Types for Programming and Reasoning

Habilitationsschrift

zur Erlangung des akademischen Grades
Dr. rer. nat. habil.
im Fachgebiet Informatik

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dr. rer. nat. Janis Voigtländer
geb. am 20. November 1977 in Halle/Saale

Dresden, im November 2008

Contents

Chapter 1

Introduction

Types play an increasingly important role in program construction, software engineering, and reasoning about programs. They serve as documentation of functionality, even as partial specifications, and can help to rule out whole classes of errors before a program is ever run. This provides qualitative guarantees and enables safe reuse of code components. In particular, static type checking constitutes a limited form of automatic correctness proof below the threshold of full, and undecidable, program verification.

An important asset for maintaining a strong type discipline, which attempts to prevent the use of code in unfit contexts by assigning types that are as precise and descriptive as possible, without forgoing the goal of also flexible reuse is the availability of *polymorphism*, first identified as a distinct concept by Strachey (1967). A polymorphic type, or type scheme, expresses that a certain functionality is offered for different concrete type instantiations, and that in a controlled fashion.

Even though the prospects of strong typing as a lightweight yet powerful formal method have already begun to influence the design of mainstream programming languages, and in particular Java and C# are embracing ever more sophisticated forms of polymorphism (Naftalin and Wadler 2006, Golding 2005), the real stronghold of typeful programming is in the area of *functional* programming languages like ML (Milner et al. 1997) and Haskell (Peyton Jones 2003). To some degree this may be an almost incidental consequence of the origin of much of the initial research into expressive types (Cardelli and Wegner 1985), but there are also profound reasons. The clear mathematical basis of the functional paradigm makes languages adhering to it particularly amenable to precise analysis and thus allows the formalization and mastery, both theoretically and practically, of very potent type systems. In fact, one of the defining characteristics of Haskell over the last two decades has been its role as a highly effective laboratory in which to explore, design, and implement advanced type-related ideas (Hudak et al. 2007).

So the forefront of type research is still in the field of functional programming, and it is also this setting in which the work reported on in this thesis explores ways of using types for high-level program construction and reasoning about the behavior of programs. Specifically, a very nice synergy arises from Haskell's type discipline

and Haskell's insistence on being a *pure*, rather than just any other "almost", functional language. The "no compromises" attitude vis-à-vis any impulse to relax the compliance with the mathematical concept of side-effect-free functions contributes to the existence of powerful reasoning techniques that connect the types of functions to those functions' possible observable behaviors.[1] One such technique is the systematic derivation of statements about program behavior from (polymorphic) types alone. Originating from Reynolds' (1983) characterization of what it means, abstractly, for a function to be fully polymorphic over some part of its type, this approach has been popularized by Wadler (1989) under the slogan of "free theorems". It combines fruitfully with algebraic techniques like equational reasoning.

One prominent application area for free theorems has been, and continues to be, the conception and study of semantics-preserving program transformations that can be used in a compiler to optimize for execution speed. This thesis reports on our furthering of that kind of applications (Chapters 6–9), but also on somewhat surprising applications outside the core area of programming language research as such (Chapters 4 and 5). Another topic that is covered is a series of investigations into the theoretical underpinning of free theorems as regards aspects of Haskell previously unaccounted for (Chapters 8 and 10).

To set the stage, though, Chapters 2 and 3 first give a brief introduction to Haskell, its abstraction facilities and associated reasoning techniques in general and on the type level, and a program transformation technique of the mentioned kind. The subsequent chapters then provide high-level overviews of the results obtained in the full papers that are appended after page 84.

1.1 List of Appended Papers

The papers included in this thesis are, in chronological order of publication:

1. J. Voigtländer. Concatenate, Reverse and Map Vanish For Free. In S.L. Peyton Jones, editor, *7th International Conference on Functional Programming, Pittsburgh, Pennsylvania, Proceedings*, volume 37(9) of *SIGPLAN Notices*, pages 14–25. ACM Press, 2002.
   DOI: 10.1145/583852.581481.

2. P. Johann and J. Voigtländer. Free Theorems in the Presence of *seq*. In X. Leroy, editor, *31st Symposium on Principles of Programming Languages, Venice, Italy, Proceedings*, volume 39(1) of *SIGPLAN Notices*, pages 99–110. ACM Press, 2004.
   DOI: 10.1145/982962.964010.

[1] As an aside, it is pleasing to see that the mentioned attitude has not prevented Haskell from becoming practically useful as well. Indeed, several success stories from different sectors of industry have been reported in the "Commercial Users of Functional Programming" workshop series (`http://www.galois.com/cufp`).

3. P. Johann and J. Voigtländer. The Impact of *seq* on Free Theorems-Based Program Transformations. *Fundamenta Informaticae*, volume 69(1–2), Special Issue "Program Transformation: Theoretical Foundations and Basic Techniques. Part 2", guest editors: A. Pettorossi and M. Proietti, pages 63–102. IOS Press, 2006.

4. J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theoretical Computer Science*, volume 388(1–3), pages 290–318. Elsevier, 2007.
DOI: 10.1016/j.tcs.2007.09.014.

5. J. Voigtländer. Proving Correctness via Free Theorems: The Case of the `destroy/build`-Rule. In R. Glück and O. de Moor, editors, *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, Proceedings*, pages 13–20. ACM Press, 2008.
DOI: 10.1145/1328408.1328412.

6. J. Voigtländer. Much Ado about Two: A Pearl on Parallel Prefix Computation. In P. Wadler, editor, *35th Symposium on Principles of Programming Languages, San Francisco, California, Proceedings*, volume 43(1) of *SIGPLAN Notices*, pages 29–35. ACM Press, 2008.
DOI: 10.1145/1328897.1328445.

7. J. Voigtländer. Semantics and Pragmatics of New Shortcut Fusion Rules. In J. Garrigue and M. Hermenegildo, editors, *9th International Symposium on Functional and Logic Programming, Ise, Japan, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, 2008.
DOI: 10.1007/978-3-540-78969-7_13.

8. J. Voigtländer. Asymptotic Improvement of Computations over Free Monads. In C. Paulin-Mohring and P. Audebaud, editors, *9th International Conference on Mathematics of Program Construction, Marseille, France, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008.
DOI: 10.1007/978-3-540-70594-9_20.

9. J. Voigtländer. Bidirectionalization for Free! In B.C. Pierce, editor, *36th Symposium on Principles of Programming Languages, Savannah, Georgia, Proceedings*. ACM Press, 2009.

10. P. Johann and J. Voigtländer. A Family of Syntactic Logical Relations for the Semantics of Haskell-like Languages. *Information and Computation*, Special Issue "Structural Operational Semantics", guest editors: R. van Glabbeek and P.D. Mosses. Elsevier, 2008.
DOI: 10.1016/j.ic.2007.11.009.

The papers are not actually appended in this chronological order. Instead, they are appended, after page 84, in the order in which we discuss their results, that is, in the order 6, 9, 1, 8, 2, 3, 5, 7, 4, 10.

Chapter 2

Haskell's Abstraction Facilities

We begin by briefly illustrating some important Haskell concepts, based on examples. This is not intended to be a thorough introduction to the language, but rather should serve to recall key ingredients of the overall programming methodology, as well as to clarify Haskell's syntax for readers more familiar with other functional languages. We also highlight ways of structuring Haskell programs by means of abstraction and introduce the technique of equational reasoning. For comprehensive accounts of the language including the discussion of features like lazy evaluation we refer the reader to the recent textbooks of Hutton (2007) and of O'Sullivan et al. (2008). We use some language extensions that are supported only by the Glasgow Haskell Compiler (`http://www.haskell.org/ghc`) at present. Throughout the thesis, we mention the appropriate compiler flags (for GHC version 6.8.2) where this is the case.

2.1 A Short Tour of Haskell

Programming in Haskell means programming with equations. For example, a function delivering for every integer n, assumed to be nonnegative, the sum of the integer values between 0 and n is given as follows:

$$\begin{aligned} &\texttt{sum} :: \mathsf{Int} \to \mathsf{Int} \\ &\texttt{sum}\ 0 \qquad\ = 0 \\ &\texttt{sum}\ (n+1) = n + (\texttt{sum}\ n) \end{aligned}$$

Note the (optional) type signature, the use of recursion/induction, and the definition by cases. This looks much like how a mathematician would typically write down a specification of the function `sum`, except for a different way of using parentheses in denoting function application.

Definition by cases is supported via so-called *pattern-matching* on the left-hand sides of equations, which is also available at other types than that of integers. For example, summing up the elements of a *list* of integer values can be done as follows:

$$\begin{aligned} &\texttt{listsum} :: [\mathsf{Int}] \to \mathsf{Int} \\ &\texttt{listsum}\ [\,] \qquad\ = 0 \\ &\texttt{listsum}\ (n : ns) = n + (\texttt{listsum}\ ns) \end{aligned}$$

The syntax of lists, as well as the way in which pattern-matching works for them, should become clear from the following example evaluation:

$$\begin{aligned} & \texttt{listsum}\ [1,2,3,42] \\ = \ & 1 + (\texttt{listsum}\ [2,3,42]) \\ = \ & 1 + (2 + (\texttt{listsum}\ [3,42])) \\ = \ & 1 + (2 + (3 + (\texttt{listsum}\ [42]))) \\ = \ & 1 + (2 + (3 + (42 + (\texttt{listsum}\ [])))) \\ = \ & 1 + (2 + (3 + (42 + 0))) \\ = \ & 48 \end{aligned}$$

In addition to existing types like integers and lists, the user can define their own types at need, in particular arbitrary *algebraic data types*. For example, a type of binary, leaf-labeled integer trees is introduced as follows:

$$\textbf{data}\ \mathsf{Tree} = \mathsf{Node}\ \mathsf{Tree}\ \mathsf{Tree} \mid \mathsf{Leaf}\ \mathsf{Int}$$

Pattern-matching is automatically available for such user-defined types as well:

$$\begin{aligned} & \texttt{treesum} :: \mathsf{Tree} \to \mathsf{Int} \\ & \texttt{treesum}\ (\mathsf{Leaf}\ n) \quad = n \\ & \texttt{treesum}\ (\mathsf{Node}\ t_1\ t_2) = (\texttt{treesum}\ t_1) + (\texttt{treesum}\ t_2) \end{aligned}$$

Often, some functionality is useful at, and can indeed be uniformly specified for, more than one particular type. For example, computing the *length* of a list should be possible completely independently of the (type of) values contained *in* the list. The desired reuse here is enabled by *polymorphism*. A polymorphic type is one in which some concrete types are replaced by type variables. The length example then takes the following form:

$$\begin{aligned} & \texttt{length} :: [\alpha] \to \mathsf{Int} \\ & \texttt{length}\ [\,] \quad = 0 \\ & \texttt{length}\ (a : as) = 1 + (\texttt{length}\ as) \end{aligned}$$

This function can be used on lists of integers, [Int], lists of Boolean values, [Bool], and even lists of trees, [Tree], lists of lists of integers, [[Int]], and so on.

Polymorphism is not only available when defining functions, but also when defining types. For example, a more general version of the above tree data type, abstracted over the type of leaves, could have been introduced as follows:

$$\textbf{data}\ \mathsf{Tree}\ \alpha = \mathsf{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha) \mid \mathsf{Leaf}\ \alpha$$

Then we could still have `treesum` with exactly the same defining equations as above, but revised type signature $\texttt{treesum} :: \mathsf{Tree}\ \mathsf{Int} \to \mathsf{Int}$, and moreover could write functions that do not depend on a particular type of leaf values. Like so, using the list concatenation operator $(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha]$:

$$\begin{aligned} & \texttt{flatten} :: \mathsf{Tree}\ \alpha \to [\alpha] \\ & \texttt{flatten}\ (\mathsf{Leaf}\ a) \quad = [a] \\ & \texttt{flatten}\ (\mathsf{Node}\ t_1\ t_2) = (\texttt{flatten}\ t_1) +\!\!+ (\texttt{flatten}\ t_2) \end{aligned}$$

List types are nothing special in Haskell. Except for some syntactic sugar, they are on an equal footing with user-defined algebraic data types. In fact, seeing [] as a type constructor of the same kind as the polymorphic version of Tree above, lists can be thought of as being introduced with the following definition:

$$\textbf{data}\ []\ \alpha = (:)\ \alpha\ ([]\ \alpha) \mid [\,]$$

Another important abstraction facility is the use of higher-order types. That is, a function argument can itself be a function. For example, the following function applies another function, which is supplied as an argument, to every element of an input list and builds an output list from the results:[1]

$$\begin{aligned}&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]\\ &\texttt{map}\ h\ [\,] \qquad = [\,]\\ &\texttt{map}\ h\ (a : as) = (h\ a) : (\texttt{map}\ h\ as)\end{aligned}$$

Now *two* type variables, α and β, are used. They keep track of the dependencies between the argument and result types of h and the types of the input and output lists, respectively.

The combination of higher-order types with polymorphism often allows to capture certain computation schemes in a reusable fashion. For example, the functions `listsum` and `length`, and even `map`, follow the same regime of taking a list apart element by element and always recursing on the remainder. The essence of this is captured by the following function:

$$\begin{aligned}&\texttt{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta\\ &\texttt{foldr}\ c\ n\ [\,] \qquad = n\\ &\texttt{foldr}\ c\ n\ (a : as) = c\ a\ (\texttt{foldr}\ c\ n\ as)\end{aligned}$$

Indeed, using this function allows alternative definitions for the functions introduced earlier to be given as follows:

$$\begin{aligned}&\texttt{listsum} :: [\mathsf{Int}] \to \mathsf{Int}\\ &\texttt{listsum} = \texttt{foldr}\ (+)\ 0\\ \\ &\texttt{length} :: [\alpha] \to \mathsf{Int}\\ &\texttt{length} = \texttt{foldr}\ (\lambda a\ r \to 1 + r)\ 0\\ \\ &\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]\\ &\texttt{map}\ h = \texttt{foldr}\ (\lambda a\ r \to (h\ a) : r)\ [\,]\end{aligned}$$

and, as a further example:

$$\begin{aligned}&(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha]\\ &(+\!\!+) = \texttt{foldr}\ (\lambda a\ r \to (a :) \circ r)\ \texttt{id}\end{aligned}$$

[1] When reading such higher-order type signatures, the function arrow "→" associates to the right. So the type $(\alpha \to \beta) \to [\alpha] \to [\beta]$ is the same as $(\alpha \to \beta) \to ([\alpha] \to [\beta])$, but not the same as $\alpha \to \beta \to [\alpha] \to [\beta]$.

The latter uses the polymorphic identity function:

$$\begin{array}{l}\texttt{id} :: \alpha \to \alpha \\ \texttt{id}\ a = a\end{array}$$

and function composition:

$$\begin{array}{l}(\circ) :: (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \\ (f \circ g)\ a = f\ (g\ a)\end{array}$$

Since polymorphism, including forms of it that are more advanced than those already seen above, is at the heart of essentially all results reported on in this thesis, the next section discusses it in some more detail.

2.2 The Virtues of Polymorphism

We have already introduced so-called *parametric polymorphism*, where the same algorithm is used for all different instantiations of a type variable. For the function $\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$ seen earlier this allows, for example:

$$\begin{array}{lcll}\texttt{map}\ (+1)\ [1,2,3] & = & [2,3,4] & \text{— with } \alpha,\beta \mapsto \mathsf{Int},\mathsf{Int} \\ \texttt{map not}\ \ [\mathsf{True},\mathsf{False}] & = & [\mathsf{False},\mathsf{True}] & \text{— with } \alpha,\beta \mapsto \mathsf{Bool},\mathsf{Bool} \\ \texttt{map even}\ [1,2,3] & = & [\mathsf{False},\mathsf{True},\mathsf{False}] & \text{— with } \alpha,\beta \mapsto \mathsf{Int},\mathsf{Bool}\end{array}$$

The concrete choice of type parameters for α and β is not given explicitly in Haskell. Rather, it is inferred automatically (while, e.g., `map not` $[1,2,3]$ would be rejected).

So far, quantification over type variables has been implicit as well. For example, the type $(\alpha \to \beta) \to [\alpha] \to [\beta]$ is actually interpreted as $\forall\alpha.\forall\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$. The positioning and scope of quantifiers can be quite important. To see why, consider the following function definition:

$$\texttt{f}\ g = (g\ [1,2,3]) + (g\ [\mathsf{True},\mathsf{False}])$$

Note that in the equation's right-hand side the function g is applied to lists of two different types. But that should be fine if we apply `f`, for example, to `length` (in which case we would expect the overall result to be 5). So it might be tempting to give `f` the type $([\alpha] \to \mathsf{Int}) \to \mathsf{Int}$. But this would fail, as it would mean that we expect `f` to be a polymorphic function which for any concrete type, say τ, takes a function of type $([\tau] \to \mathsf{Int})$ as argument and delivers an integer as result. And this τ is neither guaranteed to be Int, nor Bool, and certainly not both at the same time. So the function calls $g\ [1,2,3]$ and $g\ [\mathsf{True},\mathsf{False}]$ are bound to lead to trouble. The point is that we do not really want `f` *itself* to be polymorphic, but rather want it to be a function that takes a polymorphic function as *argument*. That is, instead of $([\alpha] \to \mathsf{Int}) \to \mathsf{Int}$, which is equivalent to $\forall\alpha.([\alpha] \to \mathsf{Int}) \to \mathsf{Int}$, we need `f` to have the type $(\forall\alpha.[\alpha] \to \mathsf{Int}) \to \mathsf{Int}$. Such *rank-2 types* (Leivant 1983) are allowed in mature Haskell implementations, and are crucial for the new program manipulation

techniques to be presented in Chapters 5–7 and 9. It is important to note, though, that this additional abstraction facility, being able to write functions that abstract over functions that abstract over types, comes at the price of type signatures no longer being optional. In particular, the equation for `f` as given above in isolation is not a legal function definition. Only when we add the type signature[2]

$$\texttt{f} :: (\textbf{forall}\ \alpha.\ [\alpha] \to \mathsf{Int}) \to \mathsf{Int}$$

it is accepted by the type checker; and so is, then, `f length`, which computes 5.

Another form of polymorphism is the so-called *ad-hoc* one, where a certain functionality is provided for different types, without necessarily the same algorithm being used in each and every instantiation. For example, an equality test for lists of integers is likely to be implemented differently than the same kind of test for integers themselves, or for trees of Boolean values. In Haskell, such overloading of functionality is supported via *type classes* (Wadler and Blott 1989). For example, the class Eq with methods == and /= is declared as follows:

$$\begin{aligned}&\textbf{class}\ \mathsf{Eq}\ \alpha\ \textbf{where}\\ &\quad (==) :: \alpha \to \alpha \to \mathsf{Bool}\\ &\quad (/=)\ \ :: \alpha \to \alpha \to \mathsf{Bool}\end{aligned}$$

For base types like Int these methods are predefined, while for other types they could be defined as in the following example:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Eq}\ \alpha \Rightarrow \mathsf{Eq}\ [\alpha]\ \textbf{where}\\ &[\,]\ \ == [\,]\ \ = \mathsf{True}\\ &(x : xs) == (y : ys) = (x == y)\ \&\&\ (xs == ys)\\ &xs\ \ /=\ ys\ \ = \texttt{not}\ (xs == ys)\end{aligned}$$

Here an equality test for elements of an arbitrary, but fixed, type is used for defining an equality test for lists of elements of that type. Without further definitions, the methods == and /= are then available for [Int], [[Int]], and so on. And the same is true for functions defined in terms of them, such as the following one:

$$\begin{aligned}&\texttt{elem} :: \textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow \alpha \to [\alpha] \to \mathsf{Bool}\\ &\texttt{elem}\ x = \texttt{foldr}\ (\lambda a\ r \to (a == x)\ ||\ r)\ \mathsf{False}\end{aligned}$$

To even increase abstraction potential, type classes come in a higher-order variant as well. Using *type constructor classes* (Jones 1995b), some functionality can be provided over a range of polymorphic type constructors like [] and Tree. The prototypical example is the Functor class, which is declared as follows:

$$\begin{aligned}&\textbf{class}\ \mathsf{Functor}\ \phi\ \textbf{where}\\ &\quad \texttt{fmap} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ (\alpha \to \beta) \to \phi\ \alpha \to \phi\ \beta\end{aligned}$$

[2]Compiler flag `-XRank2Types` is used from now on.

Its intention is to generalize the function `map` we have already discussed. In fact, an instance definition of Functor for the list type constructor is straightforward:

$$\begin{array}{l}\textbf{instance}\ \mathsf{Functor}\ []\ \textbf{where}\\ \quad \mathtt{fmap} = \mathtt{map}\end{array}$$

And given further instance definitions, `fmap` becomes available at other types like $(\alpha \to \beta) \to \mathsf{Tree}\ \alpha \to \mathsf{Tree}\ \beta$. For all such instance definitions, one typically requires the following two laws to hold:

$$\mathtt{fmap\ id} = \mathtt{id} \tag{2.1}$$

$$(\mathtt{fmap}\ f) \circ (\mathtt{fmap}\ g) = \mathtt{fmap}\ (f \circ g) \tag{2.2}$$

Unfortunately, though, Haskell does not provide any means to enforce them on the language level. Instead, the responsibility lies with the programmer here.

An important use of the generality provided by ad-hoc polymorphism over type constructors is the abstract treatment of a range of algebraic data types via the approach of *two-level types*, also pioneered by Jones (1995a). The basic idea is to split recursive type definitions by separating flat surface structure generation from recursion, where the latter is expressed via an explicit fixpoint.

For example, the flat structure generation part of lists can be captured by the following definition:

$$\textbf{data}\ \mathsf{F}_{\mathsf{List}}\ \alpha\ \beta = \mathsf{Cons}\ \alpha\ \beta \mid \mathsf{Nil}$$

Together with

$$\textbf{newtype}\ \mathsf{Fix}\ \phi = \mathsf{Wrap}\ (\phi\ (\mathsf{Fix}\ \phi))$$

we then have, for every type τ, an isomorphism between the types $[\tau]$ and $\mathsf{Fix}\ (\mathsf{F}_{\mathsf{List}}\ \tau)$. For example,

$$\mathsf{Wrap}\ (\mathsf{Cons}\ 1\ (\mathsf{Wrap}\ (\mathsf{Cons}\ 2\ (\mathsf{Wrap}\ (\mathsf{Cons}\ 3\ (\mathsf{Wrap}\ \mathsf{Nil})))))) :: \mathsf{Fix}\ (\mathsf{F}_{\mathsf{List}}\ \mathsf{Int})$$

directly corresponds to the list $[1, 2, 3] :: [\mathsf{Int}]$, and the use of **newtype** rather than **data** for the type-level fixpoint above even ensures that the "extra" Wrap constructors incur no overhead at runtime. Similarly,

$$\textbf{data}\ \mathsf{F}_{\mathsf{Tree}}\ \alpha\ \beta = \mathsf{N}\ \beta\ \beta \mid \mathsf{L}\ \alpha$$

captures trees in the sense that, for example,

$$\mathsf{Wrap}\ (\mathsf{N}\ (\mathsf{Wrap}\ (\mathsf{N}\ (\mathsf{Wrap}\ (\mathsf{L}\ \text{'a'}))\ (\mathsf{Wrap}\ (\mathsf{L}\ \text{'b'}))))\ (\mathsf{Wrap}\ (\mathsf{L}\ \text{'c'}))) :: \mathsf{Fix}\ (\mathsf{F}_{\mathsf{Tree}}\ \mathsf{Char})$$

corresponds to $(\mathsf{Node}\ (\mathsf{Node}\ (\mathsf{Leaf}\ \text{'a'})\ (\mathsf{Leaf}\ \text{'b'}))\ (\mathsf{Leaf}\ \text{'c'})) :: \mathsf{Tree}\ \mathsf{Char}$.

