Proving Correctness via Free Theorems

The Case of the destroy/build-Rule

Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany
voigt@tcs.inf.tu-dresden.de

Abstract

Free theorems feature prominently in the field of program transformation for pure functional languages such as Haskell. However, somewhat disappointingly, the semantic properties of so based transformations are often established only very superficially. This paper is intended as a case study showing how to use the existing theoretical foundations and formal methods for improving the situation. To that end, we investigate the correctness issue for a new transformation rule in the short cut fusion family. This destroy/build-rule provides a certain reconciliation between the competing foldr/build- and destroy/unfoldr-approaches to eliminating intermediate lists. Our emphasis is on systematically and rigorously developing the rule's correctness proof, even while paying attention to semantic aspects like potential nontermination and mixed strict/nonstrict evaluation.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Polymorphism; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory

Keywords correctness proofs, intermediate data structures, program transformations, rank-2 types, relational parametricity, shortcut deforestation, theorems for free

1. Introduction

Pure functional programming languages have a reputation for being exceptionally amenable to automatic program transformation, both because their efficient implementation is usually in dire need of extra optimization effort, and because their semantic foundations seem to support transformations (and corresponding correctness proofs!) in unique ways not available in other paradigms. A particularly popular class of transformations in this context is formed by those based on free theorems (Wadler 1989) as derived from relational parametricity (Reynolds 1983). Starting with Gill et al. (1993), there has been a whole industry of manufacturing transformations based on this methodology (Takano and Meijer 1995; Chitil 1999; Johann 2002; Svenningsson 2002; Voigtländer 2002; Domínguez and Pardo 2006; Coutts et al. 2007a,b; Fernandes et al. 2007; Ghani and Johann 2007). However, the semantic properties of these "short-cut-fusion-like" transformations have not always been established in as detailed and rigorous a manner as one might want, and even expect, given the alleged cleanliness of languages like Haskell (Peyton Jones 2003) in semantic regards. Discrepancies have arisen from the differences between the theoretical world of the original polymorphic lambda calculus studied by Reynolds (1983) and the reality of programming languages based on it. These differences mainly have to do with potential nontermination through general recursion in a Turing-complete language (which the original calculus is not) and with the choice between different evaluation strategies (a choice that is irrelevant in the original calculus due to its strong normalization property). For example, as first observed by Johann and Voigtländer (2004), in the presence of general recursion the destroy/unfoldr-transformation of Svenningsson (2002) can make a program more terminating than it originally was, and the presence of a polymorphic strict evaluation primitive in the otherwise nonstrict language Haskell brings further complications for destroy/unfoldr, including a potential loss of termination. For the foldr/build-transformation of Gill et al. (1993) general recursion remains without consequences, but the potentially mixed strict/nonstrict nature of evaluation in Haskell breaks total correctness. More recently, Fernandes et al. (2007) describe the correctness argument for their transformation rule as "fast and loose" and only mention that further preconditions may be needed to get correctness in the presence of selective strictness, and Coutts et al. (2007b) remain similarly vague on strictness issues as related to semantic correctness. For most of the other papers on transformation techniques cited above the situation is not much different.

It is a perfectly reasonable approach to first formulate and analyze new transformation techniques in an idealized setting without paying too much attention to peculiarities of potentially nonterminating computations. While such fast and loose reasoning (Danielsson et al. 2006) helps to form early intuition, and thus fosters the conception of new ideas for program transformation, there is a clear danger that semantic investigation stops there, and thus remains somewhat detached from programming language reality. The latter has certainly happened in the past.

In this paper we want to argue that nowadays there is no reason, or excuse, to refrain from thoroughly studying semantic properties of free theorems-based program transformations even as regards their interplay with general recursion and mixed strict/nonstrict evaluation as found in Haskell. The required theoretical foundations and formal methods have been developed. They are there to

PEPM'08, January 7–8, 2008, San Francisco, California, USA.

use, and they are easy to use. In order to demonstrate that ease, we are going to formally investigate the correctness issue for a new transformation rule in the short cut fusion family. This rule is of some independent interest, as it provides a reconciliation between the competing foldr/build- and destroy/unfoldr-approaches to eliminating intermediate lists. In fact, it is the somewhat elusive destroy/build-rule that has so far been missing, but is required to handle all meaningful combinations of consumers and producers as occurring in classical short cut fusion and its dual. However, our emphasis in this paper is on the systematic development of the new transformation's correctness proof, rather than on the rule's pragmatics. Using a yet unconsidered rule simply has the advantage that we can start our semantic investigation without having prejudices about whether or under which conditions total or partial correctness in Haskell might hold, instead setting off just from our intuitive transformation idea. We will show that, even while paying attention to aspects like potential nontermination and mixed strict/nonstrict evaluation, the construction of a formal correctness proof is nothing to shy away from. In particular, to construct such a proof successfully, one does not even need to involve oneself with the theoretical foundations of relational parametricity for the polymorphic lambda calculus (Reynolds 1983), the details of deriving free theorems (Wadler 1989), or the modifications and extensions to the theory required to properly account for general recursion and mixed evaluation (Johann and Voigtländer 2004). Such "partial ignorance" is possible because much of the underlying theory can be automated. In fact, we will have our Haskell-aware free theorems be derived by a tool that was implemented under our supervision (Böhme 2007) and demonstrates that the "for free"-part of free theorems is not just a slogan, but actual reality (and available online[1]). The formal development that, as usual for free theorems-based program transformations, remains to be done even after having derived the underlying free theorem will be completely guided by the form of the correctness statement that we want to prove. We will perform this development step by step, showing how necessary instantiations (the finding of which is usually considered the tricky part) present themselves during the course of action, without us having to pull any rabbits out of a hat. Throughout, we will also identify what further tool support would be beneficial to simplify the whole process even more.

