-Essen

DECLARE 2019

Plan for the talk

- Motivation, free theorems in teaching
- Deriving free theorems via relational parametricity
- The “conjuring lemma” as a shortcut
- Turning the idea into a practical generator
- Connection to the category theory concept of dinaturality (skipped)

Motivation from teaching

One kind of richer expressions: list comprehensions

Using a list comprehension:

```
main :: IO ()
main = drawingOf (pictures [ scene d | d <- [0..5] ])

scene :: Double -> Picture
scene d = translated d 0 (colored red triangle)
```

- With `pictures :: [ Picture ] -> Picture`.
- And a list comprehension `[ scene d | d <- [0..5] ]`.
- This is <u>not</u> exactly like a `for`-loop, for several reasons.
- Instead, it is like a mathematical set comprehension $\{ 2 \cdot n \mid n \in \mathbb{N} \}$.

Wholemeal programming on lists

We earlier had this example:

```
main :: IO ()
main = drawingOf (pictures [ scene d | d <- [0..5] ])

scene :: Double -> Picture
scene d = translated d 0 (colored red triangle)
```

- Of course, the individual evaluations will, on a sequential machine, happen in some order. And the resulting list is really a sequence, not a set. But the individual values will be independent of all that.
- Indeed, one can show that for any g and n, in Haskell:

```
          [ g x | x <- [0..n] ]
       ≡  reverse [ g x | x <- reverse [0..n] ]
```

Contrast to for-loops in Java, C, etc.

- In contrast, it is not remotely true that in an imperative language we can always replace a piece of code written like this:

```
          for (x = 0; x <= n; x++)
             result[x] = g(x);
```

  by this:

```
          for (x = n; x >= 0; x--)
             result[x] = g(x);
```

- And even for the cases where commands as above <u>are</u> equivalent, a formulation given that way is less useful than the Haskell equation we saw, or indeed its more general version:

```
          reverse [ g x | x <- xs ]
       ≡  [ g x | x <- reverse xs ]
```

Consequences of polymorphic types

- **Polymorphism has really interesting semantic consequences.**
- **For example, in the lecture last week, I mentioned that always:**

```
        reverse [ g x | x <- xs ]
      ≡ [ g x | x <- reverse xs ]
```

- **What if I told you that this holds, for arbitrary `g` and `xs`, not only for `reverse`, but for any function with type `[a] -> [a]`, no matter how it is defined?**
- **Can you give some such functions (and check the above claim)?**

Free theorems

Statements about polymorphic functions based solely on their types (Wadler 1989).

For example, that for every $f :: [a] \to [a]$ and every g and xs,

$$f\ [g\ x \mid x \leftarrow xs] = [g\ x \mid x \leftarrow f\ xs]$$

Interesting from the perspective of teaching:

- providing insight into the declarative nature of types and semantics of programs
- source of examples (finding functions for given polymorphic types, checking their properties)
- evidence for generality compared to formulations of "similar" properties (e.g., loop fusion) in C, Java, ...

(And there are various applications outside of teaching as well.)