The whole point of doing the above splitting is that we can now start to treat lists and trees, and like types, in a generic fashion. The leverage comes from defining

appropriate Functor instances, like so:

$$\begin{aligned}
&\textbf{instance}\ \mathsf{Functor}\ (\mathsf{F}_{\mathsf{List}}\ \alpha)\ \textbf{where}\\
&\quad \mathtt{fmap}\ h\ (\mathsf{Cons}\ a\ b) = \mathsf{Cons}\ a\ (h\ b)\\
&\quad \mathtt{fmap}\ h\ \mathsf{Nil}\ \ \ \ \ \ \ \ = \mathsf{Nil}\\
&\\
&\textbf{instance}\ \mathsf{Functor}\ (\mathsf{F}_{\mathsf{Tree}}\ \alpha)\ \textbf{where}\\
&\quad \mathtt{fmap}\ h\ (\mathsf{N}\ b_1\ b_2) = \mathsf{N}\ (h\ b_1)\ (h\ b_2)\\
&\quad \mathtt{fmap}\ h\ (\mathsf{L}\ a)\ \ \ \ \ = \mathsf{L}\ a
\end{aligned}$$

Then, functions abstracted over arbitrary functors, like

$$\begin{aligned}
&\mathtt{cata} :: \textbf{forall}\ \phi.\ \mathsf{Functor}\ \phi \Rightarrow (\textbf{forall}\ \beta.\ (\phi\ \beta \to \beta) \to \mathsf{Fix}\ \phi \to \beta)\\
&\mathtt{cata}\ \mathit{alg}\ (\mathsf{Wrap}\ t) = \mathit{alg}\ (\mathtt{fmap}\ (\mathtt{cata}\ \mathit{alg})\ t)
\end{aligned}$$

can be used generically. The function `cata` corresponds to `foldr` on lists and to a conceptually similar function on trees. In particular, `listsum` and `treesum` can now both be defined in terms of `cata`:

$$\begin{aligned}
&\mathtt{listsum} :: \mathsf{Fix}\ (\mathsf{F}_{\mathsf{List}}\ \mathsf{Int}) \to \mathsf{Int}\\
&\mathtt{listsum} = \mathtt{cata}\ \mathit{alg}\\
&\quad \textbf{where}\ \mathit{alg}\ \mathsf{Nil}\ \ \ \ \ \ \ \ \ = 0\\
&\qquad\qquad \mathit{alg}\ (\mathsf{Cons}\ n\ r) = n + r\\
&\\
&\mathtt{treesum} :: \mathsf{Fix}\ (\mathsf{F}_{\mathsf{Tree}}\ \mathsf{Int}) \to \mathsf{Int}\\
&\mathtt{treesum} = \mathtt{cata}\ \mathit{alg}\\
&\quad \textbf{where}\ \mathit{alg}\ (\mathsf{L}\ n)\ \ \ \ \ = n\\
&\qquad\qquad \mathit{alg}\ (\mathsf{N}\ r_1\ r_2) = r_1 + r_2
\end{aligned}$$

Such type-genericity plays an important role in Chapter 7.

2.3 Equational Reasoning

As seen, a Haskell program is just a collection of equations defining the return values of functions for given arguments. This is fundamentally different from the concept of functions or procedures in imperative or impure functional languages, where they may additionally access, and alter, some global state. A Haskell function is really a function in the mathematical sense, transferring values to values and doing nothing else. This absence of side-effects implies that every expression has a value that is independent of when it is evaluated. Clearly, two expressions having the same value can thus be replaced for each other in any program context without changing the overall semantics; a property often called *referential transparency*. And the easiest way to establish that two expressions have the same value is to observe them as the left- and right-hand sides of the same program equation. Of course, this might involve the instantiation of variables, on both sides and in exactly the same manner, that stand for abstracted parts of the function's input. Overall, this leads to a simple but powerful reasoning principle.

Since the above explanation is best substantiated by an example, we consider the following function definition:

$$\begin{aligned}
&\texttt{filter} :: \textbf{forall}\ \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
&\texttt{filter}\ p\ [\,] \qquad = [\,] \\
&\texttt{filter}\ p\ (a : as) = \textbf{if}\ p\ a\ \textbf{then}\ a : (\texttt{filter}\ p\ as) \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \texttt{filter}\ p\ as
\end{aligned}$$

Assume we want to prove that for every choice of p, h, and as (of appropriate types), the following law holds:

$$\texttt{filter}\ p\ (\texttt{map}\ h\ as) \;=\; \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ as) \tag{2.3}$$

Proceeding by induction on the list as, it suffices to establish that

$$\texttt{filter}\ p\ (\texttt{map}\ h\ [\,]) \;=\; \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ [\,]) \tag{2.4}$$

holds and that

$$\texttt{filter}\ p\ (\texttt{map}\ h\ (a : as)) \;=\; \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ (a : as)) \tag{2.5}$$

holds under the assumption that the induction hypothesis (2.3) holds for as. For the induction base (2.4), equational reasoning succeeds as follows:

$$\begin{aligned}
&\texttt{filter}\ p\ (\texttt{map}\ h\ [\,]) \\
=\ &\texttt{filter}\ p\ [\,] \\
=\ &[\,] \\
=\ &\texttt{map}\ h\ [\,] \\
=\ &\texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ [\,])
\end{aligned}$$

And for the induction step (2.3)→(2.5):

$$\begin{aligned}
&\texttt{filter}\ p\ (\texttt{map}\ h\ (a : as)) \\
=\ &\texttt{filter}\ p\ ((h\ a) : (\texttt{map}\ h\ as)) \\
=\ &\textbf{if}\ p\ (h\ a)\ \textbf{then}\ (h\ a) : (\texttt{filter}\ p\ (\texttt{map}\ h\ as)) \\
&\qquad\qquad \textbf{else}\ \ \texttt{filter}\ p\ (\texttt{map}\ h\ as) \\
=\ &\textbf{if}\ (p \circ h)\ a\ \textbf{then}\ (h\ a) : (\texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ as)) \\
&\qquad\qquad\ \ \textbf{else}\ \ \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ as) \\
=\ &\textbf{if}\ (p \circ h)\ a\ \textbf{then}\ \texttt{map}\ h\ (a : (\texttt{filter}\ (p \circ h)\ as)) \\
&\qquad\qquad\ \ \textbf{else}\ \ \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ as) \\
=\ &\texttt{map}\ h\ (\textbf{if}\ (p \circ h)\ a\ \textbf{then}\ a : (\texttt{filter}\ (p \circ h)\ as) \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \texttt{filter}\ (p \circ h)\ as) \\
=\ &\texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ (a : as))
\end{aligned}$$

While equational reasoning is employed as an auxiliary technique in most of the works included in this thesis, our main focus is on reasoning about functions *without* having access to their defining equations. How this is possible is the subject of the next chapter, along with some classical examples.

Chapter 3

Free Theorems and Program Transformations

We review why a polymorphic type may allow to derive statements about a function's behavior without knowing that function's defining equations. Then we consider an application of this approach in the area of program transformation.

3.1 Free Theorems

It is best to start with a concrete example. Consider the following type signature:

$$\texttt{f} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]$$

What does it tell us about the function `f`? For sure that it takes lists as input and produces lists as output. But we also see that `f` is polymorphic, and so must work for lists over arbitrary element types. How, then, can elements for the output list come into existence? The answer is that the output list can only ever contain elements from the input list. This is so because the function `f`, not knowing the element type of the lists it operates over, cannot possibly make up new elements of any concrete type to put into the output, such as 42 or `True`, or even `id`, because then `f` would immediately fail to have the general type **forall** $\alpha.\ [\alpha] \to [\alpha]$.

> So for any input list l (over any element type) the output list `f` l consists solely of elements from l.

But how can `f` decide which elements from l to propagate to the output list, and in which order and multiplicity? The answer is that such decisions can only be made based on the input list l. For `f` has no access to any global state or other context based on which to decide. It cannot, for example, consult the user in any way about what to do. The means by which to make decisions based on l are quite limited as well. In particular, decisions cannot possibly depend on any specifics of the elements of l. For the function `f` is ignorant of the element type, and so is prevented from analyzing list elements in any way (be it by pattern-matching, comparison

operations, or whatever). In fact, the only means for `f` to drive its decision-making is to inspect the *length* of l, because that is the only element-independent "information content" of a list.

> So for any pair of lists l and l' of same length (but possibly over different element types) the lists `f` l and `f` l' are formed by making the same position-wise selections of elements from l and l', respectively.

Now recall the function `map` from Section 2.1. Clearly, `map` h for any function h preserves the lengths of lists. So if $l' =$ `map` $h\ l$, then `f` l and `f` l' are of the same length and contain, at each position, position-wise exactly corresponding elements from l and l', respectively. Since, moreover, any two position-wise corresponding elements, one from l and one from $l' =$ `map` $h\ l$, are related by the latter being the h-image of the former, we have that at each position `f` l' contains the h-image of the element at the same position in `f` l.

> So for any list l and (type-appropriate) function h, the following law holds:

$$\texttt{f}\ (\texttt{map}\ h\ l) \quad = \quad \texttt{map}\ h\ (\texttt{f}\ l) \tag{3.1}$$

Note that during the reasoning leading up to this statement we did not (need to) consider the actual definition of `f` at all. It could have been `f` = `reverse`, or `f` = `tail`, or `f` = `take` 5, or many other choices. It just did not matter.

And this is not a one-off success. Intuitive reasoning of the same style as above can be applied to other polymorphic functions as well. For example, one can arrive at the conclusion that for every function

$$\texttt{f} :: \textbf{forall}\ \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

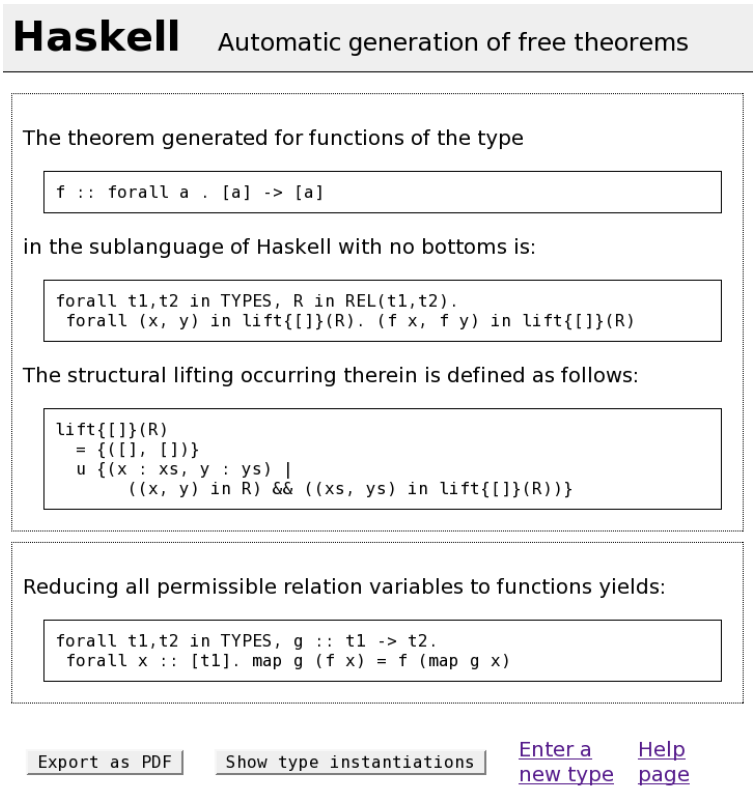
the following law holds:

$$\texttt{f}\ p\ (\texttt{map}\ h\ l) \quad = \quad \texttt{map}\ h\ (\texttt{f}\ (p \circ h)\ l) \tag{3.2}$$

The steps required to establish this are but minor extensions of the ones leading to law (3.1) above. It is only necessary to additionally factor in how `f`'s decision about which elements from an input list to propagate to the output list, and in which order and multiplicity, may now depend also on the outcomes of an input predicate, namely `f`'s first argument, on the input list's elements.

Note that law (3.2) is exactly the same as law (2.3) on page 12, except that now we claim it much more generally for all functions of `filter`'s type, not just for the particular one considered there. And there is no need for induction anymore. Better yet, the intuitive reasoning above can be put on a more formal basis. This is precisely what the methodology of deriving free theorems à la Wadler (1989) provides: a way to obtain statements like above for arbitrary function types, and in a more disciplined (and provably sound) manner than mere handwaving. We defer further discussion of this formal approach to Section 8.2. Here we only emphasize that

Wadler's approach is algorithmic enough that it can be implemented in an automatic tool. For concrete evidence, see the following screenshot of the online free theorems generator accessible at `http://linux.tcs.inf.tu-dresden.de/~voigt/ft`, after having been fed with the input "`f::[a]->[a]`":



Tool support also plays an important role in Chapter 9.

3.2 Program Transformations

Apart from obtaining algebraic laws in the style of (3.1) and (3.2) as useful ingredients for proofs by equational reasoning in general, the prime application area of free theorems has been the study of correctness of a certain class of program transformations. These "short cut fusion" techniques (Gill et al. 1993, Svenningsson 2002, Fernandes et al. 2007, and others) aim to eliminate intermediate data structures from programs written in a modular style. This is an important issue for the efficient execution of Haskell programs, given that the language strongly encourages solving an overall task by composing functions that solve subtasks.

In some cases free theorems directly lend themselves to the elimination of intermediate results. For example, the free theorem derived from the type signature

$$\texttt{length} :: \textbf{forall}\ \alpha.\ [\alpha] \to \mathsf{Int}$$

is that for every list l and (type-appropriate) function h,

$$\texttt{length}\ (\texttt{map}\ h\ l) \quad = \quad \texttt{length}\ l \tag{3.3}$$

Here it is quite obvious that a compiler which automatically replaced every instance of the left-hand side by the corresponding right-hand side would be doing the programmer a favor. But for other free theorems, like (3.1) and (3.2), the situation is less clear. Note that both

$$\texttt{f} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]$$

and

$$\texttt{f} :: \textbf{forall}\ \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

could be functions that either decrease or increase the length of the output list as compared to that of the input list; or sometimes decrease and sometimes increase the length, depending on the particular inputs. Hence, in both cases it is not certain whether it is efficiency-wise preferable to perform any `map` h "after" or "before" `f`, and thus, whether a compiler should be using the laws (3.1) and (3.2) as rewritings from left to right or conversely.

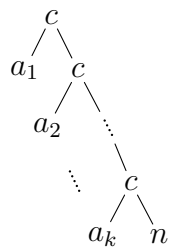
The seminal contribution of Gill et al. (1993) was to devise a transformation scheme based on free theorems that largely avoids such uncertainties (up to some very low-level implementation details). The drawback is that the functions to be dealt with are not anymore solely constrained by their types; some syntactic conditions are necessary as well. In particular, discussing only the case of intermediate *lists*, a successful fusion requires the consumer function to be defined in terms of `foldr`. Fortunately, where possible, this is good practice anyway. We have seen in Chapter 2 that `listsum`, `length`, `map`, $+\!\!+$, and `elem` can be defined so, and the same is true for `filter` and many other list-consuming functions. The producer of an intermediate list to be eliminated by fusion must be defined in terms of the following function:

$$\begin{aligned}&\texttt{build} :: \textbf{forall}\ \alpha.\ (\textbf{forall}\ \beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta) \to [\alpha]\\ &\texttt{build}\ g = g\ (:)\ [\,]\end{aligned}$$

Note that `build` has a rank-2 type as discussed in Section 2.2. To use it for producing a list of type $[\tau]$, for some concrete type τ, we must pass a polymorphic function g of type **forall** $\beta.\ (\tau \to \beta \to \beta) \to \beta \to \beta$. It is helpful to illustrate what such a g might do. Clearly, it will take two arguments, say c and n. Given these, it must return a value of the same type as n and as the return value and second argument of c. And since g must be polymorphic over that type, it is very limited in its ability to produce such a value. In fact, g can use only n and c for producing its return value. For example, it could simply return n. Or it could return $c\ a\ n$ for some $a :: \tau$, or $c\ a_1\ (c\ a_2\ n)$ for some $a_1, a_2 :: \tau$, and so on. But this is all. In fact, any g of the mentioned polymorphic type must be semantically equivalent to a function,

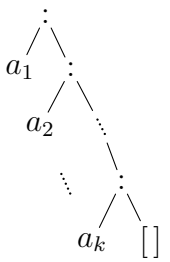
for some $k \geq 0$ and $a_1, \ldots, a_k :: \tau$, of the following form:

$$g \;=\; \lambda c\ n \rightarrow$$



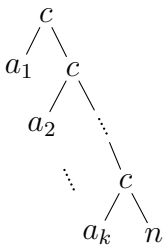
Of course, g need not be exactly of that syntactic form. But semantically, the above captures all possibilities as permitted by g's type. Now, applying `build` to g simply means to instantiate c and n to the list constructors, as follows:

$$\texttt{build}\ g \;=$$



On the other hand, the functionality of `foldr` as introduced on page 7 is to replace the list constructors in its third argument by its first and second arguments. Thus, we have:

$$\texttt{foldr}\ c\ n\ (\texttt{build}\ g) \;=$$



This suggests that the following law should hold:

$$\texttt{foldr}\ c\ n\ (\texttt{build}\ g) \;=\; g\ c\ n \tag{3.4}$$

And indeed, a free theorem can be used to formally prove this semantic equality for arbitrary c, n, and g that make the left-hand side well-typed.

Despite its simplicity, law (3.4) is very effective as an efficiency-improving program transformation. Of course, its wide applicability depends on expressing many list-producers in terms of `build`. But this is easy, and can even be automated (Chitil 1999). For example, `map` can be expressed as both a `foldr` and a `build` at the same time:

$$\texttt{map} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$$
$$\texttt{map}\ h\ as = \texttt{build}\ (\lambda c\ n \rightarrow \texttt{foldr}\ (\lambda a\ r \rightarrow c\ (h\ a)\ r)\ n\ as)$$

This enables fusion with `map` both as a consumer and as a producer. We illustrate only the latter here. Assume we want to write a function that takes a list of integers and computes the sum of their squares. An attractive, modular way of doing so is as follows:

$$\begin{aligned}&\texttt{squaresum} :: [\mathsf{Int}] \to \mathsf{Int}\\ &\texttt{squaresum}\ ns = \texttt{listsum}\ (\texttt{map}\ (\hat{}\ 2)\ ns)\end{aligned}$$

But this entails explicit creation of an intermediate list at the interface position between `map` and `listsum`. A more direct solution would be possible, but would require the programmer to forgo the modularity benefit. Fortunately, thanks to law (3.4), the compiler can find the monolithic version on its own. All it needs to do is to inline the definitions of `listsum` in terms of `foldr` and of `map` in terms of `build`, to use law (3.4) as a rewriting from left to right, and to perform some standard simplifications:

$$\begin{aligned}&\texttt{squaresum}\ ns\\ =\ &\texttt{listsum}\ (\texttt{map}\ (\hat{}\ 2)\ ns)\\ =\ &\texttt{foldr}\ (+)\ 0\ (\texttt{build}\ (\lambda c\ n \to \texttt{foldr}\ (\lambda a\ r \to c\ ((\hat{}\ 2)\ a)\ r)\ n\ ns))\\ =\ &(\lambda c\ n \to \texttt{foldr}\ (\lambda a\ r \to c\ ((\hat{}\ 2)\ a)\ r)\ n\ ns)\ (+)\ 0\\ =\ &\texttt{foldr}\ (\lambda a\ r \to (+)\ ((\hat{}\ 2)\ a)\ r)\ 0\ ns\\ =\ &\texttt{foldr}\ (\lambda a\ r \to (a \hat{}\ 2) + r)\ 0\ ns\end{aligned}$$

Note that the resulting definition works in a single pass over the input list, without creating an intermediate list. By similar steps we also obtain law (3.3) for the special case that `length` is not just any function of type **forall** α. $[\alpha] \to \mathsf{Int}$, but the particular one defined in terms of `foldr` on page 7.

Following the success of `foldr`/`build`-fusion, a whole range of program transformations based on free theorems have been developed over the years. For example, the approach has been transferred to other algebraic data types than lists (Takano and Meijer 1995, Johann 2002) and fusion rules that are dual to `foldr`/`build` in a category-theoretic sense have been considered (Takano and Meijer 1995, Svenningsson 2002). Examples of more recent work in the short cut fusion tradition are that of Coutts et al. (2007), Fernandes et al. (2007), and Ghani and Johann (2008). Our contribution reported on in Chapter 6 also builds on the short cut fusion methodology, while Section 8.4 and Chapter 9 focus on associated correctness issues. First, though, the next two chapters deal with other applications of free theorems.

Chapter 4

A Knuth-like 0-1-2-Principle for Parallel Prefix Computation

This chapter is based on the following paper:

- J. Voigtländer.
  Much Ado about Two: A Pearl on Parallel Prefix Computation.
  In P. Wadler, editor, *35th Symposium on Principles of Programming Languages, San Francisco, California, Proceedings*, volume 43(1) of *SIGPLAN Notices*, pages 29–35. ACM Press, 2008.
  (Voigtländer 2008b)

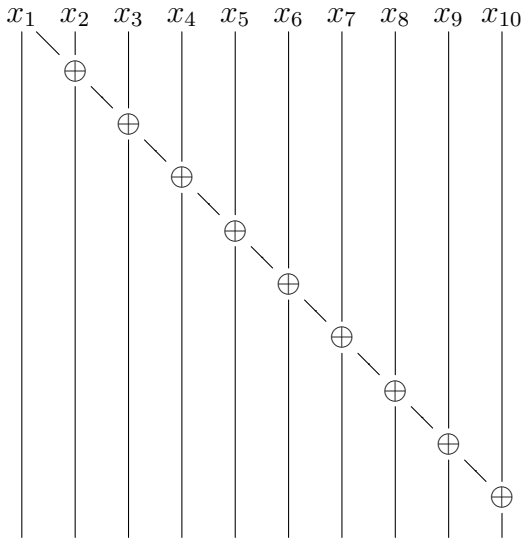
It presents an application of type-based reasoning to a real-world problem. In particular, we benefit from Haskell's mathematical rigor and its abstraction and reasoning facilities in the endeavor to analyze a whole class of algorithms.

4.1 Parallel Prefix Computation

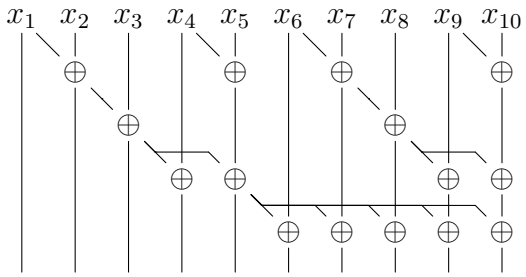
Parallel prefix computation is a task with numerous applications in the hardware and algorithmics fields (Blelloch 1993). The basic problem description is as follows:

> Given an associative binary operation $\oplus$ and inputs $x_1, \ldots, x_n$, compute the values x_1, $x_1 \oplus x_2$, $x_1 \oplus x_2 \oplus x_3$, ... up to $x_1 \oplus x_2 \oplus \cdots \oplus x_n$.

Here is an obvious solution for $n = 10$, depicted as a *prefix network* in which the inputs are provided at the top, values flow downwards along "wires" and get combined by $\oplus$-"gates", and the outputs can be read off at the bottom, from left to right:



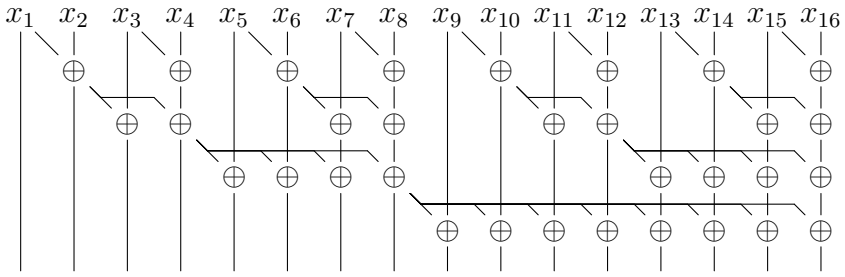
At first glance, this may appear to be the best solution possible, as it employs maximal reuse of partial results. After all, it is clear that nine applications of $\oplus$ are necessary to compute $x_1 \oplus x_2 \oplus \cdots \oplus x_{10}$ alone. So if the same nine applications yield all the other required outputs as well, what could be better? The point is that the number of applications of $\oplus$ is not the only measure of interest. For example, the above solution is inherently sequential, which leads to bad time performance. Assuming that each application of $\oplus$ requires one unit of time, the last output is not available until nine units have passed. In contrast, the following maximally parallel solution requires only four time units to deliver all outputs:



Note that thanks to the assumed associativity of $\oplus$, correctness is still guaranteed. For example, $x_1 \oplus x_2 \oplus \cdots \oplus x_7$ is now actually computed as $(((x_1 \oplus x_2) \oplus x_3) \oplus (x_4 \oplus x_5)) \oplus (x_6 \oplus x_7)$.

