Preliminary Proceedings of the

2001 ACM SIGPLAN Haskell Workshop (HW'2001)

Firenze, Italy
2nd September 2001

Ralf Hinze (editor)
Technical Report UU-CS-2001-23
Institute of Information and Computing Sciences
Utrecht University

***

Foreword

This volume contains the preliminary proceedings of the 2001 ACM SIGPLAN Haskell Workshop, which was held on 2nd September 2001 in Firenze, Italy. The final proceedings will published by Elsevier Science as an issue of Electronic Notes in Theoretical Computer Science (Volume 59).

The Haskell Workshop was sponsored by ACM SIGPLAN and formed part of the PLI 2001 colloquium on Principles, Logics, and Implementations of high-level programming languages, which comprised the ICFP/PPDP conferences and associated workshops. Previous Haskell Workshops have been held in La Jolla (1995), Amsterdam (1997), Paris (1999), and Montréal (2000).

The purpose of the Haskell Workshop was to discuss experience with Haskell, and possible future developments for the language. The scope of the workshop included all aspects of the design, semantics, theory, application, implementation, and teaching of Haskell. Submissions that discussed limitations of Haskell at present and/or proposed new ideas for future versions of Haskell were particularly encouraged. Adopting an idea from ICFP 2000, the workshop also solicited two special classes of submissions, application letters and functional pearls, described below.

Application Letters An application letter describes experience using Haskell to solve real-world problems. Such a paper may be shorter than a regular paper (but need not be), and and may be judged by interest of the application and novel use of Haskell.

Functional Pearls A functional pearl presents — using Haskell as a vehicle — an idea that is small, rounded, and glows with its own light. Such a paper may be shorter than a regular paper (but need not be), and may be judged by elegance of development and clarity of expression.

The workshop received a total of 23 submissions and after careful consideration the programme committee accepted 10 papers for presentation (6 regular, 4 functional pearls, and no application letters). Each programme committee member reviewed ten papers, possibly with the aid of an outside expert. Each paper was assigned to at least three reviewers. Final decisions were made during a virtual programme committee meeting. The selection was competitive: several good papers had to be rejected.

Ralf Hinze, Organizer and Chair
Utrecht, August 2001

Programme Committee

Acknowledgements

The programme committee thanks the following people for their assistance in evaluating the submissions:

Pablo Azero
Dennis Björklund
Magnus Carlsson
James Cheney
David Clarke
Iavor Diatchki
Jörgen Gustavsson
Thomas Hallgren
Bill Harrison
Gabriele Keller
David Lacey
Peter Ljunglöf
Andres Löh
Clare Martin
Shin-Cheng Mu
Johan Nordlander
Claudio Russo
Silvija Seres
Mark Shields
Fred Smith
Josef Svenningsson
Dave Walker
Steve Zdancewic

Special thanks are due to Andres Löh for his assistance in preparing the preliminary proceedings and to Betti Venneri for her help with organizing the workshop.

Contents

***

Functional Pearl

Derivation of a Carry Lookahead Addition Circuit

John O'Donnell [1,2]

Computing Science Department
University of Glasgow
Glasgow, United Kingdom

Gudula Rünger [3]

Fakultät für Informatik
Technische Universität Chemnitz
Chemnitz, Germany

Abstract

Using Haskell as a digital circuit description language, we transform a ripple carry adder that requires $O(n)$ time to add two n-bit words into an efficient carry lookahead adder that requires $O(\log n)$ time. The gain in speed relies on the use of parallel scan to calculate the propagation of carry bits efficiently. The main difficulty is that this scan cannot be parallelised directly since it is applied to a non-associative function. Several additional techniques are needed to circumvent the problem, including partial evaluation and symbolic function representation. The derivation given here provides a formal correctness proof, yet it also makes the solution more intuitive by bringing out explicitly each of the ideas underlying the carry lookahead adder.

1 Introduction

In this paper we use Haskell as a digital circuit description language in order to solve an important problem in hardware design: the transformation of a ripple carry adder that requires $O(n)$ time to add two n-bit words into a carry

[1] This work was supported in part by the British Council and the Deutsche Akademische Austauschdienst under the Academic Research Collaboration program.
[2] Email: `jtod@dcs.gla.ac.uk` Web: `www.dcs.gla.ac.uk/~jtod/`
[3] Email: `ruenger@informatik.tu-chemnitz.de`
Web: `www.tu-chemnitz.de/informatik/HomePages/PI/index.html`

lookahead adder, which needs only $O(\log n)$ time. This problem has great practical importance, since the clock speed of synchronous digital circuits is determined by the critical path depth, and an adder lies on the critical path in typical processor datapath architectures. In other words, by speeding up just an adder, which accounts for a few hundred logic gates, the speed of an entire chip with millions of gates can be improved.

The circuit that we design here is not new; it is related to (though different from) a circuit by Ladner and Fischer [7] (1980), and the particular variation that we develop is essentially the same as the one presented in the well known textbook on algorithms by Cormen, Leiserson and Rivest [3]. The original contributions of this paper include the formal specification, the correctness proof, and the derivation:

- Our derivation produces a precise specification of the circuit, which can be simulated or fabricated automatically. The earlier presentations give only examples of the circuit at particular word sizes, relying on the reader to figure out other cases. This can be surprisingly difficult, and is unsuitable for modern integrated circuit design, which is highly automated.
- The derivation produces a general solution that works on word size n for every natural number n.
- The carry lookahead adder is usually presented as a large and very complicated circuit, which is quite difficult to understand. In contrast, we explain it by going through a sequence of transformation steps. At each stage there is a specific technical problem to overcome and a clear strategy for solving it. This leads to a better understanding than contemplation of the final design, where several quite distinct ideas are mixed together and buried in a large network of logic gates.
- The derivation in this paper provides a correctness proof for the adder.

Although we do not claim the circuit derived here to be new, there is a sense in which it actually is new. The adders presented before operate only on fixed size words, but we derive a family of adders defined for every wordsize $n \in Nat$. For example, the adder described in [3] takes two 8-bit words and produces an 8-bit sum. *It does not work at all for any other word size.* Its time complexity is $O(1)$; indeed, it is meaningless to attribute a time complexity of $O(\log n)$ to an algorithm that lacks a parameter n.

Clearly the authors of the previous papers could have designed an adder at a different fixed word size, say 16. What they did not do was to design a general adder at size n, and there is a good reason: they did not use a formalism capable of expressing families of parameterised circuits.

In this paper, we use Hydra [10], a computer hardware description language (CHDL) embedded in Haskell. Two advantages of Hydra are central to the paper: it allows *circuit patterns* to be defined, allowing n-bit circuits, and it allows *formal equational reasoning* to be used in transforming circuits. The reader is assumed to be familiar with Haskell but not with Hydra, and the essential methods of functional hardware specification will be explained below. Links to further information on Hydra can be found on the web page for this paper:

`http://www.dcs.gla.ac.uk/~jtod/papers/2001_Adder/`

A huge benefit of CHDLs is their ability to define families of related circuits. Most CHDLs are based on imperative programming, but functional languages work far better for this application domain. In particular, this paper relies on three characteristic features of functional languages: (1) higher order functions express circuit patterns; (2) referential transparency supports equational reasoning; (3) strong typing allows the circuit types to be defined naturally, and also supports the wide variety of software tools provided by Hydra. Nonstrict semantics (lazy evaluation) is also essential to Hydra, although it happens not to be needed for the adder.

Even in a formal derivation, examples are helpful, and the reader is encouraged to experiment with a Haskell 98 program containing all the definitions in this paper. The program contains test drivers that run a number of examples, as well as comments explaining how to run it, and can be downloaded from the web page mentioned above.

A theme running through this paper is the distinction between *specification* and *implementation*. For a number of auxiliary definitions, as well as for the main result, the paper will begin with a clear specification and proceed to derive an efficient implementation. There is no need for the specification to be efficient, or for the implementation to be clear.

A related point is the distinction between *circuit specifications* and *computer programs*. The Hydra language is intended specifically for circuit design, and it is restricted to forms that correspond directly to circuits. The implementation of Hydra provides tools that will convert a circuit specification (including all the adders defined in this paper) into netlists. Hydra is implemented by embedding it in Haskell, so circuit specifications have the same syntax as Haskell. However, Hydra is not identical to Haskell, and a designer who forgets this may write a Haskell program that does not specify a digital circuit at all. This form of confusion has nothing to do with the design of Hydra or Haskell; similar problems arise with imperative CHDLs such as VHDL.

Section 2 defines precisely the problem to be solved, giving a formal specification of a binary addition circuit and also explaining how we will use Haskell to describe circuits. Section 3 introduces the combinators that will be used to specify circuit patterns, and Section 4 presents the standard ripple carry adder in this style, using a scan combinator to handle the carry propagation. The essential technique for speeding up the adder is parallel scan, which is derived formally in Section 5. However, it turns out that the particular scan used in the ripple carry adder cannot be implemented by the parallel scan algorithm because it uses a non-associative function. Section 6 solves that problem using partial evaluation. However, this introduces a new difficulty: the "circuit" now operates on functions as well as signals, and is no longer a circuit at all. Section 7 introduces a symbolic function representation, Section 8 introduces parallelism into the adder, Section 9 takes care of the final hardware details, and Section 10 concludes.

2 The Problem

A *signal* is a bit in a digital circuit; for the purposes of this paper a signal can be thought of as a value of type *Bool*. The function $\mathit{bit} :: \mathit{Signal}\ a \Rightarrow a \rightarrow \mathit{Nat}$ converts a bit value to its natural value, either 0 or 1.

A binary number is represented as a list of signals $[x_0, \ldots, x_{n-1}]$ that constitute an n-bit word, where x_0 is the most significant bit and x_{n-1} the least significant. The value represented by this word is $\mathit{bin}\ \mathit{xs}\ =\ \sum_{i=0}^{n-1} x_i 2^{n-1-i}$.

It is assumed throughout this paper that all lists have finite length. Some of the results need to be refined to handle infinite data structures, but issues of strictness are irrelevant to the derivation of the adder.

A binary adder takes a pair of n-bit words *xs* and *ys* and a carry input bit c, and it produces their sum, represented as a carry output bit c' and an n-bit sum *ss*. Instead of giving the adder two separate words *xs* and *ys*, it will receive a word $\mathit{zs} :: \mathit{Signal}\ a \Rightarrow [(a, a)]$ of pairs. The binary input words are then *map fst zs* and *map snd zs*. There are two reasons for choosing this organisation: it avoids the need for stating side conditions that *xs* and *ys* have the same length, and it simplifies the circuits we will define later. But we are not cheating—it is a standard technique in hardware design (called "bit slice" organisation) to zip the two words together in this way, because of exactly the same simplification to the design.

An adder is now defined to be any circuit with the right type that produces the right answer for arbitrary inputs.

Definition 2.1 (Adder) Let a be a signal type. An adder is a function *add* such that

$$\begin{array}{l}
\mathit{add}\ ::\ \mathit{Signal}\ a \Rightarrow a \rightarrow [(a,a)] \rightarrow (a, [a]),\\
\forall c :: a,\ \ \mathit{zs} :: [(a,a)]\ .\\
\qquad 2^n \cdot \mathit{bit}\ c'\ + \mathit{bin}\ \mathit{ss}\ =\ \mathit{bin}\ (\mathit{map}\ \mathit{fst}\ \mathit{zs}) + \mathit{bin}\ (\mathit{map}\ \mathit{snd}\ \mathit{zs}) + \mathit{bit}\ c\\
\text{where}\\
\qquad (c', \mathit{ss}) = \mathit{add}\ c\ \mathit{zs}\\
\qquad n\ =\ \mathit{length}\ \mathit{zs}\ =\ \mathit{length}\ \mathit{ss}.
\end{array}$$

Circuits will be specified in this paper using Hydra [10], a digital circuit specification language embedded within Haskell. Signals are defined as a type class that provides basic operations, such as the constant values *zero* and *one* and basic logic gates, including the inverter *inv*, the two and three input logical and-gates *and2*, *and3*, etc. Components are wired together by applying a circuit to its input signals. The values of signals will be denoted 0 and 1, although their internal representations may be different (e.g. *False* and *True*).

The majority and parity circuits provide simple examples of Hydra specifications, and they will be useful later for computing sums and carries. The *majority3* circuit takes three input signals, and returns logic 1 if two or more of the inputs are 1. The *parity3* circuit returns 1 if an odd number of the inputs are 1.

$$\begin{aligned}
&\mathit{majority3},\ \mathit{parity3}\ ::\ \mathit{Signal}\ a\ \Rightarrow a \to a \to a \to a\\
&\mathit{majority3}\ a\ b\ c\ =\ \mathit{or3}\ (\mathit{and2}\ a\ b)\ (\mathit{and2}\ a\ c)\ (\mathit{and2}\ b\ c)\\
&\mathit{parity3}\ a\ b\ c\ =\\
&\quad \mathit{or2}\ (\mathit{and2}\ (\mathit{inv}\ a)\ (\mathit{xor2}\ b\ c))\\
&\qquad\quad (\mathit{and2}\ a\ (\mathit{inv}\ (\mathit{xor2}\ b\ c)))
\end{aligned}$$

Another standard circuit is the multiplexor, which takes a control (or address) bit a, and uses it to select data inputs, which is then output. We can specify the behaviour of the 1-bit multiplexor as

$$\mathit{mux1}\ a\ x\ y\ =\ \textbf{if}\ a = \mathit{zero}\ \textbf{then}\ x\ \textbf{else}\ y$$

This specification is easy to understand, but it isn't a circuit, since if-then-else expressions are not logic gates. A central problem in circuit design is finding a way to make the available components meet a specification, and there are methodical techniques for doing this. It is straightforward to verify that the following circuit satisfies the specification of the multiplexor:

$$\begin{aligned}
&\mathit{mux1}\ ::\ \mathit{Signal}\ a \Rightarrow a \to a \to a \to a\\
&\mathit{mux1}\ a\ x\ y\ =\ \mathit{or2}\ (\mathit{and2}\ (\mathit{inv}\ a)\ x)\ (\mathit{and2}\ a\ y)
\end{aligned}$$

Two *mux1* circuits can be used to define *mux2*, which uses two address bits to select one of four data inputs. This is a typical example of the hierarchical design style used in Hydra. The *mux2* will be needed in Section 9.

$$\begin{aligned}
&\mathit{mux2}\ ::\ \mathit{Signal}\ a \Rightarrow (a,a) \to a \to a \to a \to a \to a\\
&\mathit{mux2}\ (a,b)\ w\ x\ y\ z\ =\ \mathit{mux1}\ a\ (\mathit{mux1}\ b\ w\ x)\ (\mathit{mux1}\ b\ y\ z)
\end{aligned}$$

Many basic definitions and lemmas from functional programming will be used later in the paper; some of them are summarised in Table 1.

3 Map, Fold, and Scan

The circuits we will be developing have a regular structure well suited for VLSI layout. It is best to avoid a style of specification where each component is mentioned explicitly—that would lead to a verbose design valid for only one

$$\begin{array}{rcll}
drop\ 0\ xs & = & xs & (1)\\
drop\ (i+1)\ (x:xs) & = & drop\ i\ xs & (2)\\
[e \mid [\,]] & = & [\,] & (3)\\
[f\ i \mid i \leftarrow x:xs] & = & f x : [f i \mid i \leftarrow xs] & (4)\\
[\ f\ i \mid i \leftarrow [a \mathinner{..} b]] & = & [f\ (i+k) \mid i \leftarrow [a-k \mathinner{..} b-k]] & (5)\\
[\ f\ i \mid i \leftarrow [a \mathinner{..} c] & = & [f\ i \mid i \leftarrow [a \mathinner{..} b] +\!\!+ [fi \mid i \leftarrow [b \mathinner{..} c] & (6)\\
[x] = [y] & \Leftrightarrow & x = y & (7)
\end{array}$$

Table 1
Basic Lemmas

word size, but we are seeking a concise yet generic adder specification that works for all word sizes. Furthermore, experience shows that mentioning all the bits explicitly with indices leads to cumbersome notation that is poorly suited for circuit transformation and optimisation.

The best approach is to use a higher order function—a combinator—to express the pattern by which the building blocks are composed into the full circuit. This paper uses a variety of combinators that fall into four families: *map*, *fold*, *scan*, and *sweep*. This section discusses the first three, and *sweep* is presented in Section 5.2.

The standard function *map* describes a circuit consisting of a row of identical components.

$$\begin{array}{ll}
map\ ::\ (a \to b) \to [a] \to [b] & \\
map\ f\ [\,]\ =\ [\,] & (8)\\
map\ f\ (x:xs)\ =\ f\ x\ :\ map\ f\ xs & (9)
\end{array}$$

Each component f takes an input x_i and produces an output $y_i = f\ x_i$. The entire circuit *map f* takes a word *xs* and produces an output word *ys* = *map f xs*. Normally, when f is viewed as a digital circuit, every application of f takes the same time, regardless of the input value, so *map f* takes time $O(1)$ for an n-bit word. If the component circuit f takes two inputs, then the circuit is expressed by *zipWith*.

The carry propagation across a sequence of bit positions is expressed by the standard *foldr* function:

$$\begin{array}{ll}
foldr\ ::\ (b \to a \to a) \to a \to [b] \to a & \\
foldr\ f\ a\ [\,]\ =\ a & (10)\\
foldr\ f\ a\ (x:xs)\ =\ f\ x\ (foldr\ f\ a\ xs) & (11)
\end{array}$$

A related function that will be needed later is *foldr1*, which omits the accumulator parameter a, so it is defined only if the list argument is nonempty.

The following equation states a useful relationship between *foldr* and *foldr1*:

$$\mathit{foldr1}\ f\ (\mathit{xs}{+\!\!+}[a]) \ = \ \mathit{foldr}\ f\ a\ \mathit{xs} \tag{12}$$

However, it isn't enough just to compute the carry output from one bit position—in order to compute the sum bits, we need the carry inputs to all the positions. Therefore the circuit really needs to compute the carry propagation across all possible subfields, starting from the right.

The *indices* function builds a list of indices of elements of the list.

$$\begin{aligned}
&\mathit{indices} \ :: \ \mathit{Int} \to [a] \to [\mathit{Int}]\\
&\mathit{indices}\ i\ [\,] \ = \ [\,]\\
&\mathit{indices}\ i\ (x : \mathit{xs}) \ = \ i \ : \ \mathit{indices}\ (i+1)\ \mathit{xs}
\end{aligned}$$

The *wscanr* combinator (word scan from the right) computes a list of all the partial folds. It is specified using a list comprehension showing the form of each element of the list; this form makes clear that a scan is a list of folds, and it is well suited for derivations and proofs using equational reasoning.

$$\begin{aligned}
&\mathit{wscanr}\ f\ a\ \mathit{xs} \ = \\
&\qquad [\mathit{foldr}\ f\ a\ (\mathit{drop}\ (i+1)\ \mathit{xs}) \ \mid \ i \leftarrow \mathit{indices}\ \mathit{xs}]
\end{aligned} \tag{13}$$

Since all the list arguments to fold and scan are assumed to be finite in this paper, *indices xs* $= [0\,..\ \mathit{length}\ \mathit{xs}\ {-1}]$ and an explicit enumeration can be used to generate the list of element indices:

$$\begin{aligned}
&\mathit{wscanr}\ f\ a\ \mathit{xs} \ = \\
&\quad [\mathit{foldr}\ f\ a\ (\mathit{drop}\ (i+1)\ \mathit{xs}) \ \mid \ i \leftarrow [0\,..\,\mathit{length}\ \mathit{xs} - 1]]
\end{aligned} \tag{14}$$

The *wscanr* function produces the list of *partial* folds, so the rightmost element of its result is the singleton input a, and the leftmost element of the argument list is not used at all in the result. A *wscanr* over a singleton list $[x]$ returns the list $[a]$.

$$\begin{aligned}
&\mathit{wscanr}\ f\ a\ [x]\\
&\quad = \ [\mathit{foldr}\ f\ a\ (\mathit{drop}\ (i+1)\ [x]) \ \mid \ i \leftarrow [0\,..\,0]] && \langle \mathit{13} \rangle\\
&\quad = \ [\mathit{foldr}\ f\ a\ (\mathit{drop}\ 1\ [x])]\\
&\quad = \ [\mathit{foldr}\ f\ a\ (\,)] && \langle \mathit{1,2} \rangle\\
&\quad = \ [a] && \langle \mathit{10} \rangle
\end{aligned} \tag{15}$$

Practical applications often need both the complete fold and the list of partial folds, so it is convenient also to define a function that delivers both:

$$\begin{aligned}
&\mathit{ascanr} \ :: \ (b \to a \to a) \to a \to [b] \to (a, [a])\\
&\mathit{ascanr}\ f\ a\ \mathit{xs} \ = \ (\mathit{foldr}\ f\ a\ \mathit{xs}, \ \mathit{wscanr}\ f\ a\ \mathit{xs})
\end{aligned} \tag{16}$$

These functions differ from the *scanr* defined in the Haskell Prelude. In particular, *wscanr* returns a result list with the same length as the argument list, unlike *scanr*. Furthermore, *ascanr* produces a pair containing both the fold and the scan, while *scanr* attaches the fold to the scan list and returns a result longer than the argument. The *wscanr* and *ascanr* functions have better properties for hardware design, and will be used throughout this paper.

Although the specification of *wscanr* takes quadratic time if executed naively as a computer program, it specifies a digital circuit that requires only linear time. It is useful to distinguish *specifications* from *implementations*. The role of a specification is to ease the process of reasoning about algorithms, and the remainder of this paper shows that *wscanr* does this effectively. The specification is easy to reason with because it expresses clearly and directly the value that is computed.

The clear specification can also be transformed formally into an efficient sequential linear time implementation. This is done by a method we will call *form & solution:* (1) write an equation that expresses the form of the definition, and (2) use algebra to derive the unknown parts of the equation. The form & solution technique is central to the parallelisation of scan in Section 5, and the derivation of linear time scan provides a good introduction to it. We begin by writing down the general form of the expected solution. Since we seek a linear time implementation, it is natural to try an accumulator-style definition:

$$\begin{aligned}
&\mathit{ascanr} \;::\; (b \to a \to a) \to a \to [b] \to (a, [a]) && \\
&\mathit{ascanr}\ f\ a\ [\,] \;=\; \cdots ?\ \cdots && (17)\\
&\mathit{ascanr}\ f\ a\ (x : xs) \;=\; \cdots ?\ \cdots && (18)
\end{aligned}$$

The next step is to use equational reasoning to solve for the unknown expressions, beginning with eq. (17).

Base case.

$$\begin{aligned}
&\mathit{ascanr}\ f\ a\ [\,] && &\\
&= (\mathit{foldr}\ f\ a\ [\,],\ \mathit{wscanr}\ f\ a\ [\,]) && \langle 16 \rangle &\\
&= (a,\ [\mathit{foldr}\ f\ a\ (\mathit{drop}\ (i+1)\ [\,] \;\;|\;\; i \leftarrow [0\,..\,-1]]) && \langle 10{,}13 \rangle &\\
&= (a,\ [\,]) && \langle 3 \rangle & (19)
\end{aligned}$$

Induction case. The inductive hypothesis is

$$\mathit{ascanr}\ f\ a\ xs \;=\; (\mathit{foldr}\ f\ a\ xs,\ \mathit{wscanr}\ f\ a\ xs) \qquad (20)$$

It is convenient to define names for the components of this pair:

$$\begin{aligned} a' &= \mathit{foldr}\ f\ a\ \mathit{xs} && (21)\\ \mathit{xs}' &= \mathit{wscanr}\ f\ a\ \mathit{xs} && (22) \end{aligned}$$

Then

$$\begin{aligned} &(a', \mathit{xs}')\\ &\quad = (\mathit{foldr}\ f\ a\ \mathit{xs},\ \mathit{wscanr}\ f\ a\ \mathit{xs}) && \langle \mathit{21,22}\rangle\\ &\quad = \mathit{ascanr}\ f\ a\ \mathit{xs} && \langle \mathit{20}\rangle \qquad (23) \end{aligned}$$

Now the right hand side of (18) is calculated using equational reasoning.

$$\begin{aligned} &\mathit{ascanr}\ f\ a\ (x:\mathit{xs})\\ &= (\mathit{foldr}\ f\ a\ (x:\mathit{xs}),\ \mathit{wscanr}\ f\ a\ (x:\mathit{xs})) && \langle \mathit{16}\rangle\\ &= (\mathit{foldr}\ f\ a\ (x:\mathit{xs}), && \langle \mathit{16}\rangle\\ &\qquad [\mathit{foldr}\ f\ a\ (\mathit{drop}\ (i+1)\ (x:\mathit{xs}))\\ &\qquad\quad |\quad i \leftarrow [0\ ..\ \mathit{length}\ (x:\mathit{xs})-1]])\\ &= (f\ x\ (\mathit{foldr}\ f\ a\ \mathit{xs}), && \langle \mathit{11}\rangle\\ &\qquad \mathit{foldr}\ f\ a\ (\mathit{drop}\ 1\ (x:\mathit{xs}))\ : && \langle \mathit{4}\rangle\\ &\qquad\quad [\mathit{foldr}\ f\ a\ (\mathit{drop}\ (i+1)\ (x:\mathit{xs}))\\ &\qquad\qquad |\quad i \leftarrow [1\ ..\ \mathit{length}\ \mathit{xs}]])\\ &= (f\ x\ a',\ \mathit{foldr}\ f\ a\ \mathit{xs}\ : && \langle \mathit{21,1,2}\rangle\\ &\qquad\quad [\mathit{foldr}\ f\ a\ (\mathit{drop}\ i\ \mathit{xs})\quad |\quad i \leftarrow [1\ ..\ \mathit{length}\ \mathit{xs}]]) && \langle \mathit{2}\rangle\\ &= (f\ x\ a',\ \mathit{foldr}\ f\ a\ \mathit{xs}\ : [\mathit{foldr}\ f\ a\ (\mathit{drop}\ (i+1)\ \mathit{xs}) && \langle \mathit{5}\rangle\\ &\qquad\quad |\quad i \leftarrow [0\ ..\ \mathit{length}\ \mathit{xs}-1]])\\ &= (f\ x\ a',\ \mathit{foldr}\ f\ a\ \mathit{xs}\ :\ \mathit{wscanr}\ f\ a\ \mathit{xs}) && \langle \mathit{14}\rangle\\ &= (f\ x\ a',\ a' : \mathit{xs}') && \langle \mathit{21,22}\rangle \qquad (24) \end{aligned}$$

The calculation is finished by bringing together the unknown parts of the conjecture. This results in a mathematical statement about the *ascanr* function which also serves as an efficient linear time implementation.

Theorem 3.1

$$\begin{aligned} &\mathit{ascanr}\ ::\ (b \to a \to a) \to a \to [b] \to (a, [a])\\ &\mathit{ascanr}\ f\ a\ [\,]\ =\ (a,\ [\,]) && \langle \mathit{19}\rangle\\ &\mathit{ascanr}\ f\ a\ (x:\mathit{xs}) =\\ &\quad \mathbf{let}\ (a', \mathit{xs}')\ =\ \mathit{ascanr}\ f\ a\ \mathit{xs} && \langle \mathit{23}\rangle\\ &\quad \mathbf{in}\ (f\ x\ a',\ a' : \mathit{xs}') && \langle \mathit{24}\rangle \end{aligned}$$

The proof was not given before the statement of the theorem merely for rhetorical effect. The point is that we have calculated the content of the theorem while proving it: *formal methods were used to help construct a program,* not just to prove its correctness after the fact. The theorem is an efficient executable Haskell program, but it is also a mathematical statement of a property of *ascanr*, which was specified abstractly by equation (16).

4 Ripple Carry Addition

It is an interesting exercise to start with Definition 2.1 and derive an addition circuit from first principles. We will skip that step here, and begin with a specification of the standard and well known ripple carry adder.

The inputs to the adder are a carry input bit c and a word $zs = [(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})]$ of n bit pairs. The outputs are a pair (c, ss) where c' is the carry output, and $ss = [s_0, \ldots, s_{n-1}]$ is the word of n sum bits.

Within bit position i, for $0 \leq i < n$, the adder calculates the local sum bit $s_i = \mathit{bsum}\ (x_i, y_i)\ c_{i+1}$. The carry input to position i is c_{i+1}, and the carry output $c_i = \mathit{bcarry}\ (x_i, y_i)\ c_{i+1}$. The carry input c_n to the least significant bit is defined to be the carry input c to the entire word adder, and the carry output c' from the entire adder is defined to be the carry output c_0 from the most significant bit.

A ripple carry adder contains a building block for each bit position that which takes the data bits (x, y) and a carry input and produces a sum bit s and carry output c'. This building block is traditionally called a 'full adder':

$$\mathit{fullAdd}\ ::\ \mathit{Signal}\ a \Rightarrow (a, a) \rightarrow a \rightarrow (a, a).$$

However, the crux of the derivation that follows is in handling the carry propagation, and it will simplify the notation slightly to separate the calculations of the sum and carry bits into two functions, *bsum* and *bcarry*. These correspond to standard 3-input logic gates called majority and parity.

$$\begin{aligned}
&\mathit{bsum}, \mathit{bcarry}\ ::\ \mathit{Signal}\ a \Rightarrow (a, a) \rightarrow a \rightarrow a \\
&\mathit{bcarry}\ (x, y)\ c\ =\ \mathit{majority3}\ x\ y\ c \\
&\mathit{bsum}\ (x, y)\ c\ =\ \mathit{parity3}\ x\ y\ c
\end{aligned}$$

The following equation states the relationship between these functions and a full adder:

$$\mathit{fullAdd}\ (x, y)\ c\ =\ (\mathit{bcarry}\ (x, y)\ c,\ \mathit{bsum}\ (x, y)\ c)$$

The ripple carry adder uses *ascanr* to calculate all the carry bits, followed by a map (in the form of *zipWith* that calculates the sum bits. The circuit requires $O(n)$ time, and it contains $O(n)$ logic gates. Figure 1 shows the structure of the ripple carry adder for 4-bit words. It is important to note that the figure is only an example at a fixed wordsize, while the Hydra definition

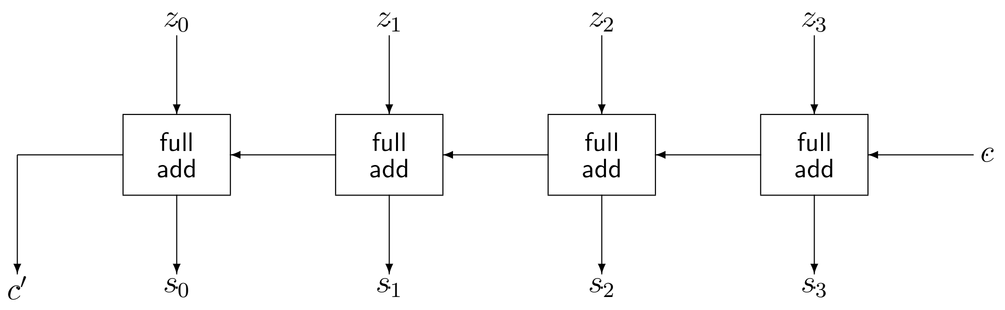


Fig. 1. Circuit diagram of *add1*

add1 is a general specification valid for all sizes $n \geq 0$.

```
add1 :: Signal a ⇒ a → [(a, a)] → (a, [a])
add1 c zs =
  let (c′, cs) = ascanr bcarry c zs
      ss = zipWith bsum zs cs
  in (c′, ss)
```

5 Parallel Scan

Since the time required by the ripple carry adder is dominated by the scan, we need to find a faster scan in order to speed up the circuit. This section derives a parallel scan algorithm that requires only $O(\log n)$ time, but which can be used only when the function being scanned is associative.

This section presents the parallel scan algorithm in detail. The material that follows is similar to the results in [11], but the full derivation is presented in order to make this paper self-contained and to give an excellent example of the power of the form & solution technique. Furthermore, the adder requires *wscanr* but [11] presented *wscanl*, and it may not be obvious how to convert the *wscanl* into *wscanr*.

The parallel scan algorithm uses a divide and conquer strategy to perform a scan in log time on a tree circuit, assuming that the function being scanned is associative. (The time is actually proportional to the height of the tree, and the algorithm works correctly even if the tree is not balanced.)

5.1 Fold and Scan Decomposition

The essence of the divide and conquer strategy is a collection of *decomposition* theorems that show how folds and scans over long lists can be broken into subproblems to be solved independently. Throughout this section we will assume that the lists are of finite length and that the functions are strict.

The fold decomposition theorem splits a long fold in the form $xs{+\!\!+}ys$ into two shorter folds over xs and ys, with a final application of f to combine the results. This theorem does *not* require f to be associative, but it also does not introduce parallelism, since there is a data dependency requiring the result of the fold over *ys* in order to calculate the fold over *xs*.

Theorem 5.1 (Fold decomposition)

$$foldr\ f\ a\ (xs +\!\!+ ys)\ =\ foldr\ f\ (foldr\ f\ a\ ys)\ xs \qquad (25)$$

The next theorem provides a useful connection between *foldr* and *foldr1*.

Theorem 5.2 (foldr/foldr1) *If xs is nonempty, then*

$$foldr\ f\ a\ xs\ =\ f\ (foldr1\ f\ xs)\ a \qquad (26)$$

The associative fold decomposition theorem introduces potential parallelism, since it breaks a long fold into two shorter folds that can be calculated independently. However, this rearrangement of the order of operations will produce a different value if the function f is not associative.

Theorem 5.3 (Associative fold decomposition) *If f is associative, then*

$$foldr1\ f\ (xs +\!\!+ ys)\ =\ f\ (foldr1\ f\ xs)\ (foldr1\ f\ ys) \qquad (27)$$

A scan decomposition theorem will also be needed. Since this theorem is less familiar than fold decomposition, we will calculate it using the form & solution method. The aim is to find a theorem in the form

$$wscanr\ f\ a\ (xs +\!\!+ ys)\ =\ wscanr\ f\ ?\ xs\ +\!\!+\ wscanr\ f\ ?\ ys \qquad (28)$$

This time the theorem can be calculated directly—no induction is needed.

$$\begin{aligned}
& wscanr\ f\ a\ (xs +\!\!+ ys) \\
&= [foldr\ f\ a\ (drop\ (i+1)\ (xs +\!\!+ ys)) && \langle 13 \rangle \\
&\qquad \mid\ i \leftarrow [0\ ..\ length\ (xs +\!\!+ ys) - 1]] \\
&= [foldr\ f\ a\ (drop\ (i+1)\ (xs +\!\!+ ys)) && \langle 6 \rangle \\
&\qquad \mid\ i \leftarrow [0\ ..\ length\ xs - 1]] \\
&\quad +\!\!+\ [foldr\ f\ a\ (drop\ (i+1)\ (xs +\!\!+ ys)) \\
&\qquad\quad \mid\ i \leftarrow [length\ xs\ ..\ length\ (xs +\!\!+ ys) - 1]] \\
&= [foldr\ f\ (foldr\ f\ a\ ys)\ (drop\ (i+1)\ xs) && \langle 25 \rangle \\
&\qquad \mid\ i \leftarrow [0\ ..\ length\ xs - 1]] \\
&\quad +\!\!+\ [foldr\ f\ a\ (drop\ (i+1)\ ys) && \langle 5,2 \rangle \\
&\qquad \mid\ i \leftarrow [0\ ..\ length\ ys - 1]] \\
&= wscanr\ f\ (foldr\ f\ a\ ys)\ xs\ +\!\!+\ wscanr\ f\ a\ ys && \langle 13 \rangle
\end{aligned}$$

Theorem 5.4

$$\begin{aligned}
& wscanr\ f\ a\ (xs +\!\!+ ys) \\
&= wscanr\ f\ (foldr\ f\ a\ ys)\ xs\ +\!\!+\ wscanr\ f\ a\ ys \qquad (29)
\end{aligned}$$

5.2 Parallel Tree Machine

The decomposition theorems express potential parallelism, but we need to turn this into actual parallelism. This requires a model of parallelism that can be used to produce digital circuits. The structure of the decomposition theorems suggests a parallel machine with a tree structure. This can be specified as an algebraic data type, with a suitable parallel machine operation.

The algebraic data type *Tree* is used to represent the structure of the circuit:

$$\textbf{data}\ \mathit{Tree}\ a \ = \ \mathit{Leaf}\ a \ \mid\ \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

The conversion function *treeWord* returns the word of values held in the leaves of a tree. This function is not part of the adder circuit, but it serves a vital role in the formal derivation of parallel scan: the specification is written in terms of lists, yet the parallel algorithm uses a tree machine, and we need a formal way to convert between them. A minor additional benefit is that *treeWord* is convenient for building software testing tools.

$$\mathit{treeWord}\ ::\ \mathit{Tree}\ a \to [a]$$

$$\mathit{treeWord}\ (\mathit{Leaf}\ x)\ =\ [x] \tag{30}$$

$$\mathit{treeWord}\ (\mathit{Node}\ x\ y)\ =\ \mathit{treeWord}\ x +\!\!+\ \mathit{treeWord}\ y \tag{31}$$

Two further functions are *mkTree*, which builds a reasonably balanced tree shape of a given size, and *wordTree*, which builds a tree representing a list, following the shape of an existing tree. Only the types are given here; the full definitions appear in the program (see Section 1).

$$\mathit{mkTree}\ ::\ \mathit{Nat} \to \mathit{Tree}\ ()$$

$$\mathit{wordTree}\ ::\ \mathit{Tree}\ b \to [a] \to \mathit{Tree}\ a$$

Now we need a specification of a parallel computation on the tree. The *sweep* combinator specifies the behaviour of a general tree circuit constructed from two building blocks: a node circuit and a leaf circuit. The behaviours of these circuits are specified by the epynomous functions.

$$\begin{array}{ll}
\mathit{sweep} & \\
\quad ::\ (a \to d \to (b,u)) & \text{— } \mathit{leaf} \text{ function} \\
\quad \to (d \to u \to u \to (u,d,d)) & \text{— } \mathit{node} \text{ function} \\
\quad \to d & \text{— root input} \\
\quad \to \mathit{Tree}\ a & \text{— leaf inputs} \\
\quad \to (u,\ \mathit{Tree}\ b) & \text{— (root output, leaf outputs)}
\end{array}$$

The leaf circuits all have a state of type a, and they provide upward-moving values of type u which they pass up the tree. Eventually the leaves will receive

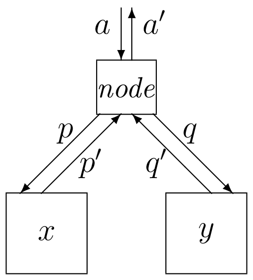


Fig. 2. Inductive case of *sweep* definition

a downward-moving value of type d, which they can then use to update their state.

$$\begin{aligned}
&\mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ a\ (\mathit{Leaf}\ x)\ = \\
&\quad \mathbf{let}\ (x', a')\ =\ \mathit{leaf}\ x\ a \\
&\quad \mathbf{in}\ (a',\ \mathit{Leaf}\ x')
\end{aligned} \tag{32}$$

Each node (see Figure 2) receives two upward messages p' and q' from its subtrees, and a downward message a from its parent. It uses these values to calculate an output a' to be sent up, and p and q to be sent down to the subtrees.

$$\mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ a\ (\mathit{Node}\ x\ y)\ = \tag{33}$$
$$\quad \mathbf{let}\ (a', p, q)\ =\ \mathit{node}\ a\ p'\ q' \tag{34}$$
$$\qquad (p', x')\ =\ \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ p\ x \tag{35}$$
$$\qquad (q', y')\ =\ \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ q\ y \tag{36}$$
$$\quad \mathbf{in}\ (a',\ \mathit{Node}\ x'\ y') \tag{37}$$

Thus the *sweep* combinator specifies a general tree circuit, where each component sends and receives on each of its ports. Naturally, it is possible to deadlock such a general tree if the leaf and node circuits are not defined properly. Most algorithms implemented with tree circuits execute with an upsweep followed by a downsweep, and the parallel scan algorithm is a typical example.

5.3 Derivation of Parallel Scan

The next step is to find—if possible—functions *leaf* and *node* that will cause the tree machine to compute an *ascanr*. This will be accomplished by conjecturing formally that it is actually possible; the conjecture will provide a set of equations stating properties that the solution must satisfy. The solution will then be found by solving the equations algebraically. The starting point is the following predicate:

$$\begin{aligned}
&\mathit{Parallel\_scanr}\ (f,\ \mathit{leaf},\ \mathit{node},\ a,\ t)\ \equiv \\
&\qquad a' \;=\; \mathit{foldr1}\ f\ (\mathit{treeWord}\ t) && (38)\\
&\quad \wedge\ \ \mathit{treeWord}\ t' \;=\; \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ t) && (39)\\
&\ \ \mathbf{where}\ (a', t') \;=\; \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ a\ t && (40)
\end{aligned}$$

No we can state a conjecture that there is a solution to the problem.

Conjecture 5.5 *Let $f :: a \to a \to a$ be associative. Then*

$$\begin{aligned}
&\exists\ \mathit{leaf}\ ::\ a \to d \to (b,u),\ \ \mathit{node}\ ::\ d \to u \to u \to (u,d,d)\ .\\
&\quad \forall a\ ::\ a,\ t\ ::\ \mathit{Tree}\ a\ .\\
&\qquad \mathit{Parallel\_scanr}\ (f, \mathit{leaf}, \mathit{node}, a, t)
\end{aligned}$$

The conjecture and predicate provide a formal specification to the problem, and we can now begin a calculation to find the solution. The calculation has two possible outcomes: if we find values of *leaf* and *node* that satisfy the equations, then the conjecture is established and we have a program (or circuit) that solves the problem. If the calculation fails to produce a result, then no conclusion can be drawn. (Fortunately that will not happen in this case!)

Base case. Let $t\ =\ \mathit{Leaf}\,x$ and $t'\ =\ \mathit{Leaf}\ x'$. The aim is to satisfy

$$\begin{aligned}
&a' \;=\; \mathit{foldr1}\ f\ (\mathit{treeWord}\ t) && \langle 38\rangle && (41)\\
&\mathit{treeWord}\ t' \;=\; \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ t) && \langle 39\rangle && (42)\\
&(a',\ \mathit{Leaf}\ x') \;=\; \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ a\ (\mathit{Leaf}\ x) && && (43)
\end{aligned}$$

Now, in order to define the *leaf* function, the values of a' and x' need to be calculated.

$$\begin{aligned}
&(a',\ \mathit{Leaf}\ x')\\
&\quad = \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ a\ (\mathit{Leaf}\ x) && \langle 40\rangle\\
&\quad = \mathbf{let}\ \ (x', a') \;=\; \mathit{leaf}\ x\ a && && (44)\\
&\qquad\ \ \mathbf{in}\ \ (a',\ \mathit{Leaf}\ x') && \langle 32\rangle && (45)\\
&a'\\
&\quad = \mathit{foldr1}\ f\ (\mathit{treeWord}\ (\mathit{Leaf}\ x)) && \langle 41\rangle\\
&\quad = \mathit{foldr1}\ f\ [x] && \langle 30\rangle\\
&\quad = x && \langle 10{,}11\rangle && (46)\\
&\mathit{treeWord}\ (\mathit{Leaf}\ x')\\
&\quad = [x'] && \langle 30\rangle && (47)\\
&\mathit{treeWord}\ (\mathit{Leaf}\ x')\\
&\quad = \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ (\mathit{Leaf}\ x)) && \langle 42\rangle
\end{aligned}$$

$$
\begin{array}{llr}
\quad = \ \mathit{wscanr}\ f\ a\ [x'] & \langle 30\rangle & \\
\quad = \ [a] & \langle 15\rangle & (48) \\
[x'] \ = \ [a] & \langle 47{,}48\rangle & (49) \\
x' \ = \ a & \langle 49{,}7\rangle & (50)
\end{array}
$$

These results can be gathered into a definition of *leaf*.

$$
\begin{array}{llr}
\mathit{leaf}\ x\ a & & \\
\quad = \ (x', a') & \langle 44\rangle & \\
\quad = \ (a, x) & \langle 50{,}46\rangle & (51)
\end{array}
$$

Induction case. Let $t = \mathit{Node}\ x\ y$ and $t' = \mathit{Node}\ x'\ y'$. There are two inductive hypotheses, one for each subtree:

$$
\begin{array}{lr}
\mathit{Parallel\_scanr}\ (f,\ \mathit{leaf},\ \mathit{node},\ p',\ x) & (52) \\
\mathit{Parallel\_scanr}\ (f,\ \mathit{leaf},\ \mathit{node},\ q',\ y) & (53)
\end{array}
$$

The aim is to find a value of the *node* function that calculates $t' = \mathit{Node}\ x'\ y'$ while satisfying the following predicate:

$$\mathit{Parallel\_scanr}\ (f,\ \mathit{leaf},\ \mathit{node},\ a,\ \mathit{Node}\ x\ y), \qquad (54)$$

which denotes

$$
\begin{array}{llr}
a' \ = \ \mathit{foldr1}\ f\ (\mathit{treeWord}\ t) & \langle 54\rangle & (55) \\
\mathit{treeWord}\ t' \ = \ \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ t) & \langle 54\rangle & (56) \\
(a', t') \ = \ \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ a\ t & \langle 54\rangle & (57)
\end{array}
$$

The inductive hypotheses denote the following equations:

$$
\begin{array}{llr}
p' \ = \ \mathit{foldr1}\ f\ (\mathit{treeWord}\ x) & \langle 52\rangle & (58) \\
\mathit{treeWord}\ x' \ = \ \mathit{wscanr}\ f\ p\ (\mathit{treeWord}\ x) & \langle 52\rangle & (59) \\
(p', x') \ = \ \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ p\ x & \langle 52\rangle & (60) \\
q' \ = \ \mathit{foldr1}\ f\ (\mathit{treeWord}\ y) & \langle 53\rangle & (61) \\
\mathit{treeWord}\ y' \ = \ \mathit{wscanr}\ f\ q\ (\mathit{treeWord}\ y) & \langle 53\rangle & (62) \\
(q', y') \ = \ \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ q\ y & \langle 53\rangle & (63)
\end{array}
$$

Now the values of the variables need to be calculated; this will enable the definition of *node*.

$$
\begin{array}{llr}
(a', t') & & \\
\quad = \ \mathit{sweep}\ \mathit{leaf}\ \mathit{node}\ a\ (\mathit{Node}\ x\ y) & \langle 57\rangle & \\
\quad = \ \mathbf{let}\ (a', p, q) \ = \ \mathit{node}\ a\ p'\ q' & \langle 33\rangle & (64)
\end{array}
$$

$$
\begin{aligned}
&\quad (p', x') \;=\; \mathit{sweep\ leaf\ node\ p\ x} && (65)\\
&\quad (q', y') \;=\; \mathit{sweep\ leaf\ node\ q\ y} && (66)\\
&\mathbf{in}\ (a',\ \mathit{Node}\ x'\ y') && (67)
\end{aligned}
$$

The root output a' is calculated by rewriting it in the form of the goal, applying the associative fold decomposition theorem, and then using the two inductive hypotheses to simplify the folds over the subtrees.

$$
\begin{aligned}
&a'\\
&\quad = \mathit{foldr1}\ f\ (\mathit{treeWord}\ (\mathit{Node}\ x\ y)) && \langle 55\rangle\\
&\quad = \mathit{foldr1}\ f\ (\mathit{treeWord}\ x \mathbin{+\!\!+} \mathit{treeWord}\ y) && \langle 31\rangle\\
&\quad = f\ (\mathit{foldr1}\ f\ (\mathit{treeWord}\ x))\ (\mathit{foldr1}\ f\ (\mathit{treeWord}\ y)) && \langle 27\rangle\\
&\quad = f\ p'\ q' && \langle 58{,}61\rangle \qquad (68)
\end{aligned}
$$

The scan is calculated using the scan decomposition theorem.

$$
\begin{aligned}
&\mathit{treeWord}\ (\mathit{Node}\ x'\ y')\\
&\quad = \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ (\mathit{Node}\ x\ y)) && \langle 56\rangle\\
&\quad = \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ x \mathbin{+\!\!+} \mathit{treeWord}\ y) && \langle 31\rangle\\
&\quad = \mathit{wscanr}\ f\ (\mathit{foldr}\ f\ a\ (\mathit{treeWord}\ y))\ (\mathit{treeWord}\ x) && \langle 29\rangle\\
&\qquad \mathbin{+\!\!+} \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ y)\\
&\quad = \mathit{wscanr}\ f\ (f\ (\mathit{foldr1}\ f\ (\mathit{treeWord}\ y))\ a) && \langle 26\rangle\\
&\qquad \mathbin{+\!\!+} \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ y)\\
&\quad = \mathit{wscanr}\ f\ (f\ q\ a)\ (\mathit{treeWord}\ x) && \langle 61\rangle \qquad (69)\\
&\qquad \mathbin{+\!\!+} \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ y)
\end{aligned}
$$

Next the root inputs p and q to the subtrees are calculated.

$$
\begin{aligned}
&\mathit{treeWord}\ (\mathit{Node}\ x'\ y')\\
&\quad = \mathit{treeWord}\ x' \mathbin{+\!\!+} \mathit{treeWord}\ y' && \langle 31\rangle\\
&\quad = \mathit{wscanr}\ f\ p\ (\mathit{treeWord}\ x) && \langle 59\rangle\\
&\qquad \mathbin{+\!\!+} \mathit{wscanr}\ f\ q\ (\mathit{treeWord}\ y) && \langle 62\rangle \qquad (70)\\
&\mathit{treeWord}\ (\mathit{Node}\ x'\ y')\\
&\quad = \mathit{wscanr}\ f\ (f\ q'\ a)\ (\mathit{treeWord}\ x) && \langle 61\rangle\\
&\qquad \mathbin{+\!\!+} \mathit{wscanr}\ f\ a\ (\mathit{treeWord}\ y) && (71)
\end{aligned}
$$

These equations can be satisfied by choosing the following definitions of p and q:

$$
\begin{aligned}
p &= f\ q'\ a && \langle 70{,}71\rangle \qquad (72)\\
q &= a && \langle 70{,}71\rangle \qquad (73)
\end{aligned}
$$

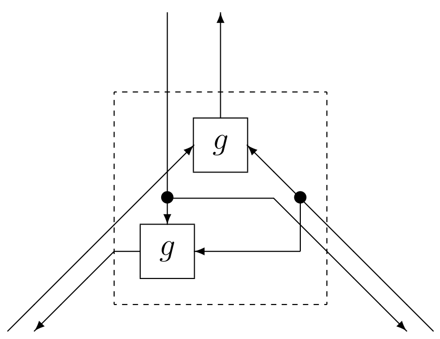


Fig. 3. Node circuit for *Tscanr*

The results of these calculations now enable the *node* function to be defined.

$$\begin{aligned} & node\ a\ p'\ q' \\ & \quad = (a', p, q) && \langle 64 \rangle \\ & \quad = (f\ p'\ q',\ f\ q'\ a,\ a) && \langle 68,72,73 \rangle \quad (74) \end{aligned}$$

The results can now be combined to define the log-time parallel *tscanr* algorithm and to establish its correctness.