2. The destroy/build-rule

Classical short cut fusion (Gill et al. 1993) uses the foldr/build-rule to eliminate intermediate lists produced by build and consumed by foldr, while its dual (Svenningsson 2002) uses the destroy/unfoldr-rule to eliminate intermediate lists produced by unfoldr and consumed by destroy. Here, foldr and unfoldr are functions from Haskell's standard libraries, while build and destroy are defined as follows:

```
build :: (∀ β. (α → β → β) → β → β) → [α]
build prod = prod (:) []

destroy :: (∀ β. (β → Maybe (α, β)) → β → γ) → [α] → γ
destroy cons xs = cons listpsi xs

listpsi :: [α] → Maybe (α, [α])
listpsi (x:xs) = Just (x, xs)
listpsi [] = Nothing
```

In addition to the two rules mentioned above, there is also a straightforward foldr/unfoldr-rule. What has been missing so far is a destroy/build-rule. It would be applicable when an intermediate list produced by a function expressible via build but not (efficiently) via unfoldr[2] is consumed by a function expressible via destroy but not via foldr (at least not in a way that benefits the efficiency of the fused program). For such an application to be worthwhile, the destroy/build-rule need not even completely eliminate an intermediate list in the same sense as the other three rules (together with subsequent, lower-level compiler optimizations) do. Instead, it would already be a win if the destroy/build-rule managed to reduce the overhead originally introduced when expressing the producer and consumer in terms of build and destroy, respectively. Let us explain.

The practical problem of how to handle a producer expressed in terms of build that does not pair up for fusion with a foldr-consumer was already encountered for the original short cut fusion rule. In order to avoid the runtime overhead incurred by (then unnecessarily) defining the producer via build, the solution proposed by Peyton Jones et al. (2001) is to define two versions of each producer: one via build and a more direct one. An appropriate "backing out" mechanism ensures that any build-version remaining after all fusion has taken place is automatically replaced by the corresponding direct one. An analogous approach is possible for the dual destroy/unfoldr-rule, but does not scale for consumers of more than one list. Consider, for example, the case zip exp_1 exp_2 with list-producing expressions exp_1 and exp_2. This is exactly the standard example for destroy/unfoldr-fusion, which works very nicely when zip consumes both its input lists via destroy and both exp_1 and exp_2 are written in terms of unfoldr. But assume the latter is possible only for one of the two, say for exp_1, but not for exp_2. Then, after the program has undergone fusion, the intermediate list corresponding to exp_1 will not be created anymore, while the one corresponding to exp_2 will still be created and, moreover, essentially an isomorphic copy of it in terms of Just (,) and Nothing will be built inside zip when that function uses destroy, and thus listpsi, to scrutinize its second argument. This duplication could even ruin the gain obtained from eliminating the intermediate list in the first argument. The "backing out" solution to this problem would be to provide a version of zip that consumes its first argument via destroy, but its second argument in direct style. However, since elsewhere in the program we could also have cases zip exp_1 exp_2 where exp_2 is expressible in terms of unfoldr while exp_1 is not, or where indeed both are not, we would altogether need to define four correlated versions of zip (as opposed to just two versions in the above scenario for foldr/build). This clearly becomes impractical. An alternative solution would be viable if we had an appropriate destroy/build-rule. Then, if a situation zip exp_1 exp_2 were encountered where, for example, exp_1 is expressible in terms of unfoldr while exp_2 is not, but the latter is at least expressible in terms of build (which will be the case much more often), we could expect destroy/unfoldr-fusion to eliminate the first intermediate list and destroy/build-fusion to deal with the second one in a way that cancels the extra effort related to the creation of Just (,)- and Nothing-structures solely to express zip in terms of destroy on both arguments in the first place.

How, then, should the destroy/build-rule look like? The definitions tell us that

```
destroy cons (build prod)
```

is equivalent to

```
cons (λys → case ys of {x:xs → Just (x, xs); [] → Nothing})
     (prod (:) [])
```

This makes it very evident how a standard list is created by *prod* using the constructors (:) and [], that list is then taken apart and essentially repackaged in Just (,)- and Nothing-structures, and finally

[1] http://linux.tcs.inf.tu-dresden.de/~voigt/ft/

[2] Conversely, unfoldr is straightforwardly written in terms of build, as well as foldr in terms of destroy, so the combination destroy/build is really the worst that can happen with respect to fusion opportunities.

cons works with those JUST (,)- and NOTHING-structures.[3] The almost obvious optimization idea would thus be to avoid the repackaging altogether and have *prod* immediately create JUST (,)- and NOTHING-structures, i.e., to replace the above as follows:

$$\mathit{cons}\ \mathsf{id}\ (\mathit{prod}\ (\lambda x\ xs \to \textsc{Just}\ (x, xs))\ \textsc{Nothing}) \qquad (1)$$

Unfortunately, this is not directly possible because of an "infinite type"-error. However, Haskell provides a workaround in terms of type isomorphisms as introduced by **newtype**-declarations. So we can define

$$\textbf{newtype}\ \mathsf{L}\ \alpha = \mathsf{L}\ \{\mathsf{unL} :: \textsc{Maybe}\ (\alpha, \mathsf{L}\ \alpha)\}$$

and let our rule replace

$$\mathsf{destroy}\ \mathit{cons}\ (\mathsf{build}\ \mathit{prod})$$

by

$$\mathit{cons}\ \mathsf{unL}\ (\mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (\mathsf{L}\ \textsc{Nothing}))$$

Due to the nature of **newtype**-declarations, unL and L are operationally identity functions, so the result of our destroy/build-rule is essentially the same as the ideal (1) above.

As indicated already in the introduction, we do not want to go into further detail about the pragmatics of destroy/build-fusion, instead turning toward the semantic investigation now. One thing that should be noted here is that there is no reason to expect any proof about the above rule to be simpler than those for other free theorems-based program transformations in the literature. The reason is that each of these earlier transformations revolves around a single rank-2 polymorphic function only, while here we have to deal with two of them at once.