Admittedly, the shorter time to output in the parallel solution comes at the expense of an increased number of $\oplus$-"gates" and more complicated "wiring". But depending on the usage scenario this can be a worthwhile allowance. In some scenarios, in particular in a hardware setting where the "wires" are real wires and the $\oplus$-"gates" are real gates, many more trade-offs (guided by architectural, delay, or other constraints) are possible and of potential benefit. Hence, a wealth of solutions has been developed over the years (Sklansky 1960, Brent and Kung 1980, Ladner and Fischer 1980, Lin and Hsiao 2004, to mention just a few). Key to all of them is to use the associativity of $\oplus$ to rearrange how partial results are computed and combined.

An obvious concern is that for correctness of such new, and increasingly complex, methods. While checking the correctness of a concrete prefix network is a straightforward, though maybe tedious, task, the real practical interest is in validating a whole method of constructing prefix networks. For that is the general nature of work on parallel prefix computation: to develop and study algorithms that yield networks for arbitrary $n \geq 1$. In the case of the completely sequential network it should be clear how to abstract from $n = 10$ to arbitrary n. But also behind the other network shown above there is a general construction principle. It is the method of Sklansky (1960), and as another example here is its instance for $n = 16$:



So if studying prefix networks really means studying methods for their construction, how should these be expressed? Clearly, it would be beneficial to have a common framework in which to describe all methods, be they classical or still under development. For then they could be more readily compared, maybe combined, and hopefully analyzed using a common set of reasoning principles, as opposed to when each method is described in a different formalism or notation. One very attractive choice for the unifying framework is to use some universal programming language. After all, by Turing completeness, this would allow to precisely capture the notion of an *algorithm* that may, or may not, be a correct solution to the parallel prefix computation task. Of course, this begs the question in terms of *which* programming language to cast the problem, algorithms, and analyses. It turns out that Haskell, with its mathematical expressivity and nice abstraction facilities, is a very good fit.

4.2 Prefix Networks in Haskell

From the problem description at the beginning of the previous section it is clear that any function implementing parallel prefix computation should have the following type:

$$\textbf{forall}\ \alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

The polymorphism over α is justified by the fact that in the problem description neither the type of the inputs $x_1, \ldots, x_n$, nor any specifics (apart from associativity) of $\oplus$ are fixed, except (implicitly) that the type on which $\oplus$ operates should be the same as that of the inputs. By providing the inputs in a variable-length list, we express our interest in algorithms that work for arbitrary n. And indeed, the prefix networks seen in the previous section are easily generated in Haskell.

For example, the completely sequential construction method is captured as follows:

$$
\begin{array}{l}
\texttt{serial} :: \textbf{forall}\ \alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
\texttt{serial}\ op\ (x : xs) = go\ x\ xs \\
\quad \textbf{where}\ go\ x\ [\,] \qquad = [x] \\
\qquad\qquad go\ x\ (y : ys) = x : (go\ (x\ `op`\ y)\ ys)
\end{array}
$$

To see how this code corresponds to the first network on page 20, consider the following evaluation:

$$
\begin{array}{ll}
 & \texttt{serial}\ (+)\ [1..10] \\
= & go\ 1\ [2..10] \\
= & 1 : (go\ (1+2)\ [3..10]) \\
= & 1 : (3 : (go\ (3+3)\ [4..10])) \\
= & 1 : (3 : (6 : (go\ (6+4)\ [5..10]))) \\
= & 1 : (3 : (6 : (10 : (go\ (10+5)\ [6..10])))) \\
= & 1 : (3 : (6 : (10 : (15 : (go\ (15+6)\ [7..10]))))) \\
= & 1 : (3 : (6 : (10 : (15 : (21 : (go\ (21+7)\ [8..10])))))) \\
= & 1 : (3 : (6 : (10 : (15 : (21 : (28 : (go\ (28+8)\ [9..10]))))))) \\
= & 1 : (3 : (6 : (10 : (15 : (21 : (28 : (36 : (go\ (36+9)\ [10])))))))) \\
= & 1 : (3 : (6 : (10 : (15 : (21 : (28 : (36 : (45 : (go\ (45+10)\ [\,])))))))))) \\
= & 1 : (3 : (6 : (10 : (15 : (21 : (28 : (36 : (45 : [55]))))))))) \\
= & [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
\end{array}
$$

The method of Sklansky (1960) is captured as follows:

$$
\begin{array}{l}
\texttt{sklansky} :: \textbf{forall}\ \alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
\texttt{sklansky}\ op\ [x] = [x] \\
\texttt{sklansky}\ op\ xs\ = us +\!\!+ vs \\
\quad \textbf{where}\ t \qquad\quad = ((\texttt{length}\ xs) + 1)\ `\texttt{div}`\ 2 \\
\qquad\qquad (ys, zs) = \texttt{splitAt}\ t\ xs \\
\qquad\qquad us \qquad = \texttt{sklansky}\ op\ ys \\
\qquad\qquad u \qquad\quad = \texttt{last}\ us \\
\qquad\qquad vs \qquad = \texttt{map}\ (u\ `op`)\ (\texttt{sklansky}\ op\ zs)
\end{array}
$$

This is already a bit more complicated than `serial`, but still expressed in a way that is nicely declarative and accessible. Confidence that this code really implements Sklansky's method can also be gained from the fact that the two parallel network pictures shown in the previous section, for $n = 10$ and $n = 16$, were automatically generated from it.[1] And more recent algorithms for parallel prefix computation can be treated in the same way.

What is only hinted at above actually extends to a whole methodology for designing, and then analyzing, hardware circuits using functional languages. An interesting introductory text containing many references is that of Sheeran (2005). Around the

[1] In the same way, namely using a separate Haskell program, the completely sequential network picture for $n = 10$ was automatically generated from the function `serial` above.

idea of specifying circuits on a very high and abstract level, and then "compiling" right down to hardware, a community is forming that attracts players from both academia and industry.[2] Our contribution reported here is a display of what powerful abstractions can buy in this context. Its reasoning is specific to parallel prefix computation, but similar results may hold for other algorithm classes of interest.

4.3 A Knuth-like 0-1-2-Principle

Assume we have a candidate function of type **forall** α. $(\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$. This function could be our attempt at implementing a classical method from the literature. Or it could be a new algorithm we have come up with or obtained by refining or combining existing ones. Indeed, it could be a function that we only hope to correctly implement parallel prefix computation, while actually it does not. To assure ourselves of its correctness, we may try an explicit proof or at least perform systematic testing. But it seems that in order to do so, we would have to consider every concrete type τ as potential instantiation for α, and for each such τ consider every (associative) operation of type $\tau \to \tau \to \tau$ as well as every input list of type $[\tau]$. Not only would this mean a lot of work, it is also unsatisfactory on a conceptual level. After all, given the rather generic problem description, we could expect that analyses of solution candidates are possible in a sufficiently generic way as well.

Here the *0-1-Principle* of Knuth (1973) comes to mind. It states that if an oblivious sorting algorithm, that is one where the sequence of comparisons performed is the same for all input sequences of any given length, is correct on Boolean valued input sequences, then it is correct on input sequences over any totally ordered value set. This greatly eases the analysis of such algorithms. Is something similar possible for parallel prefix computation? For 0-1 the answer is negative: one can give a function that is correct for all binary operations and input lists over Boolean values, but not in general. The next best thing to hope for then is that a three-valued type may suffice as a discriminator between good and bad candidate functions. And this is indeed the case.

Our 0-1-2-Principle for parallel prefix computation can be formulated as follows. Let a function

$$\texttt{candidate} :: \textbf{forall}\ \alpha.\ (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

be given and let

$$\textbf{data}\ \mathsf{Three} = \mathsf{Zero} \mid \mathsf{One} \mid \mathsf{Two}$$

If for every associative operation $(\oplus) :: \mathsf{Three} \to \mathsf{Three} \to \mathsf{Three}$ and every list $xs :: [\mathsf{Three}]$,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{serial}\ (\oplus)\ xs$$

[2] See, for example, the "Hardware Design using Functional Languages" workshop series (`http://www.hflworkshop.org`).

then the same holds for every type τ, associative $(\oplus) :: \tau \to \tau \to \tau$, and $xs :: [\tau]$. That is, correctness of `candidate` at the type Three implies its correctness at arbitrary type. Here the definition of "correctness" is "semantic equivalence to `serial` for associative operations as first input". Actually, the formal account (Voigtländer 2008b) uses a different reference implementation than `serial`, but one that is easily shown to be semantically equivalent to it by equational reasoning.

The only aspect of the overall proof to which we want to draw attention here is the role of type-based reasoning. Note that we have not put any restriction on the actual definition of `candidate`, just on its type. This is, of course, a case for working with a free theorem. The free theorem derived from `candidate`'s type is that for every choice of concrete types τ_1 and τ_2, a function $h :: \tau_1 \to \tau_2$, and operations $(\otimes) :: \tau_1 \to \tau_1 \to \tau_1$ and $(\oplus) :: \tau_2 \to \tau_2 \to \tau_2$, if for every $x, y :: \tau_1$,

$$h\ (x \otimes y) \quad = \quad (h\ x) \oplus (h\ y) \tag{4.1}$$

then for every $xs :: [\tau_1]$,

$$\texttt{map}\ h\ (\texttt{candidate}\ (\otimes)\ xs) \quad = \quad \texttt{candidate}\ (\oplus)\ (\texttt{map}\ h\ xs)$$

This free theorem's conclusion gives us a starting point for relating the behavior of `candidate` at different types, as ultimately required for the 0-1-2-Principle. Unfortunately, it is not as easy as setting $\tau_1 =$ Three and $\tau_2 = \tau$ and working from there. Instead, we found it necessary to use an indirection via the type of integer lists (and an auxiliary statement originally discovered by M. Sheeran). Also, some good choices for h, $\otimes$, and $\oplus$ must be made, associativity must be factored into establishing the precondition (4.1), and some properties of permutations are needed. But all in all, once we have the above free theorem, the proof is mainly a bunch of equational reasoning steps. It has additionally been machine-verified using the Isabelle interactive proof assistant (Böhme 2007).

Chapter 5

Semantic Bidirectionalization

This chapter is based on the following paper:

- J. Voigtländer.
  Bidirectionalization for Free!
  In B.C. Pierce, editor, *36th Symposium on Principles of Programming Languages, Savannah, Georgia, Proceedings.* ACM Press, 2009.
  (Voigtländer 2009)

It presents a novel approach to the view-update problem known from the database area, utilizing programming language theory surrounding polymorphic types.

5.1 Bidirectional Transformation

Assume we have a domain of concrete values and a function `get` that takes such a value as source and produces from it a view by abstracting from some details. Now assume this view is updated in some way, and we would like to propagate this change back to the input source. So we need another function `put` that takes the original source and an updated view and produces an updated source. Clearly, `get` and `put` should be suitably related, because otherwise the integrity of the data to be transformed by using them is threatened. In the database area, where the concrete and abstract domains will typically be relation tables or XML trees, the following conditions have been proposed (Bancilhon and Spyratos 1981):

$$\texttt{put}\ s\ (\texttt{get}\ s) = s \tag{5.1}$$

$$\texttt{get}\ (\texttt{put}\ s\ v) = v \tag{5.2}$$

$$\texttt{put}\ (\texttt{put}\ s\ v)\ (\texttt{get}\ s) = s \tag{5.3}$$

$$\texttt{put}\ (\texttt{put}\ s\ v)\ v' = \texttt{put}\ s\ v' \tag{5.4}$$

known as acceptability, consistency, undoability, and composability.

Writing and maintaining good `get`/`put`-pairs requires considerable effort. So it is natural to invest in methodologies that can reduce this burden on the programmer.

The ideal is to not have to write two separate specifications and to establish their relatedness by proving (some of) the conditions above, but to instead be able to provide only a single specification and still get both forward/backward-components. This problem has received much attention from the programming language community in recent years. For example, Foster et al. (2007) pioneered a domain-specific language approach that fences in a certain subclass of transformations, provides a supply of correctly behaving `get`/`put`-pairs on a low level, and then describes systematic and sound ways of assembling bigger bidirectional transformations from smaller ones. Another approach is to devise an algorithm that works on a syntactic representation of somehow restricted `get`-functions and tries to infer appropriate `put`-functions automatically (Matsuda et al. 2007). While *all* the approaches proposed in the literature so far have been syntactic in nature, we present one that works purely on the level of semantic values.

5.2 Bidirectionalization of Polymorphic `get`

The idea is to write, directly in the language in which the forward and backward functions shall live themselves, a higher-order function that takes `get`-functions as arguments and returns appropriate `put`-functions. It turns out that Haskell is very well up to the task. One thing to stress is that "on the semantic level" means that when prescribing how `put` will behave we are not willing, or even able, to inspect the function definition of `get`. That is, the backward component we return cannot be based on a deep analysis of the forward function's innards. This may sound crippling, and yet we can provide nontrivial, and well-behaved, `put`-functions for a wide range of (polymorphic) `get`-functions. And forgoing any possibility to "look into" `get` liberates our approach from considerable syntactic restraints. In particular, and in contrast to the situation with all previous approaches, the programmer is not anymore restricted to drawing forward functions from some sublanguage only.

Let us consider a specific example in Haskell, for simplicity working with lists only rather than with richer data structures like tables or trees. Assume our `get`-function is as follows:

```
get :: forall α. [α] → [α]
get as = take ((length as) `div` 2) as
```

Here the abstraction amounts to omitting the input list's second half. Propagating an update on the preserved first half back to the original, full list can be done with the following function:

```
put :: forall α. [α] → [α] → [α]
put as as' = let n = (length as) `div` 2
             in  if (length as') == n then as' ++ (drop n as)
                                      else  error "Shape mismatch."
```

And indeed, our higher-order function `bff` (named for an abbreviation of the full

paper's title), when applied to the above `get`, will return this `put`.[1] Of course not the exact syntactic definition of `put` that is shown above, but a functional value that is semantically equivalent to it. This is absolutely enough from an application perspective. We want automatic bidirectionalization precisely because we do not want to be bothered with thinking about the backward function. So we do not care about its syntactic form, as long as the function serves its purpose. There is a certain price to pay, namely `bff get` runs much less efficiently on its inputs than the hand-coded `put` does, in this and in other examples. But this is a different story. Here we are interested in safety and programmer (rather than program) productivity.

One aspect to be aware of is that the `put`-function given above is a partial function only. That is, it may raise an exception for unsuitable input that represents a view-update that cannot (automatically and consistently) be reconciled with the original source. Some in the related literature, notably Foster et al. (2007) and follow-on works, emphasize the static description, or even calculation, of the domain on which a `put`-function is totally defined. We instead follow Matsuda et al. (2007), accept partiality, and weaken the bidirectional properties (5.2)–(5.4) somewhat by adding definedness preconditions. Specifically, these three properties are only required to hold if `put` $s\ v$ is actually defined, and (5.4) even has the additional precondition that `put` (`put` $s\ v$) v' is defined as well. The thus revised conditions, and the original (5.1), are what we prove for polymorphic `get` and `put` = `bff get`.[2] The way we do this crucially depends on `get` being of polymorphic type, because this allows us to learn something about its behavior without having access to its defining equations.

5.3 Leveraging Free Theorems

We do not want to repeat the full development and implementation of `bff` or the associated proofs here, but at least explain some of the key ideas.

Assume that `bff` is given a function `get` :: **forall** α. $[\alpha] \to [\alpha]$ as input. How can it gain information about this function, so as to exploit that information for producing a good backward function? Note that `get` is of exactly the type discussed as first example in Section 3.1. There, we have analyzed what this type tells us about the behavior of any such function. The essence of this analysis was that such a function's behavior does not depend on any concrete list elements, but only on positional information. Now we additionally use that this positional information can even be observed explicitly, for example by applying `get` to ascending lists over integer values. Say `get` is `tail`, then every list $[0..n]$ is mapped to $[1..n]$, which allows `bff` to see that the head element of the original source is absent from the view, hence cannot be affected by an update on the view, and hence should remain unchanged when propagating an updated view back into the source. And this observation can be transferred to other source lists than $[0..n]$ just as well, even to lists over non-

[1] Well, almost. Actually, it will return this function with type **forall** α. Eq $\alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha]$.

[2] Again, almost. In general, we prove the conditions up to == rather than up to semantic equivalence. But for the typical instances of Eq used in practice, == and = totally agree.

integer types, thanks to law (3.1) from page 14. In particular, that law allows us to establish that for every list s of the same length as $[0..n]$, but over arbitrary type, we have

$$\texttt{get}\ s \quad = \quad \texttt{map}\ (s\,!!)\ (\texttt{get}\ [0..n]) \tag{5.5}$$

where (!!) :: **forall** α. $[\alpha] \to \mathsf{Int} \to \alpha$ is the operator used in Haskell for extracting a list element at a given index position, starting counting from 0.

Let us develop the above line of reasoning further, again on the `tail` example. So `bff tail` is supposed to return a good `put`. To do so, it must determine what this `put` should do when given an original source s and an updated view v. First, it would be good to find out to what element in s each element in v corresponds. Assume s has length $n + 1$. Then by applying `tail` to the same-length list $[0..n]$, `bff` (or, rather, `bff tail` = `put`) learns that the original view from which v was obtained by updating had length n, and also to what element in s each element in that original view corresponded. Being conservative, we will only accept v if it has retained that length n. For then, we also know directly the associations between elements in v and positions in the original source. Now, to produce the updated source, we can go over all positions in $[0..n]$ and fill them with the associated values from v. For positions for which there is no corresponding value in v, because these positions were omitted when applying `tail` to $[0..n]$, we can look up the correct value in s rather than in v. For the `tail` example, this will only concern position 0, for which we naturally take over the head element from s.

The same strategy works also for general `bff get`. In short, given s, produce a kind of template $s' = [0..n]$ of the same length, together with an association g between integer values in that template and the corresponding values in s. Then apply `get` to s' and produce a further association h by matching this template view versus the updated proper value view v. Combine the two associations into a single one h', giving precedence to h whenever an integer template index is found in both h and g. Thus, it is guaranteed that we will only resort to values from the original source s when the corresponding position did not make it into the view, and thus there is no way how it could have been affected by the update. Finally, produce an updated source by filling all positions in $[0..n]$ with their associated values according to h'. Some extra care is needed when matching the template view versus the updated proper value view, to produce h, for the case that an index position is encountered twice. This case occurs as soon as `get` duplicates a list element. Consider, for example, `get` $= (\lambda s \to s +\!\!+ s)$. Applied to a template $[0..n]$, it will deliver the template view $[0, \ldots, n, 0, \ldots, n]$. Under what conditions should a match between this template view and an updated proper value view be considered successful? Clearly only when equal indices match up with equal values, because only then we can produce a meaningful association reflecting a legal update.

Using the standard functions

$$\texttt{zip} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ [\alpha] \to [\beta] \to [(\alpha, \beta)]$$

and

$$\texttt{lookup} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ \mathsf{Eq}\ \alpha \Rightarrow \alpha \to [(\alpha, \beta)] \to \mathsf{Maybe}\ \beta$$

with

$$\textbf{data}\ \mathsf{Maybe}\ \beta = \mathsf{Nothing} \mid \mathsf{Just}\ \beta$$

and the obvious semantics, the strategy described above could be implemented as follows:

```
bff :: (forall α. [α] → [α]) → (forall α. Eq α ⇒ [α] → [α] → [α])
bff get [] [] = []
bff get [] v  = error "Shape mismatch."
bff get s  v  = let s' = [0..((length s) − 1)]
                    g  = zip s' s
                    h  = assoc (get s') v
                    h' = h ++ g
                in  map (λi → case lookup i h' of Just b → b) s'

assoc :: Eq α ⇒ [Int] → [α] → [(Int, α)]
assoc []       []       = []
assoc (i : is) (b : bs) = let m = assoc is bs
                          in  case lookup i m of
                                Nothing → (i, b) : m
                                Just c  → if b == c
                                          then m
                                          else error "Update violates equality."
assoc _        _        = error "Shape mismatch."
```

Note that the first two defining equations for `bff` reflect the fact that a function `get` :: **forall** α. $[\alpha] \to [\alpha]$ can map $[\,]$ only to $[\,]$, so only an empty list is accepted as updated view for an empty source. The **case** and **if** in the second defining equation for `assoc` provide for the correct treatment of duplication of list elements, by checking whether indeed equal indices match up with equal values.

The implementation given above is clearly not optimal. It makes a rather bad choice for representing the associations between integer values and values from s and v. Above, lists of pairs are used for this, namely $[(\mathsf{Int}, \alpha)]$, and `lookup` is just linear search. The full paper (Voigtländer 2009) actually uses the standard library Data.IntMap instead, with better asymptotic behavior. The implementation in the paper also differs in other, smaller ways from the one above, such as by a more refined error handling, but the key ideas are the same.

More importantly, the paper then goes on to develop semantic bidirectionalization for other functions than ones of type **forall** α. $[\alpha] \to [\alpha]$. One dimension of generalization is to consider functions that are not fully polymorphic, but may actually perform *some* operations on list elements. For example, the following function uses equality, or rather inequality, tests to remove duplicate occurrences of list

elements:

$$\begin{aligned}&\texttt{get} :: \textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]\\ &\texttt{get}\ [\,] \qquad = [\,]\\ &\texttt{get}\ (a : as) = a : (\texttt{get}\ (\texttt{filter}\ (a\ /{=})\ as))\end{aligned}$$

It is not in the reach of the bidirectionalization strategy described thus far. It cannot be given the type **forall** α. $[\alpha] \to [\alpha]$, and indeed the essential law (5.5) does not hold for it.[3] But by working with refined free theorems (Wadler 1989, Section 3.4) it is possible to treat `get`-functions of type **forall** α. Eq $\alpha \Rightarrow [\alpha] \to [\alpha]$ as well, to implement a higher-order function

$$\texttt{bff}_{\mathsf{Eq}} :: (\textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \to (\textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])$$

and to prove that every pair `get` :: **forall** α. Eq $\alpha \Rightarrow [\alpha] \to [\alpha]$ and `put` = $\texttt{bff}_{\mathsf{Eq}}$ `get` satisfies the laws (5.1)–(5.4), in their revised form discussed at the end of Section 5.2. The same goes for the type class Ord capturing ordering tests, a new higher-order function

$$\texttt{bff}_{\mathsf{Ord}} :: (\textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \to (\textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])$$

and `get`-functions like the following one:

$$\begin{aligned}&\texttt{get} :: \textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]\\ &\texttt{get} = (\texttt{take}\ 3) \circ \mathsf{List}.\texttt{sort}\end{aligned}$$

For each of `bff`, $\texttt{bff}_{\mathsf{Eq}}$, and $\texttt{bff}_{\mathsf{Ord}}$, the full paper actually only discusses the proofs for conditions (5.1) and (5.2), but those for (5.3) and (5.4) are similar.