Theorem 5.6 (Parallel scan)

$$\begin{aligned} & tscanr :: (a \to a \to a) \to a \to Tree\ a \to (a,\ Tree\ a) \\ & tscanr\ f\ a\ = \\ & \quad \mathbf{let} leaf\ x\ a\ =\ (a, x) && \langle 51 \rangle \\ & \qquad node\ a\ p'\ q'\ =\ (f\ p'\ q',\ f\ q'\ a,\ a) && \langle 74 \rangle \\ & \quad \mathbf{in}\ sweep\ leaf\ node\ a\ t \end{aligned}$$

Once again, we have used calculation by equational reasoning to derive the definition of a circuit along with its correctness proof. The *tscanr* circuit is perfectly well defined for any function f of the required type, but it computes the same result as *ascanr* only if f is associative.

Theorem 5.7 *Let* $(a', t')\ =\ tscanr\ f\ a\ t$. *If* f *is associative, then*

$$a'\ =\ foldr1\ f\ (treeWord\ t) \qquad (75)$$

$$treeWord\ t'\ =\ wscanr\ f\ a\ (treeWord\ t) \qquad (76)$$

The *tscanr* circuit derived above is essentially the same definition that appears in [11], except that paper implemented *scanl* rather than *scanr.*

Figure 4 gives an example execution of *tscanr.* This diagram may help the reader to see what is going on in parallel scan, but the real intuitions are captured by the decomposition theorems. Diagrams and examples can supplement the formalism, but they should not supplant it.

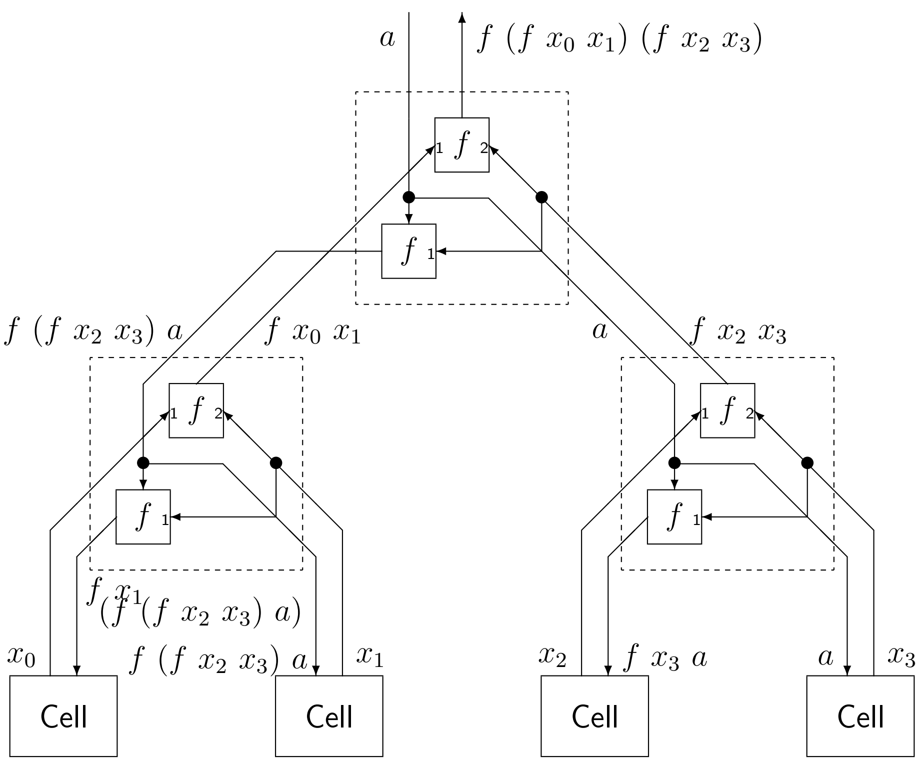


Fig. 4. Example: calculation of *tscanr* $f\ a\ [x_0, x_1, x_2, x_3]$

6 Making the Scan Associative

The time required by the ripple carry adder is dominated by the *ascanr*. The previous section has introduced a faster parallel scan, and our strategy for improving the adder is to use this to calculate the carries in log time.

Unfortunately there is an immediate stumbling block. The parallel scan algorithm requires f to be associative in order to compute *ascanr* $f\ a\ xs$ in log time, but the ripple carry adder applies *ascanr* to the *bcarry* circuit, which is not associative. Indeed, an associative function must have type $a \to a \to a$, so *bcarry* $:: (a, a) \to a \to a$ doesn't even have a suitable type.

6.1 Partial Evaluation of Scan

A useful principle in program derivation is to transform a specification to bring it as close as possible to the goal, even if the goal itself is not directly reachable. The reason is that the intermediate transformation might cause a different approach to become applicable.

Partial evaluation is a systematic method for applying this principle. The arguments to a function are partitioned into static arguments that are known in advance and dynamic arguments that will become known later. This technique is typically used in compilers: the usual idea is to have the compiler apply the functions in a program just to the static arguments that are known at compile time, in the hope that the resulting partial applications can be simplified, producing more efficient object code. In this section, we apply the same idea to the problem of carry propagation in hardware design.

Ideally, we would like the circuitry for bit position i to calculate the application *bcarry* (x_i, y_i) c_{i+1} in unit time. This is impossible, since the value of c_{i+1} must itself be computed, and it takes time for the carry propagation to ripple across the adder. However, if we think of this as a problem of higher order functional programming as well as hardware design, it becomes clear that at least the partial applications *bcarry* (x_i, y_i) can be calculated in parallel:

$$ps \; = \; map\ bcarry\ zs.$$

The partial application $p_i = bcarry\ (x_i, y_i)$ is a function $p_i :: Signal\ a \Rightarrow a \rightarrow a$ that can be used to produce the carry output in position i once the carry input is known. Meanwhile, we can go ahead and exploit the knowledge p_i has of the values of x_i and y_i, even before the carry input c_{i+1} is available.

At this stage there is nothing useful to which the p_i functions can be applied, but another idea is to compose them instead of applying them. Just as each bit position has a carry propagation function, so does a sequence of adjacent positions $i \ldots j$, for $0 \leq i \leq j < n$. Since we are interested in the carry *input* at bit position i (in order to compute the sum bit there), we define the sequence carry propagation function C_i^j as

$$\begin{aligned} C_i^j &= p_{i+1} \circ p_{i+2} \circ \cdots \circ p_j \quad \text{for } -1 \leq i < j \\ C_i^i &= id \end{aligned}$$

This function takes the carry input c_{j+1} to the least significant position of the sequence and produces the carry input c_i to the most significant position. (Note that C_{-1}^j is the carry *output* from the most significant bit.) The function can be calculated by folding the list of p functions with the composition operator, using the identity function as the unit:

$$C_i^j \; = \; foldr\ (\circ)\ id\ [p_{i+1}, \ldots, p_j],$$

for $0 \leq i \leq j < n$. In particular,

$$C_i^{n-1} \; = \; foldr\ (\circ)\ id\ [p_{i+1}, \ldots, p_{n-1}].$$

The adder circuit requires the carry input to each bit position in order to compute the corresponding sum bit, and it also needs the carry output from position 0 since this is an output of the entire circuit. Although we do not yet know the values of these carry bits, we can calculate their carry propagation functions:

$$\begin{aligned}
&\bigl(C^{n-1}_{-1}, \quad [C^{n-1}_i \mid i \leftarrow [0 \mathinner{..} n-1]]\bigr)\\
&\quad = \bigl(\mathit{foldr}\ (\circ)\ \mathit{id}\ \mathit{ps},\\
&\qquad\quad [\mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{drop}\ (i+1)\ \mathit{ps}) \mid i \leftarrow [0 \mathinner{..} n-1]]\bigr)\\
&\quad = \mathit{ascanr}\ (\circ)\ \mathit{id}\ \mathit{ps}
\end{aligned}$$

Now we are in a much better position: this entire set of functions can be calculated in log time using parallelism because *now the argument to ascanr is the associative operator* $(\circ)$. Our original problem was that *ascanr* was applied to a non-associative function. Furthermore, once the sequence carry propagation functions have been calculated, all of the actual carry bits that are needed can be calculated in $O(1)$ time simply by applying all of those functions to $c_n = c$, which is the one carry bit that we already have, since it is an input to the circuit!

This transformation can be expressed formally as two partial evaluation theorems, one each for *foldr* and *scanr*. This is a generally useful technique, and similar theorems exist for the other fold and scan functions. Theorem 6.1 has been used by Harrison [5], and Maessen has stated a weaker version of it [8]. Fischer and Ghuloum [4] described a technique for parallelising scans.

Theorem 6.1 (Partial evaluation of fold)

$$\mathit{foldr}\ f\ a\ \mathit{xs} \ = \ (\mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{map}\ f\ \mathit{xs}))\ a \tag{77}$$

Proof. Structural induction over *xs*. For the base case,

$$\begin{aligned}
&\mathit{foldr}\ f\ a\ [\,]\\
&= a\\
&= \mathit{id}\ a\\
&= (\mathit{foldr}\ (\circ)\ \mathit{id}\ [])\ a\\
&= (\mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{map}\ f\ [\,]))\ a
\end{aligned}$$

Inductive case: the hypothesis is $\mathit{foldr}\ f\ a\ \mathit{xs} \ = \ (\mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{map}\ f\ \mathit{xs}))\ a$.

$$\begin{aligned}
&\mathit{foldr}\ f\ a\ (x : \mathit{xs})\\
&= f\ x\ (\mathit{foldr}\ f\ a\ \mathit{xs})\\
&= f\ x\ (\mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{map}\ f\ \mathit{xs})\ a)\\
&= (f\ x\ \circ\ \mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{map}\ f\ \mathit{xs}))\ a\\
&= (\mathit{foldr}\ (\circ)\ \mathit{id}\ (f\ x\ :\ \mathit{map}\ f\ \mathit{xs}))\ a\\
&= (\mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{map}\ f\ (x : \mathit{xs})))\ a
\end{aligned}$$

□

The partial evaluation of scan requires an explicit application function:

$$apply\ f\ x\ =\ f\ x \tag{78}$$
$$\tag{79}$$

Theorem 6.2 (Partial evaluation of scan)

$$
\begin{aligned}
&wscanr\ f\ a\ xs\ = && (80)\\
&\quad map\ apply\ (zip\ (wscanr\ (\circ)\ (map\ f\ id)\ xs)\ (repeat\ a)) && (81)
\end{aligned}
$$

Proof. The left hand side is transformed directly into the right hand side. Let $n\ =\ length\ xs$. Then

$$
\begin{aligned}
&wscanr\ f\ a\ xs\\
&=\ [foldr\ f\ a\ (drop\ (i+1)\ xs)\ \mid\ i \leftarrow [0\mathinner{..}n-1]]\\
&=\ [foldr\ (\circ)\ id\ (map\ f\ (drop\ (i+1)\ xs))\ a\ \mid\ i \leftarrow [0\mathinner{..}n-1]]\\
&=\ [foldr\ (\circ)\ id\ (drop\ (i+1)\ (map\ f\ xs))\ a\ \mid\ i \leftarrow [0\mathinner{..}n-1]]\\
&=\ map\ apply\ [(foldr\ (\circ)\ id\ (drop\ (i+1)\ (map\ f\ xs)),\ a)\\
&\qquad \mid\ i \leftarrow [0\mathinner{..}n-1]]\\
&=\ map\ apply\ (zip\ [foldr\ (\circ)\ id\ (drop\ (i+1)\ (map\ f\ xs))\\
&\qquad \mid\ i \leftarrow [0\mathinner{..}n-1]]\ (repeat\ a)
\end{aligned}
$$

□

6.2 Associative Scan Adder

Using Theorem 6.2, we can now transform the ripple carry adder into *add2*. The structure of the circuit is shown in Figure 5.

```
add2 :: Signal a ⇒ a → [(a, a)] → (a, [a])
add2 c zs =
  let ps = map bcarry zs
      (cf, cfs) = ascanr (∘) id ps
      cs = zipWith apply cfs (repeat c)
      c′ = cf c
      ss = zipWith bsum zs cs
  in (c′, ss)
```

7 Symbolic Function Representation

In solving one problem we have created another. The adder now applies scan to an associative function, so the parallel scan method has become applicable. However, the adder now contains signals that are carry propagation functions, not carry bits. Ultimately every digital circuit must be constructed from the primitive logic components, and those operate only on bits. Before proceeding

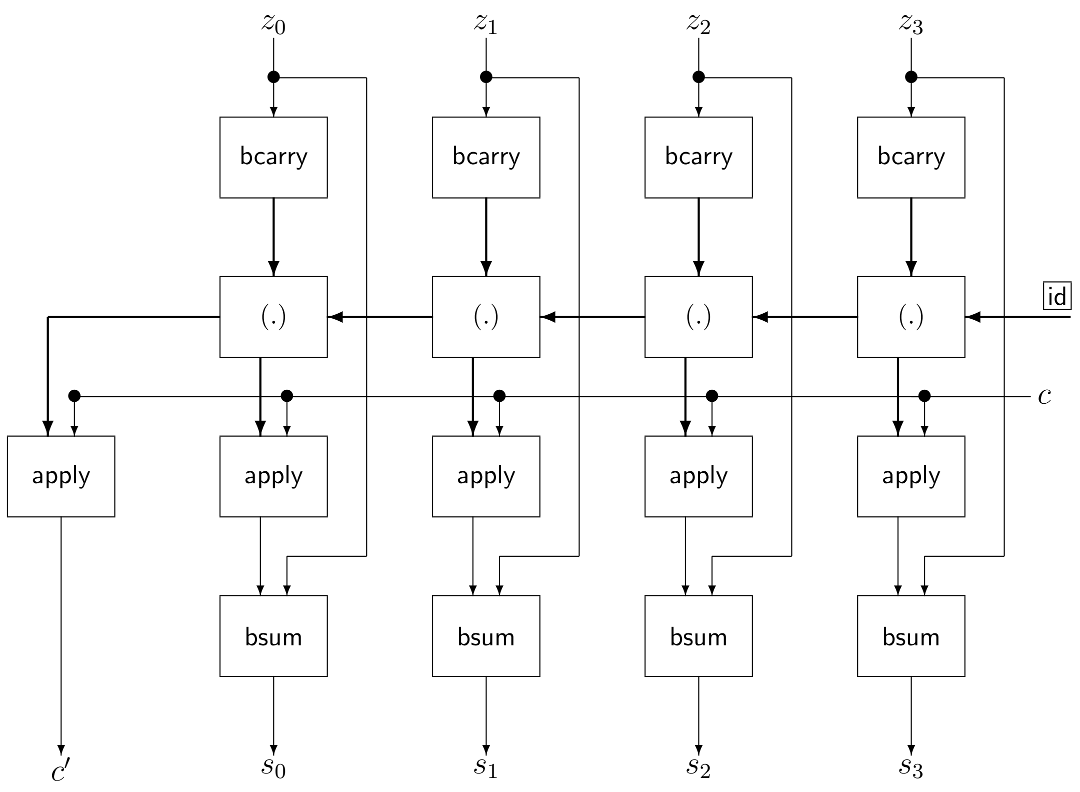


Fig. 5. Circuit diagram of *add2*

to the parallel scan, we should determine whether it will be possible to get around this difficulty—otherwise the parallelisation would be fruitless.

A function $f :: A \to B$ can be represented as a set of pairs $\{(a,\ f\ a) \mid f\ a \in A\}$; this is called the graph of f. We haven't decided yet whether to use function graphs within the adder, but the graphs still provide useful insight into the problem. Two crucial questions arise in the context of the adder:

(i) How can we compute the graphs of new carry propagation functions during the course of an addition?

(ii) How large can the function graphs become, and how can they be represented?

A partial application *bcarry* (x, y) has four possible values, since x and y are both signals restricted to 0 or 1. The complete set of partial applications can be enumerated as follows, using $f_1, \ldots, f_4$ as names for the resulting functions:

$$\begin{aligned}
\mathit{bcarry}\ (0,0) &= f_1 \\
\mathit{bcarry}\ (0,1) &= f_2 \\
\mathit{bcarry}\ (1,0) &= f_3 \\
\mathit{bcarry}\ (1,1) &= f_4
\end{aligned}$$

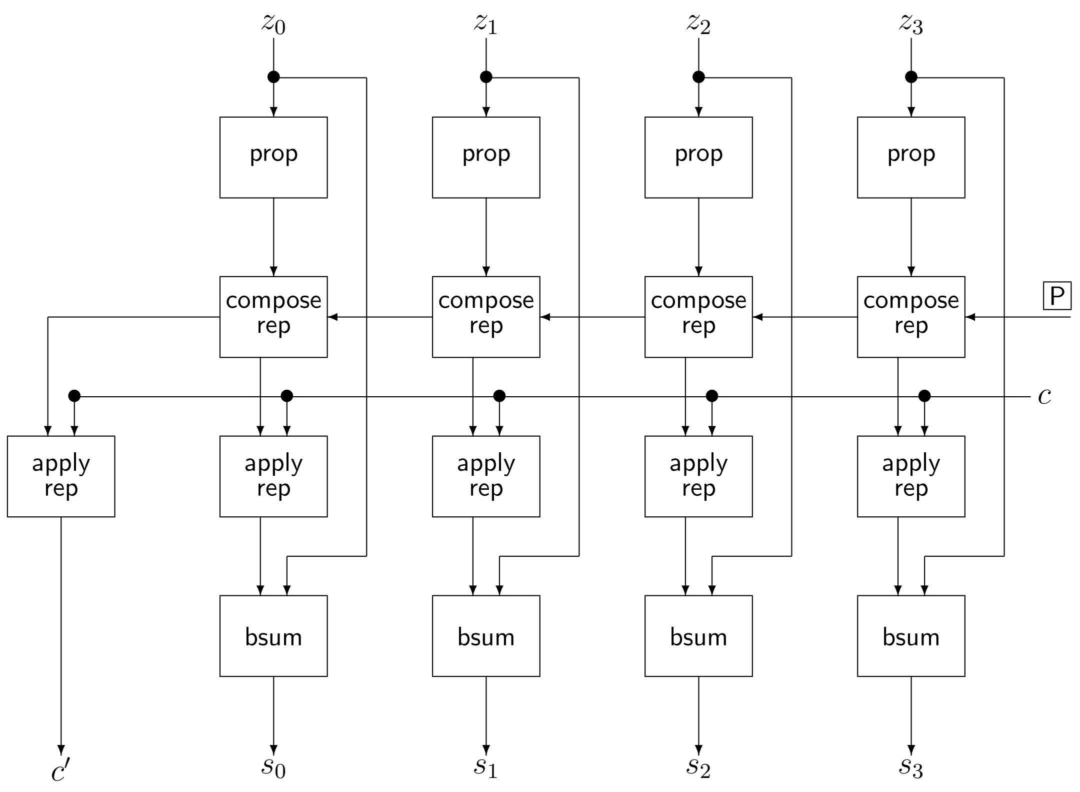


Fig. 6. Circuit diagram of *add3*

It is straightforward to check that $f_2 = f_3$, because

$$\forall c \in \{0,1\}.\ bcarry\ (0,1)\ c\ =\ bcarry\ (1,0)\ c.$$

There are traditional names for these functions [9]: f_1 is called K because it "kills" the carry (it returns 0 regardless of its carry argument); f_2 and f_3 are called P because they are the identity function, "propagating" the carry input to the output; f_4 is called G because it "generates" a carry output of 1 regardless of its argument. An arbitrary partial application of *bcarry* is representable using a finite alphabet of symbols:

$$data\ Sym\ =\ K\ \mid\ P\ \mid\ G \tag{82}$$

Each partial application of bcarry can be replaced by a full application of *bcarrySym*, which achieves the goal of getting rid of higher order functions as signals in the circuit. The definition of *bcarrySym* is slightly unusual, since it has a mixed type with signal arguments but a symbolic output. Because of this, a multiplexor cannot be used to define it, and we must resort instead to explicit testing of the input signal values. Thus *bcarrySym* is a halfway house: it operates on first order values, but its outputs are not digital circuit signals.

$bcarrySym\ ::\ Signal\ a \Rightarrow (a,a) \rightarrow Sym$
$bcarrySym\ (x,y)$

$$\begin{array}{l} \mid \ \mathit{is0}\ x \ \wedge\ \mathit{is0}\ y \ = \ K \\ \mid \ \mathit{is0}\ x \ \wedge\ \mathit{is1}\ y \ = \ P \\ \mid \ \mathit{is1}\ x \ \wedge\ \mathit{is0}\ y \ = \ P \\ \mid \ \mathit{is1}\ x \ \wedge\ \mathit{is1}\ y \ = \ G \end{array} \tag{83}$$

Now that the higher order functions inside the adder have been replaced by symbolic signals, we can no longer use $(\circ)$ and *id* to compose carry propagation functions. Therefore an explicit composition function that operates on *Sym*-represented functions needs to be defined. It is straightforward to calculate the value of this new composition operator, by considering all nine possible cases. A simpler calculation is based on the observation that K (or G) will kill (or generate) its carry output regardless of the value of its input, while P is just the identity function. The result of this calculation is the following definition:

$$\begin{array}{l} \mathit{composeSym} \ :: \ \mathit{Sym} \rightarrow \mathit{Sym} \rightarrow \mathit{Sym} \\ \mathit{composeSym}\ K\ f \ = \ K \\ \mathit{composeSym}\ P\ f \ = \ f \\ \mathit{composeSym}\ G\ f \ = \ G \end{array}$$

Finally, a new operation is needed to apply a symbolic carry function to a carry bit, getting us back into the world of signals:

$$\begin{array}{l} \mathit{applySym} \ :: \ \mathit{Signal}\ a \Rightarrow \mathit{Sym} \rightarrow a \rightarrow a \\ \mathit{applySym}\ K\ x \ = \ \mathit{zero} \\ \mathit{applySym}\ P\ x \ = \ x \\ \mathit{applySym}\ G\ x \ = \ \mathit{one} \end{array}$$

The adder can now be transformed into *add3*, using the *Sym* representation instead of partial applications. The Haskell definition and the circuit diagram (6) have exactly the same structure as *add2*.

$$\begin{array}{l} \mathit{add3} \ :: \ \mathit{Signal}\ a \Rightarrow a \rightarrow [(a,a)] \rightarrow (a,[a]) \\ \mathit{add3}\ c\ \mathit{zs} \ = \\ \quad \textbf{let}\ \mathit{ps} \ = \ \mathit{map}\ \mathit{bcarrySym}\ \mathit{zs} \\ \qquad\quad (\mathit{cf}, \mathit{cfs}) \ = \ \mathit{ascanr}\ \mathit{composeSym}\ P\ \mathit{ps} \\ \qquad\quad \mathit{cs} \ = \ \mathit{zipWith}\ \mathit{applySym}\ \mathit{cfs}\ (\mathit{repeat}\ c) \\ \qquad\quad c' \ = \ \mathit{applySym}\ \mathit{cf}\ c \\ \qquad\quad \mathit{ss} \ = \ \mathit{zipWith}\ \mathit{bsum}\ \mathit{zs}\ \mathit{cs} \\ \quad \textbf{in}\ (c', \mathit{ss}) \end{array}$$

8 Parallel Scan Adder

The adder is now transformed to use the log time *tscanr* in place of the linear time *ascanr*. This is possible because the function scanned is the associative *composeSym*. Some additional wiring rearrangements need to be introduced. The word *ps* of carry propagation functions needs to be converted by *wordTree* from a list representation to a set of tree leaves, and the result of the tree scan is *cft*, a tree-structured word that is converted by treeWord back to a list. These "impedance matching" conversions are only required to make the types match, but they have absolutely no impact on the circuit—they introduce no extra components or wires.

```
add4 :: Signala ⇒ a → [(a, a)] → (a, [a])
add4 c zs =
  let ps = map bcarrySym zs
      ps' = wordTree (mkTree (length zs)) ps
      (cf, cft) = tscanr composeSym P ps'
      cfs = treeWord cft
      cs = zipWith applySym cfs (repeat c)
      c' = applySym cf c
      ss = zipWith bsum zs cs
  in (c', ss)
```

9 Back into Hardware

The derivation is almost finished; the only remaining tasks are to replace the symbolic propagation function representations with actual digital signals, and to make the corresponding changes to the circuit components. These steps are straightforward, and could in principle be automated.

Since the *Bsym* type has three possible values, two bits are required to represent it. The circuits we are about to define will contain a lot of signals, and it will keep the definitions more readable to replace *Sym* by a type alias *BSym a*, where *a* is the hardware signal type. The signal representations of K, P and G are then defined as constant bit pairs. The actual values chosen to represent them are arbitrary, subject only to the constraint that we keep the values of the three symbols distinct. It simplifies the hardware slightly to allow both (*zero*, *one*) and (*one*, *zero*) to represent P, and we choose arbitrarily to define *repP* = (*zero*, *one*).

```
type BSym a = (a, a)
repK, repP, repG :: Signal a ⇒ BSym a
repK = (zero, zero)
```

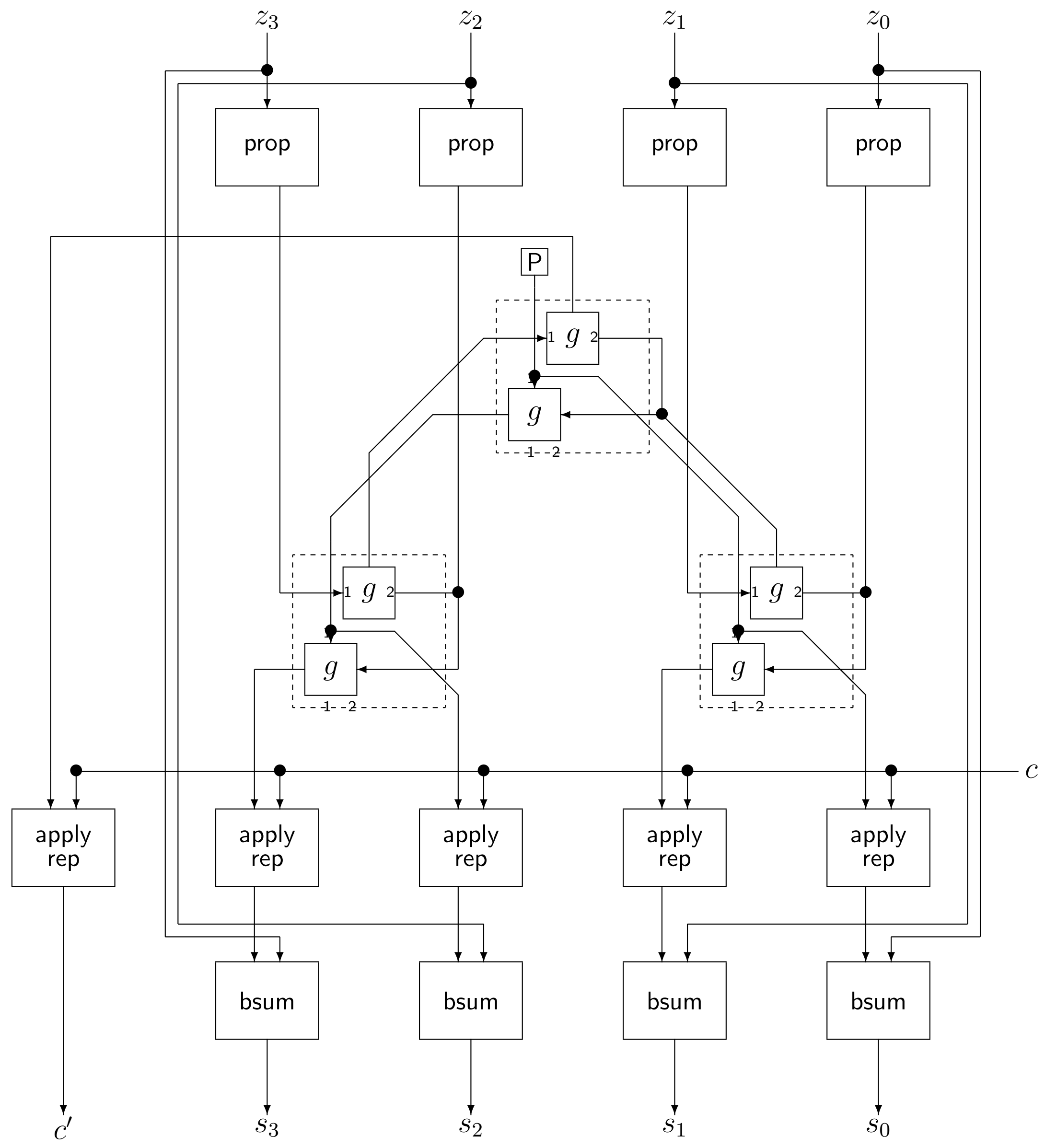


Fig. 7. Circuit diagram of *add4*

$$
\begin{aligned}
repP &= (zero, one) \\
repG &= (one, one)
\end{aligned}
$$

The symbolic circuits can now be replaced by digital implementations. The *bcarryBSym* circuit takes a pair of (x, y) of bits from the words being added and outputs the corresponding two-bit representation of the carry propagation function. In general, a circuit that implements partial applications might have to do something substantive: for example, if the number of bits in the symbolic representation is smaller than the number of input bits. In this case, however, we can choose to represent *bcarry* (x, y) by the pair (x, y). Thus K is represented by $(0, 0)$, P is represented by both $(0, 1)$ and $(1, 0)$, and G is represented by $(1, 1)$. This leads to a particularly simple definition.

$$
\begin{aligned}
&\mathit{bcarryBSym} :: \mathit{Signal}\ a \Rightarrow (a, a) \rightarrow \mathit{BSym}\ a \\
&\mathit{bcarryBSym} = \mathit{id}
\end{aligned}
$$

The remaining circuits can now be defined, but the definitions must work for both representations of P:

```
composeBSym :: Signal a ⇒ BSym a → BSym a → BSym a
composeBSym f g =
  let (g_0, g_1) = g
  in (mux2 f zero g_0 g_0 one,
       mux2 f zero g_1 g_1 one)
applyBSym :: Signal a ⇒ BSym a → a → a
applyBSym f x = mux2 f zero x x one
```

The goal has been attained: *add5* is a digital circuit that calculates the sum of two n-bit words in $O(\log n)$ time.

```
add5 :: Signal a ⇒ a → [(a, a)] → (a, [a])
add5 c zs =
  let ps = map bcarryBSym zs
      ps' = wordTree (mkTree (length zs)) ps
      (cf, cft) = tscanr composeBSym repP ps'
      cfs = treeWord cft
      cs = zipWith applyBSym cfs (repeat c)
      c' = applyBSym cf c
      ss = zipWith bsum zs cs
  in (c', ss)
```

10 Conclusion

We have transformed a linear time ripple carry adder into a log time parallel adder. The transformation proceeded in a sequence of steps, introducing the essential techniques one by one, with each change to the circuit enabling the next step to be made: partial evaluation was used to convert an inherently sequential scan into a scan over the associative composition function; a symbolic representation was introduced in order to make all the signal values first order; the tree combinator was used to implement a parallel scan; the symbolic functions were replaced by digital components.

Carry lookahead adders are often presented using an asymmetric parallel prefix network that allows only right-to-left communication, from less significant bit positions to more significant ones. This is adequate for binary addition, but not for a general processor arithmetic unit, since other ALU operations (such as comparison) require left to right communication. The structure of our circuit, consisting of a sequence of stages including a tree network, is well suited for general processor ALU design.

The circuit specification derived here is both precise and general. All necessary details are present, and the circuit can simulated using Haskell. The specification works on word size n, for every natural number n. Typical presentations of the carry lookahead adder lack this degree of precision and generality.

We have presented the derivation in a direct narrative, from the specification to the final result. This improves the elegance of the exposition, but in reality nothing goes so smoothly. Just as in ordinary programming, formal derivations sometimes become convoluted because an arbitrary choice made earlier is suboptimal, and in practice it may be necessary to make some adjustments to earlier stages in order to make the next transformation go through smoothly. For example, one might have chosen at the outset to give *bcarry* the type $a \to (a, a) \to a$, but the partial evaluation is cleaner when it has type $(a, a) \to a \to a$.

When such decisions have not been made optimally, the transformations still go through, but the notation is unnecessarily clumsy. It is then useful to go through a cleanup process, where the definitions are adjusted so that everything works out as elegantly as possible, but this can also give the misleading impression that formal transformations are more straightforward than is really the case. However, it is not true that one needs to be lucky with the original definitions in order to make progress; a more accurate conclusion is that periodic improvements to notational conventions can make the details look better.

Traditional circuit design was based on schematic diagrams, which work well for simple circuits but fail badly on large, complex designs. For this reason, computer hardware description languages (CHDLs) have become increasingly popular. CHDLs are generally based on existing programming languages, such as Ada. Other work is based on relational and functional languages; see for example [6]. In this paper we used Hydra, a CHDL based on Haskell, and concrete benefits were obtained from the use of equational reasoning, referential transparency, higher order functions, and algebraic data types.

Lava [1] is a very similar functional hardware description language; it is essentially a clone of the 1992 version of Hydra. However, there is one major difference: in its netlist generation algorithm, Lava clones the 1987 version of Hydra rather than the 1992. This is crucial, since that algorithm relies on the ability to compare pointers rather than values. This requires impure functional programming and violates referential transparency.

One can design a circuit, simulate it and transform it in Lava just as in Hydra. The problem is that when a netlist is generated for a circuit specification written in Lava (or Hydra'87), there is no guarantee that the circuit which is generated is the same as the circuit whose behaviour was verified. In other words, you can simulate a circuit with Lava and find that it works correctly, but then when you fabricate it, Lava is free to produce a *different circuit.* Correctness proofs are worthless in such an environment. For precisely this reason, a better method was introduced in Hydra '92 and all subsequent versions.

In practice this problem is not too severe, since Lava is used only for specifying circuits to be verified with conventional batch tools (such as model checkers) that come into play only *after the design is finished.* By that time there is no longer a problem, since a circuit will be fabricated using the same netlist that was used to verify it. However, Lava is unsuitable for situations where the designer wishes to use formal methods to assist *during* the design process. Hydra does not suffer from these problems, and Lava gains no advantage over Hydra through its use of the older algorithm. The loss of equational reasoning in Lava is unmitigated.

The results in this paper demonstrate how valuable formal methods can be during the design process. Formal reasoning can help the designer to create the circuit, as well as to improve its efficiency and to prove its correctness—and this paper has demonstrated all three of those activities applied to a nontrivial problem. In contrast, batch tools like model checkers give no assistance in designing a circuit, and they give little assistance in debugging it in the event that they announce the presence of an error.

Hawk [2] is another recent hardware description language similar to Hydra. It is not a clone; in particular, Hawk uses a monadic style that essentially requires the designer to write a program to construct circuits, instead of specifying circuits directly (as in Hydra and Lava). In the original Hawk, specifications had a strongly imperative flavour, since the monads performed side effects that had to occur in the right order. This problem was overcome by a complex extension to Haskell, allowing for mutual recursion among monadic actions. The kind of equational reasoning used in this paper cannot be used in Hawk. Methods for reasoning formally about imperative programs could be adapted for Hawk, but this would still leave Hawk specifications harder to use than Hydra specifications. Hawk gains no advantage over Hydra to compensate for its problems with monads.

The adder presented in this paper can be generalised to a complete ALU (arithmetic and logic unit). The tree structure derived in Section 5 for parallel scan can implement all three major variants of scan: *wscanl* (from left), *wscanr* (from right), and *wscan* (bidirectional). The adder requires only the from-right *scanr* function, but a full ALU requires all three. The circuit of Ladner and Fischer [7] uses a pattern called "recursive doubling", which is less general than the tree and which does not support all the operations required in an ALU.

Acknowledgements

We would like to thank the anonymous referees for several suggestions that helped us to improve the clarity of the paper.

References

[1] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware.* PhD thesis, Chalmers University of Technology, 2001.

[2] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*, 1998.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 29. MIT Press, 1990.

[4] J. Fischer and A. Ghuloum. Parallelizing complex scans and reductions. In *ACM Conf. on Programming Language Design and Implementation.* ACM, 1994.

[5] P. G. Harrison. Towards the synthesis of static parallel algorithms: a categorical approach. In *Proc. Working Conf. on Constructing Programs from Specifications.* IFIP, 1991.

[6] G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second Int. Conf. on Mathematics of Program Construction*, LNCS. Springer, 1992.

[7] R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM*, 4, October 1980.

[8] Jan-Willem Maessen. *Eliminating intermediate lists in pH using local transformations.* M. Eng. thesis, Massachusetts Institute of Technology, May 1994.

[9] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[10] John O'Donnell. Hardware description with recursion equations. In *Proc. IFIP 8th Int. Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382. North Holland, April 1987.

[11] John O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994.

***

FUNCTIONAL PEARL
Inverting the Burrows-Wheeler Transform

Richard Bird and Shin-Cheng Mu [1]

Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Abstract

The objective of this pearl is to derive the inverse of the Burrows-Wheeler transform from its specification, using simple equational reasoning. In fact, we derive the inverse of a more general version of the transform, proposed by Schindler.

1 Introduction

The Burrows-Wheeler Transform [1] is a method for permuting a string with the aim of bringing repeated characters together. As a consequence, the permuted string can be compressed effectively using simple techniques such as move-to-front or run-length encoding. In [4], the article that brought the BWT to the world's attention, it was shown that the resulting compression algorithm could outperform many commercial programs available at the time. The BWT has now been integrated into a high-performance utility `bzip2`, available from [6].

Clearly the best way of bringing repeated characters together is just to sort the string. But this idea has a fatal flaw as a preliminary to compression: there is no way to recover the original string unless the complete sorting permutation is produced as part of the output. Instead, the BWT achieves a more modest permutation, one that aims to bring some but not all repeated characters into adjacent positions. Moreover, the transform can be inverted using a single additional piece of information, namely an integer k in the range $0 \leq k < n$, where n is the length of the output (or input) string.

It often puzzles people, at least on a first encounter, as to exactly why the BWT is invertible and how the inversion is actually carried out. Our objective in this pearl is to derive the inverse transform from its specification

[1] Thanks to Ian Bayley, Jeremy Gibbons, Geraint Jones, and Barney Stratford for constructive criticism of earlier drafts of this paper.

by equational reasoning. In fact, we will derive the inverse of a more general version of the BWT transform, first proposed by [5].

2 Defining the BWT

The BWT is specified by two functions: $bwp :: String \rightarrow String$, which permutes the string and $bwn :: String \rightarrow Int$, which computes the supplementary integer. The restriction to strings is not essential to the transform, and we can take bwp to have the Haskell type $Ord\ a \Rightarrow [a] \rightarrow [a]$, so lists of any type will do provided there is a total ordering relation on the elements. The function bwp is defined by

(1) $bwp = map\ last \cdot lexsort \cdot rots$

The function $lexsort :: Ord\ a \Rightarrow [[a]] \rightarrow [[a]]$ sorts a list of lists into lexicographic order and is considered in greater detail in the following section. The function $rots$ returns the rotations of a list and is defined by

$$\begin{aligned} rots &:: [a] \rightarrow [[a]] \\ rots\ xs &= take\ (length\ xs)\ (iterate\ lrot\ xs) \end{aligned}$$

where $lrot\ xs = tail\ xs \mathbin{+\!\!+} [head\ xs]$, so $lrot$ performs a single left rotation. The definition of bwp is constructive, but we won't go into details – at least, not in this pearl – as to how the program can be made more efficient.

The function bwn is specified by

(2) $posn\ (bwn\ xs)\ (lexsort\ (rots\ xs)) = xs$

where $posn\ k$ applied to a list returns the element in position k. It is slightly more convenient in what follows to number positions from 1 rather than 0, so $posn\ (k+1) = (!!k)$ in Haskell-speak. In words, $bwn\ xs$ returns some position at which xs occurs in the sorted list of rotations of xs. If xs is a repeated string, then $rots\ xs$ will contain duplicates, so $bwn\ xs$ is not defined uniquely by (2).

As an illustration, consider the string `yokohama`. The rotations and the lexicographically sorted rotations are as follows:

```
1  y o k o h a m a          6  a m a y o k o h
2  o k o h a m a y          8  a y o k o h a m
3  k o h a m a y o          5  h a m a y o k o
4  o h a m a y o k          3  k o h a m a y o
5  h a m a y o k o          7  m a y o k o h a
6  a m a y o k o h          4  o h a m a y o k
7  m a y o k o h a          2  o k o h a m a y
8  a y o k o h a m          1  y o k o h a m a
```

The output of bwp is the string `hmooakya`, the last column of the second matrix, and bwn `"yokohama"` $= 8$ because row number 8 in the sorted rotations is the input string.

That the BWT helps compression depends on the probability of repetitions in the input. To give a brief illustration, an English text may contain many occurrences of words such as “this”, “the”, “that” and some occurrences of “where”, “when”, “she”, “ he” (with a space), etc. Consequently, many of the rotations beginning with “h” will end with a “t”, some with a “w”, an “s” or a space. The chance is smaller that it would end in a “x”, a “q”, or an “u”, etc. Thus the BWT brings together a smaller subset of alphabets, say, those “t”s, “w”s and “s”s. A move-to-front encoding phase is then able to convert the characters into a series of small-numbered indexes, which improves the effectiveness of entropy-based compression techniques such as Huffman-encoding. For a fuller picture of the role of the BWT in data compression, consult [1,4].

The inverse transform $unbwt :: Int \to [a] \to [a]$ is specified by

(3) $unbwt\,(bwn\,xs)\,(bwp\,xs) = xs$

To compute *unbwt* we have to show how the lexicographically sorted rotations of a list, or at least its tth row where $t = bwn\,xs$, can be recreated solely from the knowledge of its last column. To do so we need to examine lexicographic sorting in more detail.

3 Lexicographic sorting

Let $(\leq) :: a \to a \to Bool$ be a linear ordering on a. Define $(\leq_k) :: [a] \to [a] \to Bool$ inductively by

$$\begin{aligned} xs \leq_0 ys &= True \\ (x : xs) \leq_{k+1} (y : ys) &= x < y \vee (x = y \wedge xs \leq_k ys) \end{aligned}$$

The value $xs \leq_k ys$ is defined whenever the lengths of xs and ys are equal and no smaller than k.

Now, let $sort\,(\leq_k) :: [[a]] \to [[a]]$ be a stable sorting algorithm that sorts an $n \times n$ matrix, given as a list of lists, according to the ordering $\leq_k$. Thus $sort\,(\leq_k)$, which we henceforth abbreviate to $sort\,k$, sorts a matrix on its first k columns. Stability means that rows with the same first k elements appear in their original order in the output matrix. By definition, $lexsort = sort\,n$.

Define $cols\,j = map\,(take\,j)$, so $cols\,j$ returns the first j columns of a matrix. Our aim in this section is to establish the following fundamental relationship: provided $1 \leq j \leq k$ we have

(4) $cols\,j \cdot sort\,k \cdot rots = sort\,1 \cdot cols\,j \cdot map\,rrot \cdot sort\,k \cdot rots$

This looks daunting, but take $j = n$ (so $cols\,j$ is the identity), and $k = n$ (so $sort\,k$ is a complete lexicographic sorting). Then (4) states that the following transformation on the sorted rotations is the identity: move the last column to the front and resort the rows on the new first column. As we will see, this implies that the (stable) permutation that produces the first column from the last column is the same as that which produces the second from the first, and so on.

To prove (4) we will need some basic properties of rotations and sorting. For rotations, one identity suffices:

(5) $\mathit{map}\ \mathit{rrot} \cdot \mathit{rots} = \mathit{rrot} \cdot \mathit{rots}$

where *rrot* denotes a single right rotation. More generally, applying a rotation to the columns of a matrix of rotations has the same effect as applying the same rotation to the rows.