3. The correctness proof

Let T_1 and T_2 be arbitrary, but from now on fixed, types and let

$$\mathit{prod} :: \forall\ \beta.\ (\mathsf{T}_1 \to \beta \to \beta) \to \beta \to \beta$$

and

$$\mathit{cons} :: \forall\ \beta.\ (\beta \to \textsc{Maybe}\ (\mathsf{T}_1, \beta)) \to \beta \to \mathsf{T}_2$$

We want to prove a semantic connection between, preferably equivalence of,

$$\mathit{cons}\ \mathsf{listpsi}\ (\mathit{prod}\ (:)\ []) \qquad (2)$$

and

$$\mathit{cons}\ \mathsf{unL}\ (\mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (\mathsf{L}\ \textsc{Nothing})) \qquad (3)$$

We do not ponder an advance roadmap for the proof, as we hope to be guided towards our goal by simply following the directions and necessities encountered on the way. Clearly, the first thing to do is to derive the free theorems corresponding to the types of *prod* and *cons*. But actually there are not "the" free theorems, because several choices have to be made here.

1. Which language or calculus should we consider as the semantic base? Do we want to derive statements valid only for the original polymorphic lambda calculus of Reynolds (1983)? Or do we want to take an extension to general recursion into account as does Wadler (1989, Section 7)? Do we even want to be prepared for the additional presence of mixed strict/nonstrict evaluation as per the primitive seq in Haskell, with technical adjustments necessary as detailed by Johann and Voigtländer (2004, 2008) and Voigtländer and Johann (2006, 2007)?
2. Do we want general relational free theorems, or more specialized versions where the roles of relations are as far as possible taken over by functions?
3. Do we want free theorems in the traditional "symmetric" fashion (Wadler 1989; Voigtländer and Johann 2006) or do we prefer "asymmetric" ones (Johann and Voigtländer 2004; Voigtländer and Johann 2007) that allow to prove semantic inequations rather than equations?

All these variations are supported by the tool of Böhme (2007) as described in Appendix A. Here we decide to start from free theorems derived for the language including both general recursion and mixed evaluation, because those are a reality in Haskell, to work with the general case of relations prior to any specialization to functions, because such generality avoids premature commitment to instantiations that might turn out to be too specific in the further course of the proof, and to go for symmetric/equational statements, because ideally we would like to prove the semantic equivalence of (2) and (3).[4] The free theorems derived with these settings from the types of *prod* and *cons* are given in Figures 1 and 2, respectively. We emphasize that producing such statements comes with practically zero effort. We simply entered the type signatures for *prod* and *cons*, made our choices with regard to which kind of free theorems we want, copied the tool's textual output into our LaTeX source, and relied on a slightly adapted version of the λTeX style (Zadarnowski 2003) to obtain the pleasing typesetting in Figures 1 and 2. The textual output as emitted by the tool itself is shown in Appendix A.

$$\begin{array}{l}
\forall\ t_1,t_2 \in \mathsf{TYPES},\ \mathsf{R} \in \mathsf{REL}(t_1,t_2),\ \mathsf{R}\ \mathit{strict},\ \mathit{continuous},\\
\mathit{and}\ \mathit{bottom{-}reflecting}.\\
\quad \forall\ p :: \mathsf{T}_1 \to t_1 \to t_1.\\
\quad\quad \forall\ q :: \mathsf{T}_1 \to t_2 \to t_2.\\
\quad\quad\quad (((p \not\equiv \bot) \Leftrightarrow (q \not\equiv \bot))\\
\quad\quad\quad\quad \wedge\ (\forall\ x :: \mathsf{T}_1.\\
\quad\quad\quad\quad\quad\quad ((p\ x \not\equiv \bot) \Leftrightarrow (q\ x \not\equiv \bot))\\
\quad\quad\quad\quad\quad\quad \wedge\ (\forall\ (y, z) \in \mathsf{R}.\ (p\ x\ y, q\ x\ z) \in \mathsf{R})))\\
\quad\quad\quad \Rightarrow (\forall\ (v, w) \in \mathsf{R}.\ (\mathit{prod}\ p\ v, \mathit{prod}\ q\ w) \in \mathsf{R})
\end{array}$$

Figure 1. Free theorem derived for *prod*.

Note the conditions imposed on the relations in Figures 1 and 2, as well as the conditions relating to ⊥. These are exactly the kinds of technical details that need to be taken into account when wanting to produce results that also hold in the presence of general recursion and mixed evaluation, i.e., results that are meaningful for Haskell, rather than just for the polymorphic lambda calculus. Here, following the foundational work of Johann and Voigtländer (2004, 2008) and Voigtländer and Johann (2006, 2007), the appropriate conditions are automatically generated by the tool we use, and we only need to properly keep track of them in the remainder of the correctness proof. For reference, Table 1 gives the definitions of *strict*, *continuous*, and *bottom−reflecting* on relations, as well as of *strict* and *total* on functions. Only the latter two definitions will explicitly be needed in what follows, as the former ones (on relations) will eventually be reduced to them automatically.

[3] Of course, due to lazy evaluation, these three "phases" occur intermingled.

[4] Inequational free theorems typically enter the stage when equational ones either cannot be proved at all or only under severe preconditions. So switching to an asymmetric setting remains an option in case the further proof construction fails to deliver a satisfying equational result. Or, one could use the more abstract approach of Johann and Voigtländer (2008), which requires no a priori commitment to either equational or inequational free theorems.