Polymorphic functions on lists

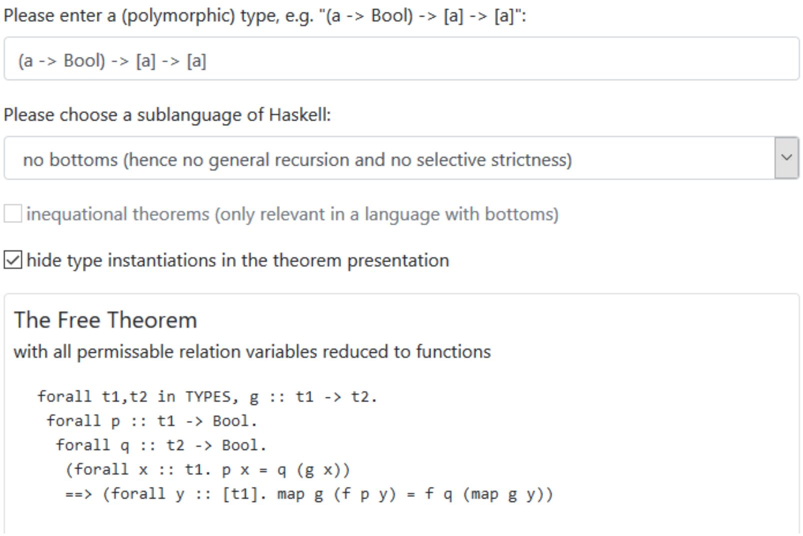
- We have already seen a lot of functions that fit this pattern:

```
head   :: [a] -> a
tail   :: [a] -> [a]
last   :: [a] -> a
init   :: [a] -> [a]
length :: [a] -> Int
null   :: [a] -> Bool
concat :: [[a]] -> [a]
```

- In concrete applications, the type variable gets instantiated appropriately: `head "abc" :: Char`.

A practical generator

At `http://free-theorems.nomeata.de/` (Joachim Breitner):



Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]":

(a -> Bool) -> [a] -> [a]

Please choose a sublanguage of Haskell:

no bottoms (hence no general recursion and no selective strictness)

☐ inequational theorems (only relevant in a language with bottoms)

☑ hide type instantiations in the theorem presentation

The Free Theorem

with all permissable relation variables reduced to functions

```
forall t1,t2 in TYPES, g :: t1 -> t2.
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall x :: t1. p x = q (g x))
   ==> (forall y :: [t1]. map g (f p y) = f q (map g y))
```

Relational parametricity (Reynolds 1983)

Free theorems – How they are usually derived

Take polymorphic type, say $f :: (a \rightarrow \text{Bool}) \rightarrow ([a] \rightarrow \text{Maybe}\ a)$, replace type variables by relation variables, for the example yielding $(\mathcal{R} \rightarrow \text{Bool}) \rightarrow ([\mathcal{R}] \rightarrow \text{Maybe}\ \mathcal{R})$, invoke a parametricity theorem stating $(f, f) \in \ldots$, unfold a given set of definitions, such as:

- base types like Bool and Int are read as identity relations,
- $\mathcal{R}_1 \rightarrow \mathcal{R}_2 = \{(f, g) \mid \forall (a, b) \in \mathcal{R}_1.\ (f\ a, g\ b) \in \mathcal{R}_2\}$
- $\text{Maybe}\ \mathcal{R} = \{(\text{N}, \text{N})\} \cup \{(\text{J}\ a, \text{J}\ b) \mid (a, b) \in \mathcal{R}\}$

... and then try to massage and simplify the resulting statement.

For example (see Section 2.2 in paper):

$$
\begin{array}{ll}
 & (f, f) \in (\mathcal{R} \rightarrow id) \rightarrow ([\mathcal{R}] \rightarrow \text{Maybe}\ \mathcal{R}) \\
\Leftrightarrow & \quad [\![\text{ definition of } \mathcal{R}_1 \rightarrow \mathcal{R}_2 \ ]\!] \\
 & \forall (a, b) \in \mathcal{R} \rightarrow id.\ (f\ a, f\ b) \in [\mathcal{R}] \rightarrow \text{Maybe}\ \mathcal{R} \\
\Leftrightarrow & \quad [\![\text{ again }]\!] \\
 & \forall (a, b) \in \mathcal{R} \rightarrow id, (c, d) \in [\mathcal{R}].\ (f\ a\ c, f\ b\ d) \in \text{Maybe}\ \mathcal{R} \\
\Leftrightarrow & \ldots
\end{array}
$$