For sorting we will need

(6) $\mathit{sort}\ k \cdot \mathit{map}\ \mathit{rrot}^k = (\mathit{sort}\ 1 \cdot \mathit{map}\ \mathit{rrot})^k$

where f^k is the composition of f with itself k times. This identity formalises the fact that one can sort a matrix on its first k columns by first rotating the matrix to bring these columns into the last k positions, and then repeating k times the process of rotating the last column into first position and stable sorting according to the first column only. Since $\mathit{map}\ \mathit{rrot}^n = \mathit{id}$, the initial processing is omitted in the case $k = n$, and we have the standard definition of *radix sort*. In this context see [2] which deals with the derivation of radix sorting in a more general setting.

It follows quite easily from (6) that

(7) $\mathit{sort}\ (k+1) \cdot \mathit{map}\ \mathit{rrot} = \mathit{sort}\ 1 \cdot \mathit{map}\ \mathit{rrot} \cdot \mathit{sort}\ k$

Finally, we will need the following properties of columns. Firstly, for arbitrary j and k:

(8) $$\mathit{cols}\ j \cdot \mathit{sort}\ k = \mathit{cols}\ j \cdot \mathit{sort}\ (j\ \mathbf{min}\ k) = \mathit{sort}\ (j\ \mathbf{min}\ k) \cdot \mathit{cols}\ j$$

In particular, $\mathit{cols}\ j \cdot \mathit{sort}\ k = \mathit{cols}\ j \cdot \mathit{sort}\ j$ whenever $j \leq k$. Since $\mathit{sort}\ j \cdot \mathit{cols}\ j$ is a complete sorting algorithm that does not depend on the order of the rows, we have

(9) $\mathit{sort}\ j \cdot \mathit{cols}\ k \cdot \mathit{perm} = \mathit{sort}\ j \cdot \mathit{cols}\ k$

whenever $j \leq k$ and *perm* is any function that permutes its argument.

With $1 \leq j \leq k$ we can now calculate:

$$\begin{array}{ll}
& \mathit{sort}\ 1 \cdot \mathit{cols}\ j \cdot \mathit{map}\ \mathit{rrot} \cdot \mathit{sort}\ k \cdot \mathit{rots} \\
= & \{\text{identity (8)}\} \\
& \mathit{cols}\ j \cdot \mathit{sort}\ 1 \cdot \mathit{map}\ \mathit{rrot} \cdot \mathit{sort}\ k \cdot \mathit{rots} \\
= & \{\text{identity (7)}\} \\
& \mathit{cols}\ j \cdot \mathit{sort}\ (k+1) \cdot \mathit{map}\ \mathit{rrot} \cdot \mathit{rots} \\
= & \{\text{identity (8)}\} \\
& \mathit{cols}\ j \cdot \mathit{sort}\ k \cdot \mathit{map}\ \mathit{rrot} \cdot \mathit{rots}
\end{array}$$

$$= \quad \{\text{identity } (5)\}$$

$$\mathit{cols}\, j \cdot \mathit{sort}\, k \cdot \mathit{rrot} \cdot \mathit{rots}$$

$$= \quad \{\text{identity } (9)\}$$

$$\mathit{cols}\, j \cdot \mathit{sort}\, k \cdot \mathit{rots}$$

Thus, (4) is established.

4 The derivation

First observe that for $1 \leq j$

$$\mathit{cols}\, j \cdot \mathit{map}\ \mathit{rrot} = \mathit{join} \cdot \mathit{fork}\,(\mathit{map}\ \mathit{last}, \mathit{cols}\,(j-1))$$

where $\mathit{join}\,(\mathit{xs}, \mathit{xss})$ is the matrix xss with xs adjoined as a new first column, and $\mathit{fork}\,(f, g)\, x = (f\ x, g\, x)$. Hence

$$\mathit{cols}\, j \cdot \mathit{sort}\, k \cdot \mathit{rots}$$

$$= \quad \{(4)\}$$

$$\mathit{sort}\, 1 \cdot \mathit{cols}\, j \cdot \mathit{map}\ \mathit{rrot} \cdot \mathit{sort}\, k \cdot \mathit{rots}$$

$$= \quad \{\text{above}\}$$

$$\mathit{sort}\, 1 \cdot \mathit{join} \cdot \mathit{fork}\,(\mathit{map}\ \mathit{last}, \mathit{cols}\,(j-1)) \cdot \mathit{sort}\, k \cdot \mathit{rots}$$

$$= \quad \{\text{since } \mathit{fork}\,(f, g) \cdot h = \mathit{fork}\,(f \cdot h, g \cdot h)\}$$

$$\mathit{sort}\, 1 \cdot \mathit{join} \cdot \mathit{fork}\,(\mathit{map}\ \mathit{last} \cdot \mathit{sort}\, k \cdot \mathit{rots}, \mathit{cols}\,(j-1) \cdot \mathit{sort}\, k \cdot \mathit{rots})$$

$$= \quad \{\text{definition of } \mathit{bwp}\}$$

$$\mathit{sort}\, 1 \cdot \mathit{join} \cdot \mathit{fork}\,(\mathit{bwp}, \mathit{cols}\,(j-1) \cdot \mathit{sort}\, k \cdot \mathit{rots})$$

Setting $\mathit{recreate}\, j = \mathit{cols}\, j \cdot \mathit{sort}\, k \cdot \mathit{rots}$, we therefore have

$$\begin{aligned} \mathit{recreate}\, 0 &= \mathit{map}\,(\mathit{take}\, 0) \\ \mathit{recreate}\,(j+1) &= \mathit{sort}\, 1 \cdot \mathit{join} \cdot \mathit{fork}\,(\mathit{id}, \mathit{recreate}\, j) \end{aligned}$$

The last equation is valid only for $j + 1 \leq k$. The Haskell code for *recreate* is given in Figure 1. The function $\mathit{sortby} :: \mathit{Ord}\ a \Rightarrow (a \to a \to \mathit{Bool}) \to [a] \to [a]$ is a stable variant of the standard function *sortBy*.

In particular, taking $k = n$ we have $\mathit{unbwt}\, t = \mathit{posn}\, t \cdot \mathit{recreate}\, n$. This implementation of *unbwt* involves computing $\mathit{sort}\, 1$ a total of n times. To avoid repeated sorting, observe that $\mathit{recreate}\, 1\ \mathit{ys} = \mathit{sort}\,(\leq)\ \mathit{ys}$, where $\mathit{sort}\,(\leq)$ sorts a list rather than a matrix of one column. Furthermore, for some suitable permutation function sp we have

```
> recreate :: Ord a => Int -> [a] -> [[a]]
> recreate 0 ys     = map (take 0) ys
> recreate (k+1) ys = sortby leq (join ys (recreate k ys))
>     where leq us vs   = head us <= head vs
>           join xs xss = [y:ys | (y,ys) <- zip xs xss]
```

Fig. 1. Computation of *recreate*

```
> unbwt :: Ord a => Int -> [a] -> [a]
> unbwt t ys = take (length ys) (thread (spl t))
>   where thread (x,j) = x:thread (spl j)
>         spl = lookup (zip [1..] (sortby (<=) (zip ys [1..])))
```

Fig. 2. Computation of *unbwt*

$$sort\,(\leq)\,ys = apply\;sp\;ys$$

where $apply :: (Int \rightarrow Int) \rightarrow [a] \rightarrow [a]$ applies a permutation to a list:

$$apply\;p\,[x_1,\ldots,x_n] = [x_{p(1)},\ldots,x_{p(n)}]$$

It follows that

$$recreate\,(j+1)\,ys = join\,(apply\;sp\;ys, apply\;sp\,(recreate\;j\;ys))$$

Equivalently,

$$recreate\;n\;ys = transpose\,(take\;n\,(iterate1\,(apply\;sp)\;ys))$$

where $transpose :: [[a]] \rightarrow [[a]]$ is the standard library function for transposing a matrix and $iterate1 = tail \cdot iterate$. The tth row of a matrix is the tth column of the transposed matrix, ie. $posn\;t \cdot transpose = map\,(posn\;t)$, so we can use the naturality of $take\;n$ to obtain

$$unbwt\;t\;ys = take\;n\,(map\,(posn\;t)\,(iterate1\,(apply\;sp)\;ys))$$

Suppose we define $spl :: Ord\;a \Rightarrow [a] \rightarrow Int \rightarrow (a, Int)$ by

$$spl\;ys \;=\; lookup\,(zip\,[1..]\,(sort\,(\leq)\,(zip\;ys\,[1..])))$$

where $lookup :: Eq\;a \Rightarrow [(a,b)] \rightarrow (a \rightarrow b)$ is a standard function and $sort\,(\leq)$ sorts a list of pairs. Then

$$spl\;ys\;j = (posn\,(sp\;j)\;ys, sp\;j)$$

Hence

$$map\,(posn\;k)\,(iterate1\,(apply\;sp)\;ys) = thread\,(spl\;ys\;k)$$

where $thread\,(x,j) = x : thread\,(spl\;ys\;j)$.

The final algorithm, written as a Haskell program, is given in Figure 2. If we use arrays with constant-time lookup, the time to compute *unbwt* is dominated by the sorting in *spl*.

```
> unbwt :: Ord a => Int -> Int -> [a] -> [a]
> unbwt k t ys = us ++ reverse (take (length ys - k) vs)
>   where us = posn t yss
>         yss = recreate k ys
>         vs = u:search k (reverse (zip yss ys)) (take k (u:us))
>         u = posn t ys

> search :: Eq a => Int -> [([a],a)] -> [a] -> [a]
> search k table xs = x:search k table' (take k (x:xs))
>                     where (x,table') = dlookup table xs

> dlookup :: Eq a => [(a,b)] -> a -> (b,[(a,b)])
> dlookup ((a,b):abs) x = if a==x then (b,abs)
>                         else (c,(a,b):cds)
>                         where (c,cds) = dlookup abs x
```

Fig. 3. Computation of Schindler's variation

5 Schindler's variation

The main variation of BWT is to exploit the general form of (4) rather than the special case $k = n$. Suppose we define

$$bwp\ k = map\ last \cdot sort\ k \cdot rots$$

This version, which sorts only on the first k columns of the rotations of a list, was considered in [5]. The derivation of the previous section shows how we can recreate the first k columns of the sorted rotations from $ys = bwp\ k\ xs$, namely by computing *recreate k ys*. Although we cannot compute the remaining columns, we can reconstruct the tth row, where $t = bwn\ k\ xs$ and

$$posn\ (bwn\ k\ xs)\ (sort\ k\ (rots\ xs)) = xs$$

The first k elements of xs are given by $posn\ t\ (recreate\ k\ ys)$, and the last element of xs is $posn\ t\ ys$. So certainly we know

$$take\ k\ (rrot\ xs) = [x_n, x_1, \ldots, x_{k-1}]$$

This list begins the *last* row of the unsorted matrix, and consequently, since sorting is stable, will be the *last* occurrence of the list in *recreate k ys*. If this occurrence is at position p, then $posn\ p\ ys = x_{n-1}$. Having discovered x_{n-1}, we know $take\ k\ (rrot^2\ xs)$. This list begins the penultimate row of the unsorted matrix, and will be either the last occurrence of the list in the sorted matrix, or the penultimate one if it is equal to the previous list. We can continue this process to discover all of $[x_{k+1}, \ldots, x_n]$ in reverse order. Efficient implementation of this phase of the algorithm requires building an appropriate data structure for repeatedly looking up elements in reverse order in the list $zip\ (recreate\ k\ ys)\ ys$ and removing them when found. A simple implementation is given in Figure 3

6 Conclusions

We have shown how the inverse Burrows-Wheeler transform can be derived by equational reasoning, and also considered the more general version proposed by Schindler. One interesting topic not touched upon is whether proposed improvements to the original BWT can also be derived in a similar style. The BWT can be modified to sort all the tails of a list rather than rotations, and in [3] it is shown how to do this in $O(n \log n)$ steps using suffix arrays. How fast it can be done in a functional setting remains unanswered, though we conjecture that $O(n(\log n)^2)$ step is the best possible.

References

[1] Burrows, M., and D. J. Wheeler, *A block-sorting lossless data compression algorithm*, Research report 124, Digital Systems Research Center (1994).

[2] Gibbons, J., *A pointless derivation of radix sort*, Journal of Functional Programming, **9**(3) (1999), 339–948.

[3] Manber, U. and M. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM Journal of Computing, **22**(5), (1993), 935–948.

[4] Nelson, M., *Data compression with the Burrows-Wheeler transform*, Dr. Dobb's Journal, September, (1996).

[5] Schindler, M., *A fast block-sorting algorithm for lossless data compression*, Data Compression Conference, (Poster Session), (1997).

[6] Seward, J.,`bzip2`, `http://sourceware.cygnus.com/bzip2/` (1999)

Genuinely Functional User Interfaces

Antony Courtney [1,2]

Dept. of Computer Science
Yale University
New Haven, CT 06520

Conal Elliott [3]

Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract

Fruit is a new graphical user interface library for Haskell based on a formal model of user interfaces. The model identifies *signals* (continuous time-varying values) and *signal transformers* (pure functions mapping signals to signals) as core abstractions, and defines GUIs compositionally as signal transformers. In this paper, we describe why we think a formal denotational model of user interfaces is useful, present our model and prototype library implementation, and show some example programs that demonstrate novel features of our library.

1 Introduction

Over the years, there have been numerous Graphical User Interface (GUI) libraries for Haskell, presenting a broad range of different programming interfaces. Some libraries, such as TkGofer [26], provide direct access to GUI facilities through the `IO` monad, and therefore have a rather imperative feel. Others, such as Fudgets [4] and FranTk [23], present qualitatively more high-level programming interfaces, so have a more declarative, functional feel.

[1] This material is based upon work supported in part by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.
[2] Email: `antony@apocalypse.org`
[3] Email: `conal@microsoft.com`

But what does it mean for one library to be more "high level" or "low level" or "functional" than another? On what basis should we make such comparisons? A pithy answer is given by Perlis [18]:

A programming language is low level when its programs require attention to the irrelevant.
–*Alan Perlis*

But how should we decide what is relevant?

Within the functional programming community, there is a strong historical connection between functional programming and *formal modeling* [1,24,25,12]. Many authors have expressed the view that functional programming languages are "high level" because they allow programs to be written in terms of an *abstract conceptual model* of the problem domain, without undue concern for implementation details.

Of course, although functional languages *can* be used in this "high level" way, this is neither a requirement nor a guarantee. It is very easy to write programs and libraries in pure Haskell that are littered with implementation details and bear little or no resemblance to any abstract conceptual model of the problem they were written to solve.

So if we wish to design a high-level interface for implementing GUIs in Haskell, it seems clear that we must first ask:

What is an abstract conceptual model of a graphical user interface?

and then embed this model in Haskell, so that there is a direct mapping from the types and functions in our library to their counterparts in the conceptual model. As far as we are aware, all previous GUI libraries for Haskell define the conceptual model of a GUI only informally, or defer to some external system (such as X Windows, Tk, Gtk, etc.) for many of the details.

In this paper, we present *Fruit*, a Functional Reactive User Interface Toolkit, based on a formal model of graphical user interfaces. The Fruit model is is based on *AFRP*, an adaptation of ideas from Functional Reactive Animation (Fran) [9,7] and Functional Reactive Programming (FRP) [14,27] to the *arrows* framework recently proposed by Hughes [15]. AFRP is based on two ideas: *signals*, which are functions from real-valued time to values, and *signal transformers*, which are functions from signals to signals. Using only the AFRP model and simple *mouse*, *keyboard* and *picture* types, we define GUIs compositionally as signal transformers.

We believe that developing a simple, precise denotational model of graphical user interfaces is valuable for a number of reasons:

- It provides a starting point for proving properties of programs with graphical interfaces, and for developing related notions of *program equivalence*.
- We can reformulate the question of whether one library is more "high level" than another in precise, objective terms, by comparing the semantic models of the libraries, and asking whether one semantic model is *more abstract*

than another [22].

- By clarifying our understanding of the abstract conceptual model of graphical interfaces, we may gain fresh insight into how to extend the model to new interaction paradigms or see systematic solutions to problems that were previously solved in an ad hoc manner.

We present examples of this final point later in the paper, when we show how continuous spatial scaling (zooming) and multiple views can be accomodated within the Fruit model.

The remainder of this paper is organized as follows. In section 2 we formally define the AFRP programming model, show how the model is embedded in Haskell, and give simple but precise definitions for some useful combinators and primitives. In section 3 we define GUIs within the AFRP model. In section 4, we develop a basic application in Fruit, and show two examples (adding continuous spatial scaling and multiple views) that demonstrate the benefits of our approach. Section 5 discusses related work. Sections 6 and 7 summarize the status of the implementation and present our conclusions.

2 AFRP Programming Model

Like Fran and FRP, the AFRP programming model is implemented as a domain-specific language embedded in Haskell [13]. In order to focus on our new language constructs, we simply assume the existence of a denotational semantics for Haskell in which Haskell functions denote (partial) functions. We define our language extensions by giving denotational definitions for our language constructs that extend this (hypothetical) Haskell semantics.

2.1 Concepts

The Fruit programming model is built around two central concepts: *signals* and *signal transformers.*

Signals

A *signal* is a function from *time* to a value:

$$\mathbf{Signal}\ \alpha \quad = \quad Time \rightarrow \alpha$$

We represent $Time$ as a non-negative real number. An example of a signal is the mouse's current (x, y) position. If $Point$ is the type of two-dimensional points, we can model the time-varying mouse position as a $Signal\ Point$.

Signal Transformers

A *signal transformer* is a function from *Signal* to *Signal*:

$$\mathbf{ST}\ \alpha\ \beta \quad = \quad Signal\ \alpha\ \rightarrow\ Signal\ \beta$$

Informally, we can think of a signal transformer as a box with an "input port" and an "output port". If we connect the box's input port to a *Signal* α value, we can observe a *Signal* β value on the output port.

A simple example of a signal transformer is the identity signal transformer. At every point in time, the identity signal transformer's output signal has the same value as its input signal. Slightly more interesting examples of signal transformers are *lifted functions* (the output signal is the point-wise application of f to the input signal), and `integral` (the output signal is the integration of the input signal over time). Note that the identity signal transformer can be defined as a lifted function, where f is the function *id*.

2.2 Abstract Types

Conceptually, signals are functions of continuous time, and signal transformers are functions from signals to signals. As has been argued elsewhere [9,7], a *continuous* model can be simpler and more natural than a discrete one when modeling animation or user interaction. However, in order to guarantee an efficient implementation on a discrete computer, signal transformers are not written directly as Haskell functions. Instead, we introduce an *abstract type constructor*, `ST`. A value of type `ST a b` denotes a signal transformer:

```
newtype ST a b = ...
```

The implementation provides a number of *primitive* signal transformers, and a set of combinators (the arrow combinators) for assembling new signal transformers from existing ones. Internally, the implementation uses discrete sampling and synchronous streams to approximate the continuous time model. It has been shown that, as the time between samples approaches zero, the discrete implementation converges to the continuous semantics in the limit [27].

Since `ST` is a Haskell type constructor, signal transformers are first-class values: we can pass them as arguments, return them as results, store them in data structures or variables, etc. In contrast, signals are not first-class values. This marks a significant departure from Fran's programming model. Fran's `Behavior` type denotes a signal in the Fruit model, and Fran uses Haskell functions to obtain the equivalent of our signal transformers.

We outlaw signals as first-class values for two reasons. First, signals alone are inherently non-modular: while we can apply point-wise transformations to the observable *output* of a signal, first-class signal values do not have an input signal. In contrast, signal transformers have both an input signal and an output signal, thus enabling us to transform both aspects of an `ST` value. In other words, only providing `ST` as first-class values guarantees that every signal in the program is always *relative* to some input signal.

Second, experience implementing Fran [6] and FRP [14] taught us that allowing signals as first class values inevitably leads to "space-time leaks" [6] in the implementation. A "space-time leak" occurs when the implementation needs the complete time-history of a signal to compute one sample value.

Defining `ST` as a `newtype` and only providing a fixed set of primitives and combinators allows us to prove that the implementation is free of space-time leaks by simple structural induction on the `ST` type.

2.3 Arrows

Recently, Hughes proposed *arrows* as a basis for building combinator libraries [15]. Concretely, *Arrow* is a Haskell type class that denotes a *computation* from some input type to some result type. In his introduction to arrows [15], Hughes presents a number of examples of arrow instances, including Haskell's built-in function type constructor `(->)` and stream processors, and gives many examples that demonstrate the utility of arrows for organizing combinator libraries.

The Arrow type class is defined as:

```
class Arrow a where
  arr :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

In the remainder of this section, we give both informal and formal definitions of each of the arrow operators for the `ST` type constructor in terms of our model.

Lifting

One of the most common and useful kinds of signal transformers is a *lifted* function, produced by the `arr` operator. The `arr` operator for signal transformers has type:

```
arr :: (b -> c) -> ST b c
```

Given any Haskell function `f` of type `(b -> c)` (i.e. a function mapping b values to c values), `arr f` denotes a signal transformer that maps a *Signal* b to a *Signal* c by applying `f` point-wise to the input signal.

For example, given the function

```
sin :: Floating a => a -> a
```

from the standard Prelude, `arr sin` is a signal transformer whose output signal at time t is `sin` applied to the signal transformer's input signal at time t.

Formally, we define `arr f` for signal transformers as follows:

$$[\![\texttt{arr f}]\!] \;=\; \lambda s : \textbf{Signal}\ \alpha\ .\ \lambda t : \mathit{Time}\ .\ [\![\texttt{f}]\!](s(t))$$

Serial Composition

The arrow infix operator `(>>>)` composes two arrows. For signal transformers, if we have:

```
fa :: ST b c
```

```
ga :: ST c d
```

then `fa >>> ga` has type `ST b d` and denotes the signal transformer that feeds its input signal to `fa`, uses `fa`'s output signal as `ga`'s input signal, and uses `ga`'s output signal as the resulting output signal.

Formally, we define serial composition for signal transformers as reverse function composition:

$$[\![\texttt{fa >>> ga}]\!] = ([\![\texttt{ga}]\!] \circ [\![\texttt{fa}]\!])$$

Widening

Given an arrow from `b` to `c`, the `first` operator widens it to be an arrow from `(b,d)` to `(c,d)`, for all types `d`. For signal transformers, the `first` operator has type:

```
first :: ST b c -> ST (b,d) (c,d)
```

Informally, `(first fa)` denotes a signal transformer that feeds the first half of its input signal (a signal of `b` values) to `fa` to produce a signal of `c` values, and pairs this with the second half of the original input signal (a signal of `d` values) to produce the output signal.

Formally, we can define `first` as:

$$[\![\texttt{first fa}]\!] =$$

$$\lambda s : \mathit{Signal}(\beta \times \gamma)\ .\ \texttt{pairZ}\ ([\![\texttt{fa}]\!]\ (\texttt{fstZ}\ s))\ \ (\texttt{sndZ}\ s)$$

where `fstZ`, `sndZ` and `pairZ` are the obvious projection and pairing functions for signals of pairs.

ArrowLoop

The names “signal” and “signal transformer” in our model suggest analogies to analog and digital signal processing and computer hardware. In those domains, *feedback cycles* are used in conjunction with the inherent propagation delay of wires to implement many interesting circuits such as flip-flops or latches. In a feedback cycle, some portion of the output signal is *fed back* as an input signal. Feedback cycles are also useful in Fruit, and are defined using the `loop` combinator [16]. The `loop` combinator is defined in the `ArrowLoop` type class:

```
class Arrow a => ArrowLoop a where
        loop :: a (b,d) (c,d) -> a b c
```

For signal transformers, if `fa` has type `ST (b,d) (c,d)`, then `loop fa` denotes a signal transformer that instantiates `fa`, and pairs the second half of `fa`'s output signal with an external input signal to form `fa`'s input signal.

Formally, we define `loop` for signal transformers as:

$$[\![\texttt{loop fa}]\!]\ =$$

$$\lambda s : \textbf{Signal}\ \beta .\ \texttt{fstZ}(\mathbf{Y}(\lambda r.[\![\texttt{fa}]\!](\texttt{pairZ}\ s\ (\texttt{sndZ}\ r))))$$

where **Y** is the standard least fixed point operator.

Discrete Events

In modeling reactive systems in general (and user interfaces in particular), we often need to model *event sources* that produce *event occurrences* at discrete points in time. For example, the left mouse button being pressed is naturally modeled as an event that occurs at some point in time. For now, we define event sources in our conceptual model as signals of `Maybe` values:[4]

$$\begin{aligned}\textbf{EventSource}\ \alpha &= \textbf{Signal}\ (\texttt{Maybe}\ \alpha)\\ &= \mathit{Time}\ \rightarrow\ (\texttt{Maybe}\ \alpha)\end{aligned}$$

If the value of an event source at time t is `Nothing`, then we say that the event source does not occur at time t. Conversely, if the value of an event source at time t is `Just v`, then we say that the event source has an occurrence at time t that carries value `v`.

As with *Signal*, *EventSource* lives in the conceptual model, and does not appear directly in our API. However, `Maybe` types appear as arguments to the `ST` type constructor when event sources are needed. We will see an example of this shortly.

2.4 Primitive Signal Transformers

Fruit defines a number of primitive signal transformers. Each such primitive has a denotational definition in terms of our formal model. The denotational definitions of these primitives are derived directly from their counterparts in Fran and FRP. We define a couple of these primitives here to give a taste of the semantics. The interested reader is referred to the denotational definitions of Fran and FRP semantics for a more complete account [9,7,27].

Piecewise Constant Signals

Given an event source, it is often useful to derive a continuous signal whose value is constant between event occurrences. This is sometimes called a "sample and hold" or "zero-order hold" circuit in the signal processing literature. The primitive `stepper` is provided for this purpose:

```
stepper :: a -> ST (Maybe a) a
```

Informally, `stepper x0` denotes a signal transformer that transforms an *EventSource* α to a *Signal* α. Initially, the output signal of `stepper x0` has value `x0`. When an event carrying value `x1` occurs on its input signal,

[4] This representation of events as continuous signals of `Maybe` values raises some thorny theoretical issues because it allows for *dense* event sources (ones that have infinitely many occurrences in a finite interval of time). We have explored some possible solutions to this problem [27], but a detailed exploration of this issue is outside the scope of this paper.

the value of the stepper's output signal becomes `x1`. The value of the output signal remains `x1` until the next event occurrence (carrying, say, `x2`), at which point the value of the output signal becomes `x2`, and so on.

Formally, we define `stepper` as follows:

$$[\![\texttt{stepper x}]\!] = \lambda s.\lambda t. \begin{cases} \texttt{a} \; \exists \texttt{a}.\exists t_a \in (0,t).((s\ t_a) = \texttt{Just a}) \\ \qquad \wedge \; \nexists t_b \in (t_a,t).((s\ t_b) = \texttt{Just b}) \\ \\ \texttt{x} \; \textit{otherwise} \end{cases}$$

Integration

The primitive signal transformer `integral` has type:

```
integral :: Floating a => ST a a
```

The output signal of `integral` is the integration of its input signal over time. Formally:

$$[\![\texttt{integral}]\!] \;=\; \lambda s.\lambda t. \int_0^t s(t)dt$$

3 Fruit: A Compositional User Interface Library

We define an interactive graphical user interface (`GUI`) as:

```
type GUI a b = ST (GUIInput, a) (Picture, b)
```

A `GUI a b` is a signal transformer that takes a graphical input signal (`GUIInput`) paired with an auxiliary semantic input signal (`a`) and produces a graphical output signal (`Picture`) paired with an auxiliary semantic output signal (`b`).

In the Fruit model, every interactive component is a value of type `GUI a b` (for some types `a` and `b`). This differs from conventional toolkits in which there are distinct types for "applications", "containers" and "components". We consider this flat type structure a feature, as it leads to a *compositional* model of user interfaces. Any two GUI values can be composed using our layout combinators to form a composite GUI in which the two child GUIs appear adjacent to one another. The result returned from the layout combinator is itself a GUI, and so can also be used in a layout combinator, displayed in a top-level window, etc.

The `GUIInput` and `Picture` signals allow the application to feed time-varying keyboard and mouse information into the GUI, and get back time-varying visual information (pictures to display). The auxiliary input and output signals allow a GUI to observe and emit extra (time-varying) information for use in the rest of the program. For example, a GUI representing a button component might provide an event source output signal that has occurrences when the button is pressed.

The `Picture` type is an abstract type that denotes a static picture that can be rendered on screen. Our prototype implementation uses a scalable vector graphics library based loosely on the graphics library defined in “The Haskell School of Expression” (SOE) [14], but any picture type that supports basic geometric primitives, bounds calculations and affine transforms would work.

3.1 The GUIInput Type

The `GUIInput` type represents the part of the input to a GUI specifically related to its visual interactive characteristics. `GUIInput` is essentially just a pair of records:

```
data Mouse = {mpos :: Point,
              lbDown :: Bool,
              rbDown :: Bool }
data Kbd = { keyDown :: [Char] }

type GUIInput = (Maybe Kbd, Maybe Mouse)
```

The `Kbd` and `Mouse` types are wrapped in `Maybe` types to account for the *focus model*. In modern window systems, there is always a foreground application that receives the keyboard and mouse input from the window system to the exclusion of all other applications running in the background. The window system typically provides a lightweight gesture (such as mouse-over or click-to-type) that allows the user to shift the focus to another application. This concept of focus model is equally applicable within a window, as we can view moving the mouse between two different visible components of a window as shifting the mouse focus from one component to the other. Keyboard focus traversal within a window (using the TAB key, for example) can be modeled analogously.

Each of the `Maybe` values in the `GUIInput` signal to a GUI are `Nothing` when the GUI does not have focus, and `Just` x (for some x) when the component has the focus. Note that, although the types of these signals are the same as a discrete event source, they are, conceptually, *not* discrete event sources. As it turns out, however, many of the event source combinators also have useful interpretations for such continuous `Maybe` signals. We will see several examples of this.

3.2 Composing GUIs

These definitions, combined with the arrow combinators and primitive behaviors from the previous section, form the basis of our GUI library. Even without any additional definitions, we can define many useful and interesting richly interactive user interfaces.

For example, we can define `mouseST` as a signal transformer that takes a `GUIInput` input signal and produces a `Point` output signal that is the mouse's current position if the GUI has the focus, or "remembers" the last position that had focus otherwise:

```
mouseST :: ST GUIInput G.Point
mouseST = arr snd >>> arr (fmap mpos)
                      >>> stepper G.origin2
```

The `G` refers to the qualified import of the graphics library; `G.origin2` is the Cartesian origin. Note that we are feeding the `Maybe Mouse` signal to the `stepper` event source combinator. The result is a continuous signal that maintains the last position of the mouse when the GUI loses mouse focus.

Using this definition, here is a definition for a GUI that draws a red ball that follows the mouse:

```
-- from the graphics library:
move :: Picture -> Point -> Picture

ballPic :: Picture
ballPic = (circle ‘withColor‘ red)

ballGUI :: GUI () ()
ballGUI = first (mouseST >>> arr (move ballPic))
```

In the above definition, `ballGUI` is given type `GUI () ()` because it neither observes nor produces any semantic signals other than its `GUIInput` input signal and its `Picture` output signal. The subexpression `(mouseST >>> arr (move ballPic))` has type `ST GUIInput Picture`. By using the `first` operator, we widen this `ST` value to one of type `ST (GUIInput,a) (Picture,a)` for all types a, and this generalized type is of course equivalent to `GUI a a`.

3.3 Running a GUI

A GUI is brought to life with the `runGUI` action, which runs a GUI in a top-level window:[5]

```
runGUI :: Unit a => GUI a b -> IO ()
```

The implementation of `runGUI` handles all low-level (imperative) communication with the graphics library to read primitive window events and draw pictures on the screen. The window displayed by the action `runGUI ballGUI`

[5] For convenience, we define a type class `Unit` with an instance for `()` and all products of `Unit`, such as `()`, `((),())`, etc. This will be convenient for simple `GUIs` composed with layout combinators, as we shall see later.

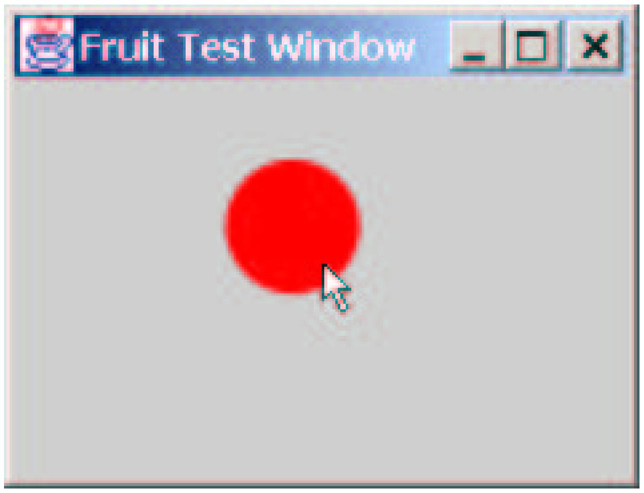


Fig. 1. Running ballGUI

is shown in figure 1.

3.4 Brief Aside: Arrows Syntactic Sugar

When defining signal transformers using the arrow combinators, we must write definitions in a *point-free* style. In the context of Fruit, this means that the names in our program refer to *signal transformers*, but we cannot name *signals* explicitly.

The arrows syntactic sugar is a proposal by Ross Paterson [16] with an existing implementation as a preprocessor. The arrows syntactic sugar allows arrows to be defined using a new syntactic form, introduced by the keyword `proc`. The `proc` form acts as a kind of *abstraction* for arrows, analogous to Haskell's built-in lambda abstraction. Within the body of a `proc`, the arrows syntactic sugar allows us to explicitly name the *signals*, and specify how the signals are connected within a signal transformer.

As with lambda abstraction, `proc` takes a *pattern* that will be matched *point-wise* over the points of the arrow. The identifiers used in the pattern may then be used within the body of the arrow. For example, we could have defined `mouseST` using the arrows syntactic sugar as:

```
mouseST :: ST GUIInput G.Point
mouseST = proc (_, mbm) -> do
  ... -- not shown (yet)
```

In this definition, the pattern `(_,mbm)` is matched against the *input type* (`GUIInput` here).

Informally, the body of an arrow definition consists of a sequence of *arrow applications* of the form:

$$
\begin{array}{rcl}
pat_1 & \xleftarrow{st_1}\!\!\prec & exp_1 \\
pat_2 & \xleftarrow{st_2}\!\!\prec & exp_2 \\
 & \ldots & \\
pat_{n-1} & \xleftarrow{st_{n-1}}\!\!\prec & exp_{n-1} \\
 & \xrightarrow{st_n}\!\!\prec & exp_n
\end{array}
$$

Each such arrow application feeds the signal described by exp_i to the signal transformer st_i. Each pat_i is matched against the output type of st_i, and introduces new *arrow-bound variables* for use in the arrow applications that follow. The final such application (which does not include a pattern) defines the output signal of the entire `proc`. Note that, in the conversion to the ASCII character set, $\xleftarrow{st}\!\!\prec$ is written as `<- st -<`. Our complete definitions for `mouseST` and `ballGUI` using the arrows syntactic sugar are thus:

```
mouseST = proc (_, mbm) -> do
  stepper G.origin2 -< fmap mpos mbm

ballGUI :: GUI () ()
ballGUI = proc (gin,_) -> do
  mouse <- mouseST -< gin
  returnA -< (move ballPic mouse,())
```

where `returnA` is defined as `arr id` in the arrows library.

The subset of the arrows syntactic sugar used in this paper is defined formally by the following translation:

```
proc p -> do { e1 -< e2 } =
  arr (\p -> e2) >>> e1

proc p -> do { p' <- e1 -< e2; A } =
  returnA &&& arr (\p -> e2) >>> second e1 >>>
  proc (p,p') -> do {A}

proc p -> do { let p' = e; A } =
  returnA &&& arr (\p -> e) >>> proc (p,p') -> do {A}

proc p -> do { rec {A}; B } =
  returnA &&& loop proc (p,pA) -> do
    { A; returnA -< (pB,pA)} >>> proc (p,pB) -> do { B }
```

3.5 Simple Components

A GUI's *auxiliary semantic* input and output signals convey semantic signals to and from the GUI not directly related to the GUI's visual behavior, and enable the GUI to be connected to the rest of the program. The Fruit library defines a number of GUI components that use these auxiliary signals.

Labels

The simplest components are labels, defined as:

```
flabel :: GUI LabelConf ()

ltext :: String -> LabelConf
```

A label is a GUI whose picture displays a text string from its auxiliary input signal, and produces no semantic output signal.

We use a trick from Fudgets [4] to specify configuration options. `LabelConf`, `ButtonConf`, etc. are simple *State* → *State* functions. These functions are very similar to the update functions generated by using Haskell's labeled field syntax, in that they will update one component of the state, but leave all others unchanged. This gives us a simple mechanism for composing property definitions (using the function composition operator '`.`') and for assigning default values for component properties. We will see an example of this shortly.

Buttons

A Fruit button (`fbutton`) is a GUI that implements a standard button control. The declaration of `fbutton` is:

```
fbutton :: GUI ButtonConf (Maybe ())

btext :: String -> ButtonConf
enabled :: Bool -> ButtonConf
```

This declares `fbutton` as a GUI that, in addition to its visual input and output signals, takes an input signal of configuration options specifying properties of the button such as the label to display in the button, whether the button is enabled, etc. The button produces an output event source that has an occurence when the button is pressed by the user. Each event occurence on the output signal carries no information other than the fact of its occurence, hence the type `Maybe ()`. Here is an example of a GUI that uses an `fbutton`:

```
butTest :: GUI () (Maybe ())
butTest = proc (inpS,_) -> do
  fbutton -< (inpS,btext "press me!")
```

The display produced by `runGUI butTest` is shown in figure 2. Note that the above example did not need to explicitly specify the button's enabled property (which is `True` by default).

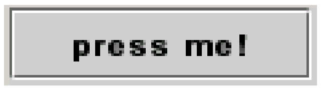


Fig. 2. A simple button

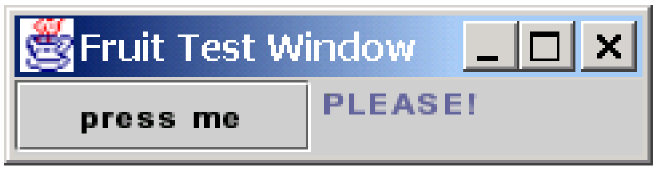


Fig. 3. Using besideGUI

3.6 Basic Layout Combinators

To be able to build more interesting interfaces, we need a mechanism to compose multiple GUIs into a larger GUI. We provide two basic *layout combinators* for this purpose:

```
aboveGUI :: GUI b c -> GUI d e -> GUI (b,d) (c,e)
besideGUI :: GUI b c -> GUI d e -> GUI (b,d) (c,e)
```

The layout combinators produce a combined GUI that behaves as the two child GUIs arranged adjacent to one another. Here is a small example that illustrates the use of `besideGUI`:

```
hello :: GUI () (Maybe (),())
hello = proc (inpS,_) -> do
  (fbutton ‘besideGUI‘ flabel) -<
        (inpS, (btext "press me",
                ltext " PLEASE! "))
```

The result of running this GUI in a top-level window with `runGUI` is shown in figure 3. A *translation transformation* has been applied to the second argument GUI to position it beside the first argument. The implementation of spatial transformation for GUIs will be described in detail in section 4.4.1.

In addition to transforming the second argument, the layout combinators must *demultiplex* the input signal into two disjoint signals to be passed to each child. This is achieved by *clipping* the `GUIInput` signal based on the mouse position: The GUI under the mouse receives the (appropriately transformed) keyboard and mouse signals, while its sibling receives `Nothing` values for the keyboard and mouse to indicate that it does not have focus. [6]

As this example illustrates, the composed GUI has auxiliary semantic input and output signals whose types are the product of the corresponding types

[6] Our current implementation of focus is based solely on mouse position. This is slightly simplistic, as modern user interface guidelines stipulate a keyboard focus cycle that is independent of the mouse focus. Extending our implementation to support such a split focus model is straightforward.

from the child GUIs. This has substantial syntactic consequenses. Programs can become complicated rather quickly, because the types of the composed GUI grow in proportion to the nesting depth of the layout. We have written numerous small example programs using our layout combinators without the arrows syntactic sugar, and have found the resulting programs to be a hopelessly unreadable mess of lifted tupling and untupling. We were exploring possible GUI-specific syntactic extensions when we encountered the arrows syntactic sugar proposal. We have been pleasantly surprised by just how well the syntactic sugar works for a specific problem domain (GUIs) for which it wasn't specifically designed.

4 Composing Applications

In this section, we demonstrate Fruit by developing a basic application. The application (a "Paddleball" game with a button for restarting the game) is small enough to allow us to study it in detail, but substantial enough to capture some of the essential issues that arise in building larger applications.

4.1 Paddleball as a GUI

Hudak [14] develops an implementation of a simple Fran-like reactive animation language, and implements "Paddleball in Twenty Lines" as a demonstration of the power and elegance of functional reactive programming. Since the Fruit model subsumes the functional reactive model on which which it is based, it was a simple matter to re-implement paddleball as a `GUI`. The complete source code is shown in figure 4.

A couple of combinators used in `pball` that we have not yet explained are:

```
-- An accumulating stepper: On every event occurence,
-- function carried with the occurence is applied to
-- the state.
stepAccum :: a -> ST (Maybe (a -> a)) a

-- replaces the value in an event occurence with
-- a new value:
ebind :: a -> Maybe b -> Maybe a
ebind = fmap . const
```

Essentially, the code for `pball` does the following:

- Sets up the ball. The ball's x and y position (`xpos` and `ypos`) are defined as the integral of velocity (`xvel` and `yvel`, respectively). The velocities are defined as piece-wise constant signals using `stepAccum`; both start at `vel` (the game velocity given as an argument to `pball`), but flip sign (`negate`) when a bounce event occurs. Note that these definitions are mutually recursive: `xpos` is defined in terms of `xvel`, which is in turn defined in terms

```
paddle :: Double -> G.Rectangle2DDouble
paddle xpos =  G.rectangle (xpos - 25) 200 50 10

-- The paddleball game, capable of playing one game
-- Output signal is an Event Source that occurs
-- when the game ends.
pball :: Double -> GUI () (Maybe ())
pball vel = proc (inpS,_) -> do
  rec xi <- integral -< xvel
      let xpos = 30 + xi
      yi <- integral -< yvel
      let ypos = 30 + yi
      let ballS = ell (xpos-12.5) (ypos-12.5) 25 25
      let ballPicS = G.shapePic ballS `G.withColor`
                          G.yellow
      xbounce <- when -< ((xpos > 175) || (xpos < 30))
      ybounce <- when -< ((ypos < 30) || hitPaddle)
      let hitPaddle = intersects ballS paddleS
      xvel <- stepAccum vel -< ebind negate xbounce
      yvel <- stepAccum vel -< ebind negate ybounce
      mpos <- mouseST -< inpS
      let paddleS = paddle (G.pointX mpos)
      let paddlePicS = G.shapePic paddleS `G.withColor`
                          G.green
      gameOver <- when -< ypos > 200
      let gamePic = G.box
            (walls `G.over` paddlePicS `G.over`
               ballPicS) gameBox
  returnA -< (gamePic,gameOver)
```

Fig. 4. Paddleball GUI source code

of `xbounce`, defined in terms of `xpos`. The `do rec ...` form of the arrows syntactic sugar takes care of setting up the appropriate connections by using the `loop` combinator (from `ArrowLoop`), and the fact that the `integral` of a signal at time t depends on the values of the input signal up to (but not including) time t ensures that the feedback cycle is well-defined.

- Sets up the Paddle: This is just a rectangle shape (`paddle`), whose x position is determined by the mouse position.
- Performs Collision Detection: This is handled by the definitions of `xbounce`, `ybounce` and `hitPaddle`. `hitPaddle` is defined by a call to the graphics library to check for the intersection of the ball (`ballS`) with the paddle (`paddleS`). `xbounce` and `ybounce` are defined using the `when` combinator:

```
when :: ST Bool (Maybe ())
```

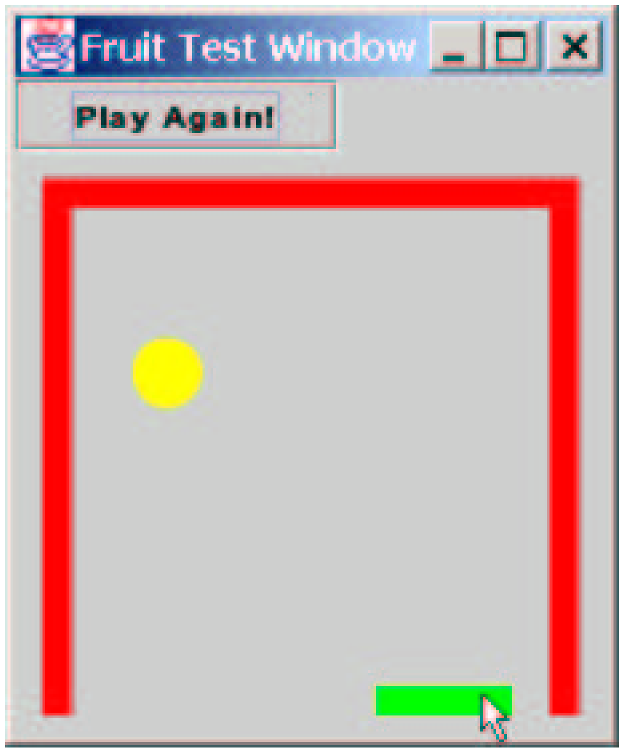


Fig. 5. Paddleball with a Restart Button

> The `when` combinator converts a continuous Boolean signal to an event source. The output event occurs when a *rising edge* is observed on the input signal.

By implementing the Paddleball game as a GUI we obtain spatial modularity (relative to Fran and FRP), and this, in turn, enables reuse: we can use the layout combinators to compose `pball` with other GUIs to form more interesting composite GUIs, and we can have as many Paddleball games active in our GUI as we wish.

4.2 Adding a "Start Over" Button.

Paddleball is a fun game, but, as defined here, it only plays one game. Our first refinement to the game is to add a restart button that allows us to play again, as show in figure 5. We define `rpball0` ("restartable" paddle ball) as follows: [7]

```
-- pbgame is a version of pball that
-- restarts the game when its input
```

[7] We have omitted the code for `pbgame` here to save space. It is easily derived from `pball` using the FRP `switch` combinator, which in AFRP has the signature:
`switch ::  ST b c -> ST (b,Maybe (ST b c)) c`
See [27] for a detailed discussion of the semantics of `switch`.