$\forall\ t_1,t_2 \in \mathsf{TYPES},\ \mathsf{R} \in \mathsf{REL}(t_1,t_2),\ \mathsf{R}\ \mathit{strict}, \mathit{continuous},$
$\mathit{and}\ \mathit{bottom{-}reflecting}.$
$\quad \forall\ p :: t_1 \to \textsc{Maybe}\ (\mathsf{T}_1, t_1).$
$\quad\quad \forall\ q :: t_2 \to \textsc{Maybe}\ (\mathsf{T}_1, t_2).$
$\quad\quad\quad (((p \neq \bot) \Leftrightarrow (q \neq \bot))$
$\quad\quad\quad\quad \wedge\ (\forall\ (x, y) \in \mathsf{R}.$
$\quad\quad\quad\quad\quad\quad (p\ x, q\ y) \in \mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id},\mathsf{R}))))$
$\quad\quad\quad \Rightarrow (\forall\ (z, v) \in \mathsf{R}.\ \mathit{cons}\ p\ z = \mathit{cons}\ q\ v)$

where

$\mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id},\mathsf{R}))$
$\quad = \{(\bot, \bot), (\textsc{Nothing}, \textsc{Nothing})\}$
$\quad \cup \{(\textsc{Just}\ x_1, \textsc{Just}\ y_1) \mid (x_1, y_1) \in \mathit{lift}\{(,)\}(\mathsf{id},\mathsf{R})\}$

$\mathit{lift}\{(,)\}(\mathsf{id},\mathsf{R})$
$\quad = \{(\bot, \bot)\}$
$\quad \cup \{((x_1, x_2), (y_1, y_2)) \mid (x_1 = y_1) \wedge ((x_2, y_2) \in \mathsf{R})\}$

Figure 2. Free theorem derived for *cons*.

R is *strict*	if $(\bot, \bot) \in \mathsf{R}$
R is *continuous*	if $(\forall\ i.\ (x_i, y_i) \in \mathsf{R}) \Rightarrow (\bigsqcup x_i, \bigsqcup y_i) \in \mathsf{R}$
R is *bottom−reflecting*	if $(x, y) \in \mathsf{R} \Rightarrow ((x \neq \bot) \Leftrightarrow (y \neq \bot))$
f is *strict*	if $f\ \bot = \bot$
f is *total*	if $(x \neq \bot) \Rightarrow (f\ x \neq \bot)$

Table 1. Restrictions on relations and functions.

Of the two statements currently at our disposal, the one in Figure 2 seems more immediately useful for proving the equivalence of (2) and (3), simply by the form of its conclusion $\mathit{cons}\ p\ z = \mathit{cons}\ q\ v$. If we instantiate as follows:

$$\begin{aligned} p &= \mathsf{listpsi} \\ z &= \mathit{prod}\ (:)\ [] \\ q &= \mathsf{unL} \\ v &= \mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (\mathsf{L}\ \textsc{Nothing}) \end{aligned}$$

we get exactly the desired equivalence.[5] Of course, given the quantifications in Figure 2, this requires that listpsi is of type $t_1 \to \textsc{Maybe}\ (\mathsf{T}_1, t_1)$ and unL is of type $t_2 \to \textsc{Maybe}\ (\mathsf{T}_1, t_2)$, i.e., that $t_1 = [\mathsf{T}_1]$ and $t_2 = \mathsf{L}\ \mathsf{T}_1$. Instantiating the statement from Figure 2 accordingly gives:[6]

$\forall\ \mathsf{R} \in \mathsf{REL}([\mathsf{T}_1],\mathsf{L}\ \mathsf{T}_1),\ \mathsf{R}\ \mathit{strict}, \mathit{continuous},$
$\mathit{and}\ \mathit{bottom{-}reflecting}.$
$\quad (((\mathsf{listpsi} \neq \bot) \Leftrightarrow (\mathsf{unL} \neq \bot))$
$\quad\quad \wedge\ (\forall\ (x, y) \in \mathsf{R}.$
$\quad\quad\quad\quad (\mathsf{listpsi}\ x, \mathsf{unL}\ y) \in \mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id},\mathsf{R}))))$
$\quad \Rightarrow ((\mathit{prod}\ (:)\ [],$
$\quad\quad\quad \mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (\mathsf{L}\ \textsc{Nothing})) \in \mathsf{R}$
$\quad\quad \Rightarrow \mathit{cons}\ \mathsf{listpsi}\ (\mathit{prod}\ (:)\ [])$
$\quad\quad\quad\quad = \mathit{cons}\ \mathsf{unL}\ (\mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))$
$\quad\quad\quad\quad\quad\quad (\mathsf{L}\ \textsc{Nothing})))$

[5] Note that, thanks to our decision to start from general relational free theorems, the statement in Figure 2 (as well as that in Figure 1) is completely symmetric with respect to taking the inverse of R. Thus, it is irrelevant whether we choose to match *cons p z* against (2) and *cons q v* against (3) or vice versa. There can be no premature commitment here.

[6] We do not yet concern ourselves with the details of the *lift*-relations here.

Here we have performed the instantiations manually in our text editor (with the output again being pretty-printed via LaTeX), but clearly some tool support for such tasks would be beneficial and should not be all too complicated to implement.

Since both listpsi and unL are by themselves partial applications, i.e., semantically equivalent to lambda-abstractions, they are clearly different from $\bot$. Thus, the

$$(\mathsf{listpsi} \neq \bot) \Leftrightarrow (\mathsf{unL} \neq \bot)$$

precondition is true and can be omitted. (Again, more tool support would be welcome to perform such nearly trivial simplifications.) This means that we could conclude our desired equivalence if we knew of an $\mathsf{R} \in \mathsf{REL}([\mathsf{T}_1],\mathsf{L}\ \mathsf{T}_1)$ that is *strict*, *continuous*, and *bottom−reflecting*, and for which both

$$\begin{aligned} &\forall\ (x, y) \in \mathsf{R}. \\ &\quad (\mathsf{listpsi}\ x, \mathsf{unL}\ y) \in \mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id},\mathsf{R})) \end{aligned} \tag{4}$$

and

$$\begin{aligned} &(\mathit{prod}\ (:)\ [], \\ &\quad \mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (\mathsf{L}\ \textsc{Nothing})) \in \mathsf{R} \end{aligned} \tag{5}$$

hold.

Note that the conclusion of the statement in Figure 1 already has a form matching that of (5). So if we could find an appropriate R such that for

$$\begin{aligned} p &= (:) \\ v &= [] \\ q &= \lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)) \\ w &= \mathsf{L}\ \textsc{Nothing} \end{aligned}$$

the preconditions from Figure 1 were fulfilled and additionally we could establish (4), then we would be done. Unfortunately, this information does not yet really help us to come up with a concrete R fulfilling those needs. But we know that an often successful heuristics in working with free theorems is to consider the special case where functions take the roles of relations. Hence, we might want to investigate the special cases of R being the graph of a function or the inverse thereof. Instead of doing such specializations manually, we can resort to tool support once more.