Free theorems – How they are usually derived

For example (see Section 2.2 in paper):

$$
\begin{array}{ll}
 & (f, f) \in (\mathcal{R} \to id) \to ([\mathcal{R}] \to \text{Maybe}\,\mathcal{R}) \\
\Leftrightarrow & [\![\text{ definition of } \mathcal{R}_1 \to \mathcal{R}_2 \ ]\!] \\
 & \forall (a, b) \in \mathcal{R} \to id.\ (f\ a, f\ b) \in [\mathcal{R}] \to \text{Maybe}\,\mathcal{R} \\
\Leftrightarrow & [\![\text{ again }]\!] \\
 & \forall (a, b) \in \mathcal{R} \to id, (c, d) \in [\mathcal{R}].\ (f\ a\ c, f\ b\ d) \in \text{Maybe}\,\mathcal{R} \\
\Leftrightarrow & \ldots
\end{array}
$$

Observations:

- Even when we in principle believe to "know" what the free theorem is, we essentially have to go through these steps.
- We have no guarantee that we will end up with a nice enough statement (depends on the massage/simplification heuristics).
- Depending on what language setting we are actually interested in, there will be deviations in the relation unfolding definitions, hence also in the derivations.

Toward a simpler approach

Category theory enthusiasts might suggest use of the concept of natural, or even dinatural, transformations at this point (Bainbridge et al. 1990).

But besides not being readily applicable to all language settings of interest, e.g., functional-logic languages, this is not exactly simple or intuitive (for most people, including myself).

Instead, we side-step the need for relational definitions, unfolding, etc., by the “conjuring lemma of parametricity”, originally devised in work exactly on parametricity for functional-logic languages (Mehner et al. 2014).

Detailed justification is in the paper, but the key point is that this conjuring lemma can be formulated and used without even mentioning all the relational machinery.

The conjuring lemma

Let τ, τ_1 and τ_2 be closed types. Let $e :: \tau$ be a term possibly involving a (but not in its own overall type, which is closed by assumption) and term variables $pre :: \tau_1 \to a$ and $post :: a \to \tau_2$, but no other free variables. Then for every $g :: \tau_1 \to \tau_2$,

$$e[\tau_1/a, id_{\tau_1}/pre, g/post] = e[\tau_2/a, g/pre, id_{\tau_2}/post] \quad (*)$$

- How could such an e look like?

  For example $e = \lambda xs \to [post\ y \mid y \leftarrow f\ [pre\ x \mid x \leftarrow xs]]$ with $f :: [a] \to [a]$. (Note that $e :: [\tau_1] \to [\tau_2]$.)

- Why is this interesting?

  Because in this case, $(*)$ specialises to

  $\lambda xs \to [g\ y \mid y \leftarrow f\ xs] \ = \ \lambda xs \to f\ [g\ x \mid x \leftarrow xs]$

Using the conjuring lemma

Toward a new practical generator

Given some f of polymorphic type, can we come up with some term e of closed type and only $pre :: \tau_1 \to a$ and $post :: a \to \tau_2$ (for any closed types τ_1 and τ_2) as free term variables?

Well, e should of course use f in some interesting way.
In essence, we want to build e around f, using *pre* and *post* to do away with the polymorphism of f.

Let's see on a few examples:

- $f :: [a] \to [a] \quad \rightsquigarrow \quad e = map\ post \circ f \circ map\ pre\ :: [\tau_1] \to [\tau_2]$
- $f :: (a \to \text{Bool}) \to [a] \to \text{Maybe}\ a$
  $\rightsquigarrow \quad e = \lambda h \to fmap\ post \circ f\ (h \circ post) \circ map\ pre$
  $\qquad :: (\tau_2 \to \text{Bool}) \to [\tau_1] \to \text{Maybe}\ \tau_2$
- $f :: (a \to \text{Bool}) \to a \to \text{Int}$
  $\rightsquigarrow \quad e = \lambda h \to f\ (h \circ post) \circ pre$
  $\qquad :: (\tau_2 \to \text{Bool}) \to \tau_1 \to \text{Int}$