```
-- event source has an occurrence:
pbgame :: Double -> GUI (Maybe ()) (Maybe ())

-- paddle ball with a reset button:
rpball0 :: Double -> GUI () ()
rpball0 vel = proc (inpS,_) -> do
  rec (picS,(pressES,_)) <-
         (fbutton `aboveGUI` (pbgame vel)) -<
           (inpS,(btext "Play Again!", pressES))
  returnA -< (picS,())
```

The definition of `rpball0` uses the `do rec...` form of the arrows syntactic sugar to feed the output event source from the reset button (`pressES`) back as an input event source to `pbgame`.

4.3 Selectively Enabling the Reset Button

When implementing GUIs, we frequently need to dynamically disable certain components of the user interface based on the program's state. Components that are disabled are typically rendered with a "grayed out" appearance to give the user a visual cue that the corresponding action is invalid.

We demonstrate this kind of programming in Fruit by disabling the reset button while a game is in progress:

```
-- like rpball0, but selectively disable
-- the restart button:
rpball1 :: Double -> GUI () ()
rpball1 vel = proc (inpS,_) -> do
  rec (picS,(restart,gameEnds)) <-
         (fbutton `aboveGUI` (pbgame vel)) -<
              (inpS,(bprops,restart))
      let bprops = (btext "Play Again!"
                  . enabled allowRestart)
      let gameDone = (ebind True gameEnds)
                  `emerge` (ebind False restart)
      allowRestart <- stepper False -< gameDone
  returnA -< (picS,())
```

The `enabled` property of the button is controlled by the `allowRestart` signal, which is `False` initially, set to `True` when a game ends, and set to `False` again when the `restart` button is pressed. As mentioned in section 3.5, the function composition operator (`.`) is used to combine button properties in the definition of `bprops`.

4.4 Exploring Modularity

Thus far our discussion has simply explored how we can implement user interfaces in a purely functional way. This is certainly an interesting academic exercise, and carries with it (we hope) the benefits of increased reasoning power that we expect from purely functional programming models. But we might be tempted to ask if there are any other purely pragmatic advantages to a purely functional approach? In this section, we explore two such advantages: *continuous spatial scaling* and *multiple views.*

4.4.1 Transforming GUIs

One difference between Fruit and every other production user interface toolkit we are aware of (for either imperative or functional languages) is that Fruit provides a uniform model and programming interface for both "low-level" interactive graphics and "high level" user interface components such as buttons. Moreover, since GUIs are first class values that denote *pure functions*, we can use higher-order operators to manipulate GUIs in useful ways.

One of the most basic higher-order functions is the function composition operator `(.)`; we use `>>>` instead, but the denotation is equivalent. (Recall that `(>>>)` is *reverse* composition, so `f >>> g = g . f` for the function space arrow.) Armed with just this operator, we can define *spatial transformation* of a `GUI`. We will define a generalized `transformGUI` operator that applies an (affine) spatial transform to a GUI to produce a new GUI:

```
transformGUI :: G.Transform ->
                    GUI b c -> GUI b c
```

Assuming that we have a basic understanding of spatial transformation for pictures, how shall we define spatial transformation of a `GUI`?

First, let's quickly review spatial transform for pictures. When we apply a spatial transform to a picture, it changes the size, position, or orientation of the picture. Consider translation of a picture by a displacement vector $(\Delta x, \Delta y)$. In general, this translation maps every (x, y) position in the original image to an (x', y') position in the new image by:

$$(x', y') \ = \ (x \ + \ \Delta x, y \ + \ \Delta y)$$

or, more generally, if *tf* represents the transformation, and `(%$)` is the *apply-transform* operator:

$$(x', y') \ = \ tf \ \texttt{\%\$} \ (x, y)$$

Note that `%$` is defined as part of the *Transformable* type class, so instance declarations may be given for any type that supports spatial transformation.

Since a `GUI`'s visual output is a signal of `Picture`, and our graphics library supports applying affine transforms to `Picture` values, we can transform a GUI's output by point-wise application of the transform to the picture output signal. But what about input?

If `g` is a `GUI`, point-wise transformation of `g`'s picture signal will map every

(x, y) position in g's coordinate system to (x', y'). In order to give an accurate input signal to g, transformGUI must map every (x', y') mouse position back to its corresponding (x, y) position in g. This suggests a general principle for transforming functions: *To transform a function (in time or in space), apply the transform point-wise to the output, and apply the inverse transform point-wise to the input.* This idea corresponds exactly to Pan's spatial "hyper-filters" [8], i.e., spatial transformations of $Image \to Image$ functions.

The implementation of transformGUI is then simply:

```
transformGUI tf g = proc (inp, b) ->
  (pic, c) <- g -< (inverse tf %$ inp, b)
  returnA -< (tf %$ pic, c)
```

This model for transforming GUIs is used in the implementation of the layout combinators to reposition their second argument GUI. The transform to apply to the second argument is determined dynamically by applying a bounds operation point-wise to the Picture signal produced by the first argument GUI.

4.4.2 Spatial Scalability

While our basic layout combinators only use basic horizontal and vertical translations, the transformGUI operator can apply *any* affine transform to a GUI. For example, here is a version of Paddleball that runs in a window 1/2 the size of the original:

```
-- uniform scaling transform (from Graphics library):
uscale :: Double -> Transform

minipb :: Double -> GUI () (Maybe ())
minipb vel =
    transformGUI (uscale 0.5) (pball vel)
```

When run, minipb displays a fully functional version of Paddleball shrunk down to postage stamp size. This type of zooming capability is obviously extremely useful for implementing vector or bitmap graphics editors, document previewers, etc. where zooming is a natural operation. But recent work in the Human/Computer Interaction (HCI) community has proposed continuous zooming can be a useful abstraction in its own right for many applications [17] [2]. Providing continuous zoom allows graphical interfaces to be designed so that users can "zoom out" for an overview of the data and "zoom in" for more detail. Pad [17] and Jazz [2] are two recent research projects that augment the widget set of a traditional imperative GUI toolkit with the abstraction of a continuously zoomable drawing surface.

The starting points for Pad and Jazz were the toolkits Tk and Swing, respectively. Because the Tk and Swing programming interfaces hide their connection with the graphics subsystem, Pad and Jazz are essentially new GUI toolkits, and require that existing applications be rewritten from scratch to take advantage of the zooming capabilities. In contrast, Fruit makes the connection to the interactive graphics subsystem seamless and explicit in the type of `GUI`. As `minipb` demonstrates, this explicit connection to interactive graphics allows us to incorporate novel ideas (such as continuous zooming) without a major restructuring of the library or completely rewriting applications.

4.5 Adding Multiple Views

Many GUI-based applications need multiple *views* on to the same underlying data set. For example, an icon editor might allow the user to open two views of the icon, one showing an editable, highly zoomed-in view where each pixel is a large square, and the other showing a preview of the icon at normal size. As the icon is edited in the zoomed view, the preview view should be updated.

We can really distinguish two kinds of views: *passive* and *active*. A *passive* view observes the underlying data set, but does not provide any means for interacting directly with the data set. The preview window of the icon editor just described is an example of such a passive view. In contrast, an *active* view is interactive: user actions in either view are reflected in other views and the underlying data set.

The requirement for multiple views is so common in user interfaces that the Model-View-Controller (MVC) design pattern has emerged as a way to structure imperative object-oriented programs to support multiple views when using imperative GUI toolkits [11].

For many practical applications (such as icon editors, illustration programs, etc.), the multiple views provided by the application are views of the same underlying (time-varying) picture, with different affine transformations applied to produce the view. For example, a zoomed-in view of an icon is a view of the same picture as a zoomed-out view; the pictures differ only by a scaling transformation. For simple cases such as this, multiple views may be added in Fruit to *any* GUI, without any pre-meditation on the part of the original GUI programmer.

Passive Views

A view can be though of as "a `GUI` with no mind of its own", as shown in figure 6. A view obtains its `Picture` signal from some external source and delivers its `GUIInput` signal to some external source. Concretely, a `view` is a `GUI` that takes a `Picture` signal as its auxiliary semantic signal, and uses this signal as its own `Picture` signal. Similarly, it delivers its `GUIInput` signal as its auxiliary output signal. This describes a simple crossover configuration that leads to the following definition:

```
view :: GUI G.Picture GUIInput
view = arr swap
```

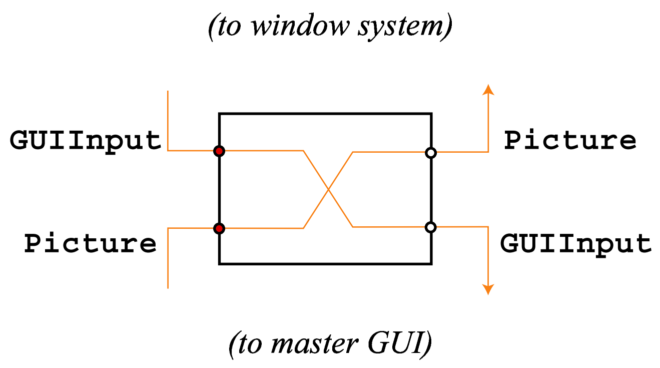


Fig. 6. Implementation of a view

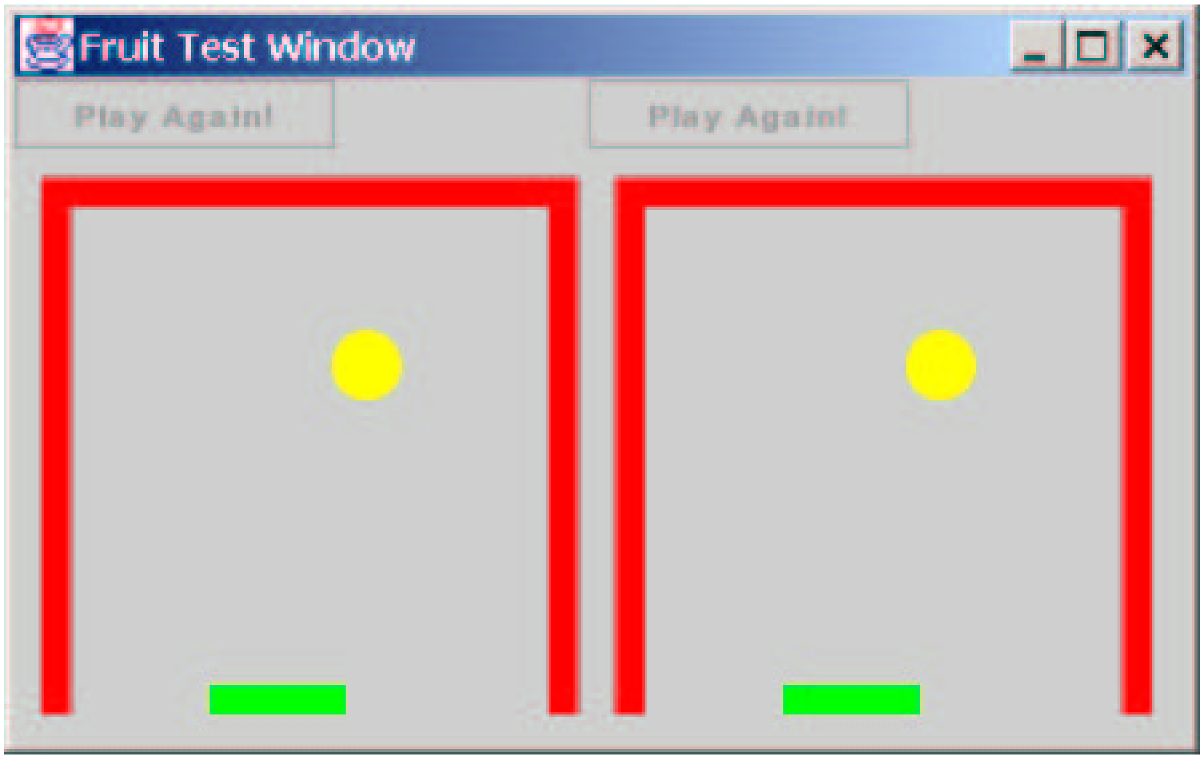


Fig. 7. Multiple Views

Given this definition, we can implement a version of Paddleball that has two views next to each other, as shown in figure 7:

```
pbview :: Double -> GUI () ()
pbview vel = proc (inpS,_) -> do
  rec (picS,(activeIn,_)) <-
        (view `besideGUI` view) -<
          (inpS,(gamePic,gamePic))
      (gamePic,_) <- (rpball1 vel) -<
          (activeIn,Nothing)
  returnA -< (picS,())
```

In this implementation, there are two views adjacent to each other. The view on the left is an *active* view, as its auxiliary input signal (`activeIn`) is fed as the input signal to the actual `rpball1` GUI. The view on the right is *passive*: Its picture signal is the same (`gamePic`) signal as the active view on the left, but its `GUIInput` signal is not connected to anything.

Multiple Active Views

Adding passive views to a GUI is certainly useful for many applications. But it is much more interesting, useful and symmetric to provide multiple *active* views, so that the user can interact with any view.

Recall from section 3.1 that we defined `GUIInput` to account for a *focus model*: At every point in time, the visual input signal to a GUI is either `(Nothing, Nothing)` (when the component does not have focus), or `(Just kbd,Just mouse)` when the GUI has mouse focus. Further, as described in section 4.4.1, the layout combinators perform *clipping* as well as transformation to ensure that only the GUI under the mouse receives (a transformed view of) the `GUIInput` signal. Recall, too, that our programming model includes a set of *event source combinators* that operate on signals of `Maybe` values.

Armed with this knowledge, we can now consider how to implement active views. In the implementation of `pbview`, each `view` is passed to the `besideGUI` layout combinator. The `besideGUI` combinator uses clipping and transformation to *demultiplex* its input signal into two signals, one for each child. At every point in time, one child's input signal is `(Just kbd, Just mouse)` while the other's is `(Nothing, Nothing)`). Regardless of which `GUI` has focus, the input signal will be transformed into the child's local coordinate system. Given this knowledge, it is a simple matter to define a `mergeGUIInput` combinator that will merge two disjoint `GUIInput` signals back in to a single signal by favoring the `Just` values and discarding the `Nothing` values. We define `mvpball` ("multi-view paddleball") as:

```
-- event merge, left-biased (from AFRP library):
emerge :: Maybe a -> Maybe a -> Maybe a
emerge mbeL mbeR = maybe mbeR id mbeL

mergeGUIInput :: GUIInput -> GUIInput ->
                    GUIInput
mergeGUIInput (mbkA,mbmA) (mbkB,mbmB) =
  (mbkA `emerge` mbkB,
   mbmA `emerge` mbmB)

mvpball :: Double -> GUI () ()
mvpball vel = proc (inpS,_) -> do
  rec (combinedPic,(leftIn,rightIn)) <-
```

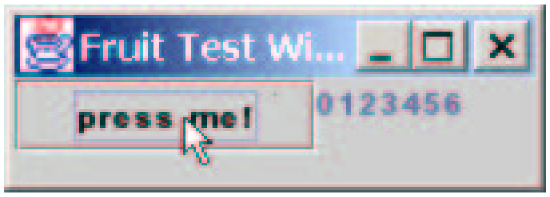


Fig. 8. Dynamic Labels Example

```
        (view `besideGUI` view) -<
            (inpS,(masterPic,masterPic))
      let mergedIn = mergeGUIInput leftIn rightIn
      (masterPic,_) <- (rpball1 vel) -<
            (mergedIn,())
  returnA -< (combinedPic,())
```

In this version of Paddleball, both views are treated symmetrically: The user can play or press the restart button in either view, and the action is reflected in both views.

Fruit is the only toolkit we are aware of that provides multiple active views "for free", without requiring any extra forethought or planning by the programmer of the original GUI.

4.6 Dynamic Interfaces

Thus far, all of the GUIs we have defined have been essentially *static* in the sense that the set of interface components visible on screen is fixed over the lifetime of the program. To support realistic user interfaces, it must be possible to dynamically add or remove components from the interface at runtime.

To demonstrate Fruit's support for dynamic interfaces, we implement an application (*dynLabels*) that dynamically adds new labels to an interface in response to a button press. The application is shown in figure 8. Every time the button is pressed, a new label is added to the interface (at the right edge of the current GUI) that display a count of how many times the button has been pressed in the program thus far. The screenshot shows the program after the button has been pressed six times.[8]

Since `GUIs` are first class values, we can maintain the current `GUI` that appears on-screen (using, say, `stepAccum` or some other accumulating signal transformer), and add to this `GUI` by using a layout combinator. Using such an accumulator in conjunction with `switch` allows us to switch from displaying one GUI to displaying the updated GUI. This pattern is so common and useful that we provide an `accumST` combinator to support it:

```
accumST :: (ST b c -> d -> ST b c)
```

[8] This example only adds components to the interface and does not remove them. Extending to allow removal as well as addition is straightforward.

```
            -> ST b c -> ST (b,Maybe d) c
```

The `accumST f st0 -< (iS,eS)` will behave initially as `(st0 -< iS)`. When an event occurs on `eS`, `accumST` passes the current signal transformer and the event occurrence value to `f` to obtain a new signal transformer. The `accumST` will then `switch` in to the signal transformer returned, which becomes the "current" signal transformer. The code for `dynLabels` is:

```
-- A set of counting labels:
countLabels :: GUI (Maybe ()) ()
countLabels =
  let addLabel :: GUI () () ->
                  Int -> GUI () ()
      addLabel labels n = labels
                  `besideGUI_` (mkLabel n)
  in proc (inpS,es) -> do
     lblNumE <- countE -< es
     (picS,_) <- accumST addLabel (mkLabel 0)
                        -< ((inpS,()),lblNumE)
     returnA -< (picS, ())

dynLabels :: GUI () ()
dynLabels = proc (inpS,_) -> do
  rec (picS,(pressES,_)) <-
        (fbutton `besideGUI` countLabels) -<
          (inpS,(btext "press me!", pressES))
  returnA -< (picS,())
```

5 Related Work

There have been many, many GUI toolkits implemented for Haskell, including Haggis [10], TkGofer [5], FranTk [23], and Fudgets [4]. These toolkits cover the spectrum from the mostly imperative (Haggis) to the mostly functional (Fudgets).

FranTk is similar to Fruit in the sense that it too uses the Fran reactive programming model (and its combinators) to specify the connections between user interface components. However, FranTk uses an imperative model for creating widgets, maintaining program state (with mutable variables or "MVars"), and wiring of cyclic connections (which occur in most GUIs, including the examples in this paper).

The closest relative to our work is Fudgets. Fudgets are implemented as *stream processors*, where each Fudget has *high level* and *low level* input and output streams. The high-level streams in Fudgets serve a role similar to the auxiliary semantic signals in our `GUI` type. The programming interface to Fudgets is very similar to that of Fruit, although Fudgets is based on discrete, asynchronous *streams*, whereas Fruit is based on continuous, synchronous *signals*.

Another difference is that Fruit is based on an abstract conceptual model of `GUIs`, whereas Fudgets is based on augmenting Haskell's stream-based I/O system with request and response types for the X Window system. Since we have not seen a formal definition of X windows, it is not clear to us what the denotational model of a Fudget is, beyond saying that it is a stream processor that emits and consumes X protocol requests. We believe that Fruit's model enables more precise reasoning about Fruit programs.

However, the Fruit programming interface is not without cost. Because any Fudget can emit an I/O request, a Fudget to perform file or network I/O can be added to a Fudget program just as easily as adding a graphical Fudget. In contrast, adding such features to Fruit would require explicit threading of the I/O actions through the Fruit program.

Finally, our work is similar to (and partially inspired by) Pike's pioneering work on Mux [20] [19], implemented in the language Newsqueak [21], a successor to Cardelli and Pike's language Squeak [3]. In Mux, every application is a *process* that communicates with the window system using CSP-style synchronous channels. The interface to each process has two input channels for keyboard and mouse input, and an output channel for producing pictures. The Mux window system itself is such a process that does simple *multiplexing* and *demultiplexing* to route messages between its input and output channels and those of its children. Thinking about composition of independent windows as multiplexing and demultiplexing is similar to our layout combinators.

The Fruit programming model owes much to its ancestors Fran and FRP. The most recent implementation of SOE FRP includes input types in the definition of `Behavior`, and an `Arrow` instance declaration for `Behavior`. The SOE FRP combinators are defined as ordinary Haskell functions, and the interface includes a primitive combinator, `runningIn`, that enables a signal to masquerade as a `Behavior`. In contrast, our interface defines every combinator as a signal transformer whose inputs are specified explicitly in its input type, and we use the arrow combinators for composition and application. Our programming interface thus gains modularity (as we can interpose functions such as spatial transformation on an `ST`'s input signal), and emphasizes the distinction between *signal transformers* and *signals*. However, we depend on the arrows syntactic sugar to make our model viable for writing real programs.

6 Current Status

We have implemented a working prototype of Fruit that is capable of running all of the examples presented here. The prototype includes a basic subset of the FRP combinators (implemented as synchronized stream processors). For visual display, Fruit uses a new vector graphics library, *Haven*, that we developed for this project. The interface to Haven is purely functional and implementation-independent, but our reference implementation uses the Java2D rendering engine. The low-level calls to Java2D are handled using another tool, *Elijah*, that provides a connection to the Java Native Interface via GreenCard, also implemented as a side project specifically for use in Fruit. We plan to release both Haven and Elijah as independent projects.

We refer to Fruit as a "prototype" only because it does not yet include a complete set of user interface components. Our focus thus far has been on figuring out the right abstract conceptual model and demonstrating that this model is viable and practical.

7 Conclusions and Future Work

In this paper, we presented a GUI toolkit for Haskell based on a formal model of graphical user interfaces. We showed how this model could be embedded in Haskell, and how the library could be used to construct a plausible example application. We also demonstrated some of the benefits of our approach, by showing how continuous spatial scaling and multiple views could be easily accomodated within the model.

Our results so far are very preliminary but encouraging. Building a library based on a formal model appears to be practical and provides some useful additional benefits, but we need to explore both of these areas in more depth.

In the short term, we plan to replace our low-level stream-based FRP implementation with a much more efficient data-driven implementation, add a complete and realistic set of widgets, and add support for efficient dynamic collections. In addition to this implementation work, we plan to further explore how we can incorporate modern user interface techniques into the model, as suggested in Section 4.4.1. And, of course, we plan to implement some real applications in Fruit, to further explore the benefits and limitations of our approach.

8 Acknowledgements

Special thanks are due to Henrik Nilsson for many patient, thoughtful discussions on the issues presented in this paper. We are also very grateful to members of the FRP research group at Yale (Paul Hudak, Liwen Huang, John Peterson, Walid Taha, Valery Trifonov and Zhanyong Wan) for their work on SOE FRP, many constructive discussions on semantics and implementation, and for reviewing earlier drafts of this paper. We would also like to thank to Magnus Carlsson, Christopher League, Ross Paterson and anonymous reviewers who read an earlier draft of this paper and provided very constructive feedback. Finally, thanks to John Hughes and Ross Paterson for their work on arrows and the arrows syntactic sugar. Without that framework, Fruit would be an interesting theoretical model with no viable implementation.

References

[1] Backus, J., *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*, Communications of the ACM **21** (1978), pp. 613–641.

[2] Bederson, B., J. Meyer and L. Good, *Jazz: An extensible zoomable user interface graphics toolkit in java*, in: *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST)*, ACM, 2000, pp. 171–180.

[3] Cardelli, L. and R. Pike, *Squeak: A language for communicating with mice*, in: B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, 1985, pp. 199–204.

[4] Carlsson, M. and T. Hallgren, "Fudgets - Purely Functional Processes with applications to Graphical User Interfaces," Ph.D. thesis, Chalmers University of Technology (1998).

[5] Claessen, K., T. Vullinghs and E. Meijer, *Structuring graphical paradigms in TkGofer*, in: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, 1997, pp. 251–262.
URL `citeseer.nj.nec.com/claessen97structuring.html`

[6] Elliott, C., *Functional implementations of continuous modelled animation*, in: *Proceedings of PLILP/ALP '98* (1998).

[7] Elliott, C., *An embedded modeling language approach to interactive 3D and multimedia animation*, IEEE Transactions on Software Engineering **25** (1999), pp. 291–308, special Section: Domain-Specific Languages (DSL).

[8] Elliott, C., *Functional images*, (to appear) Journal of Functional Programming (JFP) (2001).
URL `http://www.research.microsoft.com/~conal/papers/functional-images/`

[9] Elliott, C. and P. Hudak, *Functional reactive animation*, in: *International Conference on Functional Programming*, 1997, pp. 163–173.

[10] Finne, S. and S. P. Jones, *Composing the user interface with Haggis*, Lecture Notes in Computer Science **1129** (1996).
URL `http://citeseer.nj.nec.com/finne96composing.html`

[11] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison Wesley, Massachusetts, 1994.

[12] Henderson, P., *Functional programming, formal specification and rapid prototyping*, IEEE Transactions on Software Engineering **12** (1986), pp. 241–250.

[13] Hudak, P., *Modular domain specific languages and tools*, in: *Proceedings of Fifth International Conference on Software Reuse*, 1998, pp. 134–142.

[14] Hudak, P., "The Haskell School of Expression – Learning Functional Programming through Multimedia," Cambridge University Press, Cambridge, UK, 2000.

[15] Hughes, J., *Generalising monads to arrows*, Science of Computer Programming (2000), pp. 67–111.

[16] Paterson, R., *A new notation for arrows*, in: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2001)*, 2001.

[17] Perlin, K. and D. Fox, *Pad: An alternative approach to the computer interface*, Computer Graphics **27** (1993), pp. 57–72.

[18] Perlis, A., *Epigrams on programming*, ACM SIGPLAN Notices **17** (1982).
URL `http://www.cs.yale.edu/homes/perlis-alan/quotes.html`

[19] Pike, R., *Window systems should be transparent*, Computing Systems **1** (1988), pp. 279–296.
URL `http://citeseer.nj.nec.com/pike88window.html`

[20] Pike, R., *A concurrent window system*, Computing Systems **2** (1989), pp. 133–153.
URL `http://citeseer.nj.nec.com/pike89concurrent.html`

[21] Pike, R., *Newsqueak: A language for communicating with mice* (1989).

[22] Reynolds, J., "Theories of Programming Languages," Cambridge University Press, 1998.

[23] Sage, M., *Frantk: A declarative gui system for haskell*, in: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, 2000.
URL `http://www.haskell.org/FranTk/userman.pdf`

[24] Stoy, J. E., *Some mathematical aspects of functional programming*, in: J. Darlington, P. Henderson and D. A. Turner, editors, *Functional Programming and its Applications*, Cambridge University Press, 1982 pp. 217–252.

[25] Turner, D. A., *Functional programs as executable specifications*, Philosophical Transactions of the Royal Society of London **A312** (1984), pp. 363–388.

[26] Vullinghs, T., D. Tuinman and W. Schulte, *Lightweight GUIs for functional programming*, in: *PLILP*, 1995, pp. 341–356.
URL `citeseer.nj.nec.com/vullinghs95lightweight.html`

[27] Wan, Z. and P. Hudak, *Functional reactive programming from first principles*, in: *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

***

Named Instances for Haskell Type Classes

Wolfram Kahl [1]

Federal Armed Forces University Munich
Department of Computer Science, Institute for Software Technology
85577 Neubiberg, Germany

Jan Scheffczyk [2]

Federal Armed Forces University Munich
Werner-Heisenberg-Weg 102, App. 404
85579 Neubiberg, Germany

Abstract

Although the functional programming language Haskell has a powerful type class system, users frequently run into situations where they would like to be able to define or adapt instances of type classes only *after* the remainder of a component has been produced. However, Haskell's type class system essentially only allows late binding of type class constraints on free type variables, and not on uses of type class members at variable-free types.

In the current paper we propose a language extension that enhances the late binding capabilities of Haskell type classes, and provides more flexible means for type class instantiation. The latter is achieved via *named instances* that do not participate in automatic context reduction, but can only be used for late binding. By combining this capability with the automatic aspects of the Haskell type class system, we arrive at an essentially conservative extension that greatly improves flexibility of programming using type classes and opens up new structuring principles for Haskell library design.

We exemplify our extension through the sketch of some applications and show how our approach could be used to explain or subsume other language features as for example implicit parameters. We present a typed λ-calculus for our extension and provide a working prototype type checker on the basis of Mark Jones' "Typing Haskell in Haskell".

[1] Email: `kahl@ist.unibw-muenchen.de`
[2] Email: `jan.scheffczyk@gmx.net`

1 Introduction

One of the major success stories of Haskell is its type class system. Haskell's type classes allow a certain kind of ad-hoc polymorphism, and also enhance parameterisation of programs by allowing late binding of their members. In terms of implementations, this means that the dictionary that contains all the members of a certain instance of a class is supplied as a parameter in a late stage. However, this is not always possible, and so we find in the standard library pairs of functions like the following:

```
nub    :: (Eq a)                  => [a] -> [a]
nubBy  :: (a -> a -> Bool)        -> [a] -> [a]
sort   :: (Ord a)                 => [a] -> [a]
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

The motivation of this design is that currently Haskell allows only one instance of a given class for a given type, and provides quite a few standard instances, so it is not possible to have, for example,

- an instance `Eq Integer` that considers two integers as equal if they are equivalent modulo the 38th Mersenne prime,
- an instance `Ord String` that ignores case, or,
- given an expression type `Expr` with `instance Show Expr` producing plain text output, an additional instance `Show Expr` producing TEX output.

Therefore, case-insensitive sorting of strings (that one does not want to wrap in a `newtype` constructor[3]) has to resort to the function `sortBy`. In such simple cases, this may not be a serious problem. But frequently one designs a component around larger classes, only to notice later that the ad-hoc polymorphism provided by type classes does not easily allow ad-hoc instantiations, and the component has to be re-factored along the lines of the `-By` pattern.

In this paper we propose a language extension that allows such ad-hoc instantiations. The central idea is to allow *named instances* besides the anonymous instances of current standard Haskell, and to provide a mechanism for explicit *instance supply.* For example, TEX output of expressions might be provided for via the following instance of `Show` for `Expr`:

```
instance ExprShowTeX :: Show Expr where
  show (Power e1 e2) = '{' : show e1 ++ "}^{" ++ show e2 ++ "}"
  ...
```

[3] With the definitions
`newtype CIString = CI {unCI :: String}`
`instance Eq  CIString where compare = caseInsensitiveEqual`
`instance Ord CIString where compare = caseInsensitiveCompare`
one would have `sortBy caseInsensitiveCompare = map unCI . sort . map CI`.
The wrappers between the `newtype` isomorphisms and the application are in this case simple `map`s. In general, these wrappers can be harder to produce on the fly.

To preserve late binding possibilities, we have to suppress "context reduction" even in the presence of anonymous standard instances, so an application built on `Expr` might now have the following type:

```
calculator :: Show Expr => IO ()
```

For instantiation, we now have two possibilities: Since the type of `main` has an empty context, we can *force context reduction* via the following definition:

```
main = calculator
```

In this case, the anonymous `Show Expr` instance will be used for all occurrences of `show :: Show Expr ⇒ Expr → String`, and expressions are output (presumably) in their plain text format.

With our preliminary syntax for *explicit instance supply* we can force expressions to be output in their TEX format instead:

```
main = calculator # ExprShowTeX
```

In short, our extension unites the following features:

- the (dynamic) semantics of the current type class and instance system is preserved,
- some inferred types have larger contexts, and
- "later" binding of class members via explicit instance supply is possible.

One understanding of our extension is that we make some of the power of the target language of dictionary translations available to users, without burdening them significantly more with technical details than conventional Haskell classes do.

Another understanding also provides valuable guidance to our design. It is folklore that the Haskell type class system may be explained as an extremely restricted subset of ML module systems like the generative module system of SML [15,14], and Leroy's applicative module system [12] with manifest types [11], implemented in OCaml. In his proposal of *parameterised signatures* [6], Jones presents a nice overview of the differences between the SML and OCaml module systems, and also shows how parameterised signatures are closer to OCaml's system — the most relevant differences being the following:

- Parameterised signatures and the structures typed with parameterised signatures do not contain type components except as parameters.
- Parameterised signatures can be understood as easily producing polymorphic modules, which OCaml does not support.
- The ML module system issues of sharing and generativity are bypassed by relegating them to type checking.
- structures in the parameterised signature context are first-class values.

The system we propose in this paper is rather close to Jones' parameterised signatures, but does not go all the way to accept structures as first-class citizens. Instead, we preserve compatibility with the automatic aspects of the Haskell 98 type class system.

Let us briefly list the parallels between our understanding of Haskell type classes on the one hand, and the OCaml module system and parameterised signatures on the other hand.

- Simple Haskell 98 classes correspond to OCaml signatures, containing the class members as `value` entries, and the argument type variable as an abstract `type` entry. (Haskell 98 modules in addition feature non-abstract type entries.)

  Abstract type entries in OCaml module types have two kinds of application: The first is for hiding implementations and occurs in the construct `ConcreteModule : AbstractModuleType`, which produces a module with a hidden implementation for the abstract type entry; in Haskell, this has a parallel only in the conventional module system in the shape of abstract naming of algebraic datatypes in export lists.

  The second application is for opening up instantiation possibilities via the `with`-clause of the OCaml module language (which introduces "module constraints"), as in `AbstractModuleType with t = int`; this instantiation corresponds to the instantiation of class argument type variables via instance declarations.
- Haskell instance declarations without type class constraints[4] declare structures for the signature corresponding to the associated class, but where the type entry corresponding to the class parameter has been turned, via a `with`-clause, into a manifest type according to its instantiation by the `instance` declaration.
- Haskell classes with superclasses correspond to signature inclusion.
- Multiple abstract type entries in an OCaml module type correspond to the parameters of a multiparameter class in Haskell — these are no problem *per se*, only for automatic type inference of overloaded functions.
- Haskell instance declarations with type class constraints declare functors for which the argument types are induced by the constraints and the result type is the signature corresponding to the instantiated class, but where the type entries corresponding to the class parameters have been turned into manifest types according to their instantiation by the `instance` declaration.

But, since a class may be instantiated only once for a particular type, in Haskell 98 there can be only *one* module in scope for every instantiation of a declared module type. (The "overlapping instances" allowed in some implementations do not significantly improve the situation with respect to ad-hoc instantiation.)

[4] in the Haskell 98 report, and in large part of the literature, type class constraints are called "contexts" — to avoid confusion with other kinds of contexts we are going to use the term *constraint* throughout.

According to these analogies, there are several choices for nomenclature. Talking about dictionaries and dictionary types seems to us too much implementation oriented, and introduces additional difficulties when talking about types representing functions between dictionaries. In OCaml, the name *module type* is used both for signatures, that is, "dictionary types", and for functor types, that is "types representing functions between dictionaries", therefore we shall use the term *module type* with the same meaning. The third module type constructor of OCaml, the `with` clause introducing module type constraints, is reflected by instantiation of "signature parameters", i.e., class arguments.

OCaml also uses the term "module" both for structures and for functors, so we feel that this is most appropriate in this paper, too. In this way, everything that has a module type is a module. For non-functorial modules we may use the term *structure*; this corresponds to its usage in the context of parameterised signatures. In addition, we may subsume both instances and conventional Haskell modules under the term "structure" — conventional Haskell modules with appropriate elements can also be used for instance supply, see Sect. 3.

The named instances we introduce in this paper can then be considered as lightweight modules, and Haskell constraints indicate parameterisation that can be put to use by explicitly supplying appropriate parameter instances. Therefore, our extension eliminates the constraint that there can be at most one module for every module type; we also introduce explicit functor application and even higher-order functors. The most important aspects of the OCaml module system not covered by our extension include the following:

- Opaque types in interfaces, where not only the implementation of a type, but even its identity are hidden, can, as in parameterised signatures, be mimicked only via free type variables as class constraint arguments that occur only once, see [6, Sect. 2.1] for details.
- We did not consider nested modules at all. Very limited forms arise naturally, just by virtue of the fact that Haskell instances have to be defined inside traditional Haskell modules, but we do not consider module types specifying module entries in signatures.

Since we strive for maximal compatibility with Haskell (98), we also inherit a Haskell feature that cannot be found in ML: a Haskell function that carries a constraint in its type may be considered as an *anonymous one-element functor*. This feature poses quite a few difficulties for a coherent module-system-inspired extension of Haskell's type class system, but it also gives additional power to applications of such a system. Therefore, a coherent treatment of these *module type constraints* is the main challenge in any effort to extend the use of the Haskell class system towards what amounts to module system capabilities that cannot be found in Haskell's traditional module system.

Our current solution, which is driven by the desire to change the current language design of Haskell as little as possible, is therefore the main contribution of this paper. To better express our points, we use some new syntax; however, since throughout our extensions in this paper, we give proximity to Haskell 98 syntax higher priority than other considerations, we consider the concrete syntax of our extensions still open.

Through the desire to maximise compatibility and harmonious interaction with the Haskell 98 class system, we end up with a system that includes some rather complicated technical details, although its basic ideas are in fact simple. Therefore, we give ample space to introducing the features of our proposed extension and the motivations behind our design decisions in an informal manner. We start with introducing the simplest aspect, *named instances*, in Sections 2 and 3. The next level are functors, discussed in Sect. 4. Module type constraints as type qualifications are discussed in some detail in Sect. 5, and it turns out that rather fine-grained distinctions are necessary. Since our extensions give users more control over the satisfaction of constraints than the one-instance-per-type approach of Haskell 98, this means that we have to do *less* constraint reduction (Sect. 6). We then discuss aspects of how subclass relationships might translate into "module subtyping" in Sect. 7.

An interesting side-effect of our view of constraints is that the implicit parameters of Lewis *et al.* [13] are partly subsumed as "anonymous one-element functors with one-element structures as arguments"; we show this in Sect. 8.

In Sect. 9, we sketch a typed λ-calculus featuring module type constraints, and in Sect. 10 we briefly describe a prototype implementation of the type-inference aspects, based on Jones' "Typing Haskell in Haskell" [7].

2 Named Instances

As seen in the introduction, the first step towards enabling really late binding of class members even at types for which there are predefined instances is the ability to define and *name* non-standard instances for later reuse. Therefore, in our extended Haskell, we may provide names for instances, and, following the module analogy, understand these *named instance declarations* as declaring structures of the module type defined by the associated class declaration. For syntactic convenience, we let these names share the name space of Haskell module names, thus named instances must not use names that are also used as conventional module names. (This decision also opens up additional flexibility that will be exemplified in the next section.)

Let us now have a closer look at the named instance declaration from the introduction.

```
instance ExprShowTeX :: Show Expr where
  show (Power e1 e2) = '{' : show e1 ++ "}^{" ++ show e2 ++ "}"
  ...
```

It brings the module name "`ExprShowTeX`" into scope, and binds it to the structure determined by the body of the instance declaration (together with default definitions from the class — for sake of simplicity we shall completely ignore the question of such default definitions in the sequel).

Since "`ExprShowTeX`" is a module name, we may use the qualified identifier `ExprShowTeX.show`, which has the type `Expr` → `String`. The module name `ExprShowTeX` itself is considered to have the module type `Show Expr`, which we write "`ExprShowTeX :: Show Expr`".

Note that such a *named instance declaration* does *not* introduce a "real" instance for the class `Show` that would be available for automatic insertion wherever `show` is used at the type `Expr` → `String`.

The class member `show` still has type `(Show a)` ⇒ `a` → `String`, containing the *constraint* "`(Show a)`". This means it is parameterised over possible instances for `Show`, or, more precisely, over structures of module type `Show a`. In Haskell 98, there is no way to explicitly instantiate such parameters: it all happens implicitly, via anonymous instances and constraint reduction.

With named instances, we can have more than one instance in scope for the same type, and we introduce a possibility to explicitly supply instances as parameters. We use the infix operator "`#`" as application to instance parameters, or, as we shall say from now on, to module parameters. Therefore, "`#`" is the elimination form for types constructed with "⇒", in the same way as standard application is the elimination form for types constructed with "→". So we may write "`show # ExprShowTeX`", and this has type `Expr` → `String`. Essentially, this means that "`#`" corresponds to dictionary application. However, in dictionary translations it is necessary to make all references to dictionaries explicit, while in our system, much of the implicit character of the Haskell class system is preserved.

Let us now turn to another very simple example which hints at possible uses of named instances for structuring algebraic libraries, and which, at the same time, also illustrates some further effects. Consider the following class declaration for monoids:

```
class Monoid m where
  unit :: m
  comp :: m -> m -> m
```

Imagine further a complex body of utilities built around this class, including, as a simple example:

```
complist :: Monoid m => [m] -> m
complist = foldl comp unit
```

Desiring to use all this in integers, the user of this class in standard Haskell now faces the problem of indicating which of the two well-known monoids on integers to use for defining *the* instance of `Monoid` for `Integer`. For the other monoid, the standard escape route is `newtype`, which allows to define a different instance for the same class on a *different* type which, however, has the *same* implementation. Jones perceives the use of `newtype` declarations to "clutter up programs" [8]; this stems from the fact that they lead to artificial distinctions on essentially the *same* type.

With named instances, this problem is easily solved. We just define:

```
instance AddMonInt  :: Monoid Integer where
  unit = 0
  comp = (+)
instance MultMonInt :: Monoid Integer where
  unit = 1
  comp = (*)
```

A complex application wishing to mix both views on integers just needs to supply the respective instances to different library function invocations:

```
foo :: [[Integer]] -> Integer
foo = (complist # AddMonInt) . map (complist # MultMonInt)
```

Compare this with the Haskell 98 type inferred for the function

```
bar i x = complist $ map complist $
          replicate i $ replicate i $ (x :: Integer)
```

which is `(Monoid Integer)` $\Rightarrow$ `Int` $\rightarrow$ `Integer` $\rightarrow$ `Integer` with only *one* constraint (the application `bar # AddMonInt` will use addition on both levels of lists). Since both instances provided above are named, we assume that there is no anonymous instance of type `Monoid Integer` in scope, so the constraint would be locally unresolvable.

Even apart from the monomorphism restriction, such unresolvable constraints involving non-variable types are illegal in standard Haskell. In our setting however, admitting such constraints makes sense. The delayed constraint may be satisfied at a later stage, for example via the explicit module parameter supply `bar # MultMonInt`.

In contrast with anonymous instances, automatic export does *not* make sense for named instances. Since they share the module name space, they may be exported with `module` entries in export lists. They will be imported in the same way as other module entries — then the instance name is available as an argument to "`#`"-application, and members may be accessed as qualified identifiers. Indeed, there is no reason to forbid using conventional modules as arguments for "`#`"-application, provided they contain appropriately named members — we show an example for this in the next section.

3 Named Instances for Multi-Parameter Classes

Multi-parameter classes are recognised as extremely useful, but it is not yet clear how the design choices involving context reduction and type inference should be resolved [16]. With named instances, it is possible to use multi-parameter classes without ever defining anonymous instances for them, so problems of context reduction and type inference can essentially be avoided.

In the following, we therefore allow named instances and forbid anonymous instances for multi-parameter classes, thus reaping a significant portion of the benefits of multi-parameter classes without incurring the usual costs.

Let the following module defining a collection class be given:

```
module Coll where
class Collection c e where
  empty  :: c e
  insert :: e -> c e -> c e
  fold   :: (e -> r -> r) -> r -> c e -> r
```

Let us furthermore assume a stand-alone module that implements sets using some tricky balanced tree implementation and demands the constraint `Ord` on element types:

```
module SetColl(Set(),empty,insert,fold)where

data Set a = EmptySet | TrickyBalancedTree ... a ...

empty  :: Ord a => Set a
empty  = EmptySet

insert :: Ord a => a -> Set a -> Set a
insert = ...

fold   :: Ord a => (a -> r -> r) -> r -> Set a -> r
fold   = ...
```

As in this example, libraries built for our extended language would tend to directly implement appropriate class interfaces. This improves much upon the current practice of unqualified imports that foster the unfortunate tradition of type-indicating names like `foldSet`. Any module that imports `Coll` unqualified will have to do `qualified` import of modules like `SetColl`, because otherwise e.g. `empty` would refer both to the class member `Coll.empty` and to `SetColl.empty`.

Accordingly, consider the following module header:

```
module Main where
import Coll
import qualified SetColl
```

Inside this module, it now makes sense to consider `SetColl` as an instance of `Collection` with the type `forall a . Ord a` $\Rightarrow$ `Collection SetColl.Set a`, which is equivalent to the original (anonymous) type of the module `SetColl`. This means that this is a *fully polymorphic instance* that can be used at arbitrary (ordered) element types. The module `SetColl`, considered as a functor in this way, hides the *implementation* of the `Set` datatype constructor, but it does not hide its *identity*, nor its dependence on types provided by the functor argument — these are the effects that lead to the bypassing of the sharing issue, as discussed by Jones [6].

An application that has been designed to be independent of the implementation of collections might still use collections of different element types, so it can be defined to depend on such a polymorphic instance:

```
app_main :: (forall a . Ord a => Collection c a) => IO ()
```

Given all the above, it is now possible to instantiate this application inside module `Main` in the following way, providing an explicit module type to the conventional Haskell module `SetColl`:

```
import Application(app_main)
main = app_main #
       (SetColl :: forall a . Ord a => Collection SetColl.Set a)
```

We conjecture that for most uses of multi-parameter classes, the resulting need to specify the instance to be used will be only a small burden to the user. Perhaps it may even be a liberation not to have to think about possible overlaps, and being able to specify the intended instance, instead of having to set up a puzzle for the compiler, and hoping that the compiler will solve it correctly, and in finite time.

From the examples in this and the preceding section it should be clear that named instances together with explicit module parameter supply are a natural remedy to the commonly perceived weakness of type classes in Haskell 98 which are, citing [5], only "well suited to overloading, with a single natural implementation for each instance of a particular overloaded operator". For many applications, already this simple extension would be extremely valuable. However, once the basic correspondence between type class concepts and OCaml module concepts is established, further extensions are natural consequences of the introduction of named instances, and are discussed in the next few sections.

4 Instance Functors

In module systems, a functor is a function taking modules as arguments, and having other modules as results. In the dictionary translation of classes, instances with constraints are translated into functions between dictionaries. Since dictionaries may directly be viewed as modules, we immediately see that such instance declarations give rise to functors.

The constraints then express the types of the arguments, and the target class of the instance is the result type. On the whole, we get a *functor type*, for which we conveniently use syntax already present in Haskell. In contrast, module types that are not functors are called *atomic*.

Let us consider a facility to show lists not in the standard way with brackets and commas, but with every element on a separate line:

```
instance ShowListUL :: Show a => Show [a] where
  show xs = unlines (map show xs)
```

Now `ShowListUL` has the *functor type* `Show a` ⇒ `Show [a]`. Functor application does of course expect appropriately typed arguments, so we cannot provide `ShowListUL` as an argument to `show`; instead we have to apply the functor to some argument first, and then provide the result to `show`. For this *functor application*, we also use the infix operator "`#`": The functor application "`ShowListUL # ExprShowTeX`" has type `Show [Expr]`, and we may write

```
show # (ShowListUL # ExprShowTeX)
```

which has type `[Expr]` → `String`. However, we may not always be working with a fixed, predefined functor argument, so we need *module variables*:

```
showListUL # i = show # (ShowListUL # i)
```

The following type is then automatically inferred:

```
showListUL :: Show a => [a] -> String
```

The case where a functor takes several module arguments at first poses a problem, since in Haskell 98, the types of the following two instances (apart from being named) would be considered as completely equivalent[5]:

```
instance D1 :: (Show a, Show b) => Show (a,b) where ...
instance D2 :: (Show b, Show a) => Show (a,b) where ...
```

If we accepted this, then it would not be possible to give a reasonable semantics to applications like `D1 # ExprShowTeX`. However, since these constraint lists arise in a place where they are written by the programmer, we regard the order of these lists as intentional, so the above is equivalent to specifying curried functor types (we accept the above syntax only for backward compatibility):

```
instance D1 :: Show a => Show b => Show (a,b) where ...
instance D2 :: Show b => Show a => Show (a,b) where ...
```

5 Module Type Constraints

We now look in more detail into the problems generated by our view that constraints in the types of functions should be considered not just as class constraints, but as general *module type constraints*. The real problems come from our desire to let these module type constraints be satisfied not only by "module supply" via "`#`", but also by anonymous "default instances" via the conventional class system of Haskell. In effect, we design our extensions in such

[5] In 4.1.4 of the Haskell 98 report, the type generalisation preorder for qualified types is defined and implies that types differing only in constraints that are equal up to permutation have to be considered as equivalent. Another hint in that direction is the mention of "most general instance context" in 4.3.2.

a way that we do not break existing Haskell programs. On the contrary, we enhance the reusability of existing traditional library modules by admitting different methods of module supply for constraints, and by exposing more constraints to user-defined module supply.