Another dimension of generalization is to consider functions that deal with data structures other than lists. By employing polymorphism over type constructor classes, like Functor in Section 2.2, and type-generic programming techniques, we provide one implementation of each `bff`, $\texttt{bff}_{\mathsf{Eq}}$, and $\texttt{bff}_{\mathsf{Ord}}$ that applies to functions involving a wide range of type constructors, on both the source and the view sides. For example, the very same `bff` can be used to bidirectionalize the `get`-function shown in Section 5.2 as well as `flatten` from Section 2.1.

An online interface to the implementations from the full paper is accessible at `http://linux.tcs.inf.tu-dresden.de/~bff/cgi-bin/bff.cgi`.

[3] Consider s = "abcbabcbaccba" and $n = 12$. Then on the one hand, `get` s = "abc", but on the other hand, `map` $(s\,!!)$ (`get` $[0..n]$) = `map` $(s\,!!)$ $[0..n] = s$.

Chapter 6

List Operations Vanish for Free

This chapter is based on the following paper:

- J. Voigtländer.
  Concatenate, Reverse and Map Vanish For Free.
  In S.L. Peyton Jones, editor, *7th International Conference on Functional Programming, Pittsburgh, Pennsylvania, Proceedings*, volume 37(9) of *SIGPLAN Notices*, pages 14–25. ACM Press, 2002.
  (Voigtländer 2002)

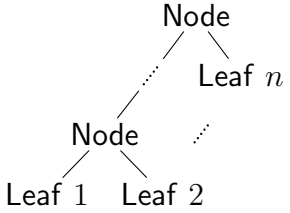
It introduces a type-based, and lightweight, approach to program transformation that can achieve asymptotic speed-ups by exploiting algebraic laws about list concatenation and other operations.

6.1 The Problem of Repeated Concatenation

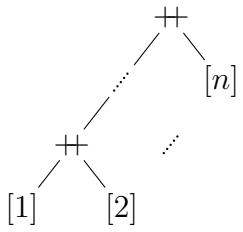
Recall the function

$$\begin{aligned}
&\texttt{flatten} :: \textbf{forall}\ \alpha.\ \mathsf{Tree}\ \alpha \to [\alpha] \\
&\texttt{flatten}\ (\mathsf{Leaf}\ a) \quad\ = [a] \\
&\texttt{flatten}\ (\mathsf{Node}\ t_1\ t_2) = (\texttt{flatten}\ t_1) +\!\!+ (\texttt{flatten}\ t_2)
\end{aligned}$$

from Section 2.1. While serving as a good specification of flattening a tree into a list, this definition has an inefficient runtime behavior due to repeated concatenate operations on intermediate lists. This is particularly evident on left-leaning trees. Consider, for example, trees of the following form:



On these, `flatten` will lead to computation tasks of the following form:



But since applications of $+\!\!+$ (both the version on page 7 and the one from the standard Haskell Prelude) require runtime linear in the lengths of their left argument lists, this will lead to an overall runtime quadratic in n.

One solution is to introduce an *accumulating parameter* as follows:

$$\begin{aligned}
&\texttt{flatten}_1 :: \textbf{forall}\ \alpha.\ \mathsf{Tree}\ \alpha \to [\alpha] \\
&\texttt{flatten}_1\ t = \texttt{flatten}'_1\ t\ [\,] \\
&\\
&\texttt{flatten}'_1 :: \textbf{forall}\ \alpha.\ \mathsf{Tree}\ \alpha \to [\alpha] \to [\alpha] \\
&\texttt{flatten}'_1\ (\mathsf{Leaf}\ a) \qquad as = a : as \\
&\texttt{flatten}'_1\ (\mathsf{Node}\ t_1\ t_2)\ as = \texttt{flatten}'_1\ t_1\ (\texttt{flatten}'_1\ t_2\ as)
\end{aligned}$$

While Wadler (1987; revised, 1989) showed how this transformation can be performed automatically, it very much depends on the syntax of the original function definition and thus is brittle to even small changes in the program. Moreover, there are situations where left-associatively nested concatenations lead to quadratic overhead but accumulating parameters are of no help at all.

A more powerful solution is to switch to an alternative representation of lists as functions, by abstraction over the list end, often called *difference lists*. In the formulation of Hughes (1986), but encapsulated as an explicitly new data type:

$$\begin{aligned}
&\textbf{newtype}\ \mathsf{DList}\ \alpha = \mathsf{DL}\ ([\alpha] \to [\alpha]) \\
&\\
&\texttt{rep} :: \textbf{forall}\ \alpha.\ [\alpha] \to \mathsf{DList}\ \alpha \\
&\texttt{rep}\ as = \mathsf{DL}\ (as\ +\!\!+) \\
&\\
&\texttt{abs} :: \textbf{forall}\ \alpha.\ \mathsf{DList}\ \alpha \to [\alpha] \\
&\texttt{abs}\ (\mathsf{DL}\ f) = f\ [\,] \\
&\\
&\texttt{nilR} :: \textbf{forall}\ \alpha.\ \mathsf{DList}\ \alpha \\
&\texttt{nilR} = \mathsf{DL}\ \texttt{id} \\
&\\
&\texttt{consR} :: \textbf{forall}\ \alpha.\ \alpha \to \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \\
&\texttt{consR}\ a\ (\mathsf{DL}\ f) = \mathsf{DL}\ ((a :) \circ f) \\
&\\
&\texttt{appendR} :: \textbf{forall}\ \alpha.\ \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \to \mathsf{DList}\ \alpha \\
&\texttt{appendR}\ (\mathsf{DL}\ f)\ (\mathsf{DL}\ g) = \mathsf{DL}\ (f \circ g)
\end{aligned}$$

Then, flattening a tree into a list in the new representation can be done using the following function:

$$\begin{aligned}&\texttt{flatten}'_2 :: \textbf{forall}\ \alpha.\ \mathsf{Tree}\ \alpha \to \mathsf{DList}\ \alpha\\&\texttt{flatten}'_2\ (\mathsf{Leaf}\ a) \quad\ = \texttt{consR}\ a\ \texttt{nilR}\\&\texttt{flatten}'_2\ (\mathsf{Node}\ t_1\ t_2) = (\texttt{flatten}'_2\ t_1)\ \texttt{`appendR`}\ (\texttt{flatten}'_2\ t_2)\end{aligned}$$

and a more efficient variant of the original function, with its original type, can be recovered as follows:

$$\begin{aligned}&\texttt{flatten}_2 :: \textbf{forall}\ \alpha.\ \mathsf{Tree}\ \alpha \to [\alpha]\\&\texttt{flatten}_2 = \texttt{abs} \circ \texttt{flatten}'_2\end{aligned}$$

But how do we know that the new $\texttt{flatten}_2$ is equivalent to the original `flatten`? We could try to argue by "distributing" `abs` over the definition of $\texttt{flatten}'_2$, using $\texttt{abs}\ \texttt{nilR} = [\,]$, $\texttt{abs}\ (\texttt{consR}\ a\ as) = a : (\texttt{abs}\ as)$, and

$$\texttt{abs}\ (\texttt{appendR}\ as\ bs) \quad = \quad (\texttt{abs}\ as) +\!\!+ (\texttt{abs}\ bs) \tag{6.1}$$

But actually the last equation does not hold in general. The reason is that there are $as :: \mathsf{DList}\ \tau$, for some concrete type τ, that are not in the image of `rep`. Consider, for example, $as = \mathsf{DL}\ \texttt{reverse}$. Then neither is $as = \texttt{rep}\ as'$ for any $as' :: [\tau]$, nor does law (6.1) hold for every bs. Any argument "by distributing `abs`" would thus have to rely on the implicit assumption that a certain discipline has been exercised when going from the original `flatten` to $\texttt{flatten}'_2$ by replacing $[\,]$, $(:)$, and $(+\!\!+)$ by `nilR`, `consR`, and `appendR` (and/or applying `rep` to explicit lists). But this implicit assumption is not immediately in reach for formal grasp. So it would be nice to be able to provide a single, conclusive correctness statement for transformations like the one above.

6.2 Our Solution

By making use of Haskell's abstraction facilities at the level of the type system, we can provide a way of writing `flatten` (and functions with similar problems) in a form closely resembling the original specification, but avoiding quadratic overhead caused by repeated concatenations, and at the same time being assured of semantic correctness. The key ingredient is a function of the following type:

$$\texttt{vanish}_{+\!+} :: \textbf{forall}\ \alpha.\ (\textbf{forall}\ \beta.\ \beta \to (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \beta) \to [\alpha]$$

Note the rank-2 polymorphism here, due to the inner **forall**. As far as the programmer is concerned, $\texttt{vanish}_{+\!+}$ has the following defining equation:

$$\texttt{vanish}_{+\!+}\ g \quad = \quad g\ [\,]\ (:)\ (+\!\!+) \tag{6.2}$$

Note the similarity of both the type and the definition to those of `build` on page 16. With the knowledge of (6.2) it is straightforward to rewrite `flatten` into the following equivalent form:

$$
\begin{aligned}
&\texttt{flatten}_3 :: \textbf{forall}\ \alpha.\ \mathsf{Tree}\ \alpha \to [\alpha] \\
&\texttt{flatten}_3\ t = \texttt{vanish}_{+\!+} \\
&\quad (\lambda \mathit{nil}\ \mathit{cons}\ \mathit{append} \to \textbf{let}\ f\ (\mathsf{Leaf}\ a) \quad\ = \mathit{cons}\ a\ \mathit{nil} \\
&\qquad\qquad\qquad\qquad\qquad\quad f\ (\mathsf{Node}\ t_1\ t_2) = (f\ t_1)\ `\mathit{append}`\ (f\ t_2) \\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ f\ t)
\end{aligned}
$$

Indeed, this abstraction from the concrete list constructors [ ] and : as well as from occurrences of ++ is no more complicated than that performed when expressing list-producers in terms of `build` as preparation for short cut fusion. And that the abstraction is done properly is guaranteed by the polymorphic type of $\texttt{vanish}_{+\!+}$'s argument. Any incomplete or inconsistent switch from [ ], :, and ++ to *nil*, *cons*, and *append* would immediately lead to a complaint by the Haskell type checker.

If (6.2) were really the syntactic definition of $\texttt{vanish}_{+\!+}$, then of course $\texttt{flatten}_3$ would not be any more efficient than the original `flatten`. The concrete list constructors and ++ would simply be put back in place, and afterwards the same computations as originally would be performed. So the crucial point is that (6.2) is actually *not* the definition of $\texttt{vanish}_{+\!+}$. Instead, that function is defined as follows:

$$
\begin{aligned}
&\texttt{vanish}_{+\!+} :: \textbf{forall}\ \alpha.\ (\textbf{forall}\ \beta.\ \beta \to (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \beta) \to [\alpha] \\
&\texttt{vanish}_{+\!+}\ g = g\ \texttt{id}\ (\lambda a\ f \to (a :) \circ f)\ (\circ)\ [\,]
\end{aligned}
$$

But a free theorem can be used to prove that (6.2) is a semantic equality that holds for this definition. So whenever the type checker accepts an expression $\texttt{vanish}_{+\!+}\ g$, the programmer is safe to read this expression as $g\ [\,]\ (:)\ (+\!+)$. So it is *as if* (6.2) were the definition of $\texttt{vanish}_{+\!+}$, which makes writing and understanding code like that of $\texttt{flatten}_3$ above very easy. Behind the scenes, however, the actual, more complicated definition of $\texttt{vanish}_{+\!+}$ is used, which gives linear runtime alike to that of $\texttt{flatten}_1$ (with accumulating parameters) and $\texttt{flatten}_2$ (with difference lists).

The connection to difference lists is very strong indeed, as comparing the arguments to g in the actual definition of $\texttt{vanish}_{+\!+}$ to the definitions of `nilR`, `consR`, and `appendR` shows. But by encapsulating the transformation in $\texttt{vanish}_{+\!+}$ we achieve the necessary discipline, enforced by the type checker, when moving from the standard representation of lists to the one with optimized concatenation. And law (6.2) provides the sought-after general correctness statement in a very concise way.

6.3 Other List Operations

The discussion so far has been specific to list concatenation. But problems with repeated, nested operations leading to quadratic overhead occur for other ways of manipulating lists (or other data structures) as well. Consider, for example, the

following naive definition of a function for shuffling a list:

$$\begin{aligned}&\texttt{shuffle} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]\\ &\texttt{shuffle}\ [\,] \qquad = [\,]\\ &\texttt{shuffle}\ (a : as) = a : (\texttt{reverse}\ (\texttt{shuffle}\ as))\end{aligned}$$

Since every application of `reverse` takes time linear in the length of its argument, the overall runtime of `shuffle` is quadratic in the length of the input list. The full paper (Voigtländer 2002) describes a methodology to manufacture `vanish`-functions for other operations than list concatenation. For `reverse` it leads to the following definition:

$$\begin{aligned}&\texttt{vanish}_{\texttt{rev}} :: \textbf{forall}\ \alpha.\ (\textbf{forall}\ \beta.\ \beta \to (\alpha \to \beta \to \beta) \to (\beta \to \beta) \to \beta) \to [\alpha]\\ &\texttt{vanish}_{\texttt{rev}}\ g = \texttt{fst}\ (g\ (\texttt{split id id})\\ &\qquad\qquad\qquad (\lambda a\ f \to \texttt{split}\ ((a :) \circ \texttt{fst} \circ f)\ (\texttt{snd} \circ f \circ (a :)))\\ &\qquad\qquad\qquad (\lambda f \to (\texttt{split snd fst}) \circ f)\\ &\qquad\qquad\qquad [\,])\end{aligned}$$

where:

$$\begin{aligned}&\texttt{fst} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ (\alpha, \beta) \to \alpha\\ &\texttt{fst}\ (a, b) = a\end{aligned}$$

$$\begin{aligned}&\texttt{snd} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ (\alpha, \beta) \to \beta\\ &\texttt{snd}\ (a, b) = b\end{aligned}$$

$$\begin{aligned}&\texttt{split} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ \textbf{forall}\ \gamma.\ (\alpha \to \beta) \to (\alpha \to \gamma) \to \alpha \to (\beta, \gamma)\\ &\texttt{split}\ f\ g\ a = (f\ a, g\ a)\end{aligned}$$

Then, the following rewrite of `shuffle`:

$$\begin{aligned}&\texttt{shuffle}' :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]\\ &\texttt{shuffle}'\ as = \texttt{vanish}_{\texttt{rev}}\\ &\quad (\lambda nil\ cons\ reverse \to \textbf{let}\ f\ [\,] \qquad = nil\\ &\qquad\qquad\qquad\qquad\qquad\quad f\ (a : as) = cons\ a\ (reverse\ (f\ as))\\ &\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ f\ as)\end{aligned}$$

has linear runtime. Its correctness with respect to the original `shuffle` is guaranteed by the following law, which can be proved using a free theorem:

$$\texttt{vanish}_{\texttt{rev}}\ g \quad = \quad g\ [\,]\ (:)\ \texttt{reverse}$$

Actually, in the presence of only partially defined or infinite lists, just an inequational variant of this law is proved, where the left-hand side may be more defined than the right-hand side. For our concrete example this becomes visible by `shuffle'` happily mapping an infinite input list to an infinite output list, while `shuffle` applied to an infinite list diverges after producing just one element of output. For more discussion on this aspect we refer the reader to the full paper.

Our general methodology builds on ideas from short cut (`foldr/build`-) fusion. Additionally, we factor in knowledge about algebraic properties of the operations under consideration. For example, $\texttt{vanish}_{+\!+}$ (as well as Wadler's and Hughes' methods introduced in Section 6.1) is built on the following associativity law:

$$(as +\!+ bs) +\!+ cs \quad = \quad as +\!+ (bs +\!+ cs) \tag{6.3}$$

while the development of $\texttt{vanish}_{\texttt{rev}}$ was informed by the observation that

$$\texttt{reverse}\ (\texttt{reverse}\ as) \quad = \quad as$$

(at least for total, finite lists as). Similarly, `vanish`-functions for `map` and `filter` can be obtained from

$$(\texttt{map}\ f) \circ (\texttt{map}\ g) \quad = \quad \texttt{map}\ (f \circ g)$$

and

$$(\texttt{filter}\ p) \circ (\texttt{filter}\ q) \quad = \quad \texttt{filter}\ (\lambda a \to (q\ a)\ \&\&\ (p\ a))$$

respectively, and combinations are also possible. Finally, the methodology is not specific to lists alone. It is equally applicable to other algebraic data types. Indeed, Chapter 7 presents a related approach that handles a whole class of data structures, so-called *free monads*.

6.4 Yet Another Possibility

To prepare for the next chapter, we draw attention to a further way of eliminating repeated list concatenation, in the spirit of difference lists but with improved guidance by the type checker and facilitating a concise correctness statement.

The idea is to use (higher-order) overloading. Specifically, one can declare a type constructor class as follows:

```
class ListLike δ where
  nil    :: forall α. δ α
  cons   :: forall α. α → δ α → δ α
  append :: forall α. δ α → δ α → δ α
```

and provide the following two instance definitions:

```
instance ListLike [] where
  nil    = []
  cons   = (:)
  append = (++)

instance ListLike DList where
  nil    = nilR
  cons   = consR
  append = appendR
```

Then, it is possible to write list-producers without early commitment to either ordinary or difference lists. For example, the following single version of `flatten`:

$$\begin{aligned}&\texttt{flatten} :: \textbf{forall}\ \alpha.\ \mathsf{Tree}\ \alpha \to (\textbf{forall}\ \delta.\ \mathsf{ListLike}\ \delta \Rightarrow \delta\ \alpha)\\&\texttt{flatten}\ (\mathsf{Leaf}\ a) \quad\ = \texttt{cons}\ a\ \texttt{nil}\\&\texttt{flatten}\ (\mathsf{Node}\ t_1\ t_2) = (\texttt{flatten}\ t_1)\ \text{`}\texttt{append}\text{`}\ (\texttt{flatten}\ t_2)\end{aligned}$$

can be used both to produce ordinary lists and to produce difference lists. The choice between the two will be made automatically by the type checker, depending on the context in which a call to `flatten` occurs. For example, in

$$\texttt{last}\ (\texttt{flatten}\ t)$$

the ordinary list representation will be used, due to the input type of

$$\texttt{last} :: \textbf{forall}\ \alpha.\ [\alpha] \to \alpha$$

If that is what we want, then fine. But if not, we can switch to the better behaved difference list representation without touching the code of `flatten` at all, by simply using the rank-2 polymorphic function

$$\begin{aligned}&\texttt{improve} :: \textbf{forall}\ \alpha.\ (\textbf{forall}\ \delta.\ \mathsf{ListLike}\ \delta \Rightarrow \delta\ \alpha) \to [\alpha]\\&\texttt{improve}\ as = \texttt{abs}\ as\end{aligned}$$

as follows:

$$\texttt{last}\ (\texttt{improve}\ (\texttt{flatten}\ t))$$

Here the (input) type of `abs` determines `flatten` to use `nilR`, `consR`, and `appendR`, leading to linear runtime. And the semantic correctness of adding `improve` at will wherever the type checker allows doing so is established by a somewhat surprising statement that can be proved using a free theorem. Namely, for every concrete type τ and every as that can be given the type **forall** δ. ListLike $\delta \Rightarrow \delta\ \tau$, it holds that

$$\texttt{improve}\ as \quad = \quad as \tag{6.4}$$

What makes this statement surprising is that it seems to say that `improve` is an identity function, while `improve`'s definition seems to say that it has incompatible input and output types. The explanation here is that the type **forall** δ. ListLike $\delta \Rightarrow \delta\ \tau$ is essentially a "subtype" of $[\tau]$. Every value of the former can also be seen as a value of type $[\tau]$, and indeed will implicitly be turned into one by the type checker if the program context so demands. But while the type checker would do this naively by using the ListLike instance in which `nil` is $[\,]$, `cons` is $(:)$, and `append` is $(+\!\!+)$, we can use `improve` to instead enforce the use of difference lists. Just as with law (6.2), the equivalence (6.4) is for the programmer's peace of mind, while behind the scenes more intricate things are going on. The work reported on in the next chapter applies this particular approach of hiding details through type constructor class abstraction to efficient programming with monads.

Chapter 7

Asymptotic Improvement of Computations over Free Monads

This chapter is based on the following paper:

- J. Voigtländer.
  Asymptotic Improvement of Computations over Free Monads.
  In C. Paulin-Mohring and P. Audebaud, editors, *9th International Conference on Mathematics of Program Construction, Marseille, France, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008.
  (Voigtländer 2008d)

It presents a program transformation to improve the time efficiency of computations over a variety of data types. Advanced mechanisms on the level of types make the transformation both powerful and easy to use.

7.1 (Free) Monads in Haskell

Since first brought to the field of programming language semantics by Moggi (1991) and then popularized for practical programming by Wadler (1992), *monads* have become everyday structures for Haskell programmers to work with. Monads allow to safely encapsulate impure features of the language (Peyton Jones and Wadler 1993, Launchbury and Peyton Jones 1995), but are also used in pure code to separate concerns and provide modular design (Liang et al. 1995). The original, mathematical definition of monads from category theory is translated into a type constructor class declaration as follows:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Monad}\ \mu\ \textbf{where} \\
\quad \texttt{return} :: \textbf{forall}\ \alpha.\ \alpha \to \mu\ \alpha \\
\quad (\gg\!=) \ :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ \mu\ \alpha \to (\alpha \to \mu\ \beta) \to \mu\ \beta
\end{array}
$$

where the following three laws are expected to hold:

$$(\texttt{return}\ a) \gg\!= k \;=\; k\ a \tag{7.1}$$

$$m \gg\!= \texttt{return} \;=\; m \tag{7.2}$$

$$(m \gg\!= k) \gg\!= h \;=\; m \gg\!= (\lambda a \to (k\ a) \gg\!= h) \tag{7.3}$$

One important Monad instance is IO, which is responsible precisely for what its name suggests. The interpretation of `return` is simply to lift pure values into the realm of monads. That is, `return` a is a computation without input/output-effects that results in a. The operator $\gg\!=$ is used for sequencing potentially effectful computations. That is, $m \gg\!= k$ is a computation that first performs the computation m, with all its input/output-effects, then takes the resulting value, passes it as argument to k, and continues with whatever computation is determined by this application. Together with some predefined primitives like the following ones:

```
getChar :: IO Char

putChar :: Char → IO ()
```

this allows to write arbitrary computations performing input/output. For example,

```
echo :: IO ()
echo = getChar >>= (λc → if c /= '*' then (putChar c) >>= (λ() → echo)
                                     else  return ())
```

is a computation that, when executed, reads characters from the standard input and immediately writes them to the standard output, until character '$*$' is read, at which point the computation finishes.

It is also entirely possible to define one's own monads. For example, the data type

```
data Tree α = Node (Tree α) (Tree α) | Leaf α
```

from Section 2.1 lends itself to an instance definition, satisfying the laws (7.1)–(7.3), as follows:

```
instance Monad Tree where
  return              = Leaf
  (Leaf a)      >>= k = k a
  (Node t1 t2) >>= k = Node (t1 >>= k) (t2 >>= k)
```

Note that the operation of $\gg\!=$ here is simply to substitute leaves by trees depending on their labels, with a potential change of the type of leaf-labels. Such monads built around a substitution operation are actually quite frequent, they constitute the class of *free monads*. And they even have something interesting to say about input/output- and related "real effects" computations, as discussed next.