Instructing the tool described in Appendix A to specialize R in the statement from Figure 1 to a function automatically yields the statement in Figure 3. In order to use that new statement to do away with the condition (5) encountered above, we now have two choices: either to instantiate it in such a way that

$$\begin{aligned} p &= (:) \\ z &= [] \\ q &= \lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)) \\ f\ z &= \mathsf{L}\ \textsc{Nothing} \end{aligned}$$

or in such a way that

$$\begin{aligned} p &= \lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)) \\ z &= \mathsf{L}\ \textsc{Nothing} \\ q &= (:) \\ f\ z &= [] \end{aligned}$$

This decision corresponds to whether the R we are looking for to satisfy (5), and (4), is expected to be the graph of function f from Figure 3 or the inverse thereof. Let us investigate the two choices separately.

Choice 1. This would mean that in the statement from Figure 3 we must instantiate $t_1 = [\mathsf{T}_1]$ (due to the type of p) and $t_2 = \mathsf{L}\ \mathsf{T}_1$ (due to the type of q). Also, since p and q, as well as their applications to an arbitrary single argument, are partial applications, they all are different from $\bot$, which allows simplifications similar to what we did before when working with the statement from Figure 2. Here, this leads to:

$$
\begin{array}{l}
\forall\ t_1,t_2 \in \mathsf{TYPES},\ f :: t_1 \to t_2,\ \textit{f strict and total}. \\
\quad \forall\ p :: \mathsf{T}_1 \to t_1 \to t_1. \\
\quad\quad \forall\ q :: \mathsf{T}_1 \to t_2 \to t_2. \\
\quad\quad\quad (((p \neq \bot) \Leftrightarrow (q \neq \bot)) \\
\quad\quad\quad\quad \wedge\ (\forall\ x :: \mathsf{T}_1. \\
\quad\quad\quad\quad\quad\quad ((p\ x \neq \bot) \Leftrightarrow (q\ x \neq \bot)) \\
\quad\quad\quad\quad\quad\quad \wedge\ (\forall\ y :: t_1.\ f\ (p\ x\ y) = q\ x\ (f\ y)))) \\
\quad\quad\quad \Rightarrow (\forall\ z :: t_1.\ f\ (\mathit{prod}\ p\ z) = \mathit{prod}\ q\ (f\ z))
\end{array}
$$

Figure 3. Specialized free theorem for *prod*.

$$
\begin{array}{l}
\forall\ f :: [\mathsf{T}_1] \to \mathsf{L}\ \mathsf{T}_1,\ f\ \textit{strict and total}. \\
\quad (\forall\ x :: \mathsf{T}_1,\ y :: [\mathsf{T}_1]. \\
\quad\quad f\ ((:)\ x\ y) = (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ x\ (f\ y)) \\
\quad \Rightarrow f\ (\mathit{prod}\ (:)\ []) = \mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (f\ [])
\end{array}
$$

The precondition of the implication in this statement together with the fact that we expect $f\ [] = \mathsf{L}\ \textsc{Nothing}$ essentially leave no room other than to consider the following definition for f (after two beta-reductions):

$$
\begin{array}{l}
f :: [\mathsf{T}_1] \to \mathsf{L}\ \mathsf{T}_1 \\
f\ (x{:}y) = \mathsf{L}\ (\textsc{Just}\ (x, f\ y)) \\
f\ [] = \mathsf{L}\ \textsc{Nothing}
\end{array}
$$

It remains to check that this f is *strict* and *total*, which it is, and we obtain from the above that for this f:

$$
\begin{array}{l}
\quad f\ (\mathit{prod}\ (:)\ []) \\
\quad\quad = \mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (\mathsf{L}\ \textsc{Nothing})
\end{array} \tag{6}
$$

Choice 2. In a completely analogous fashion, this choice would mean that in the statement from Figure 3 we must instantiate $t_1 = \mathsf{L}\ \mathsf{T}_1$ and $t_2 = [\mathsf{T}_1]$, so that then:

$$
\begin{array}{l}
\forall\ f :: \mathsf{L}\ \mathsf{T}_1 \to [\mathsf{T}_1],\ f\ \textit{strict and total}. \\
\quad (\forall\ x :: \mathsf{T}_1,\ y :: \mathsf{L}\ \mathsf{T}_1. \\
\quad\quad f\ ((\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ x\ y) = (:)\ x\ (f\ y)) \\
\quad \Rightarrow f\ (\mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (\mathsf{L}\ \textsc{Nothing})) \\
\quad\quad\quad = \mathit{prod}\ (:)\ (f\ (\mathsf{L}\ \textsc{Nothing}))
\end{array}
$$

which together with the requirement $f\ (\mathsf{L}\ \textsc{Nothing}) = []$ suggests to consider the following definition for f:

$$
\begin{array}{l}
f :: \mathsf{L}\ \mathsf{T}_1 \to [\mathsf{T}_1] \\
f\ (\mathsf{L}\ (\textsc{Just}\ (x, y))) = x{:}(f\ y) \\
f\ (\mathsf{L}\ \textsc{Nothing}) = []
\end{array}
$$

It turns out, however, that this function is not *total*, since one has $f\ (\mathsf{L}\ (\textsc{Just}\ \bot)) = \bot$. Fortunately, it is easy to obtain a *strict* and *total* function from it that still satisfies the precondition in the implication above, by introducing an irrefutable pattern as follows:

$$
\begin{array}{l}
f :: \mathsf{L}\ \mathsf{T}_1 \to [\mathsf{T}_1] \\
f\ (\mathsf{L}\ (\textsc{Just}\ \tilde{}(x, y))) = x{:}(f\ y) \\
f\ (\mathsf{L}\ \textsc{Nothing}) = []
\end{array}
$$

For this f we then obtain from the above that:

$$
f\ (\mathit{prod}\ (\lambda x\ xs \to \mathsf{L}\ (\textsc{Just}\ (x, xs)))\ (\mathsf{L}\ \textsc{Nothing})) = \mathit{prod}\ (:)\ []
$$