5.1 Ordered and Unordered Constraints

Consider the following type signatures and definitions as given:

```
f  :: Eq a => a -> c -> (a,c)
bs :: Eq b => [b]
g x = f x bs
h x = (\ ff -> ff x bs) f
```

The Haskell 98 interpreter Hugs98 and the compiler GHC-4.08 derive the typings (up to α-conversion):

```
g :: (Eq a, Eq b) => a -> (a,[b])
h :: (Eq b, Eq a) => a -> (a,[b])
```

In contrast, the compiler nhc98 derives oppositely ordered constraints — for other similar examples pairs where Hugs and GHC derive different orders, nhc98 derives the same orders.

Of course, `g` and `h` are the same function, and since in Haskell 98 the two typings are considered as equivalent, this is no problem — if explicit type signatures are added, all three systems accept any ordering of constraints.

Since every reasonable typing discipline should obey the subject reduction property — β-reduction can only lead to a more general type — this example shows that the structure of an expression induces *no natural ordering* for inferred constraints.

A canonical order might be achieved by orientation at the structure of the inferred type, for example preferring `(Eq a, Eq b) => a -> (a,[b])` because of the order of occurrence of the type variables in `a -> (a,[b])`, but this breaks down for constraints on types that do not occur in the raw typing. Therefore, unordered constraints are a natural result of type inference for Haskell 98 expressions, even in their embedding into our extension.

More precisely, it is function application that forces joining of the ordered constraints in the types of the two constituents of the application into the unordered constraint of the type of the whole application. For reasons of compatibility with the Haskell 98 view of constraints, it does not make sense to have functor types in unordered constraints. Therefore, unordered constraints may only contain atomic module types, while ordered constraints may contain even higher-order functors. For a functor type μ in the ordered constraints of the two constituents of an application we have to apply constraint reduction (see the next section), which is of course *only* possible, if the class environment contains an anonymous default functor of a type μ' such that μ and μ' both map to the same Haskell 98 functor type.

With the unordered polymorphic constraints of the example above, there is no way to allow the user to direct their satisfaction separately, since any structure `M :: Eq t` for any type `t` could satisfy both constraints.

In our extension, however, separate satisfaction of constraints can be enabled even in more general cases by providing explicit module arguments in the same way as functor arguments — this is a result of considering a function with constraints in its type as an implicit functor with an anonymous one-element result signature. So we are able to make module parameterisation explicit, and in analogy to the definition of `showListUL` in Sect. 4, we now may define:

```
k # i # j = (f # i) (bs # j)
```

For this, only the type `Eq a` ⇒ `Eq b` ⇒ `a` → `b` can be inferred, so here we have an *ordered list of constraints*.

Since the two effects may occur together, we have to partition the constraint component of types into what we are, from now on, going to call "ordered constraints" and "unordered constraints". As an example for the coexistence of ordered and unordered constraints, consider the following:

```
k' # i # j  x y = let k0 z = (f # i) z (bs # j) in
                  if x <= y && k0 x <= k0 y then k0 x else k0 y
```

Here we have `Eq a` and `Eq b` in the ordered constraints, since they are associated with the module variables `i` and `j`, and in addition `Ord a` and `Ord b` in the unordered constraints, motivated by the two occurrences of `<=`. Therefore, the following typing is inferred:

```
k' :: Eq a => Eq b => {Ord a, Ord b} => a -> a -> b
```

Note that from now on we shall write unordered constraints with braces `{}`, not with parentheses `()`. In our present design these braces, representing the unordered (Haskell 98) constraint, clearly indicate to which part of the constraint a module type belongs to. Therefore, unlike in Haskell 98, the following types should then be considered as different:

```
ff  ::  Eq a  => [a] -> Bool
ff' :: {Eq a} => [a] -> Bool
```

However, in our investigations this difference would have noticeable consequences together with a certain set of decisions concerning module type subtyping (see Sect. 7) that seems to be a useful compromise and is implemented in the prototype, but will not be discussed any further in the present paper.

Without subtypes, there is a noticeable difference only inside type class definitions:

```
class Foo a where
  foo1 ::  Foo b  => a -> b -> Bool
  foo2 :: {Foo b} => a -> b -> Bool
```

The members of this class have different types:

```
foo1 :: Foo b => {Foo a} => a -> b -> Bool
foo2 :: {Foo a, Foo b}   => a -> b -> Bool
```

As a result, the late binding capabilities of `foo2` are much more restricted than those of `foo1`.

5.2 Module Supply

It is natural to allow satisfaction of module types in unordered constraints where no ambiguity arises, such as in the following (assuming the named instance `FooChar :: Foo Char`):

```
q i = foo2 'c' (i :: Int) # FooChar
```

This has the type `{Foo Int}` ⇒ `Int` → `Bool`. We may allow this because the type of `FooChar` is not an instance of `Foo Int`.

This "automatic selection" of parameter position by actual parameters may be generalised to the ordered constraints, allowing out-of-order #-application to modules. This means that the first type-compatible argument position in the ordered constraints is used, or the *only* type-compatible element of the unordered constraints.

```
f :: Eq Int => Eq Char => Eq [Int] => Eq [Char] =>
     ([Int],[Int]) -> ([Char],[Char]) -> ([Int],[Char])

MyEqLC :: Eq [Char]

-- f # MyEqLC :: Eq Int => Eq Char => Eq [Int] =>
--             ([Int],[Int]) -> ([Char],[Char]) -> ([Int],[Char])
```

The case of several type-compatible elements in the unordered constraints has to be rejected as an unresolvable ambiguity.

This convention — which is in fact nothing more than syntactic sugar — avoids having to use module variables, and thus reduces the syntactic heaviness of supplying parameters to specific parameter positions. Although it may seem somewhat ad-hoc, we consider this usability aspect a strong argument in favour of including this feature — it is also relatively cheap to implement in the type checker and in the formal design, where the details are defined (Sect. 9). For functors, however, out-of-order application is not an option, since it makes module type inference undecidable.

5.3 Typing Functors

Finally, we have to decide how far we take module polymorphism. Consider the following definition:

```
f # k # j x y = ((==) # (k # j)) x y
```

The module variable `k` obviously must have a functor type, but we have no information about its argument type. Thus the inferred type for `f` could be `f :: (? ⇒ Eq a) ⇒ ? ⇒ a → a → Bool` — where “?” might denote a *module type variable*. Since we perceive this as an over-generalisation of very limited use, we tend to exclude module type variables from the syntax, and not to allow definitions that imply constraints with module types in which module type variables occur. The definition above may then be legalised by adding an appropriate type signature.

Functors are polymorphic by nature, but Haskell's first-order type inference makes it impossible to use arguments at polymorphic types. Extensions in Hugs and in GHC include the feature of *rank-2-types* which we adopt for our constraints. We have shown an application of a function with a second-order constraint in Sect. 3; here we show how such a function might be defined:

```
int_component    :: Collection c Integer => IO ()
string_component :: Collection c String  => IO ()

app_main :: (forall a . Ord a => Collection c a) => IO ()
app_main # coll = do int_component    # coll
                     string_component # coll
```

It would of course be more comfortable without explicit instance variables:

```
app_main :: (forall a . Ord a => Collection c a) => IO ()
app_main = do int_component
              string_component
```

This could be made possible by an extension of *forcing constraint reduction*, which is discussed at the end of the next section.

There is no intrinsic restriction of this kind of polymorphism to functors; the above example could be rewritten for `Collections` defined without constraints, for example via lists.

It would of course be most elegant if we would not have to think about such constraints while designing the application, so we would like to have:

```
app_main :: (forall a . Collection c a) => IO ()
```

However, this cannot be directly applied to `SetColl` in our current system; one would have to supply (transparently through type quantification)

```
instance PolyOrd :: (forall a . Ord a)
main = app_main # (SetColl # PolyOrd)
```

The polymorphic ordering instance might be defined via Hinze's and Peyton Jones' *derivable type classes* [3] or Hinze's fully polymorphic *atomic* instances from [2,1].

Short of requiring such *fully* polymorphic ordering instances, one might also consider polymorphism restricted to ground types generated by a limited set of type constructors, which would allow more compile-time control:

```
app_main :: (forall a BuiltFrom {Int,Maybe,(,)} . Collection c a) ⇒ IO ()
```

Obviously, this area deserves further investigation.

6 Constraint Reduction

As we have seen in the previous sections, we must have a closer look at *constraint reduction* and *module type instantiation*. We say that a module type is instantiated if its type variable(s) are made less polymorphic. In contrast, a constraint is reduced if we delete one of its module types. In addition, moving a module type from the unordered to the ordered constraint is also a kind of constraint reduction.

The concept of how constraints are reduced and module types are instantiated directly influences the use and flexibility of module type constraints. The tradeoff to be balanced is between compatibility with Haskell 98 and the desire for maximum flexibility. An eager approach to constraint reduction would enforce ultimate compatibility, but incur a severe loss of flexibility, whereas a “fairly lazy” approach is as flexible as possible, while only compromising compatibility in tolerable ways. *Fully lazy* constraint reduction is not feasible, since it would produce ambiguous constraints and inhibit polymorphic recursion [16].

Monomorphic Module Types

As we have seen in Sect. 2, monomorphic module types such as `Monoid Integer` play an essential rôle when using named instances. Therefore, we will not delete a monomorphic module type from constraints, since the user may decide at a “later” stage which structure to choose.

In order to accept Haskell 98 programs, we have to allow constraint reduction through explicit type annotations; this will then eliminate monomorphic module types for which a satisfaction is entailed from anonymous class instances. Thus, adding for example the type signature

```
bar :: Int -> Integer -> Integer
```

is only legal when a *default* instance declaration for `Monoid Integer` is in scope, and then this type signature forces `bar` to have precisely this type. This perfectly reflects the behaviour of Haskell 98.

Note that every compilable Haskell 98 program has at least one explicit type annotation[6] `main :: IO(a)`, which is forced by the Haskell compiler if it is not explicitly given by the programmer — this would be the ultimate point of forcing away delayed constraints by supplying the anonymous default instances from the Haskell 98 class system.

[6] Since the type variable `a` may be instantiated in the process, this is a somewhat special case.

Ambiguous Constraints

As seen above, there may be problems when two module types M_1 and M_2 in an *unordered* constraint are ambiguous. This is the case if there exist substitutions θ_1 and θ_2 such that $\theta_1 M_1 = \theta_2 M_2$. If, according to the definition of Peyton Jones *et al.* [16], M_1 *entails* M_2, denoted by $M_1 \Vdash M_2$, then there is a default functor Φ justifying this entailment. So we can delete the constraint M_2 and supply Φ `#` M_1 into the original parameter position of M_2. This is the *only* case where automatic constraint reduction is left intact.

Of course, explicit type annotations with ordered constraints may be used to prevent this automatic constraint reduction:

```
f :: Eq a => Eq [a] => [a] -> [a] -> Bool
f xs ys = (head xs) == (head ys) || (tail xs) == (tail ys)
```

However, we cannot allow type annotations that change ordered constraints into unordered constraints, because ordered constraints *only* arise from the use of module variables and from explicit type annotations.

Note that the definition of entailment of Peyton Jones *et al.* [16] also includes superclass declarations which we regard as projection functors. Since the problem of ambiguous constraints only involves module types related to the *same* class, we do not need to include superclass projection functors in the definition of entailment used in our context. Note further that with our notion of automatic constraint reduction, "lonely" constraints, as for example `{Eq [a]}`, are always treated as irreducible.

Forcing Constraint Reduction

As we have seen in the preceding sections, we can *force* constraint reduction via explicit type annotations. This will be *full* Haskell 98 constraint reduction without exceptions, and it can force away only from elements of the unordered constraints. Of course it needs to have the corresponding `instance` (and `class`) declarations in scope.

7 Instance Subtyping and Joint Instance Declarations

According to the above, a named variant of the default `Ord` instance for `Maybe`, with the header `instance OM :: Ord a => Ord (Maybe a)` would have the type `Ord a` $\Rightarrow$ `Ord (Maybe a)`.

Now consider the implications of the fact that `Ord` is defined as a *subclass* of `Eq`. With the usual understanding of subclass relationships, the module type `Ord (Maybe a)` should also be a subtype of the module type `Eq (Maybe a)`. This implies that wherever an instance of type `Eq (Maybe a)` can be used, an instance of type `Ord (Maybe a)` should also be acceptable.

At first sight this seems to be no problem: class declarations for classes with superclasses can be seen as containing *implicit* functor definitions for projection functors from the signature of the subclass to the signatures of the superclasses. A radical view on this subtyping relation implies that the current definition of the class `Ord` is equivalent to the following expanded version:

```
class Ord a where
  (==), (/=), (<=), (<), (>=), (>) :: a -> a -> Bool
  compare :: a -> a -> Ordering
```

This would then also support *joint instance declarations*, which have recently been proposed in a mailing list discussion. They address mainly the following scenario which is relevant to adaptability of library classes:

Assume a library class declaration

```
class C a where
  m1 :: a
  m2 :: a
```

for which a user defined the following instance:

```
instance C Int where
  m1 = 1
  m2 = 2
```

Now the library undergoes some redesign, and it is decided that splitting the class has advantages, so now we have:

```
class         C1 a where m1 :: a
class C1 a => C  a where m2 :: a
```

But in Haskell 98, this breaks the user's instance definition! Since `C1 Int` is a supertype of `C Int`, the proponents of joint instance definitions propose that the user's instance definition should be considered as legal, because it really defines a structure of type `C Int`, from which a structure of type `C1 Int` may be extracted by the corresponding superclass projection functor.

In contrast, the "conventional" instance definition

```
instance C Int where m2 = 2
```

really defines an anonymous functor of type `C1 Int` ⇒ `C Int`, which is used to *construct* a structure of type `C Int` from a previously available structure of type `C1 Int`.

In the same way, our named instance from the beginning of this section should have the type `MO :: Eq (Maybe a)` ⇒ `Ord a` ⇒ `Ord (Maybe a)`. We tend to put `Eq` first since in common understanding no instance of a subclass can be defined before there are instances of all its superclasses. Now this is quite counter-intuitive, but making this type explicit in the instance declaration would fail to indicate that the `Eq` structure of the first argument will end up as the `Eq` component of the result. Furthermore, in the presence of more than one instance at some type we run into the same multiple-inheritance problems as in C++ — just imagine additional class declarations as the following:

```
class Eq a => R a
class (R a, Ord a) => S a
```

Now we could define two joint instances for `R` and `Ord`, equipped with different equalities, and a non-joint implicit functor for `S`:

```
instance R1 :: R   Int where (==) = eq1  -- R1 :: R Int
instance O2 :: Ord Int where (==) = eq2  -- E2 :: Eq int
instance S1 :: S Int          -- S1 :: R Int => Ord Int => S Int
```

As a result, `S1` somehow contains the two different bindings `eq1` and `eq2` for the class member `(==)`, and no sensible automatically defined projection functor is available for using `(S1 # R1 # O2)` of type `S Int` at the subtype `Eq Int`, as for example in the expression "`(==) # (S1 # R1 # O2)`".

The whole topic of subtyping of module types therefore has to be treated with great care.

From this discussion it should be obvious that joint instance definitions are really independent from named instances and module type constraints, but we claim that our system is a good way to explain the issues behind joint instance definitions. This is especially so since default definitions for members in class definitions may be considered as inducing a set of functors that turn different subtypes of the defined class type into the complete class type. How these default definitions are to interact with joint instance definitions is probably much easier to analyse using our functor concept.

This problem is closely related to the problem that the identity of atomic module types (i.e., signatures) in Haskell is defined by *class name*, and not by the *contained signature*. From this point of view, implicit parameters are more honest since they use *anonymous* module types as arguments, and an accumulation of implicit parameter constraints may even be considered as a multiple-member module type. However, there is no way to supply a single multiple-member structure as an argument that instantiates all these parameters. Finding a better way to use "anonymous classes" would therefore be a useful continuation of our present work.

8 Implicit Parameters

The Haskell interpreter Hugs [4] provides an experimental extension called "implicit parameters" [13], introducing dynamic bindings. We argue that implicit parameters cover a subset of the possibilities of module type constraints, but are easier to use at least in simple cases.

A translation of the "File IO" example given by Lewis *et al.* [13] into Haskell with module type constraints will naturally use zero-parameter type classes and local instance declarations not discussed in this paper. For good measure, we throw in a joint instance definition, used at a supertype:

```
class StdIn  where stdIn  :: IO Handle
class StdOut where stdOut :: IO Handle
class {StdIn, StdOut} => StdIO

instance StdStdIO where
  stdIn  = stdin
  stdOut = stdout

getLine :: {StdIn } =>           IO String
putStr  :: {StdOut} => String -> IO ()

session :: StdIn => StdOut => IO ()
session = do putStr "What is your name?\n"
             s <- getLine
             putStr ("Hello, " ++ s ++ "!\n")

main = do h <- openFile "foo"
          instance H :: MkStdOut where stdOut = h
          session # StdStdIO # H
```

We feel that, on the one hand, the approach of module type constraints gives much more flexibility and syntactically fits better into Haskell 98. On the other hand, implicit parameters are easier to handle because the programmer can use *functions* to modify them. Therefore, both approaches might "peacefully" coexist in a Haskell environment.

9 A Typed λ-Calculus with Named Instances

In this section we formalise named instances and module type constraints by presenting a type system and a type inference algorithm for a small language corresponding to the relevant extension of a subset of Haskell 98, covering only the central aspects of our extension. We present this as a fairly standard typed λ-calculus with `let`-polymorphism.

9.1 Notation and Utility Functions

In this section we will introduce syntactical notations (see Fig. 1) and define some basic functions. We follow the common notations of [13,9].

Distinguishing λ-bound variables (x) from `let`-bound variables (p) is not really necessary, but makes the reading of formulae easier. Module variables have their own name space (which they share with Haskell 98 modules). Types are constructed from type variables via the function type constructor $\to$ and other type constructors χ.

Module types are simpler than types in that there is only the functor type constructor $:\Rightarrow$ (associating to the right) for producing non-atomic module types. As noted in Sect. 5, a constraint consists of an ordered part O and an unordered part U. The ordered part is a list of module types (and we use Haskell list syntax), while the unordered part is a *set* of *atomic* module types.

Constraints are used to construct *qualified types* σ which are then of the shape $O \rhd U \Rightarrow \tau$. For qualified types with empty constraints we just write τ.

A Haskell 98 context Γ is a finite partial function associating variables from Var with either a qualified type (QType) or a type scheme ($\mathsf{TScheme}$), where Var contains λ-bound variables and `let`-variables. An additional context Δ is provided, associating module names and module variables with module types.

λ-variables	x
`let` variables	p
Terms or expressions	$e, f, t \ ::= \ x \mid p \mid \lambda x.t \mid e\ f \mid \texttt{let}\ p = e\ \texttt{in}\ t$
Module variables MVar	i, j
Module expressions	$m \ ::= \ i \mid m_1 \mathbin{\#} m_2$
Type variables	α
Type constructors	χ
Types	$\tau \ ::= \ \alpha \mid \tau \rightarrow \tau \mid \chi\, \tau_1 \ldots \tau_n$
Class predicate symbol	κ
Module type variable	ν
Atomic module type	$\kappa\langle\tau\rangle$
Module types MType	$\mu \ ::= \ \mu^0 \mid \forall\bar{\alpha}.\mu^0 \qquad \text{where } \bar{\alpha} \subseteq \mathit{tvars}(\mu^0)$
	$\mu^0 \ ::= \ \mu^1 \mid (\forall\bar{\alpha}.\mu^1_1) :\Rightarrow \mu^1_2 \qquad \text{where } \bar{\alpha} \subseteq \mathit{tvars}(\mu^1_1)$
	$\mu^1 \ ::= \ \mu^0 \mid \mu^0 :\Rightarrow \mu^1 \mid \kappa\langle\tau\rangle \mid \nu$
Ordered constraint	$O \ ::= [\mu_1, \ldots, \mu_k]$
Unordered constraint	$U \ ::= \ \{\mu_1, \ldots, \mu_k\}$
Constraint	$O \rhd U$
Qualified type QType	$\sigma \ ::= \ O \rhd U \Rightarrow \tau$
Type scheme $\mathsf{TScheme}$	$\eta \ ::= \ \forall\bar{\alpha}.\sigma \qquad \text{where } \bar{\alpha} = \mathit{tvars}(\sigma)$
Substitution	$\theta, \hat{\theta}$
Haskell 98 context	$\Gamma : \mathsf{Var} \mathrel{+\!\!\!\!\twoheadrightarrow} (\mathsf{QType} + \mathsf{TScheme})$
Module context	$\Delta : \mathsf{MVar} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathsf{MType}$

Fig. 1. Syntax

When we write $S_1 \oplus S_2$, this denotes the union $S_1 \cup S_2$ and additionally expresses the fact that S_1 and S_2 are disjoint. We write $\mathsf{mgu}(\tau_1, \tau_2)$ resp. $\mathsf{mgu}(\mu_1, \mu_2)$ to denote the most general unifier for types τ_1 and τ_2, or module types μ_1 and μ_2, respectively.

In Sect. 5 we argued that it makes sense to accept module arguments for the first argument position expecting a matching argument type. In order to preserve type-substitutivity, we have to make sure that no earlier position unifies with the argument type. Therefore, we define the partial function fstmgu that takes as arguments a constraint and a module type μ. In case of success, fstmgu returns a substitution θ that instantiates μ, together with the constraint without μ.

- $\mathsf{fstmgu}([\mu_1, \ldots, \mu_{i-1}, \mu_i, \mu_{i+1}, \ldots, \mu_k] \rhd U, \mu) = ([\mu_1, \ldots, \mu_{i-1}, \mu_{i+1}, \ldots, \mu_k] \rhd U, \theta)$ if $\theta = \mathsf{mgu}(\mu, \mu_i)$ and no μ_j with $j < i$ is unifiable with μ, and

- $\mathsf{fstmgu}(O \rhd U \oplus \{\mu_1\}, \mu) = (O \rhd U, \theta)$ if $\theta = \mathsf{mgu}(\mu, \mu_1)$, and no element of O and no other element of U is unifiable with μ.

In addition, we define $\mathsf{delfst}(O \rhd U, \mu) = O' \rhd U'$ iff $\mathsf{fstmgu}(O \rhd U, \mu) = (O' \rhd U', \mathsf{id})$.

The notation $\mathsf{gen}(\Gamma, \sigma)$ is used when we want to denote the generic type scheme resulting from "generalisation" over the type variables in σ:

$$\mathsf{gen}(\Gamma, \sigma) = \forall \bar{\alpha}.\sigma \qquad \text{where } \bar{\alpha} = \mathsf{tvars}(\sigma) \setminus \mathsf{tvars}(\Gamma)$$

Substitutions form an upper semilattice with ordering $\preceq$, where $\theta_1 \preceq \theta_2$ iff $\exists \theta' \bullet \theta'\theta_1 = \theta_2$. We write $\theta_1 \sqcup \theta_2$ to denote the least upper bound of two substitutions in this semilattice.

Finally, we need a partial function join to join two potentially complex constraints into a single unordered constraint, if possible. Therefore, $\mathsf{join}(O_1 \rhd U_1, O_2 \rhd U_2, \Gamma)$ is defined iff all functor types in O_1 and O_2 are contained in Γ (as representants of anonymous default instances), and then its value is the union of U_1 and U_2 and the set containing all atomic module types from O_1 and O_2.

9.2 Well Typed Terms

The following type system is an extension of a standard Hindley-Milner type system. What distinguishes it is primarily the presence of the new module context Δ which keeps track of named instances. We define a term as being well typed if and only if it may be derived by the rules in Fig. 2.
Well-typedness judgements for module expressions, resp. for terms are therefore of the following shapes:

$$\Delta; \Gamma \vdash m : \mu \qquad\qquad \Delta; \Gamma \vdash t : \sigma$$

Named instances are accessed via the rule $(\mathsf{MVar})^{\mathrm{WT}}$. Functor application $(\mathsf{App}_{\#})^{\mathrm{WT}}$ is straight-forward.

The standard λ-calculus rules $(\mathsf{Var})^{\mathrm{WT}}$, $(\lambda)^{\mathrm{WT}}$, and $(\mathsf{App})^{\mathrm{WT}}$ are "mostly standard", with the following exceptions:

- the λ-bound variable needs to have an un-constrained type, since otherwise the stratification between the two type systems would be destroyed, and
- application has to join ordered constraints into an unordered constraint since ordering makes no sense here, as seen in Sect. 5.

The next four rules work on the interface between terms and module terms: Module abstraction via the $(\lambda_{\#})^{\mathrm{WT}}$ rules is similar to λ-abstraction, where polymorphic module types have to be annotated. Note that the $(\lambda_{\#})^{\mathrm{WT}}$ rules always include the module type of the bound module variable into the *ordered* constraint. In order to include polymorphic module types smoothly into our calculus, we must allow the instantiation of their type variables $\bar{\alpha}$ with arbi-

$$(\mathsf{MVar})^{\mathrm{WT}}\quad \frac{\texttt{instance}\ i : \mu \in \Delta}{\Delta;\Gamma \vdash i : \mu} \qquad (\mathsf{App}_{\#})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma \vdash i : \mu_1 :\Rightarrow \mu \qquad \Delta;\Gamma \vdash j : \mu_1}{\Delta;\Gamma \vdash (i \mathbin{\#} j) : \mu}$$

$$(\mathsf{Var})^{\mathrm{WT}}\quad \frac{e : O \rhd U \Rightarrow \tau \ \in\ \Gamma}{\Delta;\Gamma \vdash e : O \rhd U \Rightarrow \tau} \qquad (\lambda)^{\mathrm{WT}}\quad \frac{\Delta;\Gamma, x : \tau_1 \vdash e : O_2 \rhd U_2 \Rightarrow \tau_2}{\Delta;\Gamma \vdash (\lambda x.e) : O_2 \rhd U_2 \Rightarrow \tau_1 \to \tau_2}$$

$$(\mathsf{App})^{\mathrm{WT}}\quad \frac{\begin{array}{ll}\Delta;\Gamma \vdash e_1 : O_1 \rhd U_1 \Rightarrow \tau_1 \to \tau & \\ \Delta;\Gamma \vdash e_2 : O_2 \rhd U_2 \Rightarrow \tau_1 & U = \mathsf{join}(O_1 \rhd U_1, O_2 \rhd U_2, \Gamma)\end{array}}{\Delta;\Gamma \vdash (e_1\, e_2) : [] \rhd U \Rightarrow \tau}$$

$$(\lambda_{\#,1})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma, i : \mu \vdash e : O \rhd U \Rightarrow \tau \qquad \mu \not\equiv \forall\bar{\alpha}.\mu_0}{\Delta;\Gamma \vdash (\lambda_{\#} i.e) : ([\mu] +\!\!+ O) \rhd U \Rightarrow \tau}$$

$$(\lambda_{\#,2})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma, i : \mu \vdash e : O \rhd U \Rightarrow \tau}{\Delta;\Gamma \vdash (\lambda_{\#}(i{::}\mu).e) : ([\mu] +\!\!+ O) \rhd U \Rightarrow \tau}$$

$$(\mathsf{Inst}_{\#})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma \vdash i : \forall\bar{\alpha}.\mu}{\Delta;\Gamma \vdash i : [\bar{\tau}/\bar{\alpha}]\mu} \qquad (\#)^{\mathrm{WT}}\quad \frac{\Delta;\Gamma, e : O \rhd U \Rightarrow \tau, \mu \in O \vdash i : \mu}{\Delta;\Gamma \vdash (e \mathbin{\#} i) : \mathsf{delfst}(O \rhd U, \mu) \Rightarrow \tau}$$

$$(\mathsf{Let})^{\mathrm{WT}}\quad \frac{\begin{array}{ll}\Delta;\Gamma \vdash u : O_1 \rhd U_1 \Rightarrow \tau_1 & \\ \Delta;\Gamma, p : \eta \vdash t : O_2 \rhd U_2 \Rightarrow \tau_2 & \eta = \mathsf{gen}(\Gamma, O_1 \rhd U_1 \Rightarrow \tau_1)\end{array}}{\Delta;\Gamma \vdash (\texttt{let}\, p = u\, \texttt{in}\, t) : O_2 \rhd U_2 \Rightarrow \tau_2}$$

$$(\mathsf{auto})^{\mathrm{WT}}\quad \frac{\begin{array}{ll} & U_2 = \{\kappa\langle\tau_1\rangle, \ldots, \kappa\langle\tau_n\rangle\} \\ \Delta;\Gamma \vdash e : \forall\bar{\alpha}\,.\,O \rhd U \Rightarrow \tau & \exists\theta_1, \theta_2\,.\,\theta_1(\kappa\langle\chi\,\tau_1 \ldots \tau_n\rangle) \in \theta_2 U_2 \\ U = U_1 \oplus U_2 \oplus \{\kappa\langle\chi\,\tau_1 \ldots \tau_n\rangle\} & U_2 \Vdash_\Gamma \kappa\langle\chi\,\tau_1 \ldots \tau_n\rangle\end{array}}{\Delta;\Gamma \vdash e : O \rhd U_1 \oplus U_2 \Rightarrow \tau}$$

$$(\mathsf{force})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma \vdash e : O \rhd (U_1 \oplus U_2 \oplus U_3) \Rightarrow \tau \quad U_1 \cup U_3 \Vdash_\Gamma U_2 \quad U_3 = \{\mu_1, \ldots, \mu_k\}}{\Delta;\Gamma \vdash (e{::}(O +\!\!+ [\mu_1, \ldots, \mu_k]) \rhd U_1 \Rightarrow \tau) : (O +\!\!+ [\mu_1, \ldots, \mu_k]) \rhd U_1 \Rightarrow \tau}$$

Fig. 2. Well-typedness rules

trary types $\bar{\tau}$ via the $(\mathsf{Inst}_{\#})^{\mathrm{WT}}$ rule. Explicit "dictionary application" $(\#)^{\mathrm{WT}}$ is, as discussed above, not restricted to arguments matching the first argument type of the constraint.

(Non-recursive) `let` bindings are treated as usual; there is no need to add the constraints of u to those of t, since they are already taken care of via the presence of p.

The last group of rules corresponds to constraint change and constraint reduction. "Automatic" reduction only takes place in ambiguous *unordered* constraints. Note that the rule $(\mathsf{auto})^{\mathrm{WT}}$ only applies if and only if the type variables are polymorphic.

Automatic constraint reduction might be considered as problematic since we may derive *several* different types for *one* term — as Haskell 98 does in a number of cases. As an example consider `f :: Eq a, Eq [a] ⇒ [a] → [a]` which has also the type `{Eq a} ⇒ [a] → [a]`. For this reason, type inference can only be complete when the unordered constraint of a qualified type is irreducible. In addition, the rule $(\mathsf{auto})^{\mathrm{WT}}$ may only be applied (repeatedly) as last steps of the derivation of the type of u in the $(\mathsf{Let})^{\mathrm{WT}}$ rule (that is, top-level in binding groups). Reduction via $(\mathsf{auto})^{\mathrm{WT}}$ is obviously normalising because it strictly reduces the size of the unordered constraint, and because of transitivity of entailment.

Less problematic is constraint change through explicit type annotation, which is denoted by "$::$" as usual in Haskell. There are two effects possible here. The first is elimination of atomic module types from the unordered constraint if they can be entailed from anonymous instances in Γ and the remaining unordered constraint. The second allows to move atomic module types from the unordered into the ordered constraint.

9.3 Type Inference Algorithm

An only slightly more involved set of rules defines a type inference algorithm for our system. In comparison with well-typedness judgements, type inference judgements carry an additional substitution; this substitution and the inferred type are the output of the algorithm, while the two contexts Δ and Γ and the (module) term are its input:

$$\overset{\uparrow}{\theta};\overset{\downarrow}{\Delta};\overset{\downarrow}{\Gamma}\vdash \overset{\downarrow}{i}{:}\overset{\uparrow}{\mu} \qquad\qquad \overset{\uparrow}{\theta};\overset{\downarrow}{\Delta};\overset{\downarrow}{\Gamma}\vdash \overset{\downarrow}{t}{:}\overset{\uparrow}{\sigma}$$

We understand a type inference judgement to be a *result* of the algorithm if no further derivation is possible — the rule "(auto)" would otherwise introduce ambiguities. It is understood that type inference is impossible if any of the operations used in a rule invocation is undefined. The rules of the type inference algorithm as shown in Fig. 3 correspond to the rules in the preceding section.

The relationship between the type system and the inference algorithm is made precise by the following two theorems.

Theorem 9.1 (Soundness) *For every (module) term judgement resulting from the type inference algorithm, a corresponding well-typedness judgement may be derived:*

$$\theta;\Delta;\Gamma\ \vdash\ i : \mu \implies \theta\Delta;\theta\Gamma\ \vdash\ i : \mu$$
$$\theta;\Delta;\Gamma\ \vdash\ t : O \rhd U \Rightarrow \tau \implies \theta\Delta;\theta\Gamma\ \vdash\ t : O \rhd U \Rightarrow \tau$$

Theorem 9.2 (Completeness) *For every well-typedness judgement not involving rank-2 module types, a corresponding judgement may be derived via the type inference algorithm:*

$$\theta\Delta;\theta\Gamma\ \vdash\ i : \mu \implies \exists\theta_1,\theta_2,\mu_1\ .\ \theta_1;\Delta;\Gamma\ \vdash\ i : \mu_1 \wedge$$
$$\theta\Delta = \theta_2\theta_1\Delta\ \wedge\ \theta\Gamma = \theta_2\theta_1\Gamma\ \wedge\ \mu = \theta_2\mu_1$$

$$(\mathsf{MVar})^{\mathrm{TI}} \quad \frac{\mathtt{instance}\, i : \mu \in \Delta}{\mathtt{id}; \Delta; \Gamma \vdash i : \mu} \qquad (\mathsf{Inst}_{\#})^{\mathrm{TI}} \quad \frac{\theta; \Delta; \Gamma \vdash i : \forall \overline{\alpha}.\mu}{\theta; \Delta; \Gamma \vdash i : [\overline{\alpha'}/\overline{\alpha}]\mu}$$

$$(\mathsf{App}_{\#})^{\mathrm{TI}} \quad \frac{\theta_1; \Delta; \Gamma \vdash i : \mu_1 :\Rightarrow \mu \qquad \theta_2; \Delta; \Gamma \vdash j : \mu_2 \qquad \theta = \theta_1 \sqcup \theta_2 \sqcup \mathsf{mgu}(\mu_1, \mu_2)}{\theta; \Delta; \Gamma \vdash (i \# j) : \theta\mu}$$

$$(\mathsf{Var})^{\mathrm{TI}} \quad \frac{e : O \rhd U \Rightarrow \tau \in \Gamma}{\mathtt{id}; \Delta; \Gamma \vdash e : O \rhd U \Rightarrow \tau} \qquad (\lambda)^{\mathrm{TI}} \quad \frac{\theta; \Delta; \Gamma, x : \tau_1 \vdash e : O_2 \rhd U_2 \Rightarrow \tau_2}{\theta; \Delta; \Gamma \vdash (\lambda x.e) : O_2 \rhd U_2 \Rightarrow \tau_1 \to \tau_2}$$

$$(\mathsf{App})^{\mathrm{TI}} \quad \frac{\begin{array}{ll} \theta_1; \Delta; \Gamma \vdash e_1 : O_1 \rhd U_1 \Rightarrow \tau_1 \to \tau & \theta = \theta_1 \sqcup \theta_2 \sqcup \mathsf{mgu}(\tau_1, \tau_2) \\ \theta_2; \Delta; \Gamma \vdash e_2 : O_2 \rhd U_2 \Rightarrow \tau_2 & U = \mathsf{join}(O_1 \rhd U_1, O_2 \rhd U_2, \Gamma) \end{array}}{\theta; \Delta; \Gamma \vdash (e_1\, e_2) : \theta([] \rhd U \Rightarrow \tau)}$$

$$(\lambda_{\#,1})^{\mathrm{TI}} \quad \frac{\theta; \Delta; \Gamma, i : \mu \vdash e : O \rhd U \Rightarrow \tau \qquad \mu \not\equiv \forall \bar{\alpha}.\mu_0}{\theta; \Delta; \Gamma \vdash (\lambda_{\#} i.e) : ([\mu] +\!\!+ O) \rhd U \Rightarrow \tau}$$

$$(\lambda_{\#,2})^{\mathrm{TI}} \quad \frac{\theta; \Delta; \Gamma, i : \mu \vdash e : O \rhd U \Rightarrow \tau}{\theta; \Delta; \Gamma \vdash (\lambda_{\#}(i{::}\mu).e) : ([\mu] +\!\!+ O) \rhd U \Rightarrow \tau}$$

$$(\#)^{\mathrm{TI}} \quad \frac{\begin{array}{ll} \theta_1; \Delta; \Gamma \vdash e : O_1 \rhd U_1 \Rightarrow \tau & (O \rhd U, \theta_3) = \mathsf{fstmgu}(O_1 \rhd U_1, \mu) \\ \theta_2; \Delta; \Gamma \vdash i : \mu & \theta = \theta_1 \sqcup \theta_2 \sqcup \theta_3 \end{array}}{\theta; \Delta; \Gamma \vdash (e \# i) : \theta(O \rhd U \Rightarrow \tau)}$$

$$(\mathsf{Let})^{\mathrm{TI}} \quad \frac{\begin{array}{ll} \theta_1; \Delta; \Gamma \vdash u : O_1 \rhd U_1 \Rightarrow \tau_1 & \eta = \mathsf{gen}(\theta_1\Gamma, O_1 \rhd U_1 \Rightarrow \tau_1) \\ \theta_2; \Delta; \Gamma, p : \eta \vdash t : O_2 \rhd U_2 \Rightarrow \tau_2 & \theta = \theta_1 \sqcup \theta_2 \end{array}}{\theta; \Delta; \Gamma \vdash (\mathtt{let}\, p = u\, \mathtt{in}\, t) : \theta(O_2 \rhd U_2 \Rightarrow \tau_2)}$$

$$(\mathsf{auto})^{\mathrm{TI}} \quad \frac{\begin{array}{ll} & U_2 = \{\kappa\langle\tau_1\rangle, \ldots, \kappa\langle\tau_n\rangle\} \\ \theta; \Delta; \Gamma \vdash e : \forall \bar{\alpha}\, .\, O \rhd U \Rightarrow \tau & \exists \theta_1, \theta_2\, .\, \theta_1(\kappa\langle\chi\, \tau_1 \ldots \tau_n\rangle) \in \theta_2 U_2 \\ U = U_1 \oplus U_2 \oplus \{\kappa\langle\chi\, \tau_1 \ldots \tau_n\rangle\} & U_2 \Vdash_{\Gamma} \kappa\langle\chi\, \tau_1 \ldots \tau_n\rangle \end{array}}{\theta; \Delta; \Gamma \vdash e : O \rhd U_1 \oplus U_2 \Rightarrow \tau}$$

$$(\mathsf{force})^{\mathrm{TI}} \quad \frac{\theta; \Delta; \Gamma \vdash e : O \rhd (U_1 \oplus U_2 \oplus U_3) \Rightarrow \tau \qquad U_1 \cup U_3 \Vdash_{\Gamma} U_2 \qquad U_3 = \{\mu_1, \ldots, \mu_k\}}{\theta; \Delta; \Gamma \vdash (e{::}(O +\!\!+ [\mu_1, \ldots, \mu_k]) \rhd U_1 \Rightarrow \tau) : (O +\!\!+ [\mu_1, \ldots, \mu_k]) \rhd U_1 \Rightarrow \tau}$$

Fig. 3. Type inference rules

$$\theta\Delta; \theta\Gamma \vdash t : \sigma \implies \exists \theta_1, \theta_2, \sigma_1\, .\ \theta_1; \Delta; \Gamma \vdash t : \sigma_1 \wedge$$
$$\theta\Delta = \theta_2\theta_1\Delta \ \wedge\ \theta\Gamma = \theta_2\theta_1\Gamma \ \wedge\ \sigma = \theta_2\sigma_1$$

If furthermore the unordered constraint of σ is irreducible wrt. $(\mathsf{auto})^{\mathrm{WT}}$*, then the unordered constraint of σ_1 is irreducible wrt.* $(\mathsf{auto})^{\mathrm{TI}}$*, too.*

The proofs of both theorems proceed by induction on the structure of derivations. Note that Theorem 2 implies $\mu_1 \preceq \mu$ and $\sigma_1 \preceq \sigma$. Apart from that, possible types σ can only differ in unordered constraints of t. Since this holds for all possible module types μ of i and all possible types τ of t, the algorithm yields the principal types for i and t.

β-reduction is defined as usual, and it interacts sensibly with type inference. Since ordered constraints have to be joined in expression application, we obtain a weaker variant of subject reduction:

Definition 9.3 A qualified type $\sigma_1 = O_1 \rhd U_1 \Rightarrow \tau_1$ is called *weaker than* a qualified type $\sigma_2 = O_2 \rhd U_2 \Rightarrow \tau_2$ under context Γ, iff there are a substitution θ and three subsequences O_o, O_u, and O_e of O_1, such that O_1 is an interleaving of O_o, O_u, and O_e, and

$$\theta\tau_1 = \tau_2 \ , \qquad \theta O_\mathrm{o} = O_2 \ , \qquad \theta(|O_\mathrm{u}|) \cup \theta U_1 = U_2 \ , \qquad U_2 \Vdash_\Gamma \theta O_\mathrm{e} \ .$$

Theorem 9.4 (Subject reduction) *If $\theta_1; \Delta; \Gamma \ \vdash \ t_1 : \sigma_1$ holds by type inference and the term t_1 β-reduces to another term t_2, then there are θ_2 and σ_2 with $\theta_2 \preceq \theta_1$ such that σ_2 is weaker than σ_1 and the type inference judgement $\theta_2; \Delta; \Gamma \ \vdash \ t_2 : \sigma_2$ holds.*

As an aside, the above definition of the *weaker than* relation can also be used to help remedy the fact that with lazier context reduction for named instances — in the same way as with implicit parameters — user-supplied type signatures frequently hinder adaptability of Haskell code. We propose *lax type signatures*, which are to be understood as asserting a lower bound (with respect to the above *weaker than* relation) on the type of the defined entity. This implies that

```
f ::< forall a . (Eq a) => a -> b
```

would allow any of the following:

```
f :: (Eq a         ) => a -> b
f :: (Eq a         ) => a -> String
f :: (Ord a        ) => a -> b   -- modulo entailment equivalence
f :: (Eq a, Read b) => a -> b
```

Forcing the user to make the `forall` explicit follows the guideline that these lax type signatures enable users to *enforce* structure for inferred types by explicitly providing it. Writing

```
f ::< exists b . (Eq a) => a -> b
```

instead would rather have a taste of explicitly specifying degrees of freedom — but it does not specify degrees of freedom in the context. For the latter, an appropriate syntax seems to be much harder to find. Therefore, we find the `forall` variant more natural.

10 Type Checking Prototype

In order to be able to experiment with our design of module type constraints, we developed a prototype type checker. It extends the Haskell 98 type checker "Typing Haskell in Haskell" of Mark Jones [7] with module type constraints and performs type inference in the manner described in the preceding sections. We mainly extended the handling of constraints and introduced user defined type classes and (named) instances as well as type checking inside them. Furthermore some experimental features as for example subtyping (see Sect. 7) have been implemented as well as extensions that are not covered in detail by this paper (see below).

Nevertheless, the basic features of “Typing Haskell in Haskell” have not been compromised, and our prototype continues to accept the original examples provided by Mark Jones, and to infer their principal types. This serves to increase our confidence that our extension is essentially conservative. The prototype is available on the WWW [7]. Further developments will also be published at this address.

11 Conclusion and Outlook

The desire to enable later binding of Haskell type class members together with the known analogies between the Haskell type class system and ML module systems lead us to design an extension of Haskell comprising named instances, explicit instance supply, instance functors and module types. This extension incorporates a subset of Jones’ parameterised signatures, but in a way that preserves compatibility with the Haskell 98 type class system and refrains form giving structures first-class citizenship, so in this respect stays closer to the separate module language of OCaml.

In addition, the desire to remain compatible to conventional Haskell leads also to a new feature: type class constraints now have to be considered as module type constraints and thus lead to a new class of qualified types which, as we have seen, can be considered as closely related to the implicit parameters of Lewis *et al.* [13].

Apart from this, our work is of course closely related to work on first class module systems, most notably that by Russo [17,18]. We believe that extending our proposal towards first-class module system capabilities is a natural step and will most probably be necessary for many uses. Nevertheless we have intentionally left out such considerations. We consider that already our relatively small extensions have quite far-reaching consequences both conceptually and from the point of view of expressive power and (re)usability, so that it makes sense to study them in relative isolation.

The relation with work on multi-parameter classes [16] turns out to be relatively weak — module type constraints mainly offer a way to obtain many of the benefits of multi-parameter classes, short-circuiting a lot of the problems research has been centring around. We think that our extension may in fact allow more exploration of library design centred around multi-parameter classes — it would certainly have been useful in the design of the relation algebra toolkit RATH [10].

The design we presented should not be considered as an attempt at a

[7] URL: http://ist.unibw-muenchen.de/Haskell/NamedInstances/

conclusive definition of named instance features. Instead, we tried to present an apparently reasonable subset of non-trivial features amidst some discussion of the decisions involved. Due to lack of space we have concentrated on the basic idea and left out further extensions that are already implemented in our prototype and seem to prove their usefulness. A detailed discussion will be contained in the second author's diploma thesis [19].

Among these extensions are notations to directly access the anonymous *default* instances (via the special instance name `Default`) and anonymous *derived* instances (via the special instance name `Derived`). The possibility to access these instances directly makes it possible to use them as explicit functor arguments. We consider furthermore the possibility to import instances "`as Default`".

A straightforward step towards first-class instances is to make their parameterisation via module variables possible as well as their *local* definition via `let` or `where` expressions — the example in Sect. 8 shows a possible notation.

We have not discussed implementation at all — this is mainly because it does not seem to be a serious problem. Since current Haskell implementations of type classes rely on dictionary translation, implementing our features mostly amounts to extending the parser and connecting it to existing features of the implementation, which just had not been directly accessible via the user-level language before. We intend to address this in the near future.

In summary, following through the consequences of explaining the Haskell type system in terms of the ML module system is not quite as trivial as it might seem at first sight. However, it enforces useful conceptual clarifications and then gives rise to a natural extension that, in our opinion, will also be of great value to users of Haskell.

References

[1] Ralf Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. Tech. Report UU-CS-1999-28.

[2] Ralf Hinze. A new approach to generic functional programming. Technical Report IAI-TR-99-9, Institut für Informatik III, Universität Bonn, 1999.

[3] Ralf Hinze and Simon Peyton-Jones. Derivable type classes. In *Haskell Workshop 2000*, September 2000.

[4] M. P. Jones and J.C. Peterson. *Hugs 98 — A functional programming system based on Haskell 98 — User Manual*, 1999.