Doing this systematically

The following does the trick:

$$
\begin{aligned}
mono_{pre,post}(a) &= post \\
mono_{pre,post}(\mathrm{Bool}) &= id \\
mono_{pre,post}(\mathrm{Int}) &= id \\
mono_{pre,post}([\sigma]) &= map\ mono_{pre,post}(\sigma) \\
mono_{pre,post}(\mathrm{Maybe}\ \sigma) &= fmap\ mono_{pre,post}(\sigma) \\
mono_{pre,post}(\sigma_1 \to \sigma_2) &= \lambda h \to mono_{pre,post}(\sigma_2) \\
&\qquad\qquad \circ\, h \,\circ \\
&\qquad\quad mono_{post,pre}(\sigma_1)
\end{aligned}
$$

... in the sense that $e = mono_{pre,post}(\sigma)\ f$ is the term we seek if f has polymorphic type σ.

Put differently, given $f :: \sigma$, we now generate the free theorem

$$mono_{id,g}(\sigma)\ f = mono_{g,id}(\sigma)\ f$$

. . . and adding deterministic simplifications

Well, actually, we generate

$$\lfloor \mathit{mono}_{\mathit{id},g}(\sigma)\ f \rfloor = \lfloor \mathit{mono}_{g,\mathit{id}}(\sigma)\ f \rfloor$$

where:

$$\begin{aligned}
\lfloor h\ t \rfloor &= t, \quad \text{if } h = \mathit{id},\ \mathit{map}\ \mathit{id},\ \mathit{fmap}\ \mathit{id}, \\
&\qquad\ \ \text{or syntactic compositions thereof} \\
\lfloor \mathit{map}\ h\ t \rfloor &= \mathit{map}\ (\lambda v \to \lfloor h\ v \rfloor)\ t \\
\lfloor \mathit{fmap}\ h\ t \rfloor &= \mathit{fmap}\ (\lambda v \to \lfloor h\ v \rfloor)\ t \\
\lfloor (\lambda h \to \mathit{body})\ t \rfloor &= \lambda v \to \lfloor \mathit{body}[t/h]\ v \rfloor \\
\lfloor (h \circ k)\ t \rfloor &= \lfloor h\ \lfloor k\ t \rfloor \rfloor \\
\lfloor h\ t \rfloor &= h\ t
\end{aligned}$$

Thanks to the types used for syntax in the implementation, and GHC's exhaustiveness checker, we know that this simple recursive definition cannot accidentally skip any simplification opportunities.

Some category theory

About (the connection to) dinaturality

A dinatural transformation, between two bifunctors **F** and **G** of the same mixed contravariant/covariant kind over the same source category, is an indexed collection of arrows ϕ_X of types $\mathbf{G}(X, X) \to \mathbf{F}(X, X)$ such that for every g this diagram commutes:

$$\begin{tikzcd}
 & \mathbf{G}(B,B) \arrow[r, "\phi_B"] & \mathbf{F}(B,B) \arrow[dr, "\mathbf{F}(id_B, g)"] & \\
\mathbf{G}(A,B) \arrow[ur, "\mathbf{G}(g, id_B)"] \arrow[dr, "\mathbf{G}(id_A, g)"'] & & & \mathbf{F}(B,A) \\
 & \mathbf{G}(A,A) \arrow[r, "\phi_A"'] & \mathbf{F}(A,A) \arrow[ur, "\mathbf{F}(g, id_A)"'] &
\end{tikzcd}$$

One possible instantiation is to let all categories involved be **Set**, let **G** be the constant functor to the final object **1**, and let each ϕ_X be the lifted constant $f_X : \mathbf{F}(X, X)$.