Through investigation of the two choices, we have now come up with two candidates for the relation R we were seeking to satisfy (4) and (5). We could proceed manually from here, working out what the remaining condition (4) translates to when taking R to be the graph of f in Choice 1, or the inverse of the graph of f in Choice 2. But actually it seems easier to yet again use tool support and perform an automatic "functional specialization" of the statement from Figure 2 (as we did before for the one from Figure 1, leading to Figure 3), and to work onwards from there. The tool's output is shown in Figure 4.

$$
\begin{array}{l}
\forall\ t_1,t_2 \in \mathsf{TYPES},\ f :: t_1 \to t_2,\ \textit{f strict and total}. \\
\quad \forall\ p :: t_1 \to \textsc{Maybe}\ (\mathsf{T}_1, t_1). \\
\quad\quad \forall\ q :: t_2 \to \textsc{Maybe}\ (\mathsf{T}_1, t_2). \\
\quad\quad\quad (((p \neq \bot) \Leftrightarrow (q \neq \bot)) \\
\quad\quad\quad\quad \wedge\ (\forall\ x :: t_1. \\
\quad\quad\quad\quad\quad\quad (p\ x,\ q\ (f\ x)) \in \mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id}, f)))) \\
\quad\quad\quad \Rightarrow (\forall\ y :: t_1.\ \mathit{cons}\ p\ y = \mathit{cons}\ q\ (f\ y))
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id}, f)) \\
\quad = \{(\bot, \bot), (\textsc{Nothing}, \textsc{Nothing})\} \\
\quad \cup \{(\textsc{Just}\ x_1, \textsc{Just}\ y_1) \mid (x_1, y_1) \in \mathit{lift}\{(,)\}(\mathsf{id}, f)\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{lift}\{(,)\}(\mathsf{id}, f) \\
\quad = \{(\bot, \bot)\} \\
\quad \cup \{((x_1, x_2), (y_1, y_2)) \mid (x_1 = y_1) \wedge (f\ x_2 = y_2)\}
\end{array}
$$

Figure 4. Specialized free theorem for *cons*.

Doing things this way means that we have to essentially redo some of the manipulation steps that we already did on the relational free theorem from Figure 2. But all this goes through very smoothly, given what we already know. In particular, for Choice 1, i.e., for

$$
\begin{array}{l}
f :: [\mathsf{T}_1] \to \mathsf{L}\ \mathsf{T}_1 \\
f\ (x{:}y) = \mathsf{L}\ (\textsc{Just}\ (x, f\ y)) \\
f\ [] = \mathsf{L}\ \textsc{Nothing}
\end{array}
$$

and thus $t_1 = [\mathsf{T}_1]$ and $t_2 = \mathsf{L}\ \mathsf{T}_1$, we clearly still want that $p :: [\mathsf{T}_1] \to \textsc{Maybe}\ (\mathsf{T}_1, [\mathsf{T}_1])$ and $q :: \mathsf{L}\ \mathsf{T}_1 \to \textsc{Maybe}\ (\mathsf{T}_1, \mathsf{L}\ \mathsf{T}_1)$ are listpsi and unL, respectively. Also as before, this does away with the $(p \neq \bot) \Leftrightarrow (q \neq \bot)$ condition, leaving us now with:

$$
\begin{array}{l}
(\forall\ x :: [\mathsf{T}_1].\ (\mathsf{listpsi}\ x, \mathsf{unL}\ (f\ x)) \in \mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id}, f))) \\
\Rightarrow (\forall\ y :: [\mathsf{T}_1].\ \mathit{cons}\ \mathsf{listpsi}\ y = \mathit{cons}\ \mathsf{unL}\ (f\ y))
\end{array}
$$

Either from our original motivation of proving the equivalence of (2) and (3), or from what we did earlier for the relational free theorem from Figure 2, it is clear that next we want to instantiate $y = \mathit{prod}\ (:)\ []$, which gives:

$$
\begin{array}{l}
(\forall\ x :: [\mathsf{T}_1].\ (\mathsf{listpsi}\ x, \mathsf{unL}\ (f\ x)) \in \mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id}, f))) \\
\Rightarrow \mathit{cons}\ \mathsf{listpsi}\ (\mathit{prod}\ (:)\ []) = \mathit{cons}\ \mathsf{unL}\ (f\ (\mathit{prod}\ (:)\ []))
\end{array}
$$

Together with (6) this gives the desired equivalence of (2) and (3), provided we can establish the precondition, i.e., that for every $x :: [\mathsf{T}_1]$,

$$
(\mathsf{listpsi}\ x, \mathsf{unL}\ (f\ x)) \in \mathit{lift}\{\textsc{Maybe}\}(\mathit{lift}\{(,)\}(\mathsf{id}, f))
$$

But with the definition of *lift*{MAYBE}(*lift*{(,)}(id,*f*)) from Figure 4, and taking the definitions of listpsi, unL, and *f* into account as well, this is easily shown by case distinction on *x* as follows.

- If $x = \bot$, then (listpsi *x*, unL (*f x*)) = ($\bot$, $\bot$).
- If $x = []$, then (listpsi *x*, unL (*f x*)) = (NOTHING, NOTHING).
- If $x = z{:}zs$ for some $z :: T_1$ and $zs :: [T_1]$, then

$$\begin{aligned} \text{listpsi } x &= \text{JUST } (z, zs) \\ \text{unL } (f\ x) &= \text{JUST } (z, f\ zs) \end{aligned}$$

  and the pair of these is in *lift*{MAYBE}(*lift*{(,)}(id,*f*)) due to ((*z*, *zs*), (*z*, *f zs*)) ∈ *lift*{(,)}(id,*f*).

This completes the proof!