[5] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 97–136. Springer, 1995.

[6] Mark P. Jones. Using parameterized signatures to express modular strucure. In *23rd POPL*, pages 68–78. acm press, 1996.

[7] Mark P. Jones. Typing Haskell in Haskell. In *Third International Haskell Workshop 1999*, 1999.

[8] Mark P. Jones. Type classes with functional dependencies. In Smolka [20], pages 230–244.

[9] S. Peyton Jones and P. Wadler. A static semantics for Haskell. Technical Report G12 8 QQ, University of Glasgow, 1992.

[10] Wolfram Kahl and Gunther Schmidt. Exploring (finite) Relation Algebras using Tools written in Haskell. Technical Report 2000-02, Fakultät für Informatik, Universität der Bundeswehr München, October 2000.

[11] Xavier Leroy. Manifest types, modules, and separate compilation. In *21th POPL*, pages 109–122. acm press, 1994.

[12] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd POPL*. acm press, 1995.

[13] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *27th POPL*. acm press, 2000.

[14] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald Sanella, editor, *ESOP '94*, volume 788 of *LNCS*, pages 409–424. Springer, 1994.

[15] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[16] Simon L. Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space, 1997.

[17] Claudio V. Russo. *Types For Modules*. LFCS thesis ECS-LFCS-98-389, University of Edinburgh, 1998.

[18] Claudio V. Russo. First-class structures for Standard ML. In Smolka [20], pages 336–350.

[19] Jan Scheffczyk. Named instances with class in Haskell. Diploma thesis, Fakultät für Informatik, Universität der Bundeswehr München, 2001. Forthcoming. See also: http://ist.unibw-muenchen.de/Haskell/NamedInstances/.

[20] G. Smolka, editor. *ESOP 2000, 10th European Symposium on Programming*, volume 1782 of *LNCS*. Springer, March 2000.

***

A Functional Notation for Functional Dependencies

Matthias Neubauer [1] Peter Thiemann [2]

Institut für Informatik
Universität Freiburg,
Georges-Köhler-Allee Geb.079
D-79085 Freiburg i. Br., Germany

Martin Gasbichler [3] Michael Sperber [4]

Wilhelm-Schickard-Institut
Universität Tübingen
Sand 13
D-72076 Tübingen, Germany

Abstract

Functional dependencies help resolve many of the ambiguities that result from the use of multi-parameter type classes. They effectively enable writing programs at the type-level which significantly enhances the expressive power of Haskell's type system. Among the applications of this technique are the emulation of dependent types, and precise typechecking for XML and HTML combinator libraries. Unfortunately, the notation presently used for functional dependencies implies that the type-level programs are *logic* programs, but many of its applications are conceptually *functional* programs. We propose an alternative notation for functional dependencies which adds a functional-programming notation to Haskell's type classes and makes applications of functional dependencies significantly more readable. We apply the new notation to our examples and study the problems arising due to Haskell's open world assumption and overlapping instances.

Since the invention of type classes more than a decade ago [12], every new year has seen astonishing new applications and interesting extensions of the original idea. The most recent addition is Jones's incorporation of *functional dependencies* [8] which allow the formulation of tighter constraints on the instances

[1] Email: `neubauer@informatik.uni-freiburg.de`
[2] Email: `thiemann@informatik.uni-freiburg.de`
[3] Email: `gasbichl@informatik.uni-tuebingen.de`
[4] Email: `sperber@informatik.uni-tuebingen.de`

of multi-parameter type classes. A functional dependency is a declaration which specifies that a set of parameters of a type class uniquely determines another parameter. Such a specification avoids many ambiguous typings.

Functional dependencies constrain the instance declarations so that they specify a function at the type level. This functionality makes it feasible to write regular programs entirely evaluated by the type checker, opening the door for a wide range of potential applications [6,9].

Unfortunately, the use of functional dependencies is hampered by the traditional notation of type classes as relations and instances as logic programs computing these relations: As the resulting programs are getting more complex, they often become awkward and hard to read. This is not a fault of the mechanism of functional dependencies per se, but of the syntax offered by current implementations.

Thus, in this paper, we suggest a notational shift for type classes: View a type class as a type-level *function* instead of a type-level *relation*. We offer some realistic example applications of functional dependencies which would benefit significantly from such a notation. Moreover, we give a brief formal outline of how the new notation may be implemented by a translation to the traditional syntax. We present the relevant part of a larger example—type-safe combinator libraries for XML documents—and discuss some of the problems remaining.

1 Simulating Dependent Types

We start our investigation with two simple examples drawn from the realm of dependent types: formatted printing and specified list operations.

1.1 A type-safe `sprintf`

The `sprintf` function from the C-library is an example of an unsafe function that can be made type-safe by using a dependent type [1]: The first parameter of `sprintf` is a format specifier which determines the number and type of the remaining parameters. The idea here is that a type-level function computes the type of the remaining parameters from the format specifier. For this to work in Haskell, format specifiers cannot simply be strings with control characters but rather `String` constants or values from the following datatypes: [5]

```
data I f = I f
data C f = C f
data S f = S String f
```

Using these datatypes, the format specifier

```
S "Int: " (I (S ", Char: " (C ".")))
```

[5] Danvy [5] has shown that this particular example can be made to work relying only on ML-style polymorphism.

means "The literal string '`Int: `' followed by an integer followed by the literal string '`, Char: `' followed by a character and terminated by a period." A somewhat more convenient notation for the format specifier is the following:

```
S "Int: " $ I $ S ", Char: " $ C $ "."
```

The type of the format specifier contains an almost complete encoding of the format specifier itself, only omitting the string literals. In this case, the type is:

```
S (I (S (C String)))
```

A type class `SPRINTF` specifies a member function `sprintf1` which accepts a prefix, that it prepends to the output, a format specifier, and matching further arguments:

```
class SPRINTF m o | m -> o where
  sprintf1 :: String -> m -> o

instance SPRINTF String String where
  sprintf1 prefix str = prefix ++ str

instance SPRINTF a funa => SPRINTF (I a) (Int -> funa) where
  sprintf1 prefix (I a) i = sprintf1 (prefix ++ show i) a

instance SPRINTF a funa => SPRINTF (C a) (Char -> funa) where
  sprintf1 prefix (C a) c = sprintf1 (prefix ++ [c]) a

instance SPRINTF a funa => SPRINTF (S a) funa where
  sprintf1 prefix (S str a) = sprintf1 (prefix ++ str) a
```

The instance declarations comprise a logic program for computing `o` from `m` using the relation `SPRINTF`. In particular, the functional dependency `m -> o` is a kind of mode declaration.

The main entry point, `sprintf`, supplies an empty prefix to `sprintf1`:

```
sprintf :: SPRINTF m o => m -> o
sprintf = sprintf1 ""
```

For example, the type of

```
sprintIntChar = sprintf (S "Int: " $ I $ S ", Char: " $ C $ ".")
```

is

```
sprintIntChar :: Int -> Char -> String
```

When applied to an integer and a character, `sprintIntChar` yields

```
> sprintIntChar 42 'x'
"Int: 42, Char: x."
```

1.2 Specified list operations

Another classic example for the use of dependent types is length-indexed lists [13,9]. A translation of this example into Haskell first requires an encoding of natural numbers at the type level to encode list lengths:

```
data ZERO = ZERO
data SUCC n = SUCC n
```

ADD is a type class that encodes addition on these two datatypes at the type level using a functional dependency:

```
class ADD a b c | a b -> c
instance ADD ZERO b b
instance ADD a b c => ADD (SUCC a) b (SUCC c)
```

Again, the instance declarations for ADD constitute a small logic program.

Two datatypes encode heterogeneous lists on the value-level and on the type-level:

```
data NIL = NIL
data CONS x xs = CONS x xs
```

It is now possible to compute the length of a list statically via another logic program at the type level:

```
class LISTLENGTH xs len | xs -> len where
  listLength :: xs -> len

instance LISTLENGTH NIL ZERO where
  listLength NIL = ZERO
instance LISTLENGTH xs len
           => LISTLENGTH (CONS x xs) (SUCC len) where
  listLength (CONS x xs) = SUCC (listLength xs)
```

Here is a definition of an append operation on fixed-length lists:

```
class APPEND li1 li2 app | li1 li2 -> app where
  append :: li1 -> li2 -> app

instance APPEND NIL y y where
  append NIL y = y
instance APPEND xs ys app
           => APPEND (CONS x xs) ys (CONS x app) where
  append (CONS x xs) ys = CONS x (append xs ys)
```

With these declarations in place, it is natural to relate the definitions of LISTLENGTH and ADD to that of APPEND: Two concatenated lists must be as long as the sum of their lengths. The condition can be seen as a partial specification of append. Again, the constraint as a logic program is part of a revised instance declaration for APPEND:

```
instance (APPEND xs ys app,
          LISTLENGTH xs m,
          LISTLENGTH ys n,
          ADD m n s,
          LISTLENGTH app s)
           => APPEND (CONS x xs) ys (CONS x app) where
  append (CONS x xs) ys = CONS x (append xs ys)
```

Now, changing LISTLENGTH to something incompatible with the constraint results in a type error as soon as append occurs applied to a non-NIL list as first argument anywhere in a program. Consider the following buggy definition of LISTLENGTH:

```
instance LISTLENGTH NIL (SUCC ZERO) where
  listLength NIL = SUCC ZERO

instance LISTLENGTH xs len
           => LISTLENGTH (CONS x xs) (SUCC ZERO) where
  listLength (CONS x xs) = SUCC ZERO
```

As predicted, appending anything to (CONS 1 NIL) fails at compile time. Here is the error reported by Hugs:

```
> append (CONS 1 NIL) NIL
ERROR: Constraints are not consistent with functional dependency
*** Constraint       : ADD ZERO (SUCC ZERO) ZERO
*** And constraint   : ADD ZERO (SUCC ZERO) (SUCC ZERO)
*** For class        : ADD a b c
*** Break dependency : b a -> c
```

2 Functional is Better

The examples in the preceding section show that the logic-programming notation for type classes leads to poor readability in cases where the programmer really wants to define functions rather than predicates. In particular, we identify the following shortcomings:

- A functional language should express functional dependencies directly as functions. In the case of LISTLENGTH, where the listLength member mirrors the computation at the type level on the value level, the dichotomy is especially painful.
- The notation is occasionally difficult to read. Consider the final instance declaration of APPEND: The reader must discover that the type variable s is shared between the last argument of ADD and that of LISTLENGTH. In the sprintf example, nested applications are moved to the predicate set and linked via result variables.

In the following subsections, we propose a functional syntax for type-level functions and demonstrate their use with our examples.

2.1 Functional `sprintf`

The modifications to the syntax are small and only affect the class and instance declarations. The syntax of expressions does not change. The header of the class declaration expresses the functional dependency directly. The first line of an instance declaration becomes a defining equation for a type function, the keyword `where` precedes the definition of the member function:

```
class SPRINTF :: m -> o where
  sprintf1 :: String -> m -> o

instance SPRINTF String = String                  where
  sprintf1 prefix str   = prefix ++ str

instance SPRINTF (I a)  = Int -> SPRINTF a        where
  sprintf1 prefix (I a) = \i  -> sprintf1 (prefix ++ show i) a

instance SPRINTF (C a)  = Char -> SPRINTF a       where
  sprintf1 prefix (C a) = \c   -> sprintf1 (prefix ++ [c]) a

instance SPRINTF (S a)      = SPRINTF a           where
  sprintf1 prefix (S str a) = sprintf1 (prefix ++ str) a

sprintf :: m -> SPRINTF m
sprintf = sprintf1 ""
```

The main conceptual change inherent in the notation is that type classes no longer specify predicates on types but rather functions. A class declaration by itself specifies the kind of the instance declarations; a new-style declaration

```
class SPRINTF :: m -> o
```

stands for the previous functional dependency

```
class SPRINTF m o | m -> o
```

The instance declarations define the corresponding function. This notation is arguably more natural than the logic-programming notation. This is not merely the case for this specific notation, but for a number of typical applications of functional dependencies.

With a functional notation on the type-level, qualified type schemes that incorporate predicates with functional dependencies can also be expressed more concisely: As the type annotation of `sprintf` shows, we can shift the application of `SPRINTF` to the position where the resulting type occurs.

2.2 List operations

The list operations also benefit from the new notation. This example also opens an additional issue, namely the question where to put the extra specification predicates that relate `ADD`, `LISTLENGTH`, and `APPEND`.

$t ::= a$		type variable
$\mid C$		type constructor
$\mid F$		type function
$\mid t\ t$		type application
$d ::=$ `class` $F :: a_1\ \ldots\ a_n \to a$		class declaration
$\mid$ `instance` $(t_1$ `==` $t'_1; \ldots; t_m$ `==` $t'_m) \Rightarrow F\ t_1\ \ldots\ t_n$ `=` t		instance declaration

Fig. 1. Syntax for functional instances

```
class LISTLENGTH :: xs -> len where
  listLength :: xs -> len
instance LISTLENGTH NIL = ZERO where
  listLength NIL = ZERO
instance LISTLENGTH (CONS x xs) = SUCC (LISTLENGTH xs) where
  listLength (CONS x xs) = SUCC (listLength xs)

class ADD :: a b -> c
instance ADD ZERO b = b
instance ADD (SUCC a) b = SUCC (Add a b)

class APPEND :: li1 li2 -> app where
  append :: li1 -> li2 -> app
instance APPEND NIL y = y where
  append NIL y = y
instance (LISTLENGTH (APPEND xs ys) ==
           ADD (LISTLENGTH xs) (LISTLENGTH ys))
          => APPEND (CONS x xs) ys = CONS x (APPEND xs ys) where
  append (CONS x xs) ys = CONS x (append xs ys)
```

This example shows that an instance declaration can carry an *axiom* consisting of an equality constraint: `LISTLENGTH (APPEND xs ys) == ADD (LISTLENGTH xs) (LISTLENGTH ys)`. The intended semantics of this constraint is unification. The original logic program models this by sharing the type variable `s`.

3 Functional Instances

We call the applicative notation for functional dependencies *functional instances*. The implementation of functional instances does not require any new machinery in the underlying Haskell system. The implementation is a syntactic translation of functional instances into standard Haskell instance declarations with functional dependencies. The translation resembles the flattening translation for functional logic programming languages [2]. This section gives an account of this translation. Figure 1 shows the syntax for functional

$$\vdash_\delta \texttt{class}\ F :: a_1\ \dots\ a_n \to a \rightsquigarrow \texttt{class}\ F\ a_1\ \dots\ a_n\ a \mid a_1\ \dots\ a_n \to a$$

$$\frac{\vdash_\tau t \rightsquigarrow P_1 \mid t' \qquad \vdash_\sigma \{(l_1, r_1), \dots, (l_m, r_m)\} \rightsquigarrow P_2}{\vdash_\delta \texttt{instance}\ (l_1\ \texttt{==}\ r_1; \dots; l_m\ \texttt{==}\ r_m) \Rightarrow F\ t_1\ \dots\ t_n\ \texttt{=}\ t}$$

$$\rightsquigarrow \texttt{instance}\ P_1, P_2 \Rightarrow F\ t_1\ \dots\ t_n\ t'$$

$$\vdash_\tau a \rightsquigarrow \emptyset \mid a$$

$$\vdash_\tau C \rightsquigarrow \emptyset \mid C$$

$$\frac{\vdash_\tau t_1 \rightsquigarrow P_1 \mid t_1' \qquad \vdash_\tau t_2 \rightsquigarrow P_2 \mid t_2'}{\vdash_\tau t_1\ t_2 \rightsquigarrow P_1, P_2 \mid t_1'\ t_2'}$$

$$\frac{\vdash_\phi t \rightsquigarrow P \mid t'}{\vdash_\tau t \rightsquigarrow P, t'\ a \mid a} \quad a \text{ fresh}$$

$$\frac{\vdash_\phi t_1 \rightsquigarrow P_1 \mid t_1' \qquad \vdash_\tau t_2 \rightsquigarrow P_2 \mid t_2'}{\vdash_\phi t_1\ t_2 \rightsquigarrow P_1, P_2 \mid t_1'\ t_2'}$$

$$\vdash_\phi F \rightsquigarrow \emptyset \mid F$$

Fig. 2. Translation to old-style class and instance definitions

instances. An instance declaration may specify a set of axioms that must be fulfilled for the instance to apply. `instance` $F\ t_1\ \dots\ t_n$ `=` t is an abbreviation for `instance` $() \Rightarrow F\ t_1\ \dots\ t_n$ `=` t.

In addition to the constraints imposed by the grammar, in every declaration `instance` $(\dots) \Rightarrow F\ t_1\ \dots\ t_n$ `=` t, the terms $t_1, \dots, t_n$ are *constructor terms*, meaning that no type function symbols F occur in them. Also, each type function of arity n must always be fully applied to n arguments.

The translation specified in Figures 2 and 3 is defined using five judgments:

(i) The judgment $\vdash_\delta d \rightsquigarrow d'$ maps a new-style definition, d, to an old-style definition, d'. It is used for translating class and instance definitions. The translation adds a result parameter to the class definition and states the functional dependency. For an instance definition, it translates the right-hand side of the definition into a predicate set and a type term.

(ii) The judgment, $\vdash_\tau t \rightsquigarrow P \mid t'$ translates a type term, t, into a predicate

$$\vdash_\sigma \emptyset \rightsquigarrow \emptyset$$

$$\frac{\vdash_\omega (l, r) \rightsquigarrow P_1 \qquad \vdash_\sigma \{x_2, \ldots, x_n\} \rightsquigarrow P_2}{\vdash_\sigma \{(l, r), x_2, \ldots, x_n\} \rightsquigarrow P_1, P_2}$$

$$\frac{\vdash_\tau t_1 \rightsquigarrow P_1 \mid t_1' \qquad \vdash_\tau t_2 \rightsquigarrow P_2 \mid t_2'}{\vdash_\omega (t_1, t_2) \rightsquigarrow P_1, P_2, \texttt{EQV}\ t_1'\ t_2'}$$

Fig. 3. Translation of axioms

set, P, and a type, t'. Its main job is the translation of constructor terms, which is straightforward.

(iii) The translation of applications of type functions is left to the judgment, $\vdash_\phi t \rightsquigarrow P \mid t'$. It is only defined for terms t of the form $F\ t_1\ \ldots\ t_n$. All arguments, t_i, are translated using $\vdash_\tau\rightsquigarrow$ and the final rule that connects $\vdash_\phi\rightsquigarrow$ to $\vdash_\tau\rightsquigarrow$ adds the result parameter, a, as a fresh variable and moves the resulting predicate into the predicate set.

(iv) The judgment, $\vdash_\sigma \{(l_1, r_1), \ldots\} \rightsquigarrow P$, translates a set of equations, l_i `==` r_i, into a predicate, P. Its only function is to dispatch single equations to the next judgment.

(v) The judgment, $\vdash_\omega (t_1, t_2) \rightsquigarrow P$, translates a single equation into a predicate set, P. It translates t_1 and t_2 into predicates and type expressions without type functions. It enforces equality of t_1 and t_2 by making use of an auxiliary type class `EQV a b`. This class specifies that `a` and `b` must be equal:

```
class EQV a b | a -> b, b -> a
instance EQV a a
```

The functional dependencies in the class declaration specify that `EQV` is an invertible function. The only instance defines that `EQV` is the identity function.

It is a simple exercise to show that the judgment $\vdash_\delta\rightsquigarrow$ really defines a function.

Lemma 3.1 *If $\vdash_\delta d \rightsquigarrow d'$ and $\vdash_\delta d \rightsquigarrow d''$ then d' is equal to d'' up to renaming of free variables.*

4 A Larger Example: Regular Expressions

Besides giving a larger, real-life example, we also pinpoint new problems with pattern matching on types in this section.

In our work on generating valid HTML and XML documents in Haskell

[11], we model the restrictions that a DTD poses on the content of one particular XML element. The key ingredient in this model is a type class `AddTo elem subelem`, which relates each element to its corresponding admissible subelements. This approach requires that each element can be identified via its type.

A DTD specifies, in principle, a regular language of subelements for the contents of each element. It is straightforward to implement a simple recognizer for each element specified in the DTD as a regular expression matcher. Typechecking the uses of the element constructors then requires keeping track of the states of this matching automaton at the type level. The transition function is a generalization of the `AddTo` class, namely a class `NEXTSTATE subelem elem elem'`. In this approach, each element has an initial type (corresponding to the initial state of its automaton) and adding subelements changes the type of the element according to the transition function.

4.1 Data-Level Regular Expressions

Because of the laborious nature of standard type-level programs, we first present the transition function in the more familiar value-level syntax; this simplifies the presentation. First, here is a datatype for regular expressions:

```
type Symbol = String

data Regexp =
     Empty | Epsilon | Atom Symbol |
     Seq Regexp Regexp | Alt Regexp Regexp | Star Regexp
```

The transition function `nextState` maps an input symbol a and a regular expression r to a regular expression r' so that $L(r') = \{w|aw \in L(r)\}$. This is a well-known trick for matching a string to a regular expression. The same trick can also be used to translate a regular expression directly into a deterministic finite state automaton with regular expressions as its states [3]:

```
nextState :: Symbol -> Regexp -> Regexp
nextState a Empty       = Empty
nextState a Epsilon     = Empty
nextState a (Atom b)    = if a == b then Epsilon else Empty
nextState a (Seq r1 r2) = if finalState r1
                          then alt ra rb
                          else ra
                          where ra = seq (nextState a r1) r2
                                rb = nextState a r2
nextState a (Alt r1 r2) = alt (nextState a r1) (nextState a r2)
nextState a (Star r)    = seq (nextState a r) (Star r)
```

The `alt` and `seq` functions are “smart” versions of the corresponding `Alt` and `Seq` data constructors which use standard algebraic identities to simplify the resulting regular expressions on the fly; see Appendix A for their definitions.

The auxiliary `finalState` function (occasionally called `nullable` in the parsing folklore), when applied to a regular expression r, yields `True` when $\varepsilon \in L(r)$, thus identifying final states in the automaton:

```
finalState :: Regexp -> Bool
finalState Empty       = False
finalState Epsilon     = True
finalState (Atom b)    = False
finalState (Seq r1 r2) = finalState r1 && finalState r2
finalState (Alt r1 r2) = finalState r1 || finalState r2
finalState (Star r)    = True
```

4.2 Type-Level Regular Expressions

One possible approach to lifting the regular expression matcher to the type level is to explicitly compute the automaton first and translate it into a set of instance declarations. This is somewhat more involved than merely implementing the transition function. Consequently, it is more desirable to simply lift the entire computation of the automaton to the type level, preserving as much of the structure of the value-level program as possible.

The first step is to lift the declaration of regular expressions to the type level. This works just like the natural-number and fixed-length list examples above:

```
data EMPTY   = EMPTY
data EPSILON = EPSILON
data ATOM t  = ATOM t
data SEQ r s = SEQ r s
data ALT r s = ALT r s
data STAR r  = STAR r
```

In the context of the HTML combinator library, the argument type of `ATOM` is a tag such as `DL`, `DD`, or `DT`. Each of these tags is represented by a corresponding (singleton) type, like `EMPTY` or `EPSILON`.

The transition function maps a state and a symbol to the next state. Hence, it must be modeled using a type function `NEXTSTATE`:[6]

```
class NEXTSTATE :: t s -> s' where
  nextState :: t -> s -> s'
```

For readability, we omit some definitions of the `nextState` member function in the running text (see Appendix B for the complete definitions). The first two cases are simple:

```
instance NEXTSTATE t EMPTY = EMPTY where
  nextState t EMPTY = EMPTY
instance NEXTSTATE t EPSILON = EMPTY where
  nextState t EPSILON = EMPTY
```

[6] A version of the code written in the traditional notation for functional dependencies is in Appendix B.

But the next case, dealing with ATOM, raises an interesting problem: The conditional in the original program must distinguish between a matching and a non-matching input symbol. Hence, it must be able to compare types and "branch" on the results. The naive approach would be this one:

```
instance NEXTSTATE t (ATOM t) = EPSILON where ...
instance NEXTSTATE t (ATOM s) = EMPTY where ...
```

Unfortunately, the instances violate the functional dependency: all parameters are simultaneously unifiable by mapping s to t, but the results are different.

In a more expressive language, it would be possible to state that $\mathtt{t} \neq \mathtt{s}$ in the second instance. Unfortunately, it is not obvious how such constraints might be incorporated into the type language because inequations are not admissible predicates in Jones's framework of qualified types [7] (they are not closed under substitution). In addition, it is not clear if term equality is really the predicate that we want to test.

A working solution introduces an explicit equality test in the form of a type class EQUAL:

```
class EQUAL :: s t -> b where
  equal :: s -> t -> b
instance EQUAL DL DL = TRUE  where equal DL DL = TRUE
instance EQUAL DL DT = FALSE where equal DL DT = FALSE
instance EQUAL DL DD = FALSE where equal DL DD = FALSE
⋮
```

where TRUE and FALSE are the obvious singleton types. To be useful, there must be an instance definition for each pair of types, which are admissible symbol types. Subsection 4.4 discusses the problem in more detail.

With this in place, the ATOM case turns into a single instance:

```
instance NEXTSTATE t (ATOM s) = IF (Equal s t) EPSILON EMPTY where
  nextState t (ATOM s) = cond (equal s t) EPSILON EMPTY
```

where the IF type class implements a conditional on the level of types:

```
class IF :: b t1 t2 -> t where
  cond :: b -> t1 -> t2 -> t
instance IF TRUE  t1 t2 = t1 where cond TRUE t1 t2 = t1
instance IF FALSE t1 t2 = t2 where cond FALSE t1 t2 = t2
```

Interestingly, this conditional "evaluates" both branches, potentially even before the condition itself is known.

Sequences are the most difficult case. Fortunately, the techniques developed so far apply in the same way:

```
instance NEXTSTATE t (SEQ r1 r2) =
           IF (FINALSTATE r1)
              (ALT (SEQ (NEXTSTATE t r1) r2) (NEXTSTATE t r2))
              (SEQ (NEXTSTATE t r1) r2) where
```

```
  nextState t (SEQ r1 r2) =
    let ra = seq' (nextState t r1) r2
        rb = nextState t r2
    in  cond (finalState r1) ra (alt' ra rb)
```

The same holds for `ALT` and `STAR`:

```
instance NEXTSTATE t (ALT r1 r2) =
           ALT (NEXTSTATE t r1) (NEXTSTATE t r2) where
  nextState t (ALT r1 r2) =
    alt' (nextState t r1) (nextState t r2)
instance NEXTSTATE t (STAR r) =
           SEQ (NEXTSTATE t r) (STAR r) where
  nextState t (STAR r) =
    seq' (nextState r) (STAR r)
```

The definition of `FINALSTATE` is straightforward to derive from the value-level function `finalState`. It is omitted because it poses no additional problems. It relies on type-level versions of `and` and `or`, `AND` and `OR`, which are also straightforward.

4.3 Smart Constructors

In the data-level formulation, smart versions of `Alt` and `Seq` called `alt` and `seq` simplify the resulting regular expressions on-the-fly. When computing a matching automaton at the data level, this is strictly necessary to obtain a finite set of states. The type-level does not need to enumerate all the states, so it is possible to do without simplification. However, after several transitions, the type terms and their evidence-data terms grow enormously and slow down the type checker considerably.

Implementing smart constructors at the type level poses some interesting additional problems. The smart versions of `ALT` and `SEQ` are type classes called `ALT'` and `SEQ'`. Their class declarations are as follows:

```
class SEQ' :: r1 r2 -> r
class ALT' :: r1 r2 -> r
```

Their instance declarations can be easily derived from their data-level counterparts. However, they must be spelled out in considerably more detail. In particular, most variables must be eliminated from the patterns. For example:

```
instance ALT' EMPTY r     = r
instance ALT' r     EMPTY = r
```

These two instances overlap and hence the type checker rejects them. Even the implemented extensions for overlapping instances [10] do not help in this case because each instance matches the other.

One approach to make the instances non-overlapping is to define instances for all combinations of parameters `r1` and `r2` to `ALT'`. However, this results in a quadratic number of instances.

A better approach is to *sequentialize* the class ALT'. In this approach, ALT' is only responsible for dispatching on the first parameter. Dispatching on the second parameter is left to yet another type class ALT''. Sequentializing "only" doubles the number of instances. Here is the beginning of that effort:

```
instance ALT' EMPTY r2 = r2
instance ALT' r1    r2 = ALT'' r1 r2
```

The last line gives rise to an overlapping instance, but it can be resolved using the extension for overlapping instances.[7]

Ideally, an implementation of ALT'' would look as follows:

```
class ALT'' :: r1 r2 -> r
instance ALT'' r1 EMPTY = r1
instance ALT'' r1 r2    = ALT r1 r2
```

Unfortunately, the last line breaks the functional dependency: in the case where r2 = EMPTY, the last line says that ALT'' r1 EMPTY = ALT r1 EMPTY which seems to contradict the functional dependency of ALT'' (at least, this is what the implementations tell us). Consequently, all the alternatives need to be listed:

```
instance ALT'' r1 EPSILON     = ALT r1 EPSILON
instance ALT'' r1 (ATOM t)    = ALT r1 (ATOM t)
instance ALT'' r1 (SEQ s1 s2) = ALT r1 (SEQ s1 s2)
instance ALT'' r1 (ALT s1 s2) = ALT r1 (ALT s1 s2)
instance ALT'' r1 (STAR s)    = ALT r1 (STAR s)
```

4.4 The Problem with Equality

The astute reader will have asked why the type class EQUAL is defined as a function that explicitly maps a pair of types to TRUE or FALSE. It would be much more in line with typical type class programming to define a type class EQUAL2 along the following lines (in the traditional syntax):

```
class EQUAL2 a b
instance EQUAL2 DL DL
```

[7] Unfortunately, the February 2001 release of Hugs 98 is not able to correctly process the instances of ALT'. To obtain runnable code, it is necessary to expand the second line by giving the following set of instance definitions:

```
instance ALT' EMPTY r2 r2
instance (ALT'' EPSILON r2 r) => ALT' EPSILON r2 r
instance (ALT'' (ATOM t) r2 r) => ALT' (ATOM t) r2 r
instance (ALT'' (SEQ s1 s2) r2 r) => ALT' (SEQ s1 s2) r2 r
instance (ALT'' (ALT s1 s2) r2 r) => ALT' (ALT s1 s2) r2 r
instance (ALT'' (STAR r1) r2 r) => ALT' (STAR r1) r2 r
```

These instance definitions require a facility to turn off the termination guarantee of predicate rewriting (thus making type checking undecidable) because there are non-variable types in the predicates.

```
instance EQUAL2 DT DT
instance EQUAL2 DD DD
⋮
```

This type class could even be specified with a linear number of instance declarations.

Unfortunately, `EQUAL2` is not up to the job. Since the Haskell class system is designed with an open-world assumption in mind, it is impossible to extract negative information from the definition of `EQUAL2`. A predicate like `EQUAL2 DL DT` simply remains unreduced in the context. Hence, it cannot be used to select a particular instance.

Neither does the class `EQV` from Section 3 perform the desired task. Using the predicate `EQV t1 t2` immediately unifies `t1` and `t2`, thereby making it impossible to reach the `EMPTY` case for `NEXTSTATE t (ATOM s)`.

5 Towards better Type-Level Pattern Matching

While the syntax proposed in the previous sections is appealing and makes type-level programming much easier, it still leaves much to be desired. Most prominently, function definitions on the data level are still more readable than the ones on the type level, largely because data-level functions can utilize wild-card patterns and type-level functions cannot. Unfortunately, a naive translation of wild-card patterns to instances gives rise to overlapping instances. These overlapping instances are not easily resolved (witness Section 4.3), and give rise to subtle type checking problems.

Part of the problem lies in the as yet unexplored interaction of functional dependencies and overlapping instances. Since there is no underlying theory that encompasses both, there is still work to be done here.

Another part of the problem stems from the fact that type classes are the only structuring tool available for types. Data values can be classified by types in a much more rigorous fashion than types can be classified by type classes. This weak classification of types is due to the open nature of type classes: The typing of a program must not depend on the instances known at compile time. Instead, the program is guaranteed to work regardless of the instances that are added later on. Hence, it is impossible to automatically expand a wild-card pattern into the potential parameter types because this set might be different depending on the context of use of a particular module.

Moreover, the simplification example (Sec. 4.3), requires sequentializing the pattern matching. This "pattern matching by hand" seems wasteful, but is impossible to automate, as pattern-match compilation heavily relies on wild-card patterns.

A potential way out of the problem with wild-card patterns (and hence with pattern matching, too) lies in enriching the kind structure. Currently, the only kinds available are types, $*$, and arrow kinds, $\kappa \to \kappa'$. An *enumerated* *kind* defines a set of types, but in contrast to a type class, its set of types cannot be changed once it is defined.

The proposed notation for the simplification example (Sec. 4.3) is the following:

```
class ALT' r1 r2 -> r with
  r1, r2 <- { EMPTY, EPSILON, ATOM t, SEQ r1 r2, ALT r1 r2, STAR r1 }
```

Using this information, the compiler can compile pattern matching by expanding wild cards. [8]

6 Getting It Back

All examples presented in this paper exclusively rely on type classes defined using functional instances. The notation does not subsume the old notation for type classes. On the other hand, it is easy to embed old-style type classes into functional instances by always returning a dummy `Success` type from the type-level functions.

Moreover, the notation for functional instances also does not subsume functional dependencies. Specifically, functional instances do not not cover the following cases:

(i) functional dependencies which do not cover all of the variables of the type class

(ii) multiple functional dependencies in a single type class

In the first case, a class written using functional instances introduces type variables on the left-hand side which do not occur on the right-hand side of its instances. Naive implementations of type inference for functional dependencies might miss opportunities for improvement in this case. However, it is straightforward to detect unused left-hand-side type variables and omit them from the generated functional dependencies.

The second case is more involved, and it is hard to assess the value of multiple functional dependencies, as there are few published examples which make fundamental use of them. (Jones's original paper on functional dependencies [8] contains not a single concrete example for multiple dependencies, and the examples in McBride's collection [9] also work without them in concrete

[8] Curiously, it *is* already possible to define type classes that correspond to singleton kinds. For example, the class

```
class STrue true | -> true
```

has a functional dependency, which states that its parameter `true` depends on nothing. Hence, it must be a constant function (or undefined). Only a single instance declaration is possible:

```
instance STrue TRUE
```

Any use of `STrue a` in a context immediately simplifies to `a = TRUE`.

applications.) In any case, it seems it is usually possible to rewrite multiple functional dependencies into multiple type classes implementing the different modes specified by them.

7 Conclusions

We have shown examples for performing non-trivial computations on types using extensions of the Haskell class mechanism. We propose an alternative, applicative syntax for specifying type functions and we provide a translation from new-style classes and instances to the original formulation.

While we have shown how to translate the new functional instances into the old functional dependencies, we have not addressed the issue of how to do the reverse. Moreover, there is the question of how to define and handle ordinary type classes without functional dependencies in the new framework. One simple option is to keep the old notation entirely. Another option would be to reinterpret all type classes as functions rather than predicates. This means that inheritance constraints could also be formulated using equations over the values of type-class applications. The right mix is the subject of ongoing work.

Given the fragile nature of the available implementations of functional dependencies and the drawbacks listed in Section 5, we feel that the support for type-level programming is still in its infancy. To get the best of both worlds, there must be a better way of classifying types than type classes. The natural proposition here is to look to dependently typed systems and to allow lifting data-level computation to the type level [4]. This would solve the classification problem (since every lifted data value belongs to a lifted type, which is a kind), and address the problems about pattern matching and properties, too.

Also, since pattern matching has a well-defined semantics, such a step would alleviate the problems with overlapping instances. It remains, of course, to integrate this kind of computations into a type checker. This is also part of our ongoing work.

References

[1] Augustsson, L., *Cayenne—a language with dependent types*, in: P. Hudak, editor, *Proc. International Conference on Functional Programming 1998* (1998).

[2] Barbuti, R., M. Bellia, G. Levi and M. Martelli, *LEAF: A language which integrates logic, equations and functions*, in: D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, 1986 .

[3] Brzozowski, J. A., *Derivatives of regular expressions*, Journal of the ACM **11** (1964), pp. 481–494.

[4] Cardelli, L., *Phase distinctions in type theory* (1988), unpublished Manuscript.

[5] Danvy, O., *Functional unparsing*, Journal of Functional Programming **8** (1998), pp. 621–625.

[6] Hallgren, T., *Fun with functional dependencies*, in: *Joint Winter Meeting of the Departments of Science and Computer Engineering, Chalmers University of Technology and Göteborg University*, Varberg, Sweden, 2001, `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html`.

[7] Jones, M. P., "Qualified Types: Theory and Practice," Cambridge University Press, Cambridge, UK, 1994.

[8] Jones, M. P., *Type classes with functional dependencies*, in: G. Smolka, editor, *Proc. 9th European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science (2000), pp. 230–244.

[9] McBride, C., *Faking it—simulating dependent types in Haskell* (2001), `http://www.dur.ac.uk/~dcs1ctm/faking.ps`.

[10] Peyton Jones, S., M. Jones and E. Meijer, *Type classes: An exploration of the design space*, in: J. Launchbury, editor, *Proc. of the Haskell Workshop*, Amsterdam, The Netherlands, 1997, yale University Research Report YALEU/DCS/RR-1075.

[11] Thiemann, P., *Modeling HTML in Haskell*, in: *Practical Aspects of Declarative Languages, Proceedings of the Second International Workshop, PADL'00*, number 1753 in Lecture Notes in Computer Science, Boston, Massachusetts, USA, 2000, pp. 263–277.

[12] Wadler, P. and S. Blott, *How to make ad-hoc polymorphism less ad-hoc*, in: *Proc. 16th Annual ACM Symposium on Principles of Programming Languages* (1989), pp. 60–76.

[13] Xi, H. and F. Pfenning, *Dependent types in practical programming*, in: A. Aiken, editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages* (1999), pp. 214–227.

A Smart Constructors

These are the value-level definitions of the `seq` and `alt` smart constructors:

```
seq Empty r = Empty
seq r Empty = Empty
seq Epsilon r = r
seq r Epsilon = r
seq (Seq r1 r2) r3 = Seq r1 (seq r2 r3)
seq r1 r2 = Seq r1 r2
```

```
alt Empty r = r
alt r Empty = r
alt Epsilon Epsilon = Epsilon
alt Epsilon (Star r) = Star r
alt (Star r) Epsilon = Star r
alt (Alt r1 r2) r3 = Alt r1 (alt r2 r3)
alt r1 r2 = Alt r1 r2
```

B Instance Declarations for `NEXTSTATE` and `FINALSTATE`

Here is the code for the `NEXTSTATE` class in the traditional syntax for functional dependencies:

```
class NEXTSTATE t s s' | t s -> s' where
  nextState :: t -> s -> s'

instance NEXTSTATE t EMPTY EMPTY where
  nextState t EMPTY = EMPTY

instance NEXTSTATE t EPSILON EMPTY where
  nextState t EPSILON = EMPTY

instance (EQUAL s t b, IF b EPSILON EMPTY r)
           => NEXTSTATE t (ATOM s) r where
  nextState t (ATOM s) = cond (equal s t) EPSILON EMPTY

instance (FINALSTATE r1 b,
          NEXTSTATE t r2 rb,
          NEXTSTATE t r1 ra',
          SEQ' ra' r2 ra,
          ALT' ra rb rb'
          IF b ra rb' r)
           => NEXTSTATE t (SEQ r1 r2) r where
  nextState t (SEQ r1 r2) =
    let ra = seq' (nextState t r1) r2
        rb = nextState t r2
    in  cond (finalState r1) ra (alt' ra rb)

instance (NEXTSTATE t r1 r1',
          NEXTSTATE t r2 r2',
          ALT' r1' r2' r')
           => NEXTSTATE t (ALT r1 r2) r' where
  nextState t (ALT r1 r2) =
    alt' (nextState t r1) (nextState t r2)

instance (NEXTSTATE t r r',
          SEQ' r' (STAR r) r'')
           => NEXTSTATE t (STAR r) r'' where
  nextState t (STAR r) =
    seq' (nextState r) (STAR r)
```

Here is the definition of FINALSTATE in the traditional syntax:

```
class FINALSTATE s t | s -> t where
  finalState :: s -> t

instance FINALSTATE EMPTY FALSE
instance FINALSTATE EPSILON TRUE
instance FINALSTATE (ATOM t) FALSE
instance (FINALSTATE r r', FINALSTATE s s', AND r' s' u')
           => FINALSTATE (SEQUENCE r s) u'
instance (FINALSTATE r r', FINALSTATE s s', OR  r' s' u')
           => FINALSTATE (UNION r s) u'
instance FINALSTATE r r' => FINALSTATE (PLUS r) r'
instance FINALSTATE (STAR r) TRUE
instance FINALSTATE (OPTION r) TRUE
```

The definition of the IF class in the traditional syntax:

```
class IF b t1 t2 r | b t1 t2 -> r where
  cond :: b -> t1 -> t2 -> r

instance IF TRUE t1 t2 t1 where
  cond TRUE t1 t2 = t1
instance IF FALSE t1 t2 t1 where
  cond FALSE t1 t2 = t2
```

The definition of FINALSTATE requires type-level versions of the logical operations && and ||. These are two type classes AND and OR. Their definition in the traditional syntax:

```
class OR s t u | s t -> u
instance OR FALSE FALSE FALSE
instance OR FALSE TRUE  TRUE
instance OR TRUE  FALSE TRUE
instance OR TRUE  TRUE  TRUE

class AND s t u | s t -> u
instance AND FALSE FALSE FALSE
instance AND FALSE TRUE  FALSE
instance AND TRUE  FALSE FALSE
instance AND TRUE  TRUE  TRUE
```

GHood – Graphical Visualisation and Animation of Haskell Object Observations

Claus Reinke [1]

Computing Laboratory, University of Kent

Canterbury, UK

Abstract

As a possible extension to his Haskell Object Observation Debugger Hood [7], Andy Gill has described the "dynamic viewing of structures", stepping through observations instead of accumulating them into a static view. Starting from this idea, we have implemented and released an animation back-end for Hood, called GHood. Instead of the dynamic textual visualisation based on pretty-printing proposed in [7], our back-end features a dynamic graphical visualisation, based on a simple tree layout algorithm. This paper reviews the main aspects of Hood, gives a brief introduction to GHood's features and summarises our experience so far.

The visualisation of program behaviour via animations of data structure observations has uses for program comprehension and exposition, in development, debugging and education. We find that the graphical structure facilitates orientation even when textual labels are no longer readable due to scaling, suggesting advantages over a purely textual visualisation. A novel application area is opened by the use of GHood as an applet on web pages – discussions of Haskell program behaviour, e.g., in educational online material or in explanations of functional algorithms, can now easily be augmented with graphical animations of the issues being discussed.

1 Well-typed programs don't go anywhere – or do they?

The war-cry of static typing is that "well-typed programs don't go wrong", but sometimes the question is "where does this well-typed program go?", requiring a more detailed understanding of program behaviour.

For a surprisingly long time, Haskell programmers have been deprived of tools that would enable them to investigate the behaviour of their programs at a suitable level of abstraction. This lack of tool support, especially in the areas of debugging and profiling, has been quoted as one of the reasons "why no one uses functional languages" [18]. In the context of Haskell profiling, the lack has

[1] `mailto:c.reinke@ukc.ac.uk` `http://www.cs.ukc.ac.uk/people/staff/cr3/`

not been felt quite so urgently, because increasingly sophisticated lower-level tools have continued to appear (support still varies between implementations, though, and tools are implementation-specific). Still, there is a discrepancy: if programs are written in a nice high-level language, why do their dynamic aspects have to be studied in low-level terms of stack- and heap-usage? And in the area of debugging, the situation has only just started to improve.

A recent survey [3] compares three tools for tracing and debugging of lazy functional programs: Hat [20], Freja [13], and Hood [7]. All of these systems offer inspection facilities at a level close to the programming language, based on different forms of execution traces, and can be characterised on the basis of the questions they help to answer. Hat [2] takes wrong program output as starting points, enabling users to trace backwards through reduction sequences ("where did this result or output come from?"). Freja supports a technique known as declarative debugging, involving users in a dialogue that narrows down to the source of errors ("this part of your program gives the following result. Is this correct (yes/no)?"). For Hood, it is useful to imagine a data-flow model of functional program execution, with parameters flowing into operators or functions and results flowing out. On this basis, programmers can use Hood to insert probes into their programs to monitor or observe the flow of data at runtime ("what kind of data structure is flowing through here?").

Tracing tools offer high-level views into Haskell program executions. Focusing on different aspects of program behaviour, the existing tools complement each other, but it turns out that they all provide essentially static views of program execution traces, highlighting logical connections between intermediate terms instead of execution dynamics. As a possible extension to Hood, Andy Gill described the "dynamic viewing of structures", stepping through observations using a textual form of visualisation based on pretty-printing [7]. Gill implemented and demonstrated a browser back-end for Hood, based on this idea (the back-end itself is available from the Haskell CVS repository, but it is not supported by the Hood observation library, as released in July 2000; that Haskell library implements the observation combinator by accumulating observations and printing a static view at the end of program runs).

We are here concerned with extending the usefulness of Hood (the most recent of these tools, and also the only implementation-independent one) by adding dynamic views of observation traces. Starting from Gill's idea, and building on the Hood observation library, we have implemented and released a graphical animation back-end for Hood, called GHood. Instead of a dynamic textual visualisation based on pretty-printing, our back-end features a dynamic graphical visualisation, based on a simple tree layout algorithm. After reviewing the main aspects of Hood, this paper gives a brief introduction to GHood's features, demonstrates some of the new applications enabled by GHood by way of two small examples, and summarises our experience so far.

[2] Hat has since been extended considerably, and now supports several models of tracing, implemented on top of a single program execution trace (cf. Section 5.1, as well as [19,20]).

2 Hood – goodbye trace, hello observe

The pseudo-function `trace :: String -> a -> a` – not part of any Haskell language definition, but supported by all Haskell implementations – is supposed to be acting as an identity with a `String`-label. When evaluated, it returns its second parameter, but also prints its label as a side-effect. Reminiscent of the print-statements with which imperative programmers inspect their programs in the absence of proper debuggers, side-effecting output can thus be used to generate a trace of the execution of a Haskell program.

But in the end, unconstrained use of side-effecting input/output operations is no more suitable for debugging than for any other kind of input/output in a lazy functional language. Functional input/output has moved on to more systematic, declarative means of expression, which require to make effects visible in the structure, and thus in the type of programs (Chapter 3 of [16] aims to give a logical reconstruction of the main lines in this development). But this is exactly what prevents the use of these more structured means of input/output for debugging purposes, where one wants to inspect the behaviour of a given program, without having to restructure it into something else first.

Enter Hood (Haskell Object Observation Debugger). One way of understanding Hood is via a line of reasoning similar to that which led to today's functional input/output systems – it is not the idea of side-effecting operations that is at fault, it is their undisciplined use that causes problems. As the requirements of debugging differ from those of standard input/output, a similar line of reasoning will not necessarily lead to similar solutions. In standard usage, input/output is part of the program and should be reflected in its type structure whereas, for debugging purposes, the input/output-operations are part of the workbench used to inspect the program, and the original program should be disturbed as little as possible.