Assume we want to analyze input/output-code like that of `echo` above, but without directly executing it. An attractive way of doing this is to build a pure model of effectful computations by simply defining another, less opaque (than IO), monad that

also supports the primitives of interest. Borrowing from Swierstra and Altenkirch (2007), this may look as follows:

```
data IOSpec α = Return α | GetChar (Char → IOSpec α)
              | PutChar Char (IOSpec α)

instance Monad IOSpec where
  return              = Return
  (Return a)     >>= k = k a
  (GetChar f)    >>= k = GetChar (λc → (f c) >>= k)
  (PutChar c m)  >>= k = PutChar c (m >>= k)

getChar :: IOSpec Char
getChar = GetChar Return

putChar :: Char → IOSpec ()
putChar c = PutChar c (Return ())
```

Additionally, code for "simulating" the execution of such pure representations of effectful computations is given:

```
data Trace α = Finish α | Read Char (Trace α) | Write Char (Trace α)

trace :: forall α. IOSpec α → String → Trace α
trace (Return a)     cs     = Finish a
trace (GetChar f)    (c:cs) = Read c (trace (f c) cs)
trace (PutChar c m)  cs     = Write c (trace m cs)
```

Then, by changing `echo`'s type signature to `echo :: IOSpec ()`, but not changing anything in its defining equation, we have a version that can be analyzed and tested on choice input. For example, we have

```
    trace echo "ab*"
=   Read 'a' (Write 'a' (Read 'b' (Write 'b' (Read '*' (Finish ())))))
```

and if we alter the code of `echo`, say as follows:

```
echo' :: IOSpec ()
echo' = getChar >>= (λc → if c /= '*' then echo' >>= (λ() → putChar c)
                                      else return ())
```

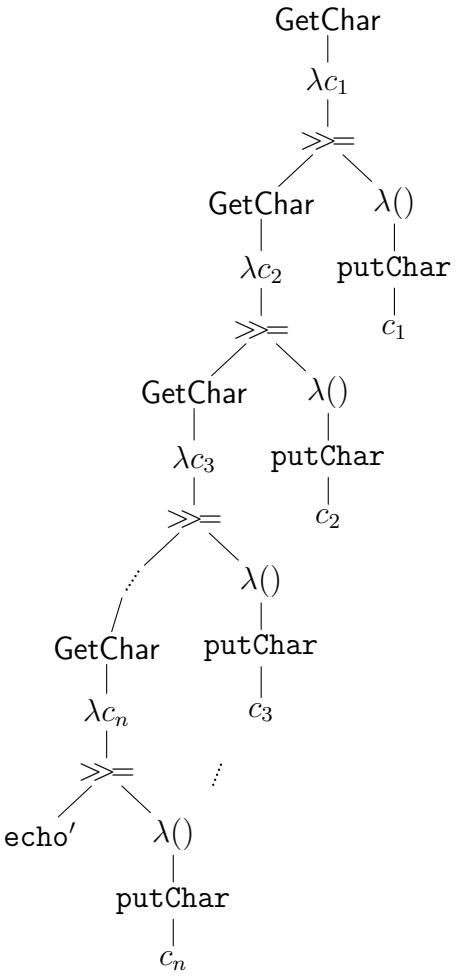
then we can very clearly observe how this affects input/output-behavior:

```
    trace echo' "ab*"
=   Read 'a' (Read 'b' (Read '*' (Write 'b' (Write 'a' (Finish ())))))
```

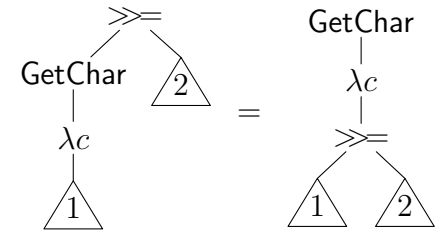
Surprisingly, though, there is a fundamental difference between using `trace` on `echo` and on `echo'`. Despite the only very small change from one to the other, the runtime of `trace echo` *cs* is linear in the length of the longest prefix of *cs* not containing '*', while that of `trace echo'` *cs* is quadratic. Why this is so, and what we can do about it, in this and in similar situations, is the topic of the current chapter.

7.2 The Problem of Repeated Substitution

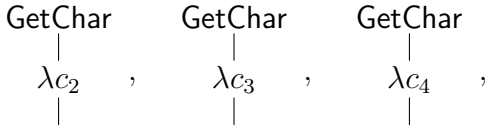
Let us consider the evaluation of `trace echo'` cs for some string cs starting with at least n non-'$*$' characters. Then at least n recursive calls to `echo'` will need to be unfolded. Inlining (`getChar` $\gg\!=$ k) :: IOSpec () to GetChar ($\lambda c \to k\ c$), this means that eventually the following computation needs to be performed:



Taking into account that the program equation defining $\gg\!=$ on a left argument of the form GetChar f can be pictured as follows:

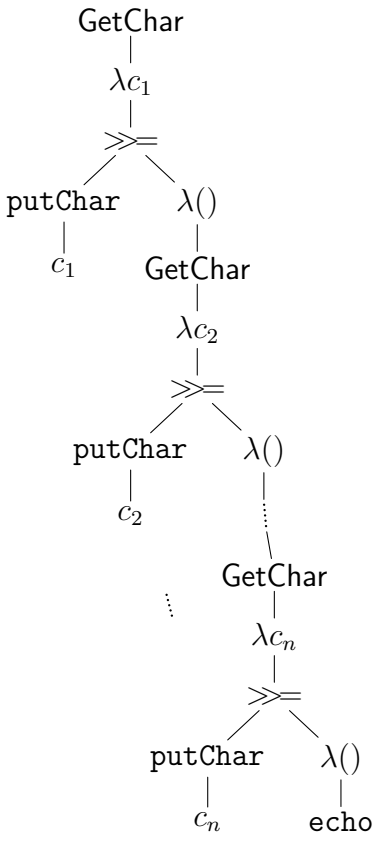


the cause of the quadratic runtime becomes apparent. The reason is simply that the fragments



and so on need to "move over" one, two, three, and so on, occurrences of $\gg=$, respectively. This gives an arithmetic sequence, the sum of which is quadratic in n.

It is instructive to compare this to the situation for `trace echo` cs. Here, the corresponding computation to be performed looks as follows:



This does not give rise to a cascade of calls to $\gg=$ as above, and thus ultimately leads to linear runtime overall.

The above considerations indicate that the problem is with left-associatively nested substitutions $\gg=$. Indeed, the situation is similar to that from Section 6.1. And just as law (6.3) on page 36 was the key to avoiding quadratic overhead caused by repeated list concatenations, we can now use law (7.3) to cure the observed inefficiency in code operating on monads.

Note that the right-hand side of (7.3) is preferable, efficiency-wise, over its left-hand side not only for the particular monad IOSpec, but for other free monads as well, such as for Tree and for other pure models of Swierstra and Altenkirch (2007). Hence, we have developed a generic solution that applies to all of them in a single stroke.

7.3 Improvement by Abstraction over `return`

Just as difference lists abstract over the end of a list, we are going to abstract over the `return`-method of a Monad instance. Encapsulated as a new data type that is parametrized over a type constructor, this can be formulated as follows:

$$\textbf{newtype}\ \mathsf{C}\ \mu\ \alpha = \mathsf{C}\ (\textbf{forall}\ \beta.\ (\alpha \to \mu\ \beta) \to \mu\ \beta)$$

It is easy to go back and forth between a monad and its "C-version" as follows:

$$\begin{aligned}&\texttt{rep} :: \textbf{forall}\ \mu.\ \mathsf{Monad}\ \mu \Rightarrow (\textbf{forall}\ \alpha.\ \mu\ \alpha \to \mathsf{C}\ \mu\ \alpha)\\&\texttt{rep}\ m = \mathsf{C}\ (m \gg\!=)\end{aligned}$$

$$\begin{aligned}&\texttt{abs} :: \textbf{forall}\ \mu.\ \mathsf{Monad}\ \mu \Rightarrow (\textbf{forall}\ \alpha.\ \mathsf{C}\ \mu\ \alpha \to \mu\ \alpha)\\&\texttt{abs}\ (\mathsf{C}\ p) = p\ \texttt{return}\end{aligned}$$

Crucially, $\mathsf{C}\ \mu$ always is itself a monad, and even is so with a constant-time implementation for the sequencing operator:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Monad}\ (\mathsf{C}\ \mu)\ \textbf{where}\\&\quad\texttt{return}\ a \quad\ = \mathsf{C}\ (\lambda h \to h\ a)\\&\quad(\mathsf{C}\ p) \gg\!= k\ = \mathsf{C}\ (\lambda h \to p\ (\lambda a \to \textbf{case}\ k\ a\ \textbf{of}\ \mathsf{C}\ q \to q\ h))\end{aligned}$$

The key to efficiency improvement by eliminating repeated substitution is now that for every concrete Monad instance κ, concrete types τ and τ', and $m :: \kappa\ \tau$ and $k :: \tau \to \kappa\ \tau'$, we have[1]

$$\texttt{rep}\ (m \gg\!= k) \quad = \quad (\texttt{rep}\ m) \gg\!= (\texttt{rep} \circ k)$$

Note that the occurrence of $\gg\!=$ on the right-hand side refers to the efficient implementation from the Monad instance $\mathsf{C}\ \kappa$.

However, it is not simply possible to more or less use the new monad C IOSpec as a drop-in replacement for IOSpec in the definition of `echo′`. The reason is that `echo′` :: IOSpec () uses `getChar` :: IOSpec Char and `putChar` :: Char → IOSpec (), for which we have no "C-equivalents" yet. More generally, $\mathsf{C}\ \kappa$ does not only have to be a monad, but also has to support equivalents of the data constructors of κ, such as Return, GetChar, and PutChar in the case $\kappa = \mathsf{IOSpec}$, and Node and Leaf in the case $\kappa = \mathsf{Tree}$. In order to treat all these cases at once, we can use the two-level types approach discussed in Section 2.2.

To recall, we have seen that for any concrete type τ, the type $\mathsf{Tree}\ \tau$ is isomorphic to the type $\mathsf{Fix}\ (\mathsf{F}_{\mathsf{Tree}}\ \tau)$, where

$$\textbf{newtype}\ \mathsf{Fix}\ \phi = \mathsf{Wrap}\ (\phi\ (\mathsf{Fix}\ \phi))$$

and

$$\textbf{data}\ \mathsf{F}_{\mathsf{Tree}}\ \alpha\ \beta = \mathsf{N}\ \beta\ \beta \mid \mathsf{L}\ \alpha$$

[1] The proof naturally depends on law (7.3) for the Monad instance κ.

Since free monads treat one distinguished data constructor differently from all others, namely the one at which actual substitution takes place, it makes sense to specialize the two-level types approach by revising the fixpoint wrapper as follows:

$$\textbf{data}\ \mathsf{Free}\ \phi\ \alpha = \mathsf{Return}\ \alpha \mid \mathsf{Wrap}\ (\phi\ (\mathsf{Free}\ \phi\ \alpha))$$

Then, for example, $\mathsf{Tree}\ \tau$ is isomorphic to $\mathsf{Free}\ \mathsf{F}'_{\mathsf{Tree}}\ \tau$ for

$$\textbf{data}\ \mathsf{F}'_{\mathsf{Tree}}\ \beta = \mathsf{N}\ \beta\ \beta$$

and $\mathsf{IOSpec}\ \tau$ is isomorphic to $\mathsf{Free}\ \mathsf{F}_{\mathsf{IO}}\ \tau$ for

$$\textbf{data}\ \mathsf{F}_{\mathsf{IO}}\ \beta = \mathsf{GetChar}\ (\mathsf{Char} \to \beta) \mid \mathsf{PutChar}\ \mathsf{Char}\ \beta$$

Moreover, the Monad instance definitions for Tree and IOSpec given in Section 7.1 are, up to the mentioned isomorphisms, just special cases of the generic definition

$$\begin{aligned}&\textbf{instance}\ \mathsf{Functor}\ \phi \Rightarrow \mathsf{Monad}\ (\mathsf{Free}\ \phi)\ \textbf{where}\\ &\quad \texttt{return} \qquad\quad = \mathsf{Return}\\ &\quad (\mathsf{Return}\ a) \gg\!= k = k\ a\\ &\quad (\mathsf{Wrap}\ t) \quad \gg\!= k = \mathsf{Wrap}\ (\texttt{fmap}\ (\gg\!= k)\ t)\end{aligned}$$

for the straightforward Functor instance definitions

$$\begin{aligned}&\textbf{instance}\ \mathsf{Functor}\ \mathsf{F}'_{\mathsf{Tree}}\ \textbf{where}\\ &\quad \texttt{fmap}\ h\ (\mathsf{N}\ b_1\ b_2) = \mathsf{N}\ (h\ b_1)\ (h\ b_2)\end{aligned}$$

and

$$\begin{aligned}&\textbf{instance}\ \mathsf{Functor}\ \mathsf{F}_{\mathsf{IO}}\ \textbf{where}\\ &\quad \texttt{fmap}\ h\ (\mathsf{GetChar}\ f) \quad = \mathsf{GetChar}\ (h \circ f)\\ &\quad \texttt{fmap}\ h\ (\mathsf{PutChar}\ c\ b) = \mathsf{PutChar}\ c\ (h\ b)\end{aligned}$$

In fact, for any Functor instance satisfying the laws (2.1) and (2.2) from page 10, the implied Monad instance satisfies the laws (7.1)–(7.3). This precisely captures all free monads in a generic way.

Now we are also in a position to treat data constructors. In particular, we need to treat the non-Return ones, that is, those that are captured in the ϕ of $\mathsf{Free}\ \phi$. In the spirit of the ListLike class from Section 6.4, we want to provide an abstract interface that can later be populated both by the original representation of a monad and by its "C-version". This motivates the following class declaration:[2]

$$\begin{aligned}&\textbf{class}\ (\mathsf{Functor}\ \phi, \mathsf{Monad}\ \mu) \Rightarrow \mathsf{FreeLike}\ \phi\ \mu\ \textbf{where}\\ &\quad \texttt{wrap} :: \textbf{forall}\ \alpha.\ \phi\ (\mu\ \alpha) \to \mu\ \alpha\end{aligned}$$

Then we can define abstract versions of `getChar` :: IOSpec Char and `putChar` :: Char → IOSpec () as follows:[3]

```
getChar :: forall μ. FreeLike F_IO μ ⇒ μ Char
getChar = wrap (GetChar return)

putChar :: forall μ. FreeLike F_IO μ ⇒ Char → μ ()
putChar c = wrap (PutChar c (return ()))
```

[2] Compiler flag `-XMultiParamTypeClasses` is needed here.
[3] Compiler flag `-XFlexibleContexts` is needed here.

and consequently give the type signature **forall** μ. FreeLike $\mathsf{F_{IO}}$ $\mu \Rightarrow \mu$ () to both `echo` and `echo′`, without changing anything in their defining equations. The idea, of course, is that we then want to use `echo` and `echo′` alternatively at type Free $\mathsf{F_{IO}}$ (), corresponding to the original type IOSpec (), or at type C (Free $\mathsf{F_{IO}}$) (), with the improved implementation of sequencing. This clearly requires appropriate instance definitions for FreeLike. One is almost trivial:[4]

```
instance Functor φ ⇒ FreeLike φ (Free φ) where
  wrap = Wrap
```

The other is somewhat intricate:

```
instance FreeLike φ μ ⇒ FreeLike φ (C μ) where
  wrap t = C (λh → wrap (fmap (λ(C p) → p h) t))
```

But it is this definition which eventually enables us to prove the "magic equivalence"

$$\texttt{improve}\ m \quad = \quad m$$

for the following function:

```
improve :: forall φ. Functor φ ⇒ (forall α. (forall μ. FreeLike φ μ ⇒ μ α)
                                             → Free φ α)
improve m = abs m
```

The proof uses both equational and type-based reasoning. More details can be found in the full paper (Voigtländer 2008d).

The benefit for the programmer is that it is now possible to transparently switch between the original representation of a free monad and the version optimized by exploiting associativity according to law (7.3). Indeed, `trace (improve echo′)` cs, for a very minor variation of `trace` that takes care of the additional Wraps, now runs in linear time, as do `trace (improve echo)` cs and `trace echo` cs.

[4] Compiler flag `-XFlexibleInstances` is needed here.

Chapter 8

Free Theorems and Selective Strictness

This chapter is based on the following two papers:

- P. Johann and J. Voigtländer.
  Free Theorems in the Presence of *seq*.
  In X. Leroy, editor, *31st Symposium on Principles of Programming Languages, Venice, Italy, Proceedings*, volume 39(1) of *SIGPLAN Notices*, pages 99–110. ACM Press, 2004.
  (Johann and Voigtländer 2004)

- P. Johann and J. Voigtländer.
  The Impact of *seq* on Free Theorems-Based Program Transformations.
  Fundamenta Informaticae, volume 69(1–2), Special Issue "Program Transformation: Theoretical Foundations and Basic Techniques. Part 2", guest editors: A. Pettorossi and M. Proietti, pages 63–102. IOS Press, 2006.
  (Johann and Voigtländer 2006)

It discusses how free theorems suffer when certain language features have to be taken into account, and what we can do about it. Of particular interest is the impact this has on program transformations like those seen in Section 3.2 and Chapter 6.

8.1 The Painful Truth about Free Theorems

In Section 3.1 we argued that for every function

$$\texttt{f} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]$$

the following law holds:

$$\texttt{f}\ (\texttt{map}\ h\ l) \quad = \quad \texttt{map}\ h\ (\texttt{f}\ l) \tag{8.1}$$

But consider the following function definition:

$$\texttt{f} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]$$
$$\texttt{f}\ l = [\texttt{head}\ (\texttt{f}\ l)]$$

Due to the recursive call in a position that is absolutely required for evaluating the element of the singleton output list, `f` l will never compute a completely total value. In fact, independently of l, it always produces a list whose single element is a looping computation. Such nontermination is usually denoted by the "undefined value" $\bot$, so that `f` l can be seen as evaluating to $[\bot]$. But this means that the left-hand side of (8.1) always evaluates to $[\bot]$, while its right-hand side evaluates to $[h\ \bot]$. Since h could be a constant function ignoring its argument, say $h = (\lambda a \to 17)$, the supposed equivalence (8.1) suddenly does not hold anymore. Let us consider this problem in some more detail.

The starting point for the discussion leading to law (8.1), or (3.1), in Section 3.1 was the observation that the output list of any `f` $::$ **forall** $\alpha.\ [\alpha] \to [\alpha]$ can only ever contain elements from the input list. But this is not true anymore when we take general recursion with its potential for nontermination into account. Because then `f` might just as well choose, for some element position of its output list, to start an arbitrary nonterminating computation, easily detectable or not. That is, while f certainly (and still) cannot possibly make up new elements of any concrete type to put into the output, such as 42 or `True`, it may very well "put" $\bot$ there, even while not knowing the element type of the lists it operates over, because conceptually $\bot$ does exist at every type. So the erstwhile claim that for any input list l the output list `f` l consists solely of elements from l has to be refined as follows.

> For any input list l the output list `f` l consists solely of elements from l and/or $\bot$.

The decisions about which elements from l to propagate to the output list, in which order and multiplicity, and where to put $\bot$ can again only be made based on the input list l, and only by inspecting its length (or running into an undefined tail or an infinite list).

> So for any pair of lists l and l' of same length (refining this notion to take partial and infinite lists into account) the lists `f` l and `f` l' are formed by making the same position-wise selections of elements from l and l', respectively, and by inserting $\bot$ at the same positions, if any.

For any $l' =$ `map` $h\ l$, we then still have that `f` l and `f` l' are of the same length and contain position-wise exactly corresponding elements from l and l', *at those positions where* `f` *takes over elements from its input rather than inserting* $\bot$. For those positions where `f` *does* insert $\bot$, which will then happen equally for `f` l and `f` l', we may only argue that the element in `f` l' contains the h-image of the corresponding element in `f` l if indeed $\bot$ is the h-image of $\bot$. This requirement is called *strictness*.

> So for any list l and, importantly, *strict* function h, the law (8.1) holds.

The example given above, essentially $\texttt{f} = (\lambda l \to [\bot])$, together with $h = (\lambda a \to 17)$ shows that it is in general not possible to drop the condition on h.

And the situation gets worse if we move on to consider further language features that are ignored in the "naive" perspective on free theorems. For example, Haskell provides a primitive `seq` with conceptually the following definition:

$$\begin{aligned}&\texttt{seq} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ \alpha \to \beta \to \beta\\&\texttt{seq}\ \bot\ b = \bot\\&\texttt{seq}\ a\ b = b\,, \text{ if } a \neq \bot\end{aligned}$$

It is typically used to selectively force the evaluation of subexpressions, circumventing lazy evaluation. This gives the advanced programmer explicit control over evaluation order (by making functions strict that would otherwise not be so), which can be important for efficiency. For example, in the function definition

$$\begin{aligned}&\texttt{foldl}' :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta\\&\texttt{foldl}'\ f\ b\ [\,] \qquad = b\\&\texttt{foldl}'\ f\ b\ (a : as) = \textbf{let}\ b' = f\ b\ a\ \textbf{in}\ \texttt{seq}\ b'\ (\texttt{foldl}'\ f\ b'\ as)\end{aligned}$$

the use of `seq` ensures that the accumulating parameter is computed immediately in each recursive step rather than constructing a complex closure which would be computed only at the very end. But the provision of `seq` also weakens what we can conclude about a function's behavior by just inspecting its type.

Consider again the following type signature:

$$\texttt{f} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]$$

Before, we have argued that when `f` decides which elements from its input list to propagate to the output list, in which order and multiplicity, and where to put $\bot$, this can only depend on the length of the input list. The rationale for this was that `f` is ignorant of the element type, and so is prevented from analyzing list elements in any way. But this is not true anymore now. Using the fully polymorphic primitive `seq`, it is possible to make `f`'s behavior depend in a certain way on whether specific elements of the input list are $\bot$ or not. It is not necessary to know those elements' type for this. Since the possible behaviors of `f` are richer now, the attendant free theorem must become weaker. Indeed, now equivalence (8.1) can only be guaranteed to hold if in addition to strictness we also require *totality* of h, where a function is called *total* if it never maps a non-$\bot$ value to $\bot$. There is also an example evidencing the need for this additional restriction. Consider the following function definition:

$$\begin{aligned}&\texttt{f} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]\\&\texttt{f}\ l = \texttt{seq}\ (\texttt{head}\ l)\ [\,]\end{aligned}$$

With the strict function $h = \bot$ and the list $l = [17]$ we have $\texttt{f}\ (\texttt{map}\ h\ l) = \bot$, but $\texttt{map}\ h\ (\texttt{f}\ l) = [\,]$, so that equivalence (8.1) fails.

It was long a folklore belief that the restriction to strict and total functions as above would be enough to always salvage free theorems even in the presence of

general recursion and selective strictness à la `seq`. But in the work reported on here we showed that this is not the case (Johann and Voigtländer 2004). For example, for the equivalence (3.2) from Section 3.1 the folklore belief would have predicted that it is again enough to require h to be strict and total. But actually it turns out that one needs to additionally require that p is not $\bot$. The good news is that it *is* possible to recover free theorems even in the presence of advanced language features. So while free theorems get weakened, and rightly so, when moving from the pure polymorphic lambda-calculus (Reynolds 1974) for which they were originally conceived towards a setting more closely resembling modern functional languages, we have a handle at things. We can revise the underlying theory, and thus make free theorems applicable in richer settings. Before saying a bit more about this, it is helpful to briefly explain how free theorems come about at all.

8.2 The Formal Background of Free Theorems

The origin of free theorems lies in Reynolds' (1983) studies about characterizing parametric polymorphism. The question approached was what it means for a polymorphic function to behave uniformly, regardless of the concrete type at which it is instantiated. Intuitively, the concept of two functions to behave the same is that they map equal arguments to equal results. But this does not really make sense when we want to compare two different instantiations of a polymorphic function. For example, when trying to compare the two instantiations of a function `f` :: **forall** α. $[\alpha] \to [\alpha]$ at types Int and Bool, we cannot say that they "behave the same" when they "map equal input lists to equal output lists". After all, one of the two instantiations under consideration maps integer lists to integer lists, while the other maps Boolean valued lists to Boolean valued lists. And there is no concept of an integer list being "equal" to a Boolean valued list. Reynolds' key idea was to move away from equality and instead consider arbitrary binary relations. Say we fix a relation between Int and Bool that relates every even integer to True and every odd integer to False. Given this relation as a base, it is straightforward to formulate a meaningful concept of an integer list and a Boolean valued list being related: we simply require that the lists are of the same length and that elements at corresponding positions are related in the way just described. For the supposed two instances of `f` at Int and Bool we can now require that they map *related* input lists to *related* output lists. If `f` is truly polymorphic, with its behavior independent of concrete choices for instantiating α, then this invariant will indeed be preserved. Actually, it will be so for *every* choice of a base relation between Int and Bool, not just for the one connecting even integers to True and odd ones to False. And what is more, this condition is not only necessary, but also sufficient. Not only will every truly polymorphic `f` preserve every relation between every pair of concrete types chosen for instantiating α, but also conversely is this universal preservation enough to establish that `f` is polymorphic in a truly uniform way. For any `f` that were to "cheat" by behaving differently for one type or another, it would be possible to

find some relation that is not preserved. It is this characterization on which Wadler (1989) built his methodology of deriving free theorems. Of course, it is necessary to make precise the idea of propagating relations from the base level to relations over lists, over functions, and so on.