It is instructive to consider also what would have happened for Choice 2, i.e., for

```
f :: L T1 → [T1]
f (L (JUST ~(x, y))) = x:(f y)
f (L NOTHING) = []
```

After some steps completely analogous to those above, we would finally have had to argue that for this function it holds that for every $x :: L\ T_1$,

$$(\text{unL } x, \text{listpsi } (f\ x)) \in \mathit{lift}\{\text{MAYBE}\}(\mathit{lift}\{(,)\}(\text{id},f))$$

But actually this does not hold for $x = L\ (\text{JUST } \bot)$, since then

$$\begin{aligned} \text{unL } x &= \text{JUST } \bot \\ \text{listpsi } (f\ x) &= \text{JUST } (\bot, \bot) \end{aligned}$$

and the pair of these is not in *lift*{MAYBE}(*lift*{(,)}(id,*f*)). This subtlety clearly underscores the importance of remaining "⊥-aware" while analyzing the semantics of program transformations for a language like Haskell. It also shows that it is wise to proceed in a systematic fashion about constructing corresponding proofs, in particular by not prematurely committing to certain instantiations, and by conclusively investigating alternatives as we did by considering both choices of specializing R down to function level.

4. Conclusion

We hope to have convinced the reader that constructing rigorous correctness proofs for free theorems-based program transformations, even while taking semantic intricacies of a language like Haskell into full account, is not as scary a task as one might think. In particular, the tool support available today is quite good already. It should be easy to add, as an additional output format beside textual output and pretty-printed LaTeX, support for output in the syntax of some proof assistant, so that existing facilities for manipulating proofs could be leveraged instead of resorting, e.g., to manual substitutions in a text editor as we have done here.

As it turned out, and in contrast to its older siblings, the destroy/build-rule does not need to be equipped with special preconditions in order to be valid as a semantic equivalence even in the presence of general recursion and mixed strict/nonstrict evaluation. This is interesting news, which clearly could not have become known without actually doing the work of constructing a proof that carefully keeps track of (and indicates where to successfully do away with) those conditions internal to the correctness argument that are mandated by the presence of fix and seq.

Like earlier short-cut-fusion-like techniques, the destroy/build-rule can be generalized from lists to other algebraic data types. For example, consider the following tree type:

```
data TREE α = NODE (TREE α) (TREE α) | LEAF α
```

It gives rise to the following two rank-2 polymorphic functions:

```
buildT :: (∀ β. (β → β → β) → (α → β) → β) → TREE α
buildT prod = prod NODE LEAF

destroyT :: (∀ β. (β → F α β) → β → γ) → TREE α → γ
destroyT cons t = cons treepsi t
```

where

```
data F α β = N β β | L α

treepsi :: TREE α → F α (TREE α)
treepsi (NODE t1 t2) = N t1 t2
treepsi (LEAF x) = L x
```

The destroyT/buildT-rule then replaces

```
destroyT cons (buildT prod)
```

by

```
cons unT (prod (λt1 t2 → T (N t1 t2)) (λx → T (L x)))
```

where

```
newtype T α = T {unT :: F α (T α)}
```

That rule's unconditional total correctness can be established in the same manner as demonstrated for the destroy/build-rule in this paper. In particular, Figure 8 in Appendix A shows that free theorems can also be automatically generated for functions involving user-defined data types, such as for

```
cons :: ∀ β. (β → F T1 β) → β → T2
```

A. Tool support

Böhme (2007) implemented a library and a shell-based application using it for generation and manipulation of free theorems. A web interface providing some of its functionality is accessible at

`http://linux.tcs.inf.tu-dresden.de/~voigt/ft`

This web interface presents the user with an input form as shown in Figure 5. Entering the type signature of *prod* and selecting the sublanguage of Haskell offered as third choice leads to the output shown in Figure 6, which corresponds to Figures 1 and 3 in the body of the paper. Likewise, entering the type signature of *cons* leads to the output shown in Figure 7, which corresponds to Figures 2 and 4.

The mentioned web page also provides the source code of the library and shell-based application. The latter offers a number of features not accessible via the web interface. For example, it allows to selectively specialize relations to graphs of functions or inverses thereof, instead of the all-or-none approach present in the web interface. It also enables the user to declare their own algebraic data types, type synonyms, type renamings, and type classes, which are then taken into account when deriving free theorems. For example, Figure 8 shows a sample session that first loads a file `trees.hs` which contains the declarations from the previous section (plus dummy declarations for the fixed types T_1 and T_2). For simplicity of interaction, the user of the web interface is instead restricted to predefined declarations from the Haskell standard libraries.

Acknowledgments

I thank the PEPM reviewers for their comments and suggestions.

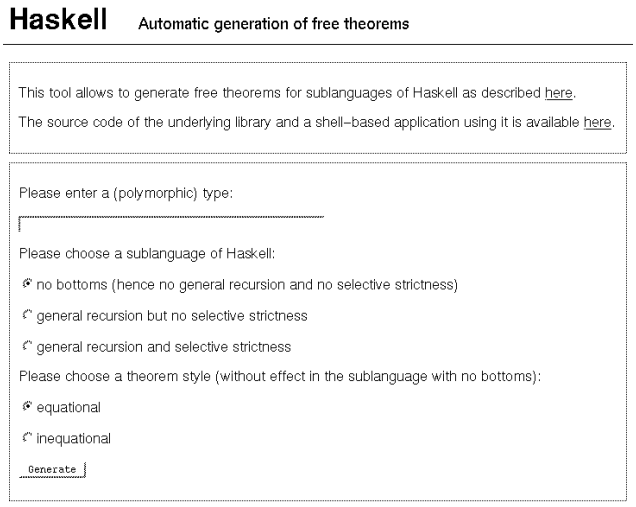

Figure 5. Input form of the web interface.