Developing this idea, Hood consists of a fairly complex library with a relatively simple interface. In fact, the type of the major function has not changed much: `observe :: Observable a => String -> a -> a`. Similar to `trace`, `observe` acts as an identity with a `String` label. But the similarities end here – calls to `trace` effectively imitate *imperative* print-statements, whereas calls to `observe` capture the intention behind print-style-debugging (indicating interest in intermediate values) in a *declarative* way, leaving the "how" of capturing and presenting information to the implementation. The combination of `observe` and its observation and presentation library eliminates all the major deficiencies of `trace`:

(i) (a) With `trace`, all information is communicated via the `String` parameter. Programmers have to add code to inspect parts of their program, and to incorporate the inspection results into the `String` labels.

(b) With `observe`, instances of the `Observable` class handle all aspects of program inspection, offering a much more convenient high-level interface. The `String` parameter is just used as a label.

(ii) (a) The extra inspection code needed to feed information into `trace` labels implies non-trivial program modifications, which run the risk of introducing bugs and changing strictness properties in the process.

(b) Predefined instances for most standard types and a combinator approach to user-defined instances of `Observable` imply smaller program modifications and ensure that strictness properties of the program under inspection are not affected by the use of `observe`.

(iii) (a) When evaluated, `trace` immediately attempts to output its label. Under a lazy evaluation strategy, this may cause other traced expressions to be evaluated, and the order of output can be confusing.

(b) Evaluation of `observe` causes information to be captured, but this is decoupled from presentation and output. In Hood, the observation events are post-processed when the observed program has terminated – observations are grouped by their labels into comprehensive summaries, which are pretty-printed as partially-known data structures.

For the full details, readers are referred to the Hood paper and documentation [7,8], but for a two-parameter constructor C in an algebraic data type, the general mechanism can be illustrated by the following pseudo-code:

```
observer (C x y) = λposition -> unsafePerformIO $
  do  sendEvent <observed constructor C at position position>
      return (C (observer x position.0) (observer y position.1))
```

where `observer` is a helper function called by `observe` (initialising *position*), and *position* records the position of the current subexpression in the observed data structure. The definition is strict in the observed (sub-)structure, forcing its evaluation to weak head normal form, but *only if* the weak head normal form of the whole expression is required by the evaluation context. On this occasion, the `observer` generates an observation event, tagged with the position information, wraps any constructor parameters in new observers, and returns the observed constructor to the evaluation context.

All those implementation details are hidden behind suitable monads and combinators, offering a simple user-level interface, and observers for most standard types are predefined. The (predefined) instance of `Observable` for lists may serve to illustrate that it is straightforward, if somewhat tedious, to make new types observable:

```
instance (Observable a) => Observable [a] where
  observer (a:as) = send ":"  (return (:) << a << as)
  observer []     = send "[]" (return [])
```

Using `observe` is equally straightforward (`runO :: IO a -> IO ()` runs an `IO`-script while taking care of observation event processing):

```
import Observe
main = runO $ print $ observe "just a list" [1..4::Int]
```

3 GHood – seeing what your program does

Using a small set of commonly implemented extensions to standard Haskell, Hood instruments existing Haskell implementations to generate observation data during program evaluation, and when the observed program terminates, the stream of observation events is postprocessed and pretty-printed. The result is a portable library that can be used with the full Haskell language.

However, there is more information in the stream of observation events than is utilised in the vanilla version of Hood. Each observation event conveys three kinds of information:

(i) *what* constructor or constant is observed?

(ii) *where* is this part of a data structure located?

(iii) *when* is this part of a data structure observed?

Hood uses location information (*where*) to collate related observations and then pretty-prints the collection of partial information (*what*) about the data structures under observation. The original Hood publication [7] mentions "We have an extension to the released version of HOOD, that includes a browser that allows dynamic viewing of structures." and includes screenshots showing dynamic pretty-printing, but this combination has yet to be released [3].

For GHood, we have taken Gill's idea of using the *when* information of observation events as a basis for animating observations as our point of departure. GHood can be characterised as a new back-end for Hood's observation library – instead of textual visualisation, based on pretty-printing, we have chosen a graphical form of visualisation, based on a simple tree-layout algorithm. The visualisation consists of displaying the structure under observation as a tree, and the animation refines the display whenever an observation event adds information. With the potential exception of functions (see section 4.2), all Haskell types are of the (recursive) sum-of-products kind, and thus have a simple mapping to a tree representation. This is not always the most natural mapping – e.g., GHood currently renders `String`s as binary lists of characters.

3.1 Implementation

We have added extension hooks in the Hood observation library: apart from initialisation and finalisation, these hooks enable additional processing of observation events, either individually, as each observation occurs (extending the `sendEvent` used in `observer`), or on the event stream as a whole, between program termination and Hood's pretty-printing. These hooks give fairly good control over the production and formatting of observation logs and could be used by other postprocessing tools. No further modifications of Hood's obser-

[3] nhc98 comes bundled with pre-release versions of the browser (from the Haskell CVS repository) and the Hood observation library, the latter modified to produce the XML-based input expected by the browser (referred to as THood in section 5.1).

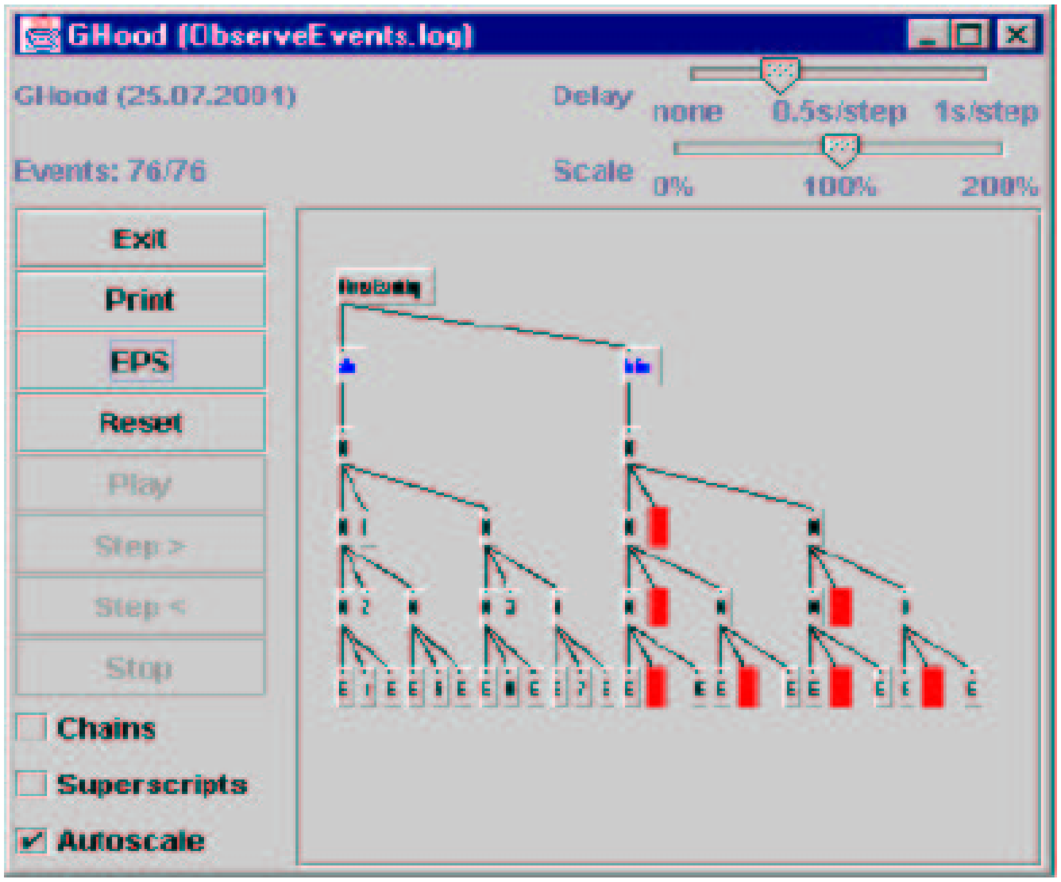


Fig. 1. GHood screenshot

vation library are necessary – the Haskell interface remains unchanged.

Using these hooks, the observation log is made available in a text file. To keep parsing of these logs in our back-end simple, log files consist of one line of plain text per observation event, giving position information and type of observation (observation label, demand for evaluation, constructor or function) for each event, as well as observation-type-specific information (arity and constructor name for observations of constructors, label text for observation labels). Observation logs can then be processed, visualised and animated in our graphical back-end GHood. The hooks give a choice between online and offline generation of external logs, with associated trade-offs: On current machines, the slow-down of programs by file i/o during evaluation in the online variant appears to be more substantial than the extra space usage by the offline version, so the latter is the default. The online version remains useful when GHood is used to debug programs that do not terminate successfully: on ghc, Hood manages to process the observation log anyway, capturing abnormal termination via exceptions, but on other Haskell implementations, only our online version of Hood generates an external log in these cases.

The GHood viewer itself is Java-based, ensuring availability on most platforms that support Haskell implementations, and it can be used with any

Haskell implementation that supports Hood (plus hooks). The graphical user interface (figure 1) is straightforward, comprising a drawing panel in which partially observed structures are displayed using a tree-layout algorithm, and a few buttons to play, stop, reset, and single-step the animation (forwards or backwards), or to print snapshots (printing produces bitmap-style Postscript, so export of vectorised encapsulated Postscript was added for use in print publications). When observation trees get large, they can be scaled down, or the panel can be scrolled, providing survey views or access to parts of the structures under observation. To provide for comprehensible automatic stepping on different platforms, controlable delays have be added between observation events in automatic animation. In the following, we focus on the observation trees, as shown in the drawing panel, but produced by the EPS export.

The main reason for implementing our own viewer was that existing graph drawing tools -as far as they have not gone commercial- appear to be limited to certain platforms or specialised towards pretty, reasonably fast (a few seconds) layout, whereas our application required portability and a quick and simple tree layout for an incrementally updated tree. The only complication resulted from the single-threaded design of Java's GUI libraries (event handlers are scheduled non-preemptively). Fortunately, GHood can be decomposed into two threads (observation tree update and GUI), only one of which requires access to the GUI, but both threads operate on the observation tree. Synchronising the threads on a per-node basis, with an atomic transaction corresponding to the processing of each observation event, appears to give a reasonable compromise between GUI responsiveness and animation progress while avoiding erroneous displays of partially updated trees.

GHood can be used as a standalone Java application or as a Java applet in web pages, and the production and visualisation of observation event logs can be decoupled. This means that online course material, documentation and publications of functional algorithms can be enhanced with dynamic visualisations without requiring a Haskell implementation on the browser side.

3.2 Observations about `unsafePerformIO` *and extension hooks*

In the implementation of `observe`, the non-standard, but commonly implemented, pseudo-function `unsafePerformIO :: IO a -> a` is used to turn an effect (logging an observation event), documented in the type of an expression, into a side-effect, so that the expression tagged with a call to `observe` can be used just as the original expression.

Traditionally, `unsafePerformIO` is seen as a means to *extend programs* with impure operations in such a way that their use, as seen from the evaluating context, can be shown to be uncritical (the prefix `unsafe` is meant to document this proof obligation). In the case of observers, however, the idea is to leave the program under observation entirely undisturbed while *extending the implementation* that runs the program. In other words, `unsafePerformIO` can also be seen as a hook provided in the Haskell evaluation mechanism.

This hook is used in `observe` to instrument the evaluator so that it performs useful logging functions when evaluating structures under obervation. And just as Hood uses an implementation hook to reuse and extend the functionality of existing Haskell implementations, GHood uses hooks in Hood to reuse the observation functionality while extending it for purposes of graphical visualisation. Such implementation extension hooks enormously simplify the implementation of portable tools, and it would seem worthwhile to create and standardise a catalogue of such hooks across Haskell implementations, moving towards portable tools that can plug into different implementations, using only the standardised extension interfaces.

Once it is understood that `unsafePerformIO` functions as an extension hook in the underlying implementation, other uses become possible as well. Instead of just logging the evaluation of some expression, the hook could be used to wait for user input before continuing the evaluation. Such user input could even be used to modify the structure under observation before passing it on to the evaluation context, enabling interactive debugging.

In the specific context of GHood, another useful implementation hook would be to the memory manager, permitting GHood to show when structures become unobservable. According to the documentation (module `Weak` in HsLibs), `addFinalizer :: a -> IO () -> IO ()` should do just that. This operation should associate an IO-script with an expression, so that the script is guaranteed to be run after the expression gets garbage collected. Unfortunately, implementation optimisations currently subvert this operation for most types, rendering it unusable in the general form.

4 GHood applications, by examples

To demonstrate the opportunities opened by GHood, we choose two examples that display non-obvious behaviour but have either been analysed recently (the breadth-first numbering problem) or can be assumed to be well-known to Haskell programmers (the interaction of non-strict evaluation with the use of `foldl` as a pattern for tail recursion). We can thus focus on the visualisation and on the information that can be derived from it. Both of the following subsections can also be seen as examples of how descriptions of functional algorithms can be augmented with animations of program behaviour. To avoid page-filling series of snapshots, we occasionally resort to radio-style textual commentaries of animations that do not easily fit into the static publication format here. Online versions of the examples discussed here are provided on the GHood home page[4], and readers are strongly encouraged to use the online animations side by side with the text here (for completeness, and to give a rough impression of the graphical animations, samples of reduced-size

[4] `http://www.cs.ukc.ac.uk/people/staff/cr3/toolbox/haskell/GHood/`

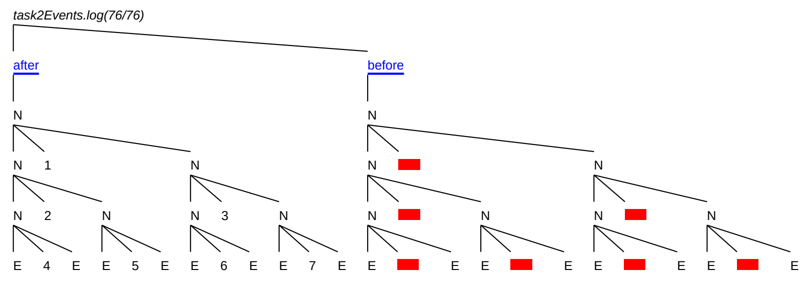


Fig. 2. End-of-run observation of breadth-first numbering

snapshot series are provided in the appendix of this paper).

4.1 Breadth-first numbering revisited

As a first small example, consider the breadth-first numbering problem proposed in a recent functional pearl [14] as "an interesting toy problem that exposes a blind spot common to many –perhaps most– functional programmers". The problem is stated as follows:

> Given a tree T, create a new tree of the same shape, but with the values at the nodes replaced by the numbers 1...|T| in breadth-first order.

Readers who have not come across this problem before are encouraged to try finding a solution for themselves before reading on (our Haskell code is in Appendix A). Originally, we tried to animate our solutions more to gain insight into the practicalities of visualisation than in the expectation to learn anything new about the problem. As a first illustration, figure 2 shows observations of two trees, one `before` and one `after` breadth-first numbering, in the final state of the animation. All observations are grouped under a root node, which also gives the name of the observation file. Below the root node come observation labels (the `String` parameters to the function `observe`), followed by tree-representations of the observed Haskell structures.

The observation labels are underlined and coloured blue[5], constructors and constants are coloured black, unobserved subexpressions (thunks) are shown as red boxes. Thunks under observation are represented as orange boxes with red outlines until their weak head normal form becomes available, and the thunk is replaced by some constructor. The typical lifecycle of a node is from "not yet inspected" (red, closed box) to "under observation, but weak head normal form not yet available" (orange, open box) to some constructor (black constructor label).

Trees are either empty (`E`) or nodes (`N`) with left and right subtree and some label, so the display in figure 2 gives the information expected from the

[5] Presentation scheme changed for publication, to facilitate readability of both colour and greyscale renderings (red and orange appear as dark and light shades of grey, respectively).

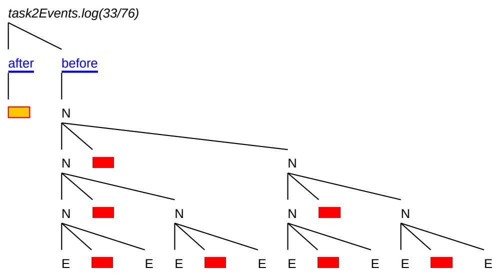


Fig. 3. A middle-of-run strictness problem

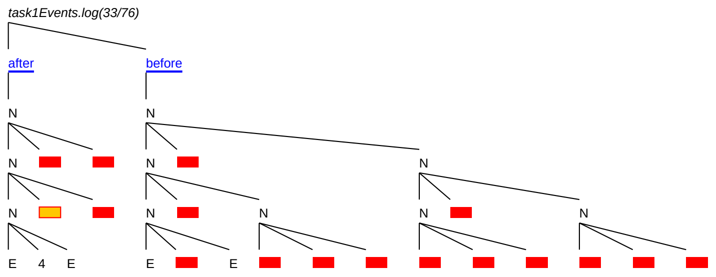


Fig. 4. Strictness problem solved?

problem specification, in that only the shape, but not the node labels of the input tree need to be inspected to construct the resulting tree, in which nodes are labeled with positive integers in breadth-first order.

The surprise came while inspecting intermediate stages of the animation – figure 3 shows an extreme situation in the middle of the run. The thunk which will evaluate to the tree after renumbering is represented as an opened box, indicating that it is being inspected by the evaluation context, but that its weak-head normal form has not yet become available. It has been in that state all the way from just after the start, while more and more of the shape of the input tree has been observed. In other words, this solution has an extreme strictness problem, inspecting parts of the input long before they should be needed! Only the very next step will replace the thunk under inspection by a node labeled `N`, with three unobserved thunks as subnodes, so no part of the result tree becomes available for observation until after all observations of the input tree shape have taken place.

Once the animation had so drastically brought this strictness problem to our attention, improving the program was not too difficult. Choosing roughly the same stage in a run of the modified program, the intermediate observation in figure 4 shows the difference quite clearly (watching the observed structures unfold dynamically during animation, it is almost impossible not to notice the difference between the two programs): parts of the resulting tree have become available for observation, right down to the first complete non-trivial sub-tree at the left, while still not all of the input tree shape has been observed.

In spite of the drastic improvement, a careful inspection of the animation for the new version shows that it still does not behave as one might expect. The relabeled tree is observed in depth-first order, whereas the input tree is observed in breadth-first order. At first, that looks reasonable: the problem specification calls for a breadth-first traversal of the input tree, and the printing routine traverses the result in depth-first order. On second thought, though, *only the computation of the new labels should depend on a breadth-first traversal of the input*, and printing the result should give the whole leftmost branch of the tree before inspecting any node labels.

At this point, we need to explain our approach to the problem and the differences between the versions. In our earliest attempts, we did indeed experience the blind spot discussed by Okasaki, though not for the reasons listed by him. Instead, our road-block was that any solution seems to involve two different views of the input trees: whereas the problem specification clearly calls for a breadth-first traversal, the easy way to describe a recursive algorithm over the trees follows their recursive structure – in depth-first order! Our very first solution side-stepped the issue in an overcautiously systematic approach, restructuring the input tree into a list of levels, then doing the relabeling (straightforward in this form), and finally rebuilding a tree of the original structure, with the new labels. But once we had managed to find at least one solution to Okasaki's problem, and identified our own blind spot on the way, we then sought to get rid of the blind spot by constructing a more suitable solution. This led to the variants described in the present paper (the original brute-force solution had similar strictness problems).

The new approach does not impose a breadth-first traversal on the input tree, but instead follows its natural recursive structure, generating a pool of "things to do" on the way. The tasks -one for each subtree- are connected by data-dependencies which represent the breadth-first traversal constraint, and it is left to the inspection of the result tree to actually cause those tasks to be evaluated, in a co-routine-like fashion. In other words, the producer of relabeled trees consumes the input trees in a depth-first traversal, and any consumer of the result tree will implicitly (by the virtues of lazy evaluation and the data dependencies set up by the producer) cause a breadth-first traversal to take place. This decoupling of the two conflicting traversals solves our blind-spot problem and gives a concise first variant of a solution, called `task1` (figures A.2, A.6, 4).

After reading Okasaki's comments [14], we noticed that his suggestion about replacing two-way queues by unidirectional queues in languages that do not support matching on both ends applied to our task pool (represented as a list, with an awkward use of `splitAt` to pattern-match at its back end). So `task1` became `task2` (figures A.3, A.5, 3) – and acquired the extreme strictness problem described earlier: Okasaki's workaround maintains queues

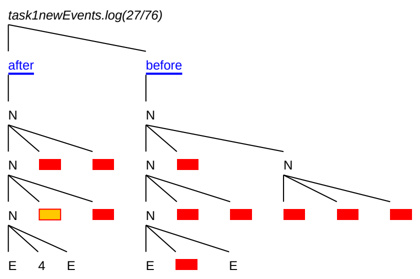


Fig. 5. Strictness problem solved!

in reversed order (so that elements can be taken from the output ends using pattern-matching), which happens to put the relabeled top node at the very end of the queue, so that the whole task queue has to be processed -and the whole input tree be observed- to get to the very first node of the result tree.

Switching back to our original variant got rid of this problem, but left another, only slightly more subtle strictness problem: to show the result tree up to the first label, as in figure 4, it should not be necessary to observe three levels of nodes in the input tree. The node labeled `4` in the result is the first at level three, so observing two levels of the input tree should suffice to compute the label! Perusing the animation again gives the embarrassing insight: just traversing the structure of the result tree seems to trigger the breadth-first traversal of the input tree, even before any labels are inspected. And indeed, this variant takes the result structure from the task pool that was set up to enforce the breadth-first traversal. Separately passing the structure of the input tree and filling in the labels computed on demand solves this problem, and the animation of our final variant, `task1new` (figures A.4, A.7, 5), exhibits a nice, demand-driven pattern of observations.

Note that this kind of dynamic strictness problem, where parts of inputs are demanded too early, differs from the kind of problems that could be investigated using static strictness information (is a part of input ever demanded or not at all?). If the iteration bounds that guarantee termination of a strictness inference system can be increased in cases where termination is obvious for other reasons, the best information such a system could give corresponds to that deducible from figure 2. But that information is the same for all variants of the solution!

4.2 A well-known strictness problem

Recursive algorithms over lists can often be expressed more concisely as folds, avoiding explicitly recursive definitions. For lists, there are two standard fold operators, `foldr` and `foldl`, which combine the list elements by right- and left-associative operators, respectively. More generally, a fold operator replaces constructors in a parameter structure by operators of appropriate arity, thus

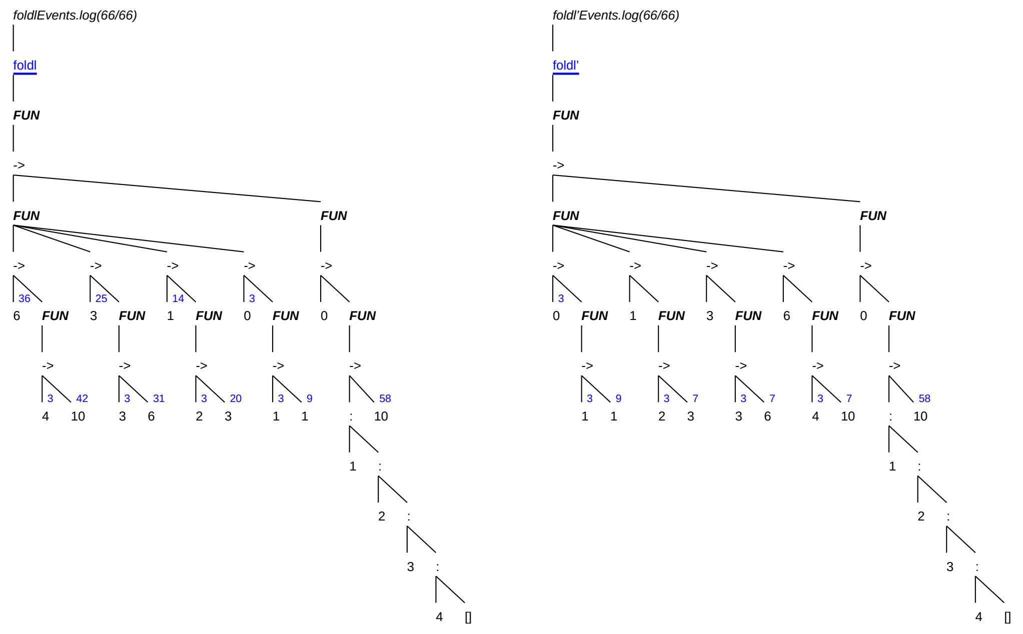


Fig. 6. `foldl` versus `foldl'` – tail recursion with (non-)strict accumulator

expressing the recursive structure of the algorithm in terms of the recursive structure of its input. Viewed in these more general terms, `foldr` expresses a standard recursion along the list structure, whereas `foldl` expresses a tail recursion with an accumulator. Such tail recursions are usually associated with constant stack-usage.

```
foldr op c []     = c
foldr op c (x:xs) = x `op` (foldr op c xs)

foldl op c []     = c
foldl op c (x:xs) = foldl op (op c x) xs
```

As many Haskell programmers discover for the first time in more complex programs, this idea does not quite work – for large inputs their programs can run out of stack space in spite of the careful use of tail recursion! This is quite a common experience, and so it seems worthwhile to see how much of the problem reveals itself by careful analysis of an example, using only the graphical animation of observations. The reader should keep in mind that this subsection is not concerned with new aspects of folds – rather, it serves to illustrate the novel ways of explaining more or less well-known properties of functional algorithms, made possible by visualisation tools such as GHood.

Figure 6 (left) shows an end-of-animation snapshot of the call:

```
observe "foldl" foldl (+) 0 [1..4::Int]
```

To make up for the lack of animation here, nodes in this figure are annotated with superscripts giving the number of observation events between the beginning of their observation and the availability of their weak head normal form (shown only if that number exceeds one). As in Hood, the observed part of a function is presented as a finite map of input/output pairs. Those pairs are labeled with arrows here, so `FUN{6->FUN{4->10},3->FUN{3->6},..}` represents a function f that, when applied to `6`, returned a function that, when applied to `4`, returned `10` (f was also applied to `3`, and returned a function that, when applied to `3`, returned `6`). The overall picture tells us that `foldl` is a ternary function, mapping a binary function (itself applied four times, as there are four pairs in its map) to a function, that maps the integer `0` to a function, that maps the list `[1,2,3,4]` to the integer `10`.

In the animation, several phases can be distinguished. First, `foldl` itself is observed to reveal its arity, then evaluation demands that its result be observed (the box corresponding to this thunk is opened). Before this becomes available, the spine of the input list is observed in full, which in itself is a stumbling block in many programs operating on lists of substantial size: the whole length of the input list is created in memory before any other computations take place (the spine of the list can be collected immediately, but the thunks for its elements take up space, even though these elements are not yet about to be inspected). Using `foldr` would avoid this problem, at the expense of linear stack usage.

Next, observation of the result of applying the binary operator is demanded, leading to a demand for the first parameter of this application. This, in turn, demands observation of the result of another application of the operator, and so on, creating a chain of thunks under observation until the demand for the first parameter of the fourth application is fulfilled by observing the second parameter to `foldl`. After that point, the chain unwinds step by step, demanding successive observations of all input list elements before, finally, the result of the call to `foldl` becomes observable.

Returning to the annotated snapshot in figure 6 (left), we see that some 58 events passed during observation of the final result, `10`, and that the chain consisted of computing, starting in this sequence `6+4->10`, `3+3->6`, `1+2->3`, and `0+1->1`, and terminating in reversed order, taking 42, 31, 20, and 9 observed steps, respectively. In summary, the call to `foldl` was indeed tail recursive, but it only observed the spine of the input list and delivered a thunk involving the list elements as an interim result. Evaluating this thunk then unfolded another, implicit recursion (corresponding to the evaluation of a nested arithmetical expression) with just the kind of linearly growing stack-usage (the chain of opened boxes) we wanted to avoid.

The obvious countermeasure is to force evaluation of the accumulator to avoid this split into a tail-recursive thunk construction and a not tail-recursive evaluation of that thunk, e.g., by using the call-by-value applicator `$!`:

```
foldl' op c []     = c
foldl' op c (x:xs) = (foldl' op $! (op c x)) xs
```

The new annotated end-of-animation snapshot in figure 6 (right) already indicates a major change. With the exception of the final result, no more than 9 observation events occur between the beginning of a node observation and the availability of its weak head normal form. As those delays roughly correspond to stack usage, getting rid of the ghost-recursion has established the bound on stack usage that was the original goal. The order of applications of the binary operator seems to have changed as well.

Going through the full animation sequence shows further differences: the spine and elements of the input list are now inspected in a stepwise fashion, interleaved with applications of the binary operator, now in the sequence `0+1->1`, `1+2->3`, `3+3->6`, and `6+4->10`. This ordering ensures that intermediate results are already available when demanded by the next application and is the result of forcing the evaluation of the accumulator. So, not only has the unbounded use of stack space been avoided, but a space leak (observing the full spine of the input list -thus creating implicit thunks for all elements- long before its elements are inspected) has been plugged as well.

4.3 Summary, and further examples

The examples in this section have been chosen to be small, relatively well-known, yet displaying interesting behaviour and illustrating different aspects of GHood. In the case of breadth-first numbering, animation of observations was used during algorithm development and helped to discover unexpected properties of early program variants, as well as pointing to the source of the problems. In the case of `foldl`, the algorithm and problems are usually considered to be well-known, but resurface with surprising reliability, and the animation was used to demonstrate and explain how a tail-recursive function could still lead to linear resource usage for intermediate structures. The examples differ in another notable aspect: for breadth-first numbering, the tree layout imposed by GHood naturally matches the trees in the problem, whereas the tree layout is rather less natural for `foldl`.

In both examples, observation of unexpected behaviour could be traced back to problems and led to modifications of the programs observed. It would be misleading, though, to assume that the main use of GHood was in debugging – it just happens that understanding what a program does can be a useful asset in debugging (declarative debugging, as in Freja [13], suggests that such an understanding is not always necessary). For a nice example of how animation of observations can aid program comprehension outside of debugging, readers are again referred to the GHood home page: the online examples include an animated observation of Colin Runciman's Haskell implementation of the "wheel sieve" algorithm for generating prime numbers [17]. The program is considerably more complex than the examples discussed here, and the animation provides a nice complement to the discussion in the JFP paper.

5 Evaluation, related and further work

5.1 *Experience, feedback, and evaluation*

After some internal testing at UKC, first versions of GHood were made available to the Haskell community in January 2001. Since then, we have received a lot of positive feedback, very few feature requests, and problem reports have mostly been limited to problems with the Java 2 runtime installations on which our viewer depends. This suggests that the tool, while far from perfect, is already considered good enough to fill its niche. In other words, while our current users might welcome refinements of the current features, such improvements will not be considered essential unless they reflect changes in the basic approach. Our plans for GHood are thus limited to completion of the modifications currently under development (see below), to be incorporated in a final release later this year.

In March, we also had the opportunity to visit[6] the functional programming group in York and take part in a repetition of the usability study described in [3], with updated variants of the same tools. Though limited to case studies in debugging, the experiment provided a host of useful feedback and ideas. The most important outcome was that the tools (Freja, Hat, and GHood) had actually managed to explore, and partially fill, *different* niches in the area of debugging Haskell programs. Each tool was useful for debugging, but each tool was useful in a different way, and more than once, we would have wanted an easy way to switch from one tool to another – not only with the same Haskell implementation, but in the same debugging session, taking the current debugging state and investigating it from a different perspective. As the Hat trace seems to contain most of the information needed for each of the tools, the York group has now started to move in that direction, and first results are visible in the new Hat toolsuite bundled with the just-released nhc98-1.04 [20,19] (the suite includes a variant of Hood-style observation, implemented on top of Hat's redex trails instead of Hood's observation library).

In the following, we distinguish between Hood -the Haskell library released in July 2000, GHood -the graphical back-end for Hood described in this paper, and THood, by which we refer to the version of Hood that comes bundled with nhc. The latter includes Gill's textual browser from the Haskell CVS repository, and a pre-release version of the Hood library, modified to generate the XML input expected by the browser. In its current pre-release form, THood suffers from differences to the released Hood (this is easily repaired) and from a lack of automated animation (only single-stepping forwards and backwards and jumps to beginning and end of observations are provided).

All Hood backends inherit the core functionality and some limitations from the library. In practice, the most annoying limitation is the need to inspect and modify the source code in order to import the module `Observe` and to

[6] This visit was supported by EPSRC grant number GR/M81953.

define instances of the class `Observable` for all non-standard data types, as far as values of these types need to be observed (this set of types needs to be closed with respect to embedded types). Further modifications include a call to `runO` in `main` and running the implementation with options indicating extensions beyond Haskell 98. In contrast to calls to `observe`, which indicate programmer intentions, these modifications are implied, boring, and error-prone. Even though errors introduced in the process are isolated from the program, easily spotted and fixed, they could be avoided entirely by automating these tasks (Malcolm Wallace suggested using Drift to generate the instances of `Observable`). The main problem with calls to `observe` is to identify program positions where such calls will provide useful information.

The York experiment was limited to debugging, and as far this is concerned, the most useful feature of GHood surprisingly turned out to be information about what is not there: again and again, unevaluated thunks provided shortcuts to spotting bugs (one example was a bugged compiler in which a symboltable lookup managed to return values without the symboltable ever being observed). Both Hood and THood indicate unevaluated thunks as simple underscores, and neither shows temporal relations between different observations (Hood has no animation, THood treats observations under different labels separately). GHood, in contrast, displays unevaluated thunks in clearly visible red, and animates all observations under a single root node, facilitating comprehension of interrelationships. Deriving information from non-available data (thunks) seems to take some getting-used-to, though: the important connection is that Hood-based tools show what the program sees, so if GHood does not show the value of a thunk, there is no need for the debugger to know the value, simply because the program never asks for that value.

Of the tools in the experiment, GHood seemed to cope best with large structures, but it was not entirely without problems in this regard: scaling (both in time and in space) is useful because the graphical structure supports orientation even when textual labels are no longer readable, but because of this graphical structure, small structures are not represented as compactly as in Hood or THood. If THood would be extended with automated animation, it would be at an advantage for small, not inherently tree-like structures, such as the observation of `foldl`. For slightly larger observations, such as the lazy wheel sieve, THood's compact representation can no longer entirely make up for the lack of scaling (scaling the pretty-printed representation to point size would give a graphic represention without much structure, but it would be interesting to compare that representation to GHood's).

GHood extends Hood, so the static pretty-printed observations are still available to complement the dynamic graphic visualisation, but some graphs, especially `String`s, should be represented more compactly, to improve readability. Another problem concerns navigation in large structures: the standard two-scrollbars solution is rather unsuited for concurrently navigating in both dimensions and needs to be replaced, and although both survey views and zooming to details are currently supported, they should not exclude each other. On a related note, we should point out that Hood-based animation tools not only enable programmers to focus on the parts of the program to be observed, decoupling program size from the size of observations. To some extent, the level of abstraction at which to animate program observations can also be controlled: at the level described in section 3, entirely different approaches to the breadth-first numbering problem, such as the brute-force level-and-reconstruct approach, will display similar behaviour, even though their behaviour would differ substantially under more detailed observations.

Other issues include online versus offline generation of observation logs (cf. section 3.1), observability of η-conversion (`observe "f" f` shares a single observation label between all uses of f, whereas `\x->observe "f" f x` creates separate observation labels for each call), the need to remove calls to `observe` to avoid clutter (GHood should be extended to permit selective observation), and the need for "packaging" of observations, preserving the connection between them (for instance, several local variable bindings in a function body).

As mentioned earlier, the approach taken by Hood and GHood does not in principle exclude interactive debugging, and the February 2001 release of Hugs (`www.haskell.org/hugs`) offers support for a built-in variant of Hood, called HugsHood, which heads in this direction by supporting breakpoints. Similarly, there is no fundamental reason against online visualisation (during program execution) but our current offline approach to visualisation has opened new application areas beyond debugging.

5.2 Other related work

The idea to visualise and animate the execution of functional programs in order to gain insights into their behaviour is an old one. For an overview of the problems and opportunities see Sandra Foubister's thesis [5]. We are not aware of a survey covering this area, but various proposals and even implementations have been put forward, including Foubister's "hint" tool and an animation of a G-machine implementation using the graph layout tool daVinci [15], not to mention proposals for specially designed visual functional languages. More recent incarnations of the idea include a graphical debugger/tracer in the Curry Integrated Development EnviRonment CIDER [11], and the Kiel Interactive Evaluation Laboratory [2] for a simple first-order subset of ML. For completeness, text-based navigation through reduction sequences should also be mentioned, as in the DrScheme environment [4] or in the reduction systems in the Berkling and Kluge tradition [10].

Animation of observations in GHood is distinctly different from traditional text- or graphics-based animation or navigation of reduction sequences. Comparing our experience with GHood and with textual single-stepping through reduction sequences, as afforded, e.g., by the reduction systems developed by Kluge et. al. [10,6], we find both disadvantages and advantages.

At first, the disadvantages seem overwhelming: without any extra effort by programmers, reduction systems provide a direct experience of the operational semantics, as well as navigation, editing, and selective reduction of parts of intermediate programs in a reduction sequence. GHood, as a back-end for Hood, only animates observations of intermediate structures. Observations are approximations of weak head normal forms of those intermediates, and the animation shows the sequence in which parts of structures under observation are inspected. This allows only indirect conclusions about the program behaviour. In practice, it can be rather difficult to try and infer the algorithm from the visualisation alone but, starting with a conjecture or some approximate understanding of the program behaviour, it tends to be straightforward to confirm or refute such hypotheses in the visualisation.

On the positive side, graphical visualisation is more suitable for overviews of larger programs and of animation sequences, where textual information is no longer readable. The observational approach also makes it easier to focus visualisation on interesting aspects of program behaviour, excluding both unobserved parts of programs and intermediate expression representations on the way to weak head normal forms. Nevertheless, observation graphs for realistic programs grow quickly, demanding further work on the user interface.

The general problem faced by developers of execution monitoring tools is the need to use (and most likely create) specially instrumented implementations. As a consequence of the efforts involved, such specialised implementations tend to support only small subsets of the original languages, visualisation often takes place at the implementation level, and the specialised implementations do not evolve with the language and its standard implementations. Tools based on specialised implementations are by definition not portable, and if separate implementations are needed for normal and for visualisation use, differences in evaluation mechanisms may occur.

Another alternative is to use a separate evaluator with built-in execution animation facilities and to provide mappings between subsets of that evaluator's language and subsets of the language to be extended with execution monitoring. Wolfram Kahl has demonstrated this approach with his term-graph-based program development and transformation environment HOPS [9], but it means that two evaluators, their languages, and the mapping between them have to be kept in synch, not to mention portability issues.

Hood avoids all these problems by using a commonly implemented implementation hook (`unsafePerformIO`) to instrument existing Haskell implementations, reusing and extending their functionality. The resulting library is portable and can be used with the full Haskell language. GHood uses hooks in Hood to reuse the observation functionality while extending it for purposes of dynamic graphical visualisation, using Java as a widely available implementation platform. Reflecting on the success of these hook-based solutions, implementation hooks turn out to be (application-specific) residues of more general meta-programming infra-structure.

In language communities with successful tool-building traditions, such as Lisp, Prolog, and Smalltalk, tool development seems to rely on well-developed infra-structures for meta-programming and reflection. At a prototype stage, the key idea is to write a meta-interpreter (between a few lines and a page of code for these languages) that reuses existing implementation functionality, and then to instrument the meta-interpreter for purposes of monitoring (animation in our case). Successful prototype tools can then be implemented more efficiently, often using standard techniques. To achieve efficiency, the meta-interpreter should delegate standard functionality to the standard evaluator with as little overhead as possible. In such embedded meta-interpreters, only the extra functionality (e.g., for program monitoring) incurs interpretative overhead, and if suitable extension interfaces to the standard evaluator are available (aka reflection or introspection capabilities), the meta-interpreter *becomes* the standard interpreter, instrumented via its extension hooks.

In the context of declarative debugging, Naish and Barbour [12] have used this idea to design a "portable lazy functional declarative debugger" which could be implemented in the functional language to be debugged, assuming a single impure primitive, called `dirt` (display intermediate reduced term).

Haskell neither supports reflection[7] nor does it offer well-documented interfaces to implementation functionality (cf. the SML/NJ `Compiler` structure [1]), or other typical parts of a meta-programming infra-structure. Its syntax is more complex than Lisp's S-expressions, and reusable parsers for full Haskell have only recently started to appear, but the parsers in the various Haskell implementations remain practically unaccessible; all Haskell implementations internally build up a symbol-table, associating identifiers with attributes, such as types or strictness, but there is no standard interface by which Haskell programs could load a Haskell program and query the symbol-table information.

6 Conclusions

GHood is a new back-end for Hood, providing graphical visualisation and animation of Haskell program execution. Unlike traditional approaches to graph reduction animation, GHood is not based on a special-purpose implementation, but extends and reuses existing Haskell implementations, via Hood. The visualisation itself is also different, in that it does not animate reductions of terms to normal form, but inspection of terms by their evaluation contexts: instead of evolution of a term through intermediate representations, an animation shows refinement of information about a term in a single representation.

Portable tools such as Hood and GHood depend critically on being able to instrument and thus reuse existing Haskell implementations by means of extension hooks, and the ease with which tool implementers can reuse existing

[7] How to do this properly in a statically typed, pure, and non-strict functional language is another research direction that would merit more attention

implementation functionality has an important impact on the development of tools for Haskell. We suggest that a common (implementation-independent) infra-structure for meta-programming and reflection in Haskell, with standard interfaces to implementation functionality, could improve the basis for Haskell tool development, and that both the general framework and specific implementation extension hooks should become a focus of research.

In the present paper, we have focussed on illustrating the way in which GHood can be used to help comprehension of Haskell program behaviour, using small examples from everyday practice. Our own experience and feedback from users shows that dynamic observation of intermediate structures is a useful addition to the Haskell programmer's toolbox. Although the 'd' in Hood stands for "debugger", we prefer to see GHood as a workbench: Haskell programmers can use it to set up and perform experiments involving dynamic aspects of their programs. Such experiments can be used to validate theories of program behaviour or they can deliver the data points from which such theories can be abstracted. For both uses, experiments have to be set up and the data be interpreted carefully, so Hood and GHood are tools that can inform thinking about programs, but they cannot replace such thinking.

We hope to see GHood or similar tools for the visualisation of functional program behaviour used in education (online course material), documentation, and publication (online supplements to articles on functional algorithms). Instructors might want to consider the motivational aspect as well – several correspondents commented the first pre-releases with the words "GHood is cool!". Another correspondent remarked "finally, I can *show* my colleagues what non-strict evaluation means".

References

[1] *Standard ML of New Jersey*, `http://www.smlnj.org`.

[2] Berghammer, R. and M. Tiedt, *Kiel Interactive Evaluation Laboratory*, Technical report, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel (1999), `http://www.informatik.uni-kiel.de/~progsys/kiel.html`.

[3] Chitil, O., C. Runciman and M. Wallace, *Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs*, in: *Proceedings of the 12th International Workshop on Implementation of Functional Languages, Aachen, Germany, September 4th - 7th 2000, LNCS 2011*, 2001, pp. 176–193.

[4] Clements, J., M. Flatt and M. Felleisen, *Modeling an Algebraic Stepper*, in: *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001*, Lecture Notes in Computer Science **2028** (2001).

[5] Foubister, S., "Graphical Application and Visualisation of Lazy Functional Computation ," Ph.D. thesis, Department of Computer Science, University of York (1995).

[6] Gärtner, D. and W. Kluge, *π-RED$^+$: An interactive compiling graph reduction system for an applied λ-calculus*, Journal of Functional Programming **6** (1996).

[7] Gill, A., *Debugging Haskell by Observing Intermediate Data Structures*, in: G. Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop* (2000).

[8] Gill, A., *The Haskell Object Observation Debugger*, `http://www.haskell.org/hood/` (2000).

[9] Kahl, W., *HOPS – The Higher Object Programming System*, `http://ist.unibw-muenchen.de/kahl/HOPS/`.

[10] Kluge, W. E., *A User's Guide for the Reduction System π-RED$^+$*, Technical Report 9419, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel (1994), `http://www.informatik.uni-kiel.de/~base/`.

[11] Koj, J., "Eine graphische Programmierumgebung für deklarative Programmiersprachen," Master's thesis, RWTH Aachen, Germany (2000), the Curry Integrated Development EnviRonment `http://www.informatik.uni-kiel.de/~pakcs/cider/`.

[12] Naish, L. and T. Barbour, *Towards a portable lazy functional declarative debugger*, Australian Comp. Science Communications **18** (1996), pp. 401–408.

[13] Nilsson, H., "Declarative Debugging for Lazy Functional Languages," Ph.D. thesis, Department of Computer and Information Science, Linköpings Universitet, S-581 83, Linköping, Sweden (1998).

[14] Okasaki, C., *Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design* , in: *International Conference on Functional Programming ICFP'2000, Montreal, Canada*, 2000, pp. 131–136.

[15] Panne, S., *Graph Visualization with daVinci*, `http://www.pms.informatik.uni-muenchen.de/mitarbeiter/panne/haskell_libs/daVinci.html`.

[16] Reinke, C., "Functions, Frames, and Interactions – completing a λ-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments," Ph.D. thesis, Faculty of Engineering, Christian-Albrechts-University, Kiel (1997), Technical Report 9804, Institute of Computer Science, May 1998, `http://www.cs.ukc.ac.uk/people/staff/cr3/publications/phd.html` .

[17] Runciman, C., *Lazy wheel sieves and spirals of primes*, Journal of Functional Programming **7** (1997), pp. 219–225.

[18] Wadler, P., *Functional Programming: Why no one uses functional languages*, SIGPLAN Notices **33** (1998), pp. 23–27.

[19] Wallace, M., O. Chitil, T. Brehm and C. Runciman, *Multiple-View Tracing for Haskell: a New Hat*, in: *ACM SIGPLAN Haskell Workshop, Firenze, Italy*, 2001.

[20] York Functional Programming Group, *Hat - The Haskell Tracer*, `http://www.cs.york.ac.uk/fp/hat/` (2001).

A Source code and animation sequences

A note on the use of animation sequences: online animations for all examples are available on the GHood home page. Snapshot samples of animation sequences are included in this appendix for archival reasons, but as the static medium cannot portray the advantages of dynamic visualisation, the online animations should be preferred, if at all possible. Readers without access to the online animations will find it helpful to print or display this appendix separately from the main text, so that they can see both side by side without having to jump back and forth.

```
import Observe

data Tree a = E | N (Tree a) a (Tree a) deriving (Show)

instance Observable a => Observable (Tree a) where
  observer E         = send "E" (return E)
  observer (N l x r) = send "N" (return N << l << x << r)

main = printO $ observe "after" $ bfnum  $ observe "before" xxx
  where { xxx = N xx 2 xx; xx = N x 1 x; x = N E 0 E }
```

Fig. A.1. task-based breadth-first numbering, common prefix

```
-- for non-empty tree, fork out immediate subtrees (l,r) as
-- new tasks, build result from sub-results (l',r')
task n ~[]      E         = (n  ,[]   ,E)
task n ~[l',r'] (N l x r) = (n+1,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = t':rs'
  where
    (n',tp',t') = task n r t
    ts'         = taskM n' (ts++tp')
    (rs',r)     = splitAt (length ts) ts'

bfnum t = head $ taskM (1::Integer) [t]
```

Fig. A.2. `task1` – task-based breadth-first numbering, first attempt

```
task n ~rs         E         = (n  ,rs,[]   ,E)
task n ~(r':l':rs) (N l x r) = (n+1,rs,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = rs'++[t']
  where
    (n',rs',tp',t') = task n ts' t
    ts'             = taskM n' (ts++tp')

bfnum t = head $ taskM (1::Integer) [t]
```

Fig. A.3. `task2` – task-based breadth-first numbering, more elegant?

```
task n ~[]      E         = (n  ,[]   ,E)
task n ~[l',r'] (N l x r) = (n+1,[l,r],N l' n r')

taskM n []     = []
taskM n (t:ts) = t':rs'
  where
    (n',tp',t') = task n r t
    ts'         = taskM n' (ts++tp')
    (rs',r)     = splitAt (length ts) ts'

fillIn E         ~E            = E
fillIn (N l _ r) ~(N l' x' r') = N (fillIn l l') x' (fillIn r r')

bfnum t = fillIn t $ head $ taskM (1::Integer) [t]
```

Fig. A.4. `task1new` – task-based breadth-first numbering, improved!