First, every quantification over type variables is replaced by quantification over relation variables. For example, given the type signature `f` :: **forall** α. $[\alpha] \to [\alpha]$ we obtain $\forall\mathcal{R}.\ [\mathcal{R}] \to [\mathcal{R}]$. Then, there is a systematic way of reading such expressions over relations as relations themselves. In particular,

- base types like Int are read as identity relations,
- for relations $\mathcal{R}$ and $\mathcal{S}$ we have

  $$\mathcal{R} \to \mathcal{S} = \{(f, g) \mid \forall (a, b) \in \mathcal{R}.\ (f\ a, g\ b) \in \mathcal{S}\}$$

  and
- for "type schemes" τ and τ' with at most one free variable, say α, and a function $\mathcal{F}$ on relations such that every relation $\mathcal{R}$ between concrete types τ_1 and τ_2, denoted $\mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2)$, is mapped to a relation $\mathcal{F}\ \mathcal{R} \in \mathit{Rel}(\tau[\tau_1/\alpha], \tau'[\tau_2/\alpha])$, we have

  $$\forall\mathcal{R}.\ \mathcal{F}\ \mathcal{R} = \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2).\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}\ \mathcal{R}\}$$

  (Here, u_{τ_1} is the instantiation of a value u of type **forall** α. τ at the type τ_1, and similarly for v_{τ_2}. So far, we have always left type instantiation implicit, and we will continue to do so in what follows, except for Chapter 10.)

Also, every type constructor is read as an appropriate construction on relations. For example, the list type constructor maps every relation $\mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2)$ to the relation $[\mathcal{R}] \in \mathit{Rel}([\tau_1], [\tau_2])$ defined by

$$[\mathcal{R}] = \{([\,], [\,])\} \cup \{(a : as, b : bs) \mid (a, b) \in \mathcal{R},\ (as, bs) \in [\mathcal{R}]\}$$

and similarly for other algebraic data types.

Free theorems are now derived from the fact, proved once and then used over and over again, that every value of a concrete type is related to itself by the relational interpretation of that type. For the example `f` :: **forall** α. $[\alpha] \to [\alpha]$ this means that any such `f` satisfies $(\mathtt{f}, \mathtt{f}) \in \forall\mathcal{R}.\ [\mathcal{R}] \to [\mathcal{R}]$, which by unfolding some of the above definitions is equivalent to having for every τ_1, τ_2, $\mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2)$, $l :: [\tau_1]$, and $l' :: [\tau_2]$ that $(l, l') \in [\mathcal{R}]$ implies $(\mathtt{f}\ l, \mathtt{f}\ l') \in [\mathcal{R}]$, or, specialized to the function level ($\mathcal{R} \mapsto h$, and thus $[\mathcal{R}] \mapsto \mathtt{map}\ h$), for every $h :: \tau_1 \to \tau_2$ and $l :: [\tau_1]$ that `f` (`map` h l) = `map` h (`f` l). This finally provides the formal, and systematic, counterpart to the intuitive reasoning seen earlier.

Of course, this all depends on the mentioned fact that every value is related to itself by the relation built from its type. Reynolds proved it for the pure polymorphic lambda-calculus, but when extending the calculus towards a real programming language this proof has to be revisited, adapted, extended, and in some cases even largely redone.

8.3 Free Theorems in the Presence of `seq`

To take care of general recursion and thus potential nontermination, Wadler (1989, Section 7) mandates that only *strict* relations should be allowed as interpretations of types. That is, $Rel(\tau_1, \tau_2)$ is not anymore taken to be the collection of all relations between types τ_1 and τ_2, but only of those that contain the pair $(\bot, \bot)$.[1] By ensuring that the required constructions on relations preserve strictness, in particular by explicitly including the pair $(\bot, \bot)$ in $[\mathcal{R}] \in Rel([\tau_1], [\tau_2])$ and in the relational interpretations of other algebraic data types, one gets a treatment of free theorems that is sound again. In particular, one gets law (8.1) under the condition that h is a strict function, precisely as argued on the intuitive level in the first half of Section 8.1.

When additionally including `seq`, slightly more subtle issues need to be addressed. First of all, the collection of relations available for interpreting types needs to be further constrained. It is important to note that the restrictions getting imposed on relations when considering additional language features are not the result of arbitrary decisions. Rather, they directly have to do with the idea in the first part of Section 8.2 that relations are used as kind of "invariants" that tie together two instances of a polymorphic function. The more expressivity the language has at hand without being bound to concrete types, such as $\bot$ and `seq`, the fewer such invariants we can expect to be preserved. Or, in other words, since free theorems stem from considerations about what the language is *not* able to do in a polymorphic way, exactly the right weakenings have to be found whenever a new primitive is added that gives the language new possibilities of *doing* things polymorphically. The best way to find out what restrictions become necessary is to derive a free theorem for the new primitive, but according to the "old" relational framework.

Recall that the type of `seq` is **forall** α. **forall** β. $\alpha \to \beta \to \beta$. So we would expect that $(\texttt{seq}, \texttt{seq}) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \mathcal{R} \to (\mathcal{S} \to \mathcal{S})$, and thus that for every $\mathcal{R} \in Rel(\tau_1, \tau_2)$, $\mathcal{S} \in Rel(\tau_1', \tau_2')$, $(a_1, a_2) \in \mathcal{R}$, and $(b_1, b_2) \in \mathcal{S}$, it holds that $(\texttt{seq}\ a_1\ b_1, \texttt{seq}\ a_2\ b_2) \in \mathcal{S}$. But just requiring $\mathcal{R}$ and $\mathcal{S}$ to be strict is not enough to guarantee this. In particular, it is absolutely possible to choose a strict $\mathcal{R}$ with $(\bot, a_2) \in \mathcal{R}$ for some $a_2 \neq \bot$ and a strict $\mathcal{S}$ with $(b_1, b_2) \in \mathcal{S}$ but $(\bot, b_2) \notin \mathcal{S}$ for some b_1 and b_2. Then $(\bot, a_2) \in \mathcal{R}$ and $(b_1, b_2) \in \mathcal{S}$, but not $(\texttt{seq}\ \bot\ b_1, \texttt{seq}\ a_2\ b_2) \in \mathcal{S}$. The problem apparently is that imposing strictness on a relation only means that $\bot$ must *at least* be related to $\bot$. It does not mean that $\bot$ may not *also* be related to some non-$\bot$ values. And this leads to the counterexample just mentioned. The appropriate restriction that prevents this is to require that relations quantified over in the definition of $\forall\mathcal{R}.\ \mathcal{F}\ \mathcal{R}$ should not only be strict, but also *bottom-reflecting*. That is, $Rel(\tau_1, \tau_2)$ is not anymore taken to be the collection of all strict relations between types τ_1 and τ_2, but only of those that are strict and in addition do not contain a pair (a_1, a_2) with

[1] Actually, relations must be strict *and continuous* (in the domain-theoretic sense). However, we mainly focus on interpreting type variables not by general relations, but by relation graphs of Haskell functions. These, and any relations constructed from them by propagation along the type structure, will always be continuous, so we leave the continuity condition implicit in what follows.

either $a_1 = \bot$ and $a_2 \neq \bot$ or $a_1 \neq \bot$ and $a_2 = \bot$.

But this refinement is not yet enough to recover free theorems in the presence of `seq`. Having adapted the choice of relations to quantify over in such a way that $(\texttt{seq}, \texttt{seq}) \in \forall\mathcal{R}.\ \forall\mathcal{S}.\ \mathcal{R} \to (\mathcal{S} \to \mathcal{S})$ holds, only means that we have ensured that the new base case in the overall proof of "every value of a concrete type is related to itself by the relational interpretation of that type" goes through. That proof is by induction over typing derivations (with rules like those shown in Section 10.1), and we need to recheck all the "old" induction cases as well. As it turns out, one of them depends on the fact that any restriction we impose on relations, such as strictness and bottom-reflectingness, should not only hold for the relations chosen in the definition of $\forall\mathcal{R}.\ \mathcal{F}\ \mathcal{R}$, but should also be preserved throughout the propagation of relations along the type structure. And unfortunately, with the definition

$$\mathcal{R} \to \mathcal{S} = \{(f, g) \mid \forall(a, b) \in \mathcal{R}.\ (f\ a, g\ b) \in \mathcal{S}\}$$

it is not the case that for every choice of strict and bottom-reflecting relations $\mathcal{R}$ and $\mathcal{S}$ we get a strict and bottom-reflecting relation $\mathcal{R} \to \mathcal{S}$. So this definition needs to be adapted by explicitly enforcing bottom-reflectingness. That change to the definition, in turn, affects another induction case. But in the end this "adaptation cycle" can be completed, and we truly recover free theorems that are valid in the presence of both $\bot$ and `seq`. More details can be found in the original paper (Johann and Voigtländer 2004).

One additional, and crucial, ingredient of our approach worth mentioning here is that we perform a certain break of symmetry to obtain more liberal free theorems. Note that all free theorems seen so far have been stating semantic equivalences. When we moved to a richer language setting, more or less severe preconditions became necessary to maintain those equivalences. For example,

$$\texttt{f}\ (\texttt{map}\ h\ l) \quad = \quad \texttt{map}\ h\ (\texttt{f}\ l)$$

as derived from `f` :: **forall** α. $[\alpha] \to [\alpha]$ could only be maintained for h that are both strict and total, and we even saw how the equivalence fails for a certain `f` and a certain h that is strict but not total. In that particular case failure showed up by the left-hand side becoming less defined than the right-hand side, $\bot$ vs. $[\,]$. Thus, a natural question is whether this is a general phenomenon. Can we somehow "decompose" the above equivalence into

$$\texttt{f}\ (\texttt{map}\ h\ l) \quad \sqsubseteq \quad \texttt{map}\ h\ (\texttt{f}\ l) \tag{8.2}$$

and

$$\texttt{f}\ (\texttt{map}\ h\ l) \quad \sqsupseteq \quad \texttt{map}\ h\ (\texttt{f}\ l)$$

for a suitable definedness order $\sqsubseteq$ in such a way that the inequational versions hold under more liberal preconditions than required for the full equivalence? The answer is yes. By introducing asymmetry into the relational interpretations of types we

obtain such a treatment. For example, we can then show that law (8.2) holds for any `f` :: **forall** α. $[\alpha] \to [\alpha]$ potentially involving $\bot$ and `seq` and any h that is strict. Totality of h is not required for this direction of semantic approximation, for this particular free theorem. The ability to study semantic inequations rather than equations is also helpful in analyzing the impact of `seq` on short cut fusion and related program transformations, as reported on in the next section.

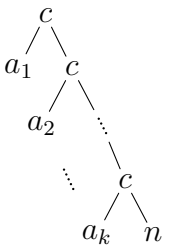
8.4 The Impact on Program Transformations

In Section 3.2 we have encountered the following `foldr`/`build`-rule:

$$\texttt{foldr}\ c\ n\ (\texttt{build}\ g) \quad = \quad g\ c\ n \tag{8.3}$$

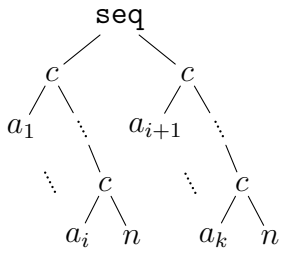
and discussed how applying it from left to right can improve the efficiency of programs. The semantic correctness of this rule was attributed to the facts that the type of `build` enforces g to be polymorphic, g :: **forall** β. $(\tau \to \beta \to \beta) \to \beta \to \beta$ for some concrete type τ, and that this means that g must be semantically equivalent to a function, for some $k \geq 0$ and $a_1, \ldots, a_k :: \tau$, of the following form:

$g \quad = \quad \lambda c\ n \to$



But the argument establishing the latter was based on the "naive" setting of Section 3.1 (and Section 8.2), while now we know that in a more realistic programming language things may change. And indeed, general recursion and selective strictness provide new ways for g to behave while maintaining its polymorphic type. For example, it can now take the form

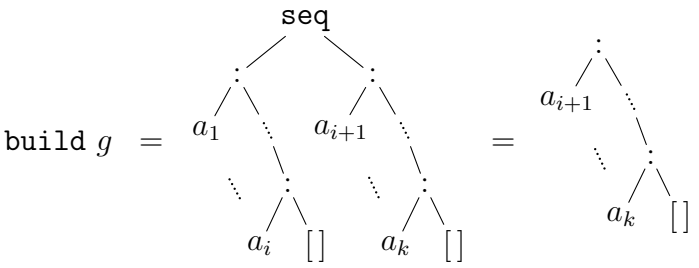
$g \quad = \quad \lambda c\ n \to$



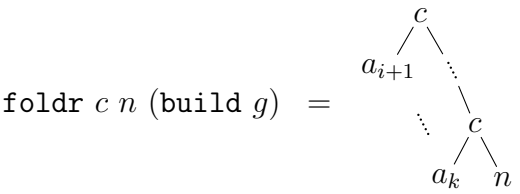
or even

$g \quad = \quad \lambda c\ n \to$

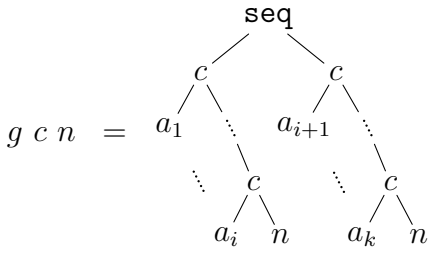
And for such g the semantic equivalence in rule (8.3) can break. Consider the first one. For it, we have:



and thus:



while:



Clearly, it can happen that $c\ a_1\ (\cdots\ (c\ a_i\ n)\cdots)$ is $\bot$ while $c\ a_{i+1}\ (\cdots\ (c\ a_k\ n)\cdots)$ is not. In that case, the right-hand side of (8.3) becomes $\bot$ while the left-hand side is not. That such a decrease of definedness is possible when incautiously applying `foldr/build`-fusion was first observed by Voigtländer (2002, Appendix B). A more realistic failure scenario (than for rather artificial g as above) is discussed by Johann and Voigtländer (2008, Section 6).

The main extension in the second paper on which this chapter is based (Johann and Voigtländer 2006, Section 8), over the first one, is precisely an in-depth analysis of this impact of `seq` on the correctness of `foldr/build`-fusion and related transformations. On the theoretical side, we identify conditions under which such transformations remain fully correct and also investigate under which (weaker) conditions at least "partial correctness" in the sense of semantic approximation can be guaranteed.

For `foldr/build`-fusion we find that the semantic equivalence in (8.3) in general only holds under the conditions that neither $c\ \bot\ \bot$ nor n are $\bot$, but that the following inequational law holds without preconditions:

$$\texttt{foldr}\ c\ n\ (\texttt{build}\ g) \quad \sqsupseteq \quad g\ c\ n \tag{8.4}$$

For our transformation technique reported on in Chapter 6 we find that law (6.2) becomes

$$\texttt{vanish}_{+\!+}\ g \quad \sqsupseteq \quad g\ [\,]\ (:)\ (+\!+) \tag{8.5}$$

and similarly for `vanish`-functions that deal with other operations than list concatenation. Note that the quality of "equivalence gets down-graded to approximation" is of an opposite nature for our technique from Chapter 6 than it is for `foldr/build`-fusion. That is, while (8.4) is intended to be used as a rewriting from left to right, meaning that the program after transformation may be less defined than the one before, the efficiency improvement direction in (8.5) is from right to left, so that the worst thing which can happen[2] is that the program becomes more defined when applying our transformation. Finally, we consider a transformation that is dual (in a category-theoretic sense) to `foldr/build`-fusion.

Takano and Meijer (1995) and later more pragmatically oriented Svenningsson (2002) proposed to express list-producing and -consuming functions, respectively, in terms of

$$\begin{aligned}&\texttt{unfoldr} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ (\beta \to \mathsf{Maybe}\ (\alpha, \beta)) \to \beta \to [\alpha]\\ &\texttt{unfoldr}\ f\ b = \textbf{case}\ f\ b\ \textbf{of}\ \mathsf{Nothing} \quad \to [\,]\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Just}\ (a, b') \to a : (\texttt{unfoldr}\ f\ b')\end{aligned}$$

and

$$\begin{aligned}&\texttt{destroy} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \gamma.\ (\textbf{forall}\ \beta.\ (\beta \to \mathsf{Maybe}\ (\alpha, \beta)) \to \beta \to \gamma) \to [\alpha] \to \gamma\\ &\texttt{destroy}\ h = h\ (\lambda b \to \textbf{case}\ b\ \textbf{of}\ [\,] \quad \to \mathsf{Nothing}\\ &\qquad\qquad\qquad\qquad\qquad\quad a : b' \to \mathsf{Just}\ (a, b'))\end{aligned}$$

Then the rule

$$\texttt{destroy}\ h\ (\texttt{unfoldr}\ f\ b) \quad = \quad h\ f\ b \tag{8.6}$$

can be used to eliminate intermediate lists in some cases where `foldr/build`-fusion is not effective. For example, it can handle `zip` as a fusion-friendly consumer of two lists, provided the function is expressed as follows:

$$\begin{aligned}&\texttt{zip} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ [\alpha] \to [\beta] \to [(\alpha, \beta)]\\ &\texttt{zip}\ as\ bs = \texttt{destroy}\ (\lambda p_1\ x_1 \to \texttt{destroy}\ (\lambda p_2\ x_2 \to go\ p_1\ p_2\ x_1\ x_2)\ bs)\ as\\ &\quad \textbf{where}\ go\ p_1\ p_2\ x_1\ x_2 = \textbf{case}\ p_1\ x_1\ \textbf{of}\\ &\qquad\qquad\qquad\qquad \mathsf{Nothing} \quad \to [\,]\\ &\qquad\qquad\qquad\qquad \mathsf{Just}\ (a, x_1') \to \textbf{case}\ p_2\ x_2\ \textbf{of}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Nothing} \quad \to [\,]\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Just}\ (b, x_2') \to (a, b) : (go\ p_1\ p_2\ x_1'\ x_2')\end{aligned}$$

The semantic correctness of the `destroy/unfoldr`-rule is intended to follow from a free theorem obtained for the polymorphic type of h. But we demonstrated that

[2] There are indeed examples where it *does* happen.

even in the presence of just general recursion, without selective strictness, only an inequational variant of (8.6) holds, replacing $=$ by $\sqsubseteq$, and that things get even more complicated when selective strictness is also considered. Specifically, we find that we then must impose the preconditions $f \neq \bot$ and $f\ \bot \in \{\bot, \mathsf{Just}\ \bot\}$ to obtain

$$\texttt{destroy}\ h\ (\texttt{unfoldr}\ f\ b) \quad \sqsubseteq \quad h\ f\ b$$

while f being strict and total and never returning $\mathsf{Just}\ \bot$ is needed to obtain

$$\texttt{destroy}\ h\ (\texttt{unfoldr}\ f\ b) \quad \sqsupseteq \quad h\ f\ b$$

Concrete counterexamples show that none of these preconditions (or of those in the equational variant of (8.4)) can be dropped. Beside such theoretical investigations, the extended paper also discusses pragmatic aspects of the semantically weakened transformations. Natural questions are how to cope with transformation rules that potentially increase or decrease definedness, whether a compiler can statically check required preconditions, and how many of the typically encountered fusion instances are actually affected by the presence of `seq`.

Chapter 9

New Developments in Short Cut Fusion

This chapter is based on the following two papers:

- J. Voigtländer.
  Proving Correctness via Free Theorems: The Case of the `destroy/build`-Rule.
  In R. Glück and O. de Moor, editors, *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, Proceedings*, pages 13–20. ACM Press, 2008.
  (Voigtländer 2008a)

- J. Voigtländer.
  Semantics and Pragmatics of New Shortcut Fusion Rules.
  In J. Garrigue and M. Hermenegildo, editors, *9th International Symposium on Functional and Logic Programming, Ise, Japan, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, 2008.
  (Voigtländer 2008c)

It describes our investigations into new (and variants of old) short cut fusion rules, along with systematic proof construction.

9.1 The `destroy/build`-Rule

As seen in the previous chapter, program transformations based on free theorems are prone to semantic defect when advanced language features enter the picture. And even when restricted to a more naive language setting, the construction of complete correctness proofs has sometimes been considered a kind of black art. Not only does one have to derive an appropriate free theorem, but then one is typically left with several degrees of freedom as to how to instantiate relations and (function) values in the obtained statement. And only the right choice, plus a certain amount of equational reasoning, will lead to a complete overall proof. Our position, though, is

that more mysteriousness has been attributed to this whole process than is justified or necessary. To make this point, the first paper on which the current chapter is based (Voigtländer 2008a) performs a case study on proving the correctness of an until then unknown transformation rule.

In that case study, we purposefully deviate from the "by-hand-approach" that was used in proving the results reported on in Section 8.4. Instead, we try to harness tool support as much as possible and to perform the overall proof construction in a mostly systematic manner. The ideal is that starting from an intuitive transformation idea it should be possible to establish formal correctness, and to identify any necessary preconditions, in a goal-driven fashion, that is, without too much foresight and without too much additional invention being required on the way.

The new transformation rule considered is a rule that applies when a list-consumer expressed in terms of `destroy` encounters a list-producer expressed in terms of `build`. This is a situation that could not be handled previously, but that does occur in practice, because neither of `foldr`/`build`- and `destroy`/`unfoldr`-fusion subsumes the other in transformation power. In particular, while `foldr` can be written using `destroy`, and `unfoldr` can be written using `build`, neither can every function expressible via `build` be (efficiently) expressed via `unfoldr`, nor can every function expressible via `destroy` be expressed via `foldr` in a way that would benefit efficiency (not even after fusion). As a consequence, if one wants to maximize potential profit from both `foldr`/`build`- and `destroy`/`unfoldr`-fusion, one ends up with programs mixing all four functions. And then it can happen that a list-consumer $\mathtt{f}_1$ which was written with `destroy` in the hope that it may match up with an `unfoldr`-expressed producer (and which indeed may do so at some other place in the overall program) encounters, in some call instance, a list-producer $\mathtt{f}_2$ which was written with `build` in the dual hope. The `destroy`/`build`-rule we propose, while not causing as aggressive optimization as the `foldr`/`build`- and `destroy`/`unfoldr`-rules do by itself, at least reduces the overhead originally introduced when expressing $\mathtt{f}_1$ and $\mathtt{f}_2$ via `destroy` and `unfoldr` rather than in direct style.