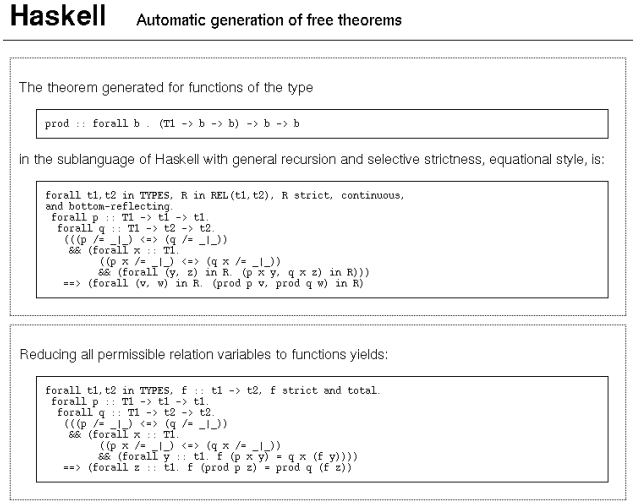

Figure 6. Tool output for *prod*.

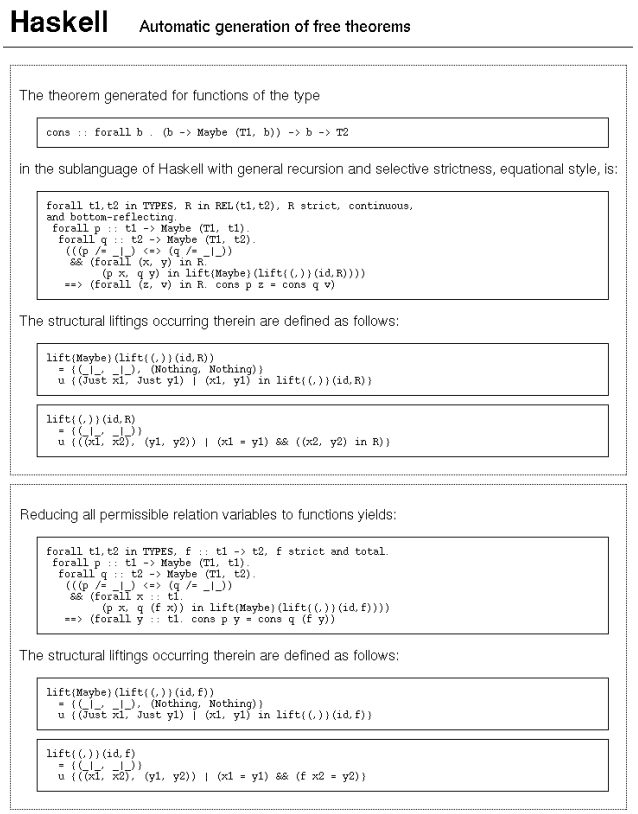

Figure 7. Tool output for *cons*.

References

S. Böhme. Free theorems for sublanguages of Haskell. Master's thesis, Technische Universität Dresden, 2007.

O. Chitil. Type inference builds a short cut to deforestation. In *International Conference on Functional Programming, Proceedings*, pages 249–260. ACM Press, 1999.

D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting Haskell strings. In *Practical Aspects of Declarative Languages, Proceedings*, volume 4354 of *LNCS*, pages 50–64. Springer-Verlag, 2007a.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *International Conference on Functional Programming, Proceedings*, pages 315–326. ACM Press, 2007b.

N.A. Danielsson, R.J.M. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Principles of Programming Languages, Proceedings*, pages 206–217. ACM Press, 2006.

F. Domínguez and A. Pardo. Program fusion with paramorphisms. In *Mathematically Structured Functional Programming, Proceedings*, Electronic Workshops in Computing. British Computer Society, 2006.

J.P. Fernandes, A. Pardo, and J. Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell Workshop, Proceedings*, pages 95–106. ACM Press, 2007.

N. Ghani and P. Johann. Monadic augment and generalised short cut fusion. *Journal of Functional Programming*, 17(6):731–776, 2007.

A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.

P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002.

P. Johann and J. Voigtländer. Free theorems in the presence of *seq*. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

P. Johann and J. Voigtländer. A family of syntactic logical relations for the semantics of Haskell-like languages. *Information and Computation*, 2008. To appear.

S.L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

S.L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop, Proceedings*, pages 203–233, 2001.

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

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM Press, 2002.

A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 306–313. ACM Press, 1995.

J. Voigtländer. Concatenate, reverse and map vanish for free. In *International Conference on Functional Programming, Proceedings*, pages 14–25. ACM Press, 2002.

```
> :load trees.hs
Loading ‘trees.hs’ ... found 10 declarations.
> :declaration F
data F alpha beta
    = N beta beta
    | L alpha
> :seq-equational
The current language subset is ‘seq-equational’.
> cons :: forall beta. (beta -> F T1 beta) -> beta -> T2
The free theorem for the type signature

  cons :: forall beta . (beta -> F T1 beta) -> beta -> T2

in the language subset ‘seq-equational’ is:

  forall t1,t2 in TYPES, R in REL(t1,t2), R strict, continuous,
  and bottom-reflecting.
   forall p :: t1 -> F T1 t1.
    forall q :: t2 -> F T1 t2.
     (((p /= _|_) <=> (q /= _|_))
      && (forall (x, y) in R. (p x, q y) in lift{F}(id,R)))
     ==> (forall (z, v) in R. cons p z = cons q v)

cons > :specialise R
The free theorem for the type signature

  cons :: forall beta . (beta -> F T1 beta) -> beta -> T2

in the language subset ‘seq-equational’ is:

  forall t1,t2 in TYPES, f :: t1 -> t2, f strict and total.
   forall p :: t1 -> F T1 t1.
    forall q :: t2 -> F T1 t2.
     (((p /= _|_) <=> (q /= _|_))
      && (forall x :: t1. (p x, q (f x)) in lift{F}(id,f)))
     ==> (forall y :: t1. cons p y = cons q (f y))

cons > :lifts

  lift{F}(id,f)
    = {(_|_, _|_)}
    u {(N x1 x2, N y1 y2) | (f x1 = y1) && (f x2 = y2)}
    u {(L x1, L y1) | x1 = y1}
```

Figure 8. The shell-based application in action.

J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics. Technical Report TUD-FI06-02, Technische Universität Dresden, 2006.

J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theoretical Computer Science*, 388(1–3):290–318, 2007.

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

P. Zadarnowski. LaTeX style, 2003.