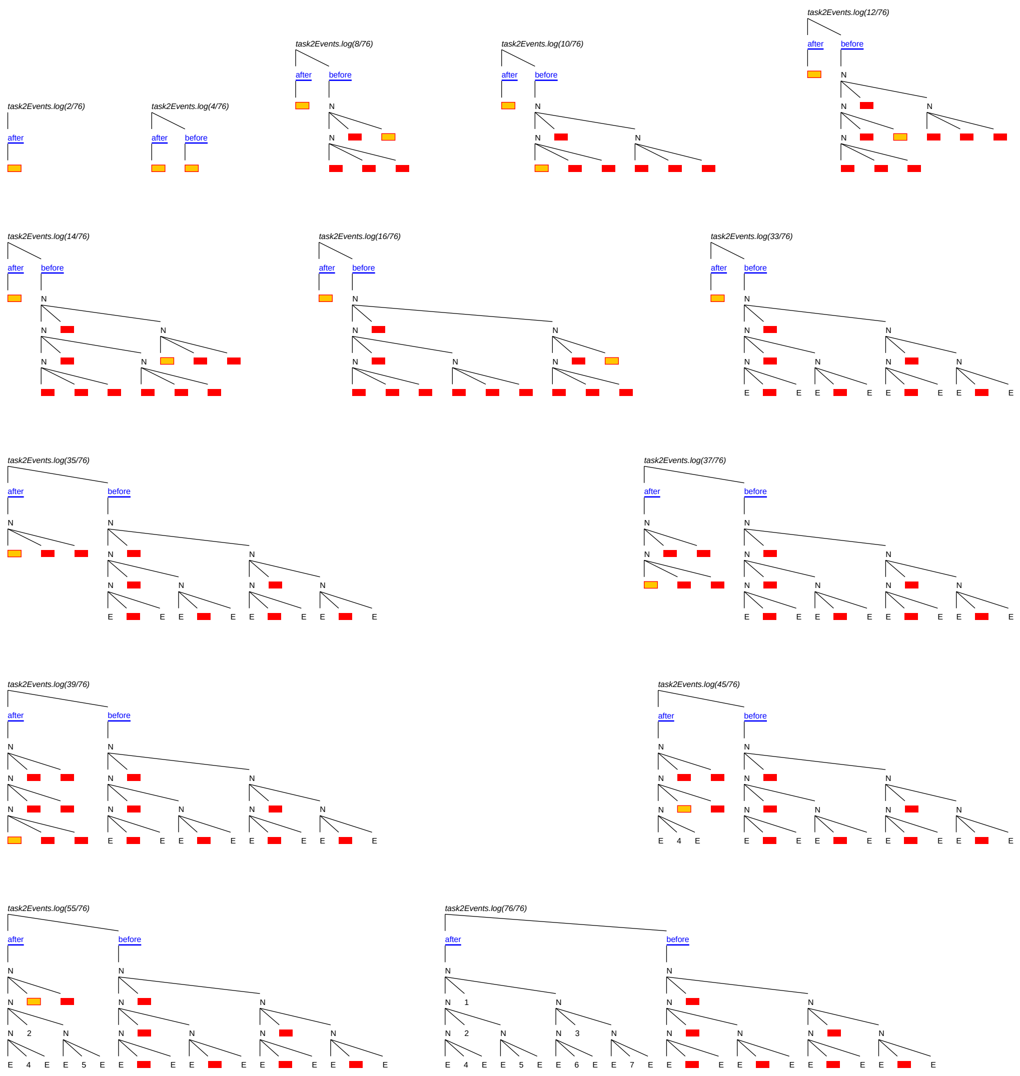


Fig. A.5. `task2`: steps 2, 4, 8, 10, 12, 14, 16, 33, 35, 37, 39, 45, 55, and 76

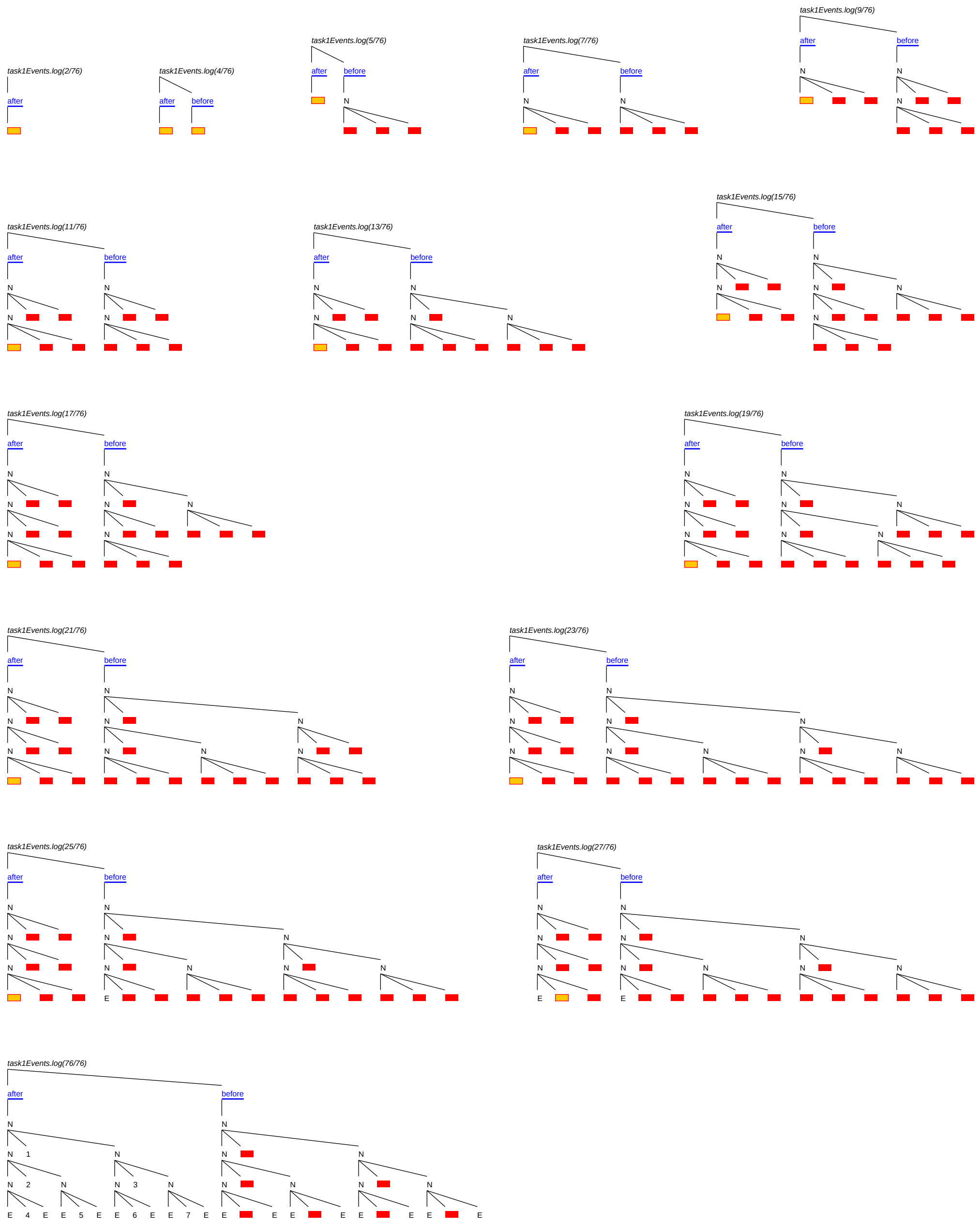


Fig. A.6. `task1`: steps 2, 4, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, and 76

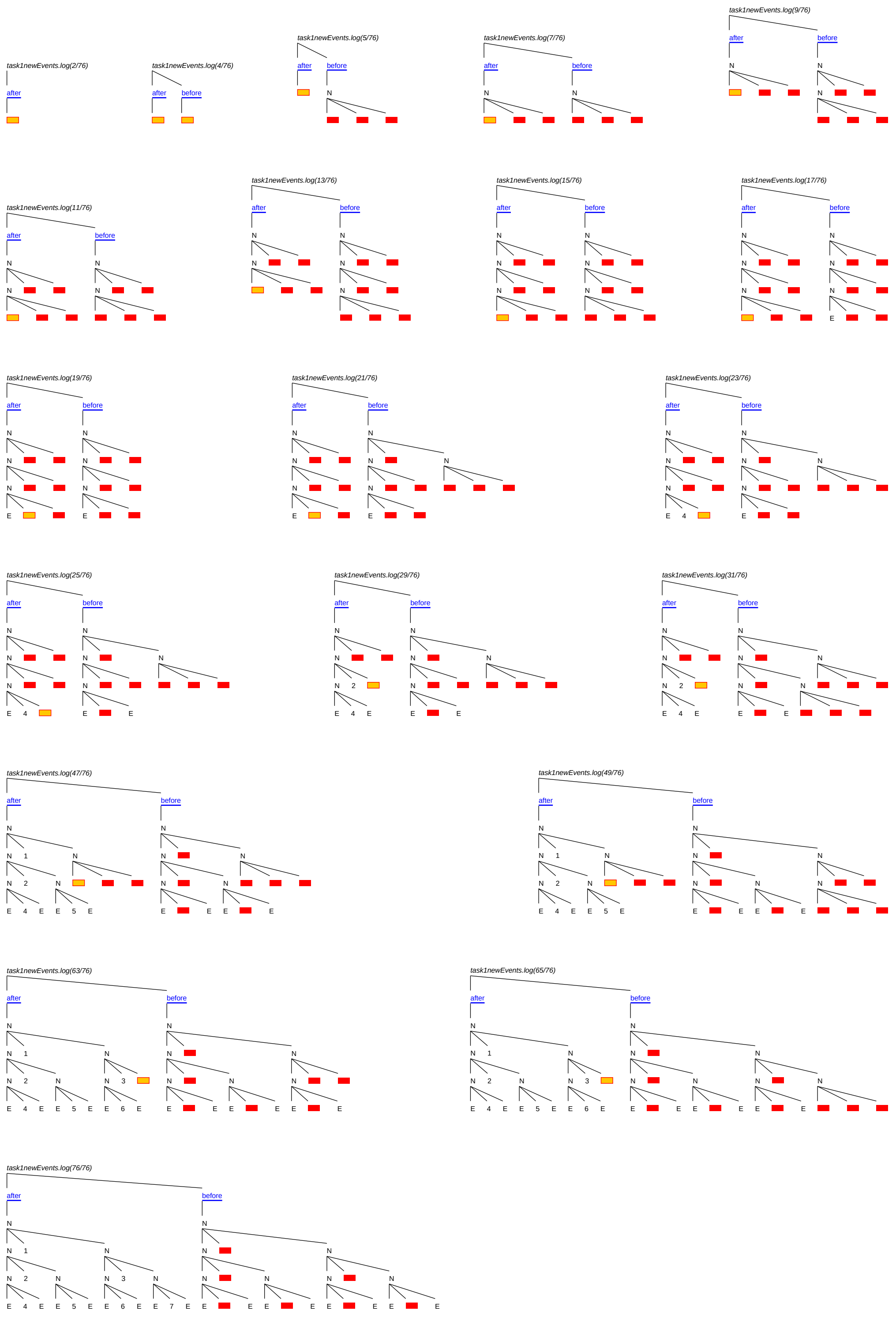


Fig. A.7. `task1new`: steps 2, 4, 5, 7, 9, 11, 13, 15, 17, 19, 21, . . . and 76

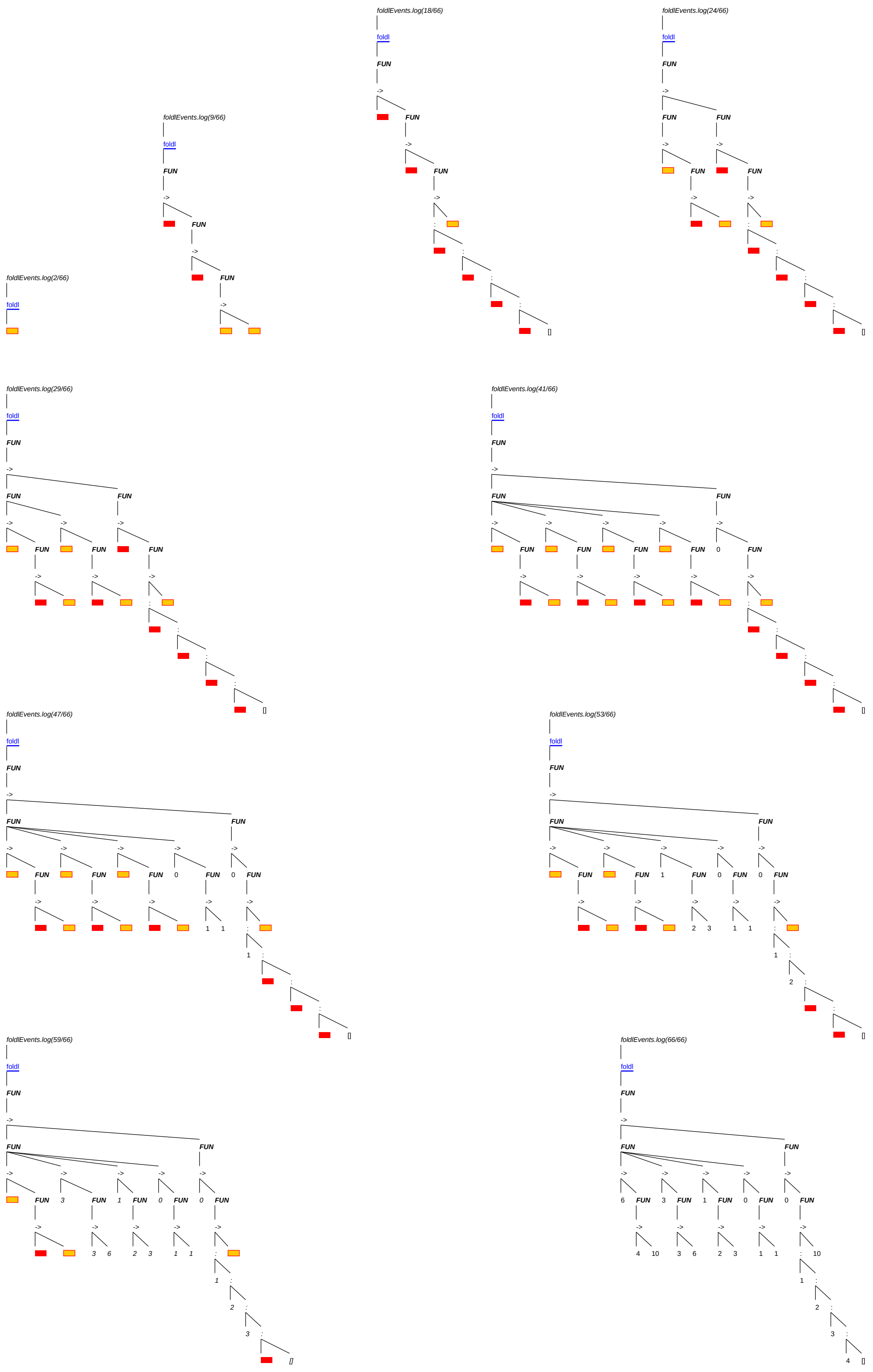


Fig. A.8. `foldl` steps 2, 9, 18, 24, 29, 41, 47, 53, 59 and 66

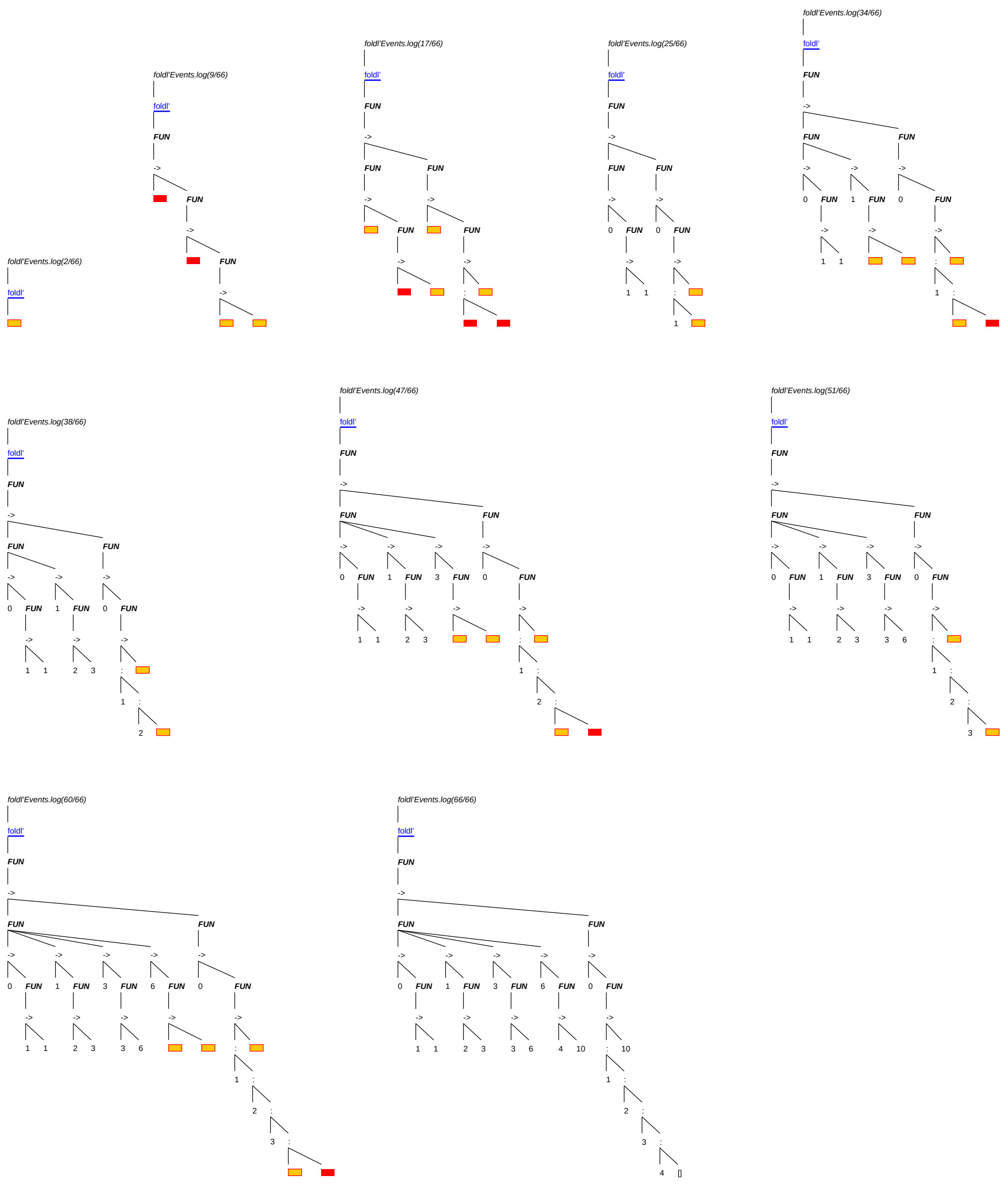


Fig. A.9. `foldl'`: steps 2, 9, 17, 25, 34, 38, 47, 51, 60 and 66

***

Multiple-View Tracing for Haskell: a New Hat

Malcolm Wallace, Olaf Chitil, Thorsten Brehm and
Colin Runciman

University of York, UK
{malcolm,olaf,thorsten,colin}@cs.york.ac.uk

Abstract

Different tracing systems for Haskell give different views of a program at work. In practice, several views are complementary and can productively be used together. Until now each system has generated its own trace, containing only the information needed for its particular view. Here we present the design of a trace that can serve several views. The trace is generated and written to file as the computation proceeds. We have implemented both the generation of the trace and several different viewers.

1 Introduction

Usually, a computation is treated as a black box that performs input and output actions, but whose internal workings are invisible. As programmers, however, we may want to look into the black box to understand how the different parts of the program cause the computation to perform the observed actions. Often the computation does not perform the intended actions and we have to determine which parts of the program cause the erroneous behaviour. Even if a program is correct, we may desire to understand its parts better by seeing "how it works"; especially when we have to modify a program that we did not write ourselves. Also for teaching it is sometimes useful to "see" a computation.

In [2] we compared the Haskell tracing systems Freja [1] [5,6] and HOOD [2] [3] with our Haskell tracer Hat [3] [9,10,11]. The main conclusion of our comparison was that each system gives a unique view of a computation and these views are usefully complementary. In experiments, we discovered that after using one system to help track a bug to a certain point, users often wanted to change to another system to continue the search, or to confirm their suspicions.

[1] http://www.ida.liu.se/~henni
[2] http://www.haskell.org/hood
[3] http://www.cs.york.ac.uk/fp/hat

All three tracing systems take a two-phase approach to tracing: during the computation information describing the computation is collected in a data structure, the trace. After termination of the computation the trace is viewed. An advantage of a trace as a concrete data structure is that it liberates the viewer from the time arrow of the computation. However, each system creates its own trace, containing only the information required for its particular view. We noted in [2] that Hat's trace, called a Redex Trail, contains nearly all the information contained in Freja's trace. Hence we decided to extend the Redex Trail structure to the Augmented Redex Trail structure (ART). With separate tools we can view an ART trace in at least three different ways: à la Freja, à la Hat and à la HOOD. Whereas Freja, HOOD and the old Hat system generated their traces in main memory, the new Hat writes the ART trace to file as computation proceeds. Hat's new architecture has the following advantages:

- As a stand-alone description of a computation, the ART trace serves as interface between trace generation and trace viewing. The two phases become completely separate.
- A trace in file supports sequential access and forms of indexed search that were not feasible for heap-based traces.
- The ART trace clarifies the relationships between the different views of a computation. It suggests ways for integrating different views and creating new views.
- The size of the trace is no longer bound by the size of the main memory but only by the far larger size of the file store.
- The trace is no longer transient but can be archived for later viewing.
- Trace system developers only need to implement trace generation once for several views.
- The user only pays the cost of generating a trace once for several views.

In Section 2 we review the Redex Trail of the old Hat system, while Section 3 briefly illustrates some alternative tracing views. In Section 4 we develop in several steps the new Augmented Redex Trail structure. In Section 5 we describe how several tools for different views obtain their information from the ART trace. In Section 6 we outline the generation of the ART trace. In Section 7 we discuss ideas for future work. Section 8 concludes.

We have modified Hat to produce the ART trace and have implemented new tools for viewing the trace in the style of Freja and HOOD. The system has been publicly released as Hat 1.04.

2 The Redex Trail Model

Let us view a computation abstractly as a series of rewrite steps. Starting from a single expression (main), at each step a reducible expression (redex) is replaced by another expression (its reduct), by instantiating the LHS of an equation, and replacing it with the corresponding instance of the RHS. Eventually, only irreducible expressions (values) remain.

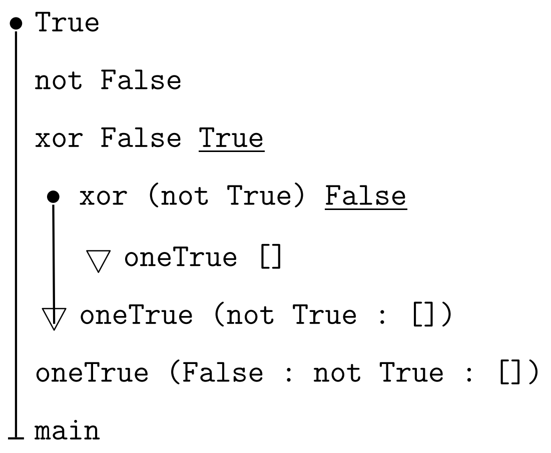
The original Redex Trail structure is a directed graph, recording copies of all values and redexes, with a backward link from each reduct (and each proper subexpression contained within it) to the parent redex that created it.

2.1 Example

$$\begin{array}{ll}
\mathit{oneTrue} & :: [\mathit{Bool}] \rightarrow \mathit{Bool} \\
\mathit{oneTrue}\ [] & = \mathit{False} \\
\mathit{oneTrue}\ (x : \mathit{xs}) & = \mathit{xor}\ x\ (\mathit{oneTrue}\ \mathit{xs}) \\
 & \\
\mathit{xor} & :: \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool} \\
\mathit{xor}\ x\ \mathit{True} & = \mathit{not}\ x \\
\mathit{xor}\ \_\ \mathit{False} & = \mathit{True} \\
 & \\
\mathit{main} & = \mathit{print}\ (\mathit{oneTrue}\ [\mathit{False}, \mathit{not}\ \mathit{True}])
\end{array}$$

Fig. 1. Example program

The small example program shown in Figure 1 produces the Redex Trail illustrated in Figure 2. A subexpression with a different parent is represented as a box within a box. A solid arrow denotes the parent relationship. Of course, the user is not expected to see and understand a complete graph of this nature. A tool called the Redex Trail viewer permits the whole graph to be explored interactively one expression at a time, as illustrated in this snapshot:



```
• True
  not False
  xor False True
   • xor (not True) False
      ▽ oneTrue []
   ▽ oneTrue (not True : [])
  oneTrue (False : not True : [])
  main
```

Each redex is shown on a separate line. The parent of an expression is shown below it. (Because parents are shown below their children in the viewer, we have drawn the full graph in Figure 2 similarly.) The parent of a whole

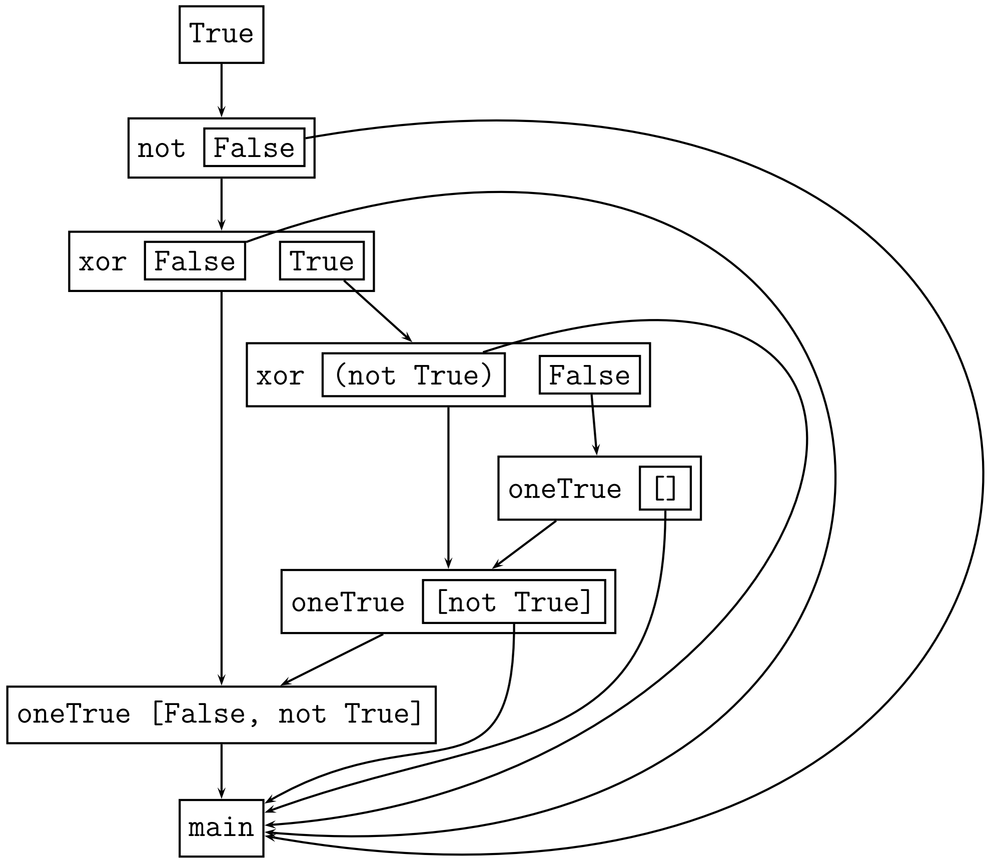


Fig. 2. An example of a Redex Trail

redex starts in the same column, whereas the parent of a proper subexpression is further indented. Underlining, and colouring if available, is used to show to which subexpression a parent belongs.

2.2 Structure

Figure 3 formalises the Redex Trail structure as a set of concrete Haskell types. An *App* node represents a redex in the obvious way, with a trace for the function and separate traces for each argument. It also contains the *parent* trace describing the redex that created it. Each function and argument is likewise of *Trace* type and therefore has its own, possibly different, parent. The *SrcPos* type records the location in the source code of the relevant application site on the RHS of a definition.

A *Const* node represents an irreducible value (an *Atom*), such as an integer, character, or a constructor or named function from the program, represented simply as a string identifier. (In the latter case, a *SrcPos* is associated with the identifier to record its static definition site.) A *Const* node also has a

$$\begin{array}{llll}
\textbf{data}\ \mathit{Trace} & = \mathit{App} & \{\mathit{fun} :: \mathit{Trace},\ \mathit{args} :: [\mathit{Trace}] \\
 & & ,\ \mathit{parent} :: \mathit{Trace},\ \mathit{src} :: \mathit{SrcPos}\ \} \\
 & \mid\ \mathit{Const} & \{\mathit{value} :: \mathit{Atom} \\
 & & ,\ \mathit{parent} :: \mathit{Trace},\ \mathit{src} :: \mathit{SrcPos}\ \} \\
 & \mid\ \mathit{Root} & \\
\textbf{data}\ \mathit{SrcPos} & = \mathit{SrcPos} & \{\mathit{file} :: \mathit{FilePath} \\
 & & ,\ \mathit{line} :: \mathit{Int},\ \mathit{column} :: \mathit{Int}\ \} \\
 & \mid\ \mathit{NoPos} & \\
\textbf{data}\ \mathit{Atom} & \multicolumn{2}{l}{= \mathit{Id}\ \mathit{String}\ \mathit{SrcPos} \mid \mathit{IntVal}\ \mathit{Int} \mid \mathit{CharVal}\ \mathit{Char} \mid \ldots}
\end{array}$$

Fig. 3. The original Redex Trail structure.

parent redex and a *SrcPos* to indicate its dynamic origin. For instance, two uses of the function f in a computation may have different positions in the source code, because they are introduced to the trace by rewriting different redexes.

Finally, it is possible for a redex to have no parent, represented simply as *Root*. This clearly occurs at the very start of the computation, namely for the *main* function. It also applies in the case of other top-level pattern-bindings (CAFs).

3 Alternative Views

We would like to adapt the original Redex Trail structure to support additional styles of viewing, such as Algorithmic Debugging and Observations. So what do these views look like? And what information do they require from the trace?

3.1 Algorithmic Debugging

Algorithmic Debugging is a well-known technique in declarative languages [8], implemented for a subset of Haskell by the Freja system [5]. The algorithm locates an error in a program, given a user who can provide answers to a sequence of questions. Each question concerns a reduction of a redex to a result, presented as an equation. The user should answer *yes* if the equation is correct with respect to his intentions, and *no* otherwise. After some number of questions, the system identifies an incorrect function definition.

Here is such a session for the example program of Figure 1. The symbol '_' represents an expression that has never been evaluated and whose value hence cannot have influenced the computation.

```
1> oneTrue (False:_:[]) = True   (Y/?/N): n
2> oneTrue (_:[]) = True   (Y/?/N): n
3> oneTrue [] = False   (Y/?/N): y
4> xor _ False = True   (Y/?/N): n
```

```
Error located!
Bug found in: xor _ False = True
```

Freja creates an Evaluation Dependency Tree (EDT) as its trace structure. Figure 4 shows the EDT for this example. Each node of the tree is a reduction. The tree is basically the derivation/proof tree for a call-by-value reduction with miraculous stops where expressions are not needed for the result. The call-by-value structure ensures that the tree structure reflects the program structure and that arguments are maximally evaluated.

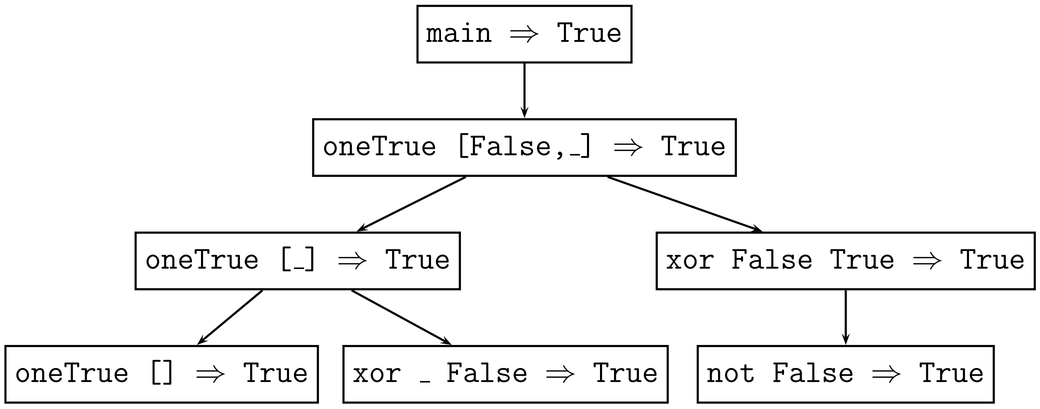


Fig. 4. An Evaluation Dependency Tree

The viewer dialogue walks the tree, presenting each node as a question – some answers permit some branches of the tree to be ignored.

To allow algorithmic debugging starting from a Redex Trail rather than an EDT, we need to add the ability to extract redexes from deep within the Trail – for instance, the very first question is about the reduction of *main*, which lies at the farthest tip of the Redex Trail graph. Furthermore, we need to record dependency information in the opposite direction – the reduction of an expression depended on what sub-reductions?

3.2 Observations

HOOD [3] allows observation of individual values within computations. The programmer annotates the expression(s) of interest in the program source with the combinator `observe`. For each annotation, HOOD records the value in all of its intermediate stages of evaluation, so that after termination of the computation the observed expression can be shown to exactly the degree to which it was demanded.

By a few clever tricks, HOOD can record not only data values, but also functional values, again only to the degree they are really used in the computation. Thus a functional value is recorded as a bag of actual argument/result

```
data StoredEvent    = Value  { value  :: String
                             , inside :: Ref    , position :: Int }
                    | Constr { name   :: String, arity    :: Int
                             , inside :: Ref    , position :: Int }
type Observation    = Ref  → StoredEvent
data ObservedValue  = Value  { value :: String }
                    | Constr { name      :: String
                             , arguments :: [ObservedValue] }
```

Fig. 5. The two HOOD observation structures. StoredEvents are recorded as a simple sequence during the computation, but the viewer must later traverse the StoredEvents to construct an ObservedValue that can be displayed.

mappings [4].

Here we illustrate the output when observing the functions `oneTrue` and `xor` in our example program. The symbol '_' again represents an unevaluated expression.

```
oneTrue (False:_:[]) = True
oneTrue (_:[]) = True
oneTrue [] = False

xor False True = True
xor _ False = True
```

The HOOD trace structure (sketched in Figure 5) is a sequence of individual 'events'. Every event represents the creation of a data value or constructor (in WHNF) during the computation. It has a backwards link to identify the enclosing data structure of which it is a component, and the particular argument position it occupies. Each constructor also has a note of how many argument 'holes' it can accept. A functional value is treated like a data constructor with two components, an argument and a result.

At viewing time, the HOOD viewer transforms the stored structure internally by constructing forward links from each constructor to the final value of each of its components. This can be expensive for a large structure.

The most notable difference between these structures and the Redex Trail is that HOOD stores only irreducible values, not reducible expressions. Another major difference is that HOOD records only individual annotated values, not a full trace of the whole computation.

In another paper at this workshop Reinke describes a graphical viewer for HOOD observations [7].

4 The Augmented Redex Trail Structure

The EDT structure used in Freja's algorithmic debugging is very similar to a substructure of the Redex Trail, but with pointers reversed. Most of the information in HOOD's Observations can be derived from either an EDT or a Redex Trail by searching through those structures for named values and source positions. The Redex Trail lacks some small pieces of information that would permit the reconstruction of an EDT by the reversal of pointers. This same lack also prevents the reconstruction of Observations.

In this section we extend the original structure in stages to become the Augmented Redex Trail, or ART structure.[4]

4.1 Linearisation and Explicit Sharing

The original Redex Trail is an ephemeral heap-based structure, but we want to store the trace in a file so it is persistent, does not depend on connecting a viewer to a live program, and can be accessed many times by different tools.

The original Redex Trail type *Trace* (in Figure 3) is self-recursive, so to place it in file requires the structure to be linearised. Linearisation gives two benefits: we can write the trace into the file one node at a time; and we can access a part of the trace piecemeal without necessarily following all possible paths. In both cases, efficiency is important: when generating, sequential writing is best; when viewing, the viewer tool should need to read only a small fragment of the whole structure.

Another benefit of linearising the structure is that it makes sharing of nodes explicit. In the original graph model, sharing and cycles are implicit via the self-recursive type, but in the new model this information is revealed directly to the viewing tool. There is a great deal of sharing in a trace of a typical program.

```
data Expr  = App    { fun    :: Ref, args :: [Ref]
                    , parent :: Ref, src  :: SrcPos }
           | Const  { value  :: Atom
                    , parent :: Ref   , src :: SrcPos }
           | Root
type Trace = Ref  → Expr
data Ref   = NoRef  | Ref FilePos deriving Eq
```

Fig. 6. The linearised Redex Trail structure.

Figure 6 shows how the trace structure changes to accommodate linearisation. We rename the original *Trace* datatype to *Expr*, and all self-recursive

[4] The concrete types presented in this section are a slightly abstracted view of the real trace structures in our current implementation.

uses of the type become explicit pointers *Ref*. The new *Trace* type defines a unique mapping from *Ref*s to *Expr*s. The *Root* constructor is replaced by the null reference *NoRef*. Every *Expr* node of the graph can be written to or read from a file individually. A *Ref* can also be updated in-place (see Section 4.2).

The trace structure as a whole is a sequence of *Expr* nodes, and the evaluation order of the program is apparent from the implicit ordering of *Expr* nodes in the file.

4.2 Redexes with Results

The original Redex Trail structure records all of the intermediate steps in a reduction sequence, but the links are made only in one direction – backwards – allowing exploration only from 'effects' to 'causes' in a viewer. However, both Algorithmic Debugging and Observations present equations in their viewers, and hence require a 'forward' link from every redex (LHS) to its reduct (RHS).

In Figure 7 we once again modify the concrete Haskell representation of the trace structure, this time to incorporate the 'forward' or *result* links.

```
data Expr = App     { fun    :: Ref , args   :: [Ref]
                    , parent :: Ref , src    :: SrcPos
                    , status :: Eval, result :: Ref }
          | Const   { value  :: Atom
                    , parent :: Ref  , src    :: SrcPos
                    , status :: Eval , result :: Ref }
data Eval = Applied | Blackholed | Completed | Value
```

Fig. 7. The core of the Augmented Redex Trail structure.

Although it may appear that the *parent* and *result* pointers simply represent the same relation with directions reversed, this is not so. A redex has at most one outgoing result arc – to the single expression it is rewritten to – but it can have many incoming parent arcs, because it is the creator of all subexpressions within the reduct. Hence, a result arc represents equality whereas a parent arc represents only inclusion.

A *Const* representing a basic value (e.g. integer/character) has a *parent* but no *result*; a *Const* which is an identifier (e.g. a top-level pattern-binding) could have a *result*, but no *parent*; often a *Const* identifier (e.g. local pattern-bindings) has both a *parent* and a *result*.

4.3 Unevaluated Expressions

Together with the *result* pointer we introduced a *status* :: *Eval* field. Even though an *Expr* is created, the expression it represents may never be evaluated.

This has consequences for how a viewer should interpret the *result* pointer.[5]

Every *Expr* can potentially go through three possible states. Initially, it is created by rewriting another expression according to some equation (its *status* is *Applied* and its *result* pointer is undefined). At some later point in the computation, the value of the expression may be demanded, or 'entered' (*status* now becomes *Blackholed*). At that point the *result* pointer is set to the newly written *Expr* that represents the reduct. Later again, the expression may become evaluated to a result expression (although this of course may still contain unevaluated subexpressions). At this point, the lazy evaluation mechanism does an 'update', overwriting the original expression with its result. In the trace, however, we do not overwrite the *Expr*, but update only the *status* to *Completed* (see §6.2). The final possible *Eval* status of *Value* is for irreducible expressions: an *Atom*, or an *App* with a data constructor in the function position. The *result* pointer for a *Value* is undefined.

4.4 Example

The augmented version of the Redex Trail graph from Figure 2 is shown in Figure 8. The subexpression relationship is now shown as a pointer (solid line). Parents are shown as dotted lines, and results as dashed lines. Expressions are annotated with their *status*.

4.5 Entry Points to the Trace

Every ART viewing tool needs an entry point at which to begin its presentation to the user. These entry points can be different for different tools.

The entry point for algorithmic debugging is the 'beginning' of the computation, the evaluation of the *main* function. In every ART trace, the *Expr* for *main* is the first *Expr* in the generated sequence.

The entry point for some other viewers is at the 'end' of computation, for instance when reconstructing a virtual stack-trace from an error message, or when exploring a Redex Trail backwards from the program output. This suggests that both the program output and any error messages must be recorded in the trace, since they are the 'end-points' of the program. Output and errors are easily added to the ART structure as strings with parent pointers. The output need not be monolithic; it can be spread across many strings; however we do not discuss here the various possible ways to split the output, nor how to store it in the file in a manner that permits quick access.

Other viewers may have variable entry points. For instance, a HOOD-style observation may need a named function or source position, and retrieve the relevant information by linear search through the trace.

[5] The original Redex Trail structure had a kind of node (*Sat*) which incorporated aspects of both the *result* pointer and the *status* marker, but these nodes were transient, removed from the graph once an expression was evaluated. The *Sat* node did not permanently record the full information required to forward-link every redex to its reduct.

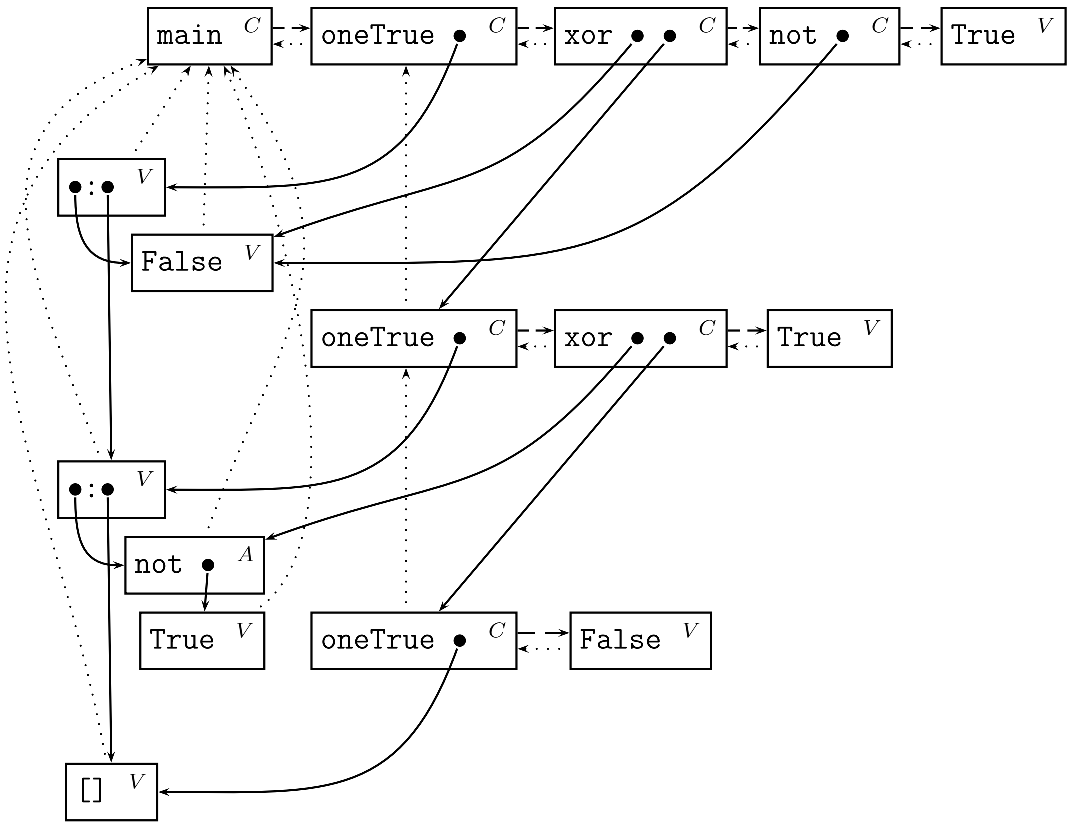


Fig. 8. A Trace

4.6 Other Refinements – Lambda, Do, Case, If, Guard

Not all expressions in Haskell are either a simple value, an application of a constructor to arguments, or an application of a named function to arguments. We also have anonymous functions (lambda expressions), monadic bindings (do statements), and various forms of conditional operations (ifs, cases, and guards). For conditionals, it is important to record in the trace not merely the final result, but how the decision was reached to take a particular branch. We follow much the same treatment for these extra constructs as in the original Redex Trail structure. For lambda expressions, we introduce a new kind of *Atom* to represent any function without a name, and the various kinds of conditional are handled by introducing new kinds of *Expr*s, each of which differs only slightly from the standard *App* kind.

4.7 Projections

There are situations that are slightly tricky to record, due to interesting consequences of the lazy evaluation model. One such is projections. In a definition like *id* $x = x$, the question is, who is the parent of x? In one sense, it is *id*, yet in another sense, *id* is merely passing on the value without touching it, so x's parent is really whatever expression created it, not *id*. Once again, we follow the original Redex Trail structure by introducing another special kind of *Expr*, the *Proj* node, which can be thought of as attaching an additional projective parent to the referenced redex.

4.8 Trusting

Finally, it is sometimes desirable *not* to record all reductions in the trace structure – we *trust* some function definitions, such as those in the Standard Prelude [10]. There are two main reasons for trusting. The first reason is to improve performance. Trace files are very large and quite slow to write. If we know that certain parts of the trace are not of interest, it makes sense to omit them. The second reason is to reduce the amount of information presented to the user of a trace-viewing tool.[