By simply inlining the definitions of `destroy` and `build`, we see that for every choice of concrete types τ and τ' and functions

$$g :: \textbf{forall}\ \beta.\ (\tau \to \beta \to \beta) \to \beta \to \beta$$

and

$$h :: \textbf{forall}\ \beta.\ (\beta \to \mathsf{Maybe}\ (\tau, \beta)) \to \beta \to \tau'$$

the expression

$$\mathtt{destroy}\ h\ (\mathtt{build}\ g) \tag{9.1}$$

is equivalent to

$$\begin{array}{l} h\ (\lambda b \to \textbf{case}\ b\ \textbf{of}\ [\,] \quad\ \to \mathsf{Nothing} \\ \qquad\qquad\qquad\qquad\quad\ a : b' \to \mathsf{Just}\ (a, b'))\ (g\ (:)\ [\,]) \end{array} \tag{9.2}$$

This shows that a standard list is created by g using the constructors : and [ ] and that essentially an isomorphic copy of that list in terms of Just $(\_, \_)$ and Nothing is built when h traverses it using, repeatedly, its function argument $(\lambda b \to \textbf{case}\ b\ \textbf{of}\ \cdots)$. Internally, h then works with those Just $(\_, \_)$- and Nothing-structures. To avoid the "repackaging" of lists, an almost obvious optimization idea is to have g immediately create lists in terms of Just $(\_, \_)$ and Nothing instead. The function argument supplied to h could then be trivial, not involving any pattern-matching. While the supposed

$$h\ (\lambda b \to b)\ (g\ (\lambda a\ b \to \mathsf{Just}\ (a, b))\ \mathsf{Nothing})$$

is rejected due to an occurs check during type unification, the following variant works:

$$\begin{aligned}&\textbf{newtype}\ \mathsf{L}\ \alpha = \mathsf{L}\ (\mathsf{Maybe}\ (\alpha, \mathsf{L}\ \alpha))\\&h\ (\lambda(\mathsf{L}\ b) \to b)\ (g\ (\lambda a\ b \to \mathsf{L}\ (\mathsf{Just}\ (a, b)))\ (\mathsf{L}\ \mathsf{Nothing}))\end{aligned} \tag{9.3}$$

Due to the nature of **newtype**-definitions, no extra overhead is incurred. Even though they do not look so, both L and $(\lambda(\mathsf{L}\ b) \to b)$ are operationally identity functions. We do not further discuss the pragmatics of transforming expression (9.1) into expression (9.3), but instead focus on what this transformation means for the semantics of a program. The question of interest is whether, and maybe under which preconditions, (9.1) and (9.3) are equivalent, or maybe weaker, one approximates the other.

The first step is to derive free theorems from the types of g and h given above. Here they are, in the "equational setting" (no asymmetry introduced when interpreting types as relations), and already specialized to the function level:

$$\begin{aligned}&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2, f \text{ strict and total. } \forall p :: \tau \to \tau_1 \to \tau_1.\ \forall q :: \tau \to \tau_2 \to \tau_2.\\&\quad (((p \neq \bot) \Leftrightarrow (q \neq \bot))\\&\qquad \wedge (\forall x :: \tau.\ ((p\ x \neq \bot) \Leftrightarrow (q\ x \neq \bot)) \wedge (\forall y :: \tau_1.\ f\ (p\ x\ y) = q\ x\ (f\ y))))\\&\quad \Rightarrow (\forall z :: \tau_1.\ f\ (g\ p\ z) = g\ q\ (f\ z))\end{aligned}$$

and

$$\begin{aligned}&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2, f \text{ strict and total.}\\&\forall p :: \tau_1 \to \mathsf{Maybe}\ (\tau, \tau_1).\ \forall q :: \tau_2 \to \mathsf{Maybe}\ (\tau, \tau_2).\\&\quad (((p \neq \bot) \Leftrightarrow (q \neq \bot)) \wedge (\forall x :: \tau_1.\ (p\ x, q\ (f\ x)) \in \mathit{lift}_{\mathsf{Maybe}}(\mathit{lift}_{(,)}(\mathtt{id}, f))))\\&\quad \Rightarrow (\forall y :: \tau_1.\ h\ p\ y = h\ q\ (f\ y))\end{aligned}$$

where

$$\begin{aligned}\mathit{lift}_{\mathsf{Maybe}}(\mathit{lift}_{(,)}(\mathtt{id}, f)) &= \{(\bot, \bot), (\mathsf{Nothing}, \mathsf{Nothing})\}\\&\quad \cup \{(\mathsf{Just}\ x_1, \mathsf{Just}\ y_1) \mid (x_1, y_1) \in \mathit{lift}_{(,)}(\mathtt{id}, f)\}\\\mathit{lift}_{(,)}(\mathtt{id}, f) &= \{(\bot, \bot)\}\\&\quad \cup \{((x_1, x_2), (y_1, y_2)) \mid (x_1 = y_1)\ \wedge\ (f\ x_2 = y_2)\}\end{aligned}$$

It should be emphasized that no manual derivation effort is required for obtaining these two statements. In fact, the "Export as PDF" facility of the online free

theorems generator accessible at `http://linux.tcs.inf.tu-dresden.de/~voigt/ft` produces almost exactly the above as output, up to only minor differences in typesetting.

The conditions imposed on f and the conditions relating to $\bot$ are precisely the kinds of technical details that need to be taken into account when wanting to prove correctness results that also hold in the presence of general recursion and `seq`, i.e., results that are meaningful for Haskell, rather than just for the pure polymorphic lambda-calculus. Here, following the foundational work reported on in Section 8.3, they are automatically generated by the tool we use, and we only need to properly keep track of (and hopefully do away with) them in the remainder of the proof.

To proceed with the proof, we can let ourselves be guided by the form of the overall correctness statement that we are interested in. For example, the desired equivalence of (9.1), or rather (9.2), on the one hand and (9.3) on the other, matches nicely with the conclusion $h\ p\ y = h\ q\ (f\ y)$ of the second of the two free theorems obtained above. Starting from this observation, and with some finite proof branching where required instantiations are not immediately obvious, we can step by step develop a successful proof. By simply following the directions and necessities encountered on the way, pieces fall into place and eventually even a concrete definition for f (in both free theorems above) is essentially forced on us. There is no need to pull any rabbits out of a hat. In the end, some easy checks suffice to establish that

$$\texttt{destroy}\ h\ (\texttt{build}\ g) \quad = \quad h\ (\lambda(\mathsf{L}\ b) \to b)\ (g\ (\lambda a\ b \to \mathsf{L}\ (\mathsf{Just}\ (a, b)))\ (\mathsf{L}\ \mathsf{Nothing}))$$

holds as a semantic equivalence. In particular, no preconditions relating to $\bot$ remain, because all expressions mandated to be non-$\bot$ in the two free theorems above happen to become explicit λ-abstractions.

The first paper reported on here (Voigtländer 2008a) exercises this whole process through, almost tutorial style, for the particular case of the `destroy/build`-rule. And, quite pleasingly, the presented way of reasoning rigorously about program transformations based on free theorems, while *not* ignoring advanced language features, scales to other transformation rules as well. Specifically, it was also employed in the exploratory study performed in the second paper on which the current chapter is based (Voigtländer 2008c), reported on in the next three sections.

9.2 Circular Short Cut Fusion

We already mentioned at the end of Section 3.2 that `foldr/build`-fusion was just a beginning for ever new related transformation rules to be developed. In one strand of development, Fernandes et al. (2007) started an investigation into how to deal with situations in which a list-producer needs to pass additional context information to the consumer.

Consider, for example, the following variation of the function `filter` from page 12 that in addition to selecting those elements from the input list which fulfill the input

predicate also outputs their number:

$$\begin{aligned}
&\texttt{filterAndCount} :: \textbf{forall}\ \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to ([\alpha], \mathsf{Int}) \\
&\texttt{filterAndCount}\ p\ [\,] \qquad\quad = ([\,], 0) \\
&\texttt{filterAndCount}\ p\ (a : as) = \textbf{if}\ p\ a\ \textbf{then let}\ (as', l) = \texttt{filterAndCount}\ p\ as \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ (a : as', l + 1) \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{else}\ \ \texttt{filterAndCount}\ p\ as
\end{aligned}$$

This additional information could be used in a function that, for the special case that the list elements are floating-point numbers, consumes such an output pair and normalizes list elements through division by the provided length:

$$\begin{aligned}
&\texttt{normalize} :: ([\mathsf{Float}], \mathsf{Int}) \to [\mathsf{Float}] \\
&\texttt{normalize}\ (as, l) = \texttt{foldr}\ (\lambda a\ r \to (a\ /\ (\texttt{fromIntegral}\ l)) : r)\ [\,]\ as
\end{aligned}$$

The functions in terms of which Fernandes et al. (2007) propose to define such producers and consumers are as follows:

$$\begin{aligned}
&\texttt{buildp} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \gamma.\ \textbf{forall}\ \delta.\ (\textbf{forall}\ \beta.\ (\alpha \to \beta \to \beta) \to \beta \to \gamma \to (\beta, \delta)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \gamma \to ([\alpha], \delta) \\
&\texttt{buildp}\ g\ u = g\ (:)\ [\,]\ u
\end{aligned}$$

$$\begin{aligned}
&\texttt{pfold} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ \textbf{forall}\ \delta.\ (\alpha \to \beta \to \delta \to \beta) \to (\delta \to \beta) \to ([\alpha], \delta) \to \beta \\
&\texttt{pfold}\ h_1\ h_2\ (as, z) = \texttt{foldr}\ (\lambda a\ r \to h_1\ a\ r\ z)\ (h_2\ z)\ as
\end{aligned}$$

Note that the function argument g of `buildp` takes, beside abstract list constructors, an additional input parameter. That parameter may be used to guide both the construction of the output list and the production of the additional context information to be passed to the consumer.

To facilitate fusion, the above examples can be expressed in terms of `buildp` and `pfold` as follows:

$$\begin{aligned}
&\texttt{filterAndCount} :: \textbf{forall}\ \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to ([\alpha], \mathsf{Int}) \\
&\texttt{filterAndCount}\ p = \texttt{buildp}\ g \\
&\quad \textbf{where}\ g\ c\ n\ [\,] \qquad\quad = (n, 0) \\
&\qquad\quad g\ c\ n\ (a : as) = \textbf{if}\ p\ a\ \textbf{then let}\ (as', l) = g\ c\ n\ as \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ (c\ a\ as', l + 1) \\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{else}\ \ g\ c\ n\ as
\end{aligned}$$

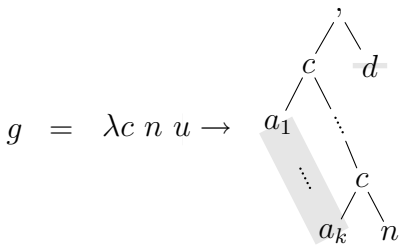
$$\begin{aligned}
&\texttt{normalize} :: ([\mathsf{Float}], \mathsf{Int}) \to [\mathsf{Float}] \\
&\texttt{normalize} = \texttt{pfold}\ (\lambda a\ r\ l \to (a\ /\ (\texttt{fromIntegral}\ l)) : r)\ (\lambda l \to [\,])
\end{aligned}$$

Finally, Fernandes et al. (2007) provide the following semantic equivalence, intended to be used as a fusion rule from left to right:

$$\begin{aligned}
\texttt{pfold}\ h_1\ h_2\ (\texttt{buildp}\ g\ u) \ \ = \ \ &\textbf{let}\ b = g\ (\lambda a\ r \to h_1\ a\ r\ z)\ (h_2\ z)\ u \\
&\qquad\ z = \texttt{snd}\ b \\
&\textbf{in}\ \ \texttt{fst}\ b
\end{aligned} \tag{9.4}$$

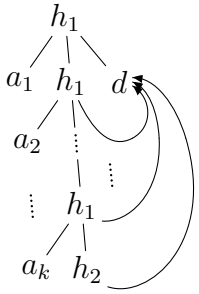
Note the circular dependency between b and z in the right-hand side, preventing use of the rule in an eagerly evaluated language. That it is correct in Haskell, at least in a naive setting, can be explained as follows.

Let $g :: \textbf{forall}\ \beta.\ (\tau_1 \to \beta \to \beta) \to \beta \to \tau_2 \to (\beta, \tau_3)$ for concrete types τ_1, τ_2, and τ_3. Intuitively, all that g can do, given arguments c, n, and u, is to return a pair of which the first component is $c\ a_1\ (\cdots\ (c\ a_k\ n)\cdots)$ for some $k \geq 0$ and $a_1, \ldots, a_k :: \tau_1$ and of which the second component is some $d :: \tau_3$, where all of k, $a_1, \ldots, a_k$, and d may depend on, and only on, the provided $u :: \tau_2$. For example, it is not possible that d depends on n, because the type of n is unknown inside g, and so g is very limited in what it can do with n. In graphical form:

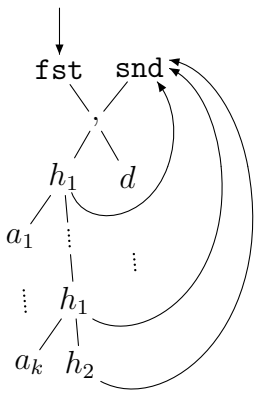


where the shading indicates which output parts can depend on which input.

Now let us have a look at the left- and right-hand sides of the `pfold`/`buildp`-rule for a fixed such g and fixed h_1, h_2, and u. On the left, `pfold` h_1 h_2 (`buildp` g u) unfolds to **case** g (:) [] u **of** $(as, z) \to$ `foldr` $(\lambda a\ r \to h_1\ a\ r\ z)$ $(h_2\ z)$ as, and thus to:



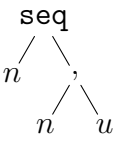
And on the right, the circular bindings in the **let**-block lead to a cyclic structure:



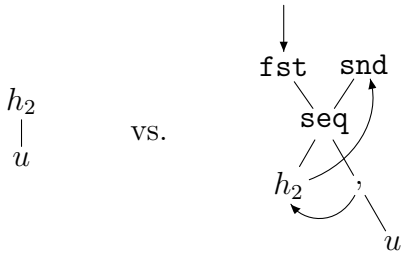
with the arrow at the top indicating where evaluation is to be started.

Looking at the two graphs just shown, it becomes clear why the left- and right-hand sides of (9.4) are equivalent, and also why the circular dependency in the right-hand side is not harmful. However, the explanation just given depends on a simplified view on g, in particular on the assumption that g makes no use of `seq`. So what if, instead, g had a form like the following one:

$$g \;=\; \lambda c\ n\ u \rightarrow$$



assuming $\tau_2 = \tau_3$? Then by comparing the left- and right-hand sides of (9.4) we get:



Clearly, the right-hand side will fail to terminate if h_2 is strict, because then the cycle through `seq`, h_2, and `snd` is truly vicious. But the left-hand side may very well terminate even for strict h_2, given that $h_2\ u \neq \bot$ is very well possible even then. This shows that (9.4) is not anymore a semantic equivalence in the presence of `seq`.

To avoid the above problem, we can demand that $h_2\ \bot \neq \bot$. Additionally, a similar restriction needs to be imposed on h_1 as well. Then, the paper reported on here (Voigtländer 2008c) proves that (9.4) is indeed a semantic equivalence provided $h_1\ \bot\ \bot\ \bot \neq \bot$ and $h_2\ \bot \neq \bot$. In addition to this full correctness result for a more realistic language setting than considered by Fernandes et al. (2007), we also proved that even in the presence of general recursion and selective strictness the partial correctness result

$$\begin{array}{rcl} \texttt{pfold}\ h_1\ h_2\ (\texttt{buildp}\ g\ u) & \sqsupseteq & \textbf{let}\ b = g\ (\lambda a\ r \rightarrow h_1\ a\ r\ z)\ (h_2\ z)\ u \\ & & \quad\ \ z = \texttt{snd}\ b \\ & & \textbf{in}\ \ \texttt{fst}\ b \end{array}$$

holds, without any preconditions. In doing so, the proof development approach described in Section 9.1 was very useful, in particular that the employed free theorems generator is able to automatically derive inequational free theorems as well.

Following this careful semantic account of circular `pfold`/`buildp`-fusion, an alternative rule is invented that trades circularity for higher-orderedness and thus attains better semantic properties, as well as becoming usable again, in principle at least, in an eagerly evaluated language.

9.3 Higher-Order Short Cut Fusion

The idea to replace circular definitions by higher-order ones is a classical one. For example, Pettorossi and Proietti (1987) describe it using the terminology of import and export of information. When applying it to `pfold`/`buildp`-fusion, we clearly want to preserve the advantages of the circular rule such as elimination of the intermediate list and effective handling of the additional result produced by `buildp` and used by `pfold`. It turns out that this is indeed possible, as follows:

$$\begin{aligned}\texttt{pfold}\ h_1\ h_2\ (\texttt{buildp}\ g\ u) \quad = \quad & \textbf{let}\ b = g\ (\lambda a\ r\ z \to h_1\ a\ (r\ z)\ z)\ (\lambda z \to h_2\ z)\ u \\ & \textbf{in}\ \ (\texttt{fst}\ b)\ (\texttt{snd}\ b)\end{aligned}$$

Note that there is no circular dependency in the right-hand side, and that the rule is higher-order in the sense that the first component of the tuple computed in b will be a function of type $\tau_3 \to \tau$ for whatever type τ the corresponding component of b in (9.4) has.

For examples of the new rule in action we refer the reader to the full paper (Voigtländer 2008c). Here we only mention that, crucially, we have proved that even in the presence of general recursion and selective strictness the above semantic equivalence holds unconditionally. Moreover, the full paper weighs the circular and higher-order flavors of `pfold`/`buildp`-fusion against each other as regards their sharing behavior. It turns out that the circular version fares better here. In the presence of the so-called *full laziness transformation* (Peyton Jones and Lester 1991) it can avoid certain recomputations that may surface when applying the higher-order version. A "repair" for the higher-order fusion rule is given as well, but in order to be effective requires slight adaptations to either or both of the functions `pfold` and `buildp`.

Finally, we note that Pardo et al. (2009) have recently taken our higher-order `pfold`/`buildp`-rule and generalized it to other algebraic data types than lists, as well as transferred it (and circular short cut fusion) to a setting of programs involving monads.

9.4 `foldr`/`build` and `destroy`/`unfoldr`, Revisited

The experiences made with the higher-order `pfold`/`buildp`-rule, in particular that it holds unconditionally as a semantic equivalence even when `seq` is present, have led us to revisit the veteran `foldr`/`build`- and `destroy`/`unfoldr`-rules discussed in Sections 3.2 and 8.4. And indeed, the realization that the good semantic properties of the higher-order rule stem from the fact that the "extra" λ-abstractions over z prevent g from encountering a $\bot$-value when combining its arguments, even though h_1 and/or h_2 might very well contain or produce such values, motivates a higher-order variation of the `foldr`/`build`-rule which can be proved fully correct without preconditions.

In an attempt to make the resulting rule also pragmatically effective, rather than just semantically correct, we "defunctionalize" it by performing an indirection via an extra, and almost trivial, data type instead of via the protective λ-abstractions. Whether this will ultimately lead to a practical and robust fusion framework is still unclear and depends on continued effort to be invested in more low-level compiler optimizations (Peyton Jones 2007). For a corresponding variant of the `destroy`/`unfoldr`-rule initial experiments seem quite promising. For more details, we again refer the reader to the full paper (Voigtländer 2008c).

Chapter 10

Moving to Operational Semantics

This chapter is based on the following two papers:

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

- P. Johann and J. Voigtländer.
  A Family of Syntactic Logical Relations for the Semantics of Haskell-like Languages.
  Information and Computation, Special Issue "Structural Operational Semantics", guest editors: R. van Glabbeek and P.D. Mosses. Elsevier, 2008.
  (Johann and Voigtländer 2008)

It gives a very high-level overview of our work on putting free theorems for Haskell on a firm theoretical basis. We do not go into formal detail here, instead letting the papers speak for themselves in this regard.

10.1 A Core Calculus for Haskell

In Section 8.3 we have reported on our study of free theorems for a language setting including general recursion and selective strictness, and thus for more realistic situations than were considered previously. That work was performed by using a denotational semantics framework (Schmidt 1986), which seemed appropriate as it allowed a rather intuitive approach to the problems at hand. However, one issue with this is that there is not really a "standard denotational model for Haskell" anywhere in the literature. Even the language specification (Peyton Jones 2003) is remarkably silent on how the mathematical meaning of a given program is to be determined. On the other hand, one could say that there *is* a mostly agreed upon

operational model for Haskell somewhat hidden inside the language implementations. Hence, in follow-on work we have studied the theoretical foundations of free theorems in an operational semantics framework. Specifically, we have dealt with an extension of the pure polymorphic lambda-calculus that resembles the intermediate language Core used inside the Glasgow Haskell Compiler.

We do not model full Core, but even so we get closer to results that we can with a high degree of confidence claim to apply to real Haskell. For example, in favor of an intuitive exposition, Johann and Voigtländer (2004, 2006) did not even formally describe the exact sublanguage of Haskell, with syntax and static and dynamic semantics, for which free theorems were developed. Rather, that line of work relied on a "commonly accepted, informal" understanding as actually used in much of the literature concerning the reasoning about Haskell programs. Also, type abstraction and type instantiation were left implicit, i.e., were not reflected in the syntax of expressions. This agrees with surface Haskell, but loses important structural information that needs to be used in the inductive proof of "every value (actually, every expression) of a concrete type is related to itself by the relational interpretation of that type". And finally, there were also some fundamental theoretical aspects that we could not fully satisfactorily address in the denotational setting.

The lambda-calculus studied in the first paper this chapter reports on (Voigtländer and Johann 2007) is given by the following typing axioms and rules, from which also the syntax of expressions and types can be read off:[1]

$$\Gamma, x :: \tau \vdash x :: \tau$$

$$\frac{\Gamma, x :: \tau \vdash M :: \tau'}{\Gamma \vdash (\lambda x :: \tau.M) :: \tau \to \tau'} \qquad \frac{\Gamma \vdash F :: \tau \to \tau' \qquad \Gamma \vdash A :: \tau}{\Gamma \vdash F\ A :: \tau'}$$

$$\frac{\alpha, \Gamma \vdash M :: \tau}{\Gamma \vdash (\Lambda\alpha.M) :: \forall\alpha.\tau} \qquad \frac{\Gamma \vdash G :: \forall\alpha.\tau}{\Gamma \vdash G_{\tau'} :: \tau[\tau'/\alpha]}$$

$$\Gamma \vdash \mathbf{nil}_\tau :: \tau\text{-}\mathit{list} \qquad \frac{\Gamma \vdash H :: \tau \qquad \Gamma \vdash T :: \tau\text{-}\mathit{list}}{\Gamma \vdash (H : T) :: \tau\text{-}\mathit{list}}$$

$$\frac{\Gamma \vdash L :: \tau\text{-}\mathit{list} \qquad \Gamma \vdash M_1 :: \tau' \qquad \Gamma, h :: \tau, t :: \tau\text{-}\mathit{list} \vdash M_2 :: \tau'}{\Gamma \vdash \mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\} :: \tau'}$$

$$\frac{\Gamma \vdash F :: \tau \to \tau}{\Gamma \vdash \mathbf{fix}(F) :: \tau} \qquad \frac{\Gamma \vdash A :: \tau \qquad \Gamma \vdash B :: \tau'}{\Gamma \vdash \mathbf{seq}(A, B) :: \tau'}$$

Note that type abstraction and instantiation are now explicit in the syntax of expressions as $\Lambda\alpha.M$ and $G_{\tau'}$, respectively, and that general recursion is captured via an explicit fixpoint primitive. As a sole, but representative, algebraic data type we include lists. Now, for example, the function `map` can be defined as the following expression M and then satisfies $\emptyset \vdash M :: \tau$, where $\emptyset$ is the empty typing environment

[1] Throughout, Γ is a typing environment of the form $\alpha_1, \ldots, \alpha_n, x_1 :: \tau_1, \ldots, x_m :: \tau_m$.

and $\tau = \forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha\text{-}list \to \beta\text{-}list$:

$$\begin{array}{l}\mathbf{fix}(\lambda m :: \tau.\Lambda\alpha.\Lambda\beta.\lambda h :: \alpha \to \beta.\lambda l :: \alpha\text{-}list.\mathbf{case}\ l\ \mathbf{of}\ \{\mathbf{nil} \quad \Rightarrow \mathbf{nil}_\beta; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad a : as \Rightarrow (h\ a) : (m_{\alpha\beta}\ h\ as)\})\end{array}$$

10.2 A Small-Step Operational Semantics

We provide a simple stack-based abstract machine for our calculus. First, a subset of the set of all (well-typed) expressions is identified as *values* by the following grammar:

$$V ::= \lambda x :: \tau.M \mid \Lambda\alpha.M \mid \mathbf{nil}_\tau \mid M : M$$

Then, redex/reduct-pairs are defined as follows:

$$\begin{array}{rll}(\lambda x :: \tau.M)\ A & \rightsquigarrow M[A/x] & \\ (\Lambda\alpha.M)_\tau & \rightsquigarrow M[\tau/\alpha] & \\ \mathbf{case}\ \mathbf{nil}_\tau\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\} & \rightsquigarrow M_1 & \\ \mathbf{case}\ H : T\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\} & \rightsquigarrow M_2[H/h, T/t] & \\ \mathbf{fix}(F) & \rightsquigarrow F\ \mathbf{fix}(F) & \\ \mathbf{seq}(A, B) & \rightsquigarrow B & \text{if } A \text{ is a value}\end{array}$$

We also need a notion of reduction in context, so a grammar for *stacks* of *evaluation frames* is given as follows:

$$\begin{array}{l}S ::= Id \mid S \circ E \\ E ::= (-\ M) \mid\ -_\tau\ \mid (\mathbf{case} - \mathbf{of}\ \{\mathbf{nil} \Rightarrow M;\ x : x \Rightarrow M\}) \mid \mathbf{seq}(-, M)\end{array}$$

and the small-step transition relation is then defined by:

$$\begin{array}{rll}(S, E\{M\}) & \rightarrowtail (S \circ E, M) & \text{if } M \text{ is not a value} \\ (S \circ E, M) & \rightarrowtail (S, E\{M\}) & \text{if } M \text{ is a value} \\ (S, M) & \rightarrowtail (S, M') & \text{if } M \rightsquigarrow M'\end{array}$$

where the notation $E\{M\}$ stands for the result of replacing '$-$' by the expression M in the evaluation frame E. Intuitively, the first two transition rules navigate an expression to detect the next redex to be reduced, while the third rule performs a small-step reduction in a given evaluation context. Note that $\rightarrowtail$ is deterministic, but not terminating (due to **fix**). An expression is said to evaluate to a value, denoted $M \Downarrow V$, if $(Id, M) \rightarrowtail^* (Id, V)$ for the empty stack Id and the reflexive, transitive closure $\rightarrowtail^*$ of $\rightarrowtail$. If for a given expression M there exists such a value V, then M is said to converge, $M\Downarrow$, otherwise to diverge, $M\Uparrow$.

Two expressions are considered to be semantically equivalent if and only if they lead to the same overall observable behavior, convergence or divergence, whenever putting them in some larger, and arbitrary, program context. The aim of the work reported on in this chapter was to precisely characterize this notion of semantic equivalence, and/or a corresponding notion of semantic approximation, at a given

type by an inductively generated relational interpretation of that type à la Section 8.2. The desired "every expression is related to itself by the relational interpretation of its type" is simply a special case of this, given that semantic equivalence and semantic approximation are reflexive notions by nature.

10.3 Characterization Results and Extensions

The technical approach we used is very close to the one that was pioneered by Pitts (2000, 2005) for dealing with general recursion in either purely strict or purely nonstrict calculi, but without taking selective strictness into account. Indeed, in a precursor work (Voigtländer and Johann 2006) to that reported on here we showed how to extend Pitts' results to selective strictness à la `seq`, i.e., to the calculus and semantics just presented in the previous two sections. Of course, this crucially depended on the insights regarding `seq` and necessary adaptations to the process of interpreting types as relations reported on in Section 8.3. We transferred the modifications found to work in the denotational setting to the operational setting and established their proper interaction with Pitts' machinery, as well as formal properties specific to selective strictness. This led to a "`seq`-aware" version of Pitts' characterization result for semantic equivalence.

In the journal version (Voigtländer and Johann 2007) we then make the step to an "inequational" setting by performing the break of symmetry also mentioned in Section 8.3. This focus on semantic approximation, rather than equivalence, again allows a finer-grained analysis of computational behavior than a fully symmetric treatment does. As one example application we put those results from Section 8.4 relating to `foldr/build`-fusion on a firmer theoretical basis than available before.

In a certain sense, the results reported on in this chapter so far really only reconfirm what we already learned in the less completely formalized earlier studies. No radical revision of our perspective on free theorems for Haskell became necessary, and while the full proofs do contain some interesting technical details, they are also long and sometimes tedious. But still, that was effort well spent. By showing that our initial intuitive approach can indeed be embellished into a fully formal account, the former attains justification. So we, and others, can safely continue to use it in applications such as those presented in earlier chapters.

Also, we expanded on the operational semantics treatment to push yet closer to Haskell reality. One deviation of the calculus shown in Section 10.1 from the Glasgow Haskell Compiler's Core language is that it provides no sharing construct. Thus, when expressing the function `foldl'` from page 49 we have to write it as:

$$\begin{array}{l}\mathbf{fix}(\lambda g :: \forall\alpha.\forall\beta.(\beta \to \alpha \to \beta) \to \beta \to \alpha\text{-}list \to \beta.\\ \quad \Lambda\alpha.\Lambda\beta.\lambda f :: \beta \to \alpha \to \beta.\lambda b :: \beta.\lambda l :: \alpha\text{-}list.\\ \quad\quad \mathbf{case}\ l\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow b;\ a : as \Rightarrow \mathbf{seq}(f\ b\ a, g_{\alpha\beta}\ f\ (f\ b\ a)\ as)\})\end{array}$$

Note the duplication of the subexpression $f\ b\ a$. To avoid this, we have in later

work replaced

$$\frac{\Gamma \vdash A :: \tau \qquad \Gamma \vdash B :: \tau'}{\Gamma \vdash \mathbf{seq}(A, B) :: \tau'}$$

by

$$\frac{\Gamma \vdash A :: \tau \qquad \Gamma, x :: \tau \vdash B :: \tau'}{\Gamma \vdash \mathbf{let!}\ x = A\ \mathbf{in}\ B :: \tau'}$$

Thus, we can express `foldl'` more naturally as follows:

$$\begin{array}{l}\mathbf{fix}(\lambda g :: \forall\alpha.\forall\beta.(\beta \to \alpha \to \beta) \to \beta \to \alpha\text{-}list \to \beta.\\ \quad \Lambda\alpha.\Lambda\beta.\lambda f :: \beta \to \alpha \to \beta.\lambda b :: \beta.\lambda l :: \alpha\text{-}list.\\ \quad\quad \mathbf{case}\ l\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow b;\ a : as \Rightarrow \mathbf{let!}\ b' = f\ b\ a\ \mathbf{in}\ g_{\alpha\beta}\ f\ b'\ as\})\end{array}$$

More fundamentally, though, the second paper this chapter reports on (Johann and Voigtländer 2008) introduces a Haskell-like error primitive with typing axiom

$$\Gamma \vdash \mathbf{error}_\tau(i) :: \tau$$

where the number i represents a descriptive argument. The motivation is to appropriately distinguish different causes of program failure, e.g., divergence versus failed pattern-matching versus arithmetic errors, because such distinctions are significant in practice. In denotational terms, we do not want to conflate all possible erroneous behaviors into a single $\bot$.

In the operational semantics we then have additional possible behaviors beside $M{\Downarrow}$ and $M{\Uparrow}$, namely $M{\not\downarrow}i$ if there is a transition sequence from M in the context of the empty stack to an $\mathbf{error}(i)$-call in the context of an arbitrary stack. Actually, in an orthogonal drive towards implementation reality, the transition relation itself is also changed to now work over untyped expressions. To this end, a type-erasure transformation is defined that drops the type annotations in the binding occurrences of variables in λ-abstractions, eliminates all type abstractions and instantiations, omits the type subscripts of **nil** and **error**, and leaves the input expression otherwise unchanged. The stack-based abstract machine is correspondingly changed, in particular by dropping the value form $\Lambda\alpha.M$, the evaluation frame form $-_\tau$, and the redex/reduct-pair definition describing the interaction of those two. Moreover, the evaluation frame form $\mathbf{seq}(-, M)$ is replaced by $\mathbf{let!}\ x = -\ \mathbf{in}\ M$ and $\mathbf{seq}(A, B) \rightsquigarrow B$ by $\mathbf{let!}\ x = A\ \mathbf{in}\ B \rightsquigarrow B[A/x]$, still only for A being a value.

The switch to a type-erasing semantics is important to more faithfully model actual implementations, in particular that type abstraction and instantiation carry no computational content. It also rectifies a mismatch between the original denotational treatment of Johann and Voigtländer (2004) and the first operational treatment of Voigtländer and Johann (2007) as pertains the relational interpretation of $\forall$-types. On the technical side, the interaction between the low-level semantics definition on untyped expressions and the intended reasoning on a higher, typed level poses an interesting challenge.

One final aspect of the theory developed in the second paper (Johann and Voigtländer 2008) we want to report on here is that it is built around a very abstract notion of relating the observable behavior of computations. Clearly, once

we semantically distinguish different failure causes, their relative definedness needs to be considered. Many potential choices arise here, such as considering nonterminating programs as strictly less defined than programs that terminate via **error**, considering these situations incomparable, or adding even more structure, among different kinds of explicit **error**s or with respect to relating such finite failures and normally terminating programs. In order not to predetermine such design decisions, which may very well depend on the usage scenario for free theorems and program transformations based on them, we parametrize our whole development by a preorder embodying the various possible choices. A further advantage of this abstract approach is that we can deal with semantic equivalence and (either direction of) semantic approximation in a unified manner.

When, for example, investigating the correctness of `foldr`/`build`-fusion, we do not anymore need to prove two separate results for when the transformed program is at least as ($\sqsubseteq$) or at most as ($\sqsupseteq$) defined as the original one. Instead, a single proof parametrized by an abstract preorder suffices. Concrete results for semantic equivalence and different directions of semantic approximation, and for different concepts of relating failure causes, can then be read off by just instantiating this parameter in different ways. This high reuse potential is capitalized on in a concrete case study (Johann and Voigtländer 2008, Section 6).

Chapter 11

Conclusion

The story of type-based thinking and reasoning about programs is only at its beginning. We expect to see it having a big impact also on practical software construction in the coming years. One possible scenario is that ideas and features first developed and studied in the context of perceivedly mere academic languages and type systems continue to slowly trickle into the mainstream, as has happened with the inclusion of parametric polymorphism into Java and C#, and of first-class functions (i.e., λ-abstractions) into the latter. Scala (Odersky 2006) is an experiment into how a resulting hybrid object-oriented/functional language with strong ties to type theory research may look like. Another possible scenario is a more radical paradigm shift. Microsoft's LINQ project, which directly builds on Haskell work, demonstrates that this is not as absurd a prospect as it may seem (Meijer 2008).

More generally, formal methods are definitely gaining tract in software engineering. Extended static checking (Flanagan et al. 2002), contract-oriented design (Meyer 1992, Barnett et al. 2006), and interactive proof of the adherence of programs to logical specifications (Guttag et al. 1993) or refinement of such specifications into programs (Abrial 1996), all are approaches to improve the quality of software, with increasing degrees of ambition regarding the extent of provided guarantees. In a suitably controlled setting they could certainly profit from techniques like free theorems. For side-effect-free code parts that essentially correspond to pure functions, say, statements derived from sufficiently expressive types could be used to automatically discharge proof obligations, or auxiliary lemmas could be generated for potential use in interactive proofs.

And Haskell does not even represent the upper end of the expressiveness spectrum of type systems. Currently, one could say that the object-oriented research community looks towards Haskell when seeking a yardstick of how strong types could be (Wadler 2006, Dominus 2008). But there is more to discover beyond Haskell. For example, *dependent types* in languages like Coq (Bertot and Castéran 2004), Epigram (McKinna 2006), and Agda (Norell 2007) capture semantically richer properties of functions, and could thus lead to new heights in expressiveness for applications in the spirit of free theorems.

Less speculatively, there are still a number of interesting directions to pursue that are more bound to the particular niche of type-based reasoning in which we have been working as reported in this thesis. For example, the applications from Chapters 4 and 5 could be further developed, and similar applications outside the core area of programming language research be identified. On the theory side, there are still features of Haskell that are not fully taken care of in current formal accounts. For example, the interplay between free theorems and type classes, as used for some of the results in Chapter 5 and (even for type *constructor* classes) in Section 6.4 and Chapter 7, has been described by Wadler (1989, Section 3.4) in principle, but has never really been formalized. Also, the semantic treatment of finite failure reported on in Section 10.3 still differs from the reality in Haskell implementations which actually come with a certain degree of nondeterminism in error handling (Peyton Jones et al. 1999). In recent work (Stenger and Voigtländer 2008) we are approaching this issue. We would also like to investigate whether it is possible to develop a theory of free theorems that provides statements not only about semantic equivalences or approximations, but also about more intensional aspects such as the relative efficiency of two expressions. For this, we expect to benefit from machinery developed in the context of "improvement theories" (Sands 1996, Moran and Sands 1999, Gustavsson and Sands 2001).

We should also mention that despite our stance on the importance of reconciling free theorems with the reality of Haskell and its advanced features, some of our applications have been examined only in the "naive language setting" so far. Notably, the results reported on in Chapters 4, 5, and 7 currently ignore the presence of `seq`. But transferring them to a more accurate model of Haskell should now be more or less routine tasks. More tool support would certainly be welcome. As a possible implementation strategy for such further mechanization effort we intend to look at that employed by Silva and Oliveira (2008).

Bibliography

J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981. DOI: 10.1145/319628.319634.

M. Barnett, B.-Y.E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 364–387. Springer-Verlag, 2006. DOI: 10.1007/11804192_17.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.

G.E. Blelloch. Prefix sums and their applications. In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufmann, 1993.

S. Böhme. Much ado about two. Formal proof development. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/MuchAdoAboutTwo.shtml`, 2007.

R.P. Brent and H.T. Kung. The chip complexity of binary arithmetic. In *ACM Symposium on Theory of Computing, Proceedings*, pages 190–200. ACM Press, 1980. DOI: 10.1145/800141.804666.

L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985. DOI: 10.1145/6041.6042.

O. Chitil. Type inference builds a short cut to deforestation. In *International Conference on Functional Programming, Proceedings*, volume 34(9) of *SIGPLAN Notices*, pages 249–260. ACM Press, 1999. DOI: 10.1145/317636.317907.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *International Conference on Functional Programming, Proceedings*, volume 42(9) of *SIGPLAN Notices*, pages 315–326. ACM Press, 2007. DOI: 10.1145/1291151.1291199.

M.J. Dominus. Atypical types (Invited talk). At *Object-Oriented Programming, Systems, Languages, and Applications*, 2008.

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. DOI: 10.1145/1291201.1291216.

C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation, Proceedings*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM Press, 2002. DOI: 10.1145/543552.512558.

J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007. DOI: 10.1145/1232420.1232424.

N. Ghani and P. Johann. Short cut fusion of recursive programs with computational effects. In *Trends in Functional Programming, Draft Proceedings*, 2008.

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. DOI: 10.1145/165180.165214.

T. Golding. *Professional .NET 2.0 Generics*. Wrox, 2005.

J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *International Conference on Functional Programming, Proceedings*, volume 36(10) of *SIGPLAN Notices*, pages 265–276. ACM Press, 2001. DOI: 10.1145/507546.507667.

J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

P. Hudak, R.J.M. Hughes, S.L. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *History of Programming Languages, Proceedings*, pages 12-1–12-55. ACM Press, 2007. DOI: 10.1145/1238844.1238856.

R.J.M. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986. DOI: 10.1016/0020-0190(86)90059-1.

G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002. DOI: 10.1023/A:1022982420888.

P. Johann and J. Voigtländer. Free Theorems in the Presence of seq. In *Principles of Programming Languages, Proceedings*, volume 39(1) of *SIGPLAN Notices*, pages 99–110. ACM Press, 2004. DOI: 10.1145/982962.964010.

P. Johann and J. Voigtländer. The Impact of seq on Free Theorems-Based Program Transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.

P. Johann and J. Voigtländer. A Family of Syntactic Logical Relations for the Semantics of Haskell-like Languages. *Information and Computation*, 2008. DOI: 10.1016/j.ic.2007.11.009.

M.P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, Tutorial Text*, volume 925 of *LNCS*, pages 97–136. Springer-Verlag, 1995a.

M.P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995b.

D.E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980. DOI: 10.1145/322217.322232.

J. Launchbury and S.L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995. DOI: 10.1007/BF01018827.

D. Leivant. Polymorphic type inference. In *Principles of Programming Languages, Proceedings*, pages 88–98. ACM Press, 1983. DOI: 10.1145/567067.567077.

S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In *Principles of Programming Languages, Proceedings*, pages 333–343. ACM Press, 1995. DOI: 10.1145/199448.199528.

Y.-C. Lin and J.-W. Hsiao. A new approach to constructing optimal parallel prefix circuits with small depth. *Journal of Parallel and Distributed Computing*, 64(1): 97–107, 2004. DOI: 10.1016/j.jpdc.2003.09.004.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming, Proceedings*, volume 42(9) of *SIGPLAN Notices*, pages 47–58. ACM Press, 2007. DOI: 10.1145/1291220.1291162.

J. McKinna. Why dependent types matter (Invited talk). In *Principles of Programming Languages, Proceedings*, volume 41(1) of *SIGPLAN Notices*, pages 1–14. ACM Press, 2006. DOI: 10.1145/1111320.1111038.

E. Meijer. Fundamentalist functional programming (Keynote address). In *Generative Programming and Component Engineering, Proceedings*, page 99. ACM Press, 2008. DOI: 10.1145/1449913.1449929.

B. Meyer. Applying "Design by contract". *IEEE Computer*, 25(10):40–51, 1992. DOI: 10.1109/2.161279.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML — Revised*. MIT Press, 1997.

E. Moggi. Notions of computation and monads. *Information and Computation*, 93 (1):55–92, 1991. DOI: 10.1016/0890-5401(91)90052-4.

A.K. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Principles of Programming Languages, Proceedings*, pages 43–56. ACM Press, 1999. DOI: 10.1145/292540.292547.

M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly, 2006.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

M. Odersky. The Scala experiment: Can we provide better language support for component systems? (Invited talk). In *Principles of Programming Languages, Proceedings*, volume 41(1) of *SIGPLAN Notices*, pages 166–167. ACM Press, 2006. DOI: 10.1145/1111320.1111052.

B. O'Sullivan, D.B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly, 2008.

A. Pardo, J.P. Fernandes, and J. Saraiva. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *Partial Evaluation and Program Manipulation, Proceedings*. ACM Press, 2009.

A. Pettorossi and M. Proietti. Importing and exporting information in program development. In *Partial Evaluation and Mixed Computation, Proceedings*, pages 405–425. North-Holland, 1987.

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

S.L. Peyton Jones. Call-pattern specialisation for Haskell programs. In *International Conference on Functional Programming, Proceedings*, volume 42(9) of *SIGPLAN Notices*, pages 327–337. ACM Press, 2007. DOI: 10.1145/1291220.1291200.

S.L. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5):479–506, 1991. DOI: 10.1002/spe.4380210505.

S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Principles of Programming Languages, Proceedings*, pages 71–84. ACM Press, 1993. DOI: 10.1145/158511.158524.

S.L. Peyton Jones, A. Reid, C.A.R. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Programming Language Design and Implementation, Proceedings*, volume 34(5) of *SIGPLAN Notices*, pages 25–36. ACM Press, 1999. DOI: 10.1145/301631.301637.

A.M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000. DOI: 10.1017/S0960129500003066.

A.M. Pitts. Typed operational reasoning. In B.C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 245–289. MIT Press, 2005.

J.C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Proceedings*, pages 408–423. Springer-Verlag, 1974.

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

D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2): 175–234, 1996. DOI: 10.1145/227699.227716.

D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

P.F. Silva and J.N. Oliveira. 'Galculator': Functional prototype of a Galois-connection based proof assistant. In *Principles and Practice of Declarative Programming, Proceedings*, pages 44–55. ACM Press, 2008. DOI: 10.1145/1389449.1389456.

J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(6):226–231, 1960.

F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. Technical Report TUD-FI08-08, Technische Universität Dresden, 2008.

C. Strachey. Fundamental concepts in programming languages. Lecture notes for a course at the *International Summer School in Computer Programming*, 1967. Reprint appeared in *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000. DOI: 10.1023/A:1010000313106.

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming, Proceedings*, volume 37(9) of *SIGPLAN Notices*, pages 124–132. ACM Press, 2002. DOI: 10.1145/583852.581491.

W. Swierstra and T. Altenkirch. Beauty in the beast: A functional semantics for the awkward squad. In *Haskell Workshop, Proceedings*, pages 25–36. ACM Press, 2007. DOI: 10.1145/1291201.1291206.

A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 306–313. ACM Press, 1995. DOI: 10.1145/224164.224221.

J. Voigtländer. Concatenate, Reverse and Map Vanish For Free. In *International Conference on Functional Programming, Proceedings*, volume 37(9) of *SIGPLAN Notices*, pages 14–25. ACM Press, 2002. DOI: 10.1145/583852.581481.

J. Voigtländer. Proving Correctness via Free Theorems: The Case of the destroy/build-Rule. In *Partial Evaluation and Semantics-Based Program Manipulation, Proceedings*, pages 13–20. ACM Press, 2008a. DOI: 10.1145/1328408.1328412.

J. Voigtländer. Much Ado about Two: A Pearl on Parallel Prefix Computation. In *Principles of Programming Languages, Proceedings*, volume 43(1) of *SIGPLAN Notices*, pages 29–35. ACM Press, 2008b. DOI: 10.1145/1328897.1328445.

J. Voigtländer. Semantics and Pragmatics of New Shortcut Fusion Rules. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, 2008c. DOI: 10.1007/978-3-540-78969-7_13.

J. Voigtländer. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008d. DOI: 10.1007/978-3-540-70594-9_20.

J. Voigtländer. Bidirectionalization for Free! In *Principles of Programming Languages, Proceedings*. ACM Press, 2009.

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. DOI: 10.1016/j.tcs.2007.09.014.

P. Wadler. Faith, evolution, and programming languages: from Haskell to Java to Links (Invited talk). In *Object-Oriented Programming, Systems, Languages, and Applications, Proceedings*, page 508. ACM Press, 2006. DOI: 10.1145/1176617.1176623.

P. Wadler. The concatenate vanishes. Note, University of Glasgow, 1987; revised, 1989.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989. DOI: 10.1145/99370.99404.

P. Wadler. The essence of functional programming (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 1992. DOI: 10.1145/143165.143169.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages, Proceedings*, pages 60–76. ACM Press, 1989. DOI: 10.1145/75277.75283.

For copyright reasons, the full papers are not appended to this electronic version of the thesis. Author versions are available online at:

- Much Ado about Two: A Pearl on Parallel Prefix Computation.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/popl202-voigtlaender.pdf`
- Bidirectionalization for Free!
  `http://wwwtcs.inf.tu-dresden.de/~voigt/popl09-2.pdf`
- Concatenate, Reverse and Map Vanish For Free.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/p114-voigtlaender.pdf`
- Asymptotic Improvement of Computations over Free Monads.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/mpc08.pdf`
- Free Theorems in the Presence of *seq*.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/p76-voigtlaender.pdf`
- The Impact of *seq* on Free Theorems-Based Program Transformations.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/seqFinal.pdf`
- Proving Correctness via Free Theorems: The Case of the `destroy/build`-Rule.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/pepm09-voigtlaender.pdf`
- Semantics and Pragmatics of New Shortcut Fusion Rules.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/flops.pdf`
- Selective strictness and parametricity in structural operational semantics, inequationally.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/TCS.pdf`
- A Family of Syntactic Logical Relations for the Semantics of Haskell-like Languages.
  `http://wwwtcs.inf.tu-dresden.de/~voigt/iandc.pdf`