About (the connection to) dinaturality

That is,

$$\begin{tikzcd}
 & \mathbf{1} \arrow[r, "f_B"] & \mathbf{F}(B,B) \arrow[dr, "\mathbf{F}(id_B, g)"] & \\
\mathbf{1} \arrow[ur, "id_{\mathbf{1}}"] \arrow[dr, "id_{\mathbf{1}}"'] & & & \mathbf{F}(B,A) \\
 & \mathbf{1} \arrow[r, "f_A"'] & \mathbf{F}(A,A) \arrow[ur, "\mathbf{F}(g, id_A)"'] &
\end{tikzcd}$$

Consequently, $\mathbf{F}(id_B, g)\ f_B = \mathbf{F}(g, id_A)\ f_A$, if f_X is typed $\mathbf{F}(X, X)$ for a bifunctor $\mathbf{F}$ of appropriate kind. Using the *Hom* bifunctor, we can indeed turn polymorphic types σ into such bifunctors:

$$\begin{aligned}
&\mathbf{F}_a && (X, Y) = Y \\
&\mathbf{F}_{\mathsf{K}} && (X, Y) = \mathsf{K} \\
&\mathbf{F}_{(\mathsf{C}\ \sigma)} && (X, Y) = \mathsf{C}\ (\mathbf{F}_\sigma\ (X, Y)) \\
&\mathbf{F}_{\sigma_1 \to \sigma_2} && (X, Y) = \mathit{Hom}(\mathbf{F}_{\sigma_1}\ (Y, X), \mathbf{F}_{\sigma_2}\ (X, Y))
\end{aligned}$$

About (the connection to) dinaturality

So now we have that if $f :: \sigma$, then

$$\mathbf{F}_\sigma\,(id_B, g)\ f_B = \mathbf{F}_\sigma\,(g, id_A)\ f_A$$

for the bifunctor arising from:

$$\begin{array}{ll}
\mathbf{F}_a & (X, Y) = Y \\
\mathbf{F}_{\mathsf{K}} & (X, Y) = \mathsf{K} \\
\mathbf{F}_{(\mathsf{C}\ \sigma)} & (X, Y) = \mathsf{C}\ (\mathbf{F}_\sigma\,(X, Y)) \\
\mathbf{F}_{\sigma_1 \to \sigma_2} & (X, Y) = Hom(\mathbf{F}_{\sigma_1}\,(Y, X), \mathbf{F}_{\sigma_2}\,(X, Y))
\end{array}$$

Taking into account that the action of *Hom* on arrows is that $Hom(f, g)$ is the function $h \mapsto g \circ h \circ f$, this is exactly the statement

$$mono_{id,g}(\sigma)\ f = mono_{g,id}(\sigma)\ f$$

which was used earlier, since one can show that the definitions are related by, on the arrow level, $\mathbf{F}_\sigma\,(f, g) = mono_{f,g}(\sigma)$.

Conclusion

- Free theorems provide useful statements about pure, polymorphic functions based solely on their types.
- Their derivation “the standard way” can be very tedious.
- Using a shortcut we can get many of the interesting statements more simply.
- A practical generator can be implemented quite neatly.
- There is no deep theoretical advance here (actually some rediscovery of category theory concepts), but nice pragmatics.
- It would really be nice to have a practical generator for a functional-logic language.

References

E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990.

S. Mehner, D. Seidel, L. Straßburger, and J. Voigtländer. Parametricity and proving free theorems for functional-logic languages. In *Principles and Practice of Declarative Programming, Proceedings*, pages 19–30. ACM Press, 2014.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

When it “doesn’t work”

For types like $f :: (a \to a) \to (a \to a)$ we lose some generality.

The general free theorem would be:

$$(g \circ h = k \circ g) \Rightarrow (g \circ f\ h = f\ k \circ g)$$

We instead generate (essentially):

$$g \circ f\ (p \circ g) = f\ (g \circ p) \circ g$$

Why? And what does “like” mean above?

In a nutshell, “because” of: $(a^{+} \to a^{-})^{-} \to (a^{-} \to a^{+})^{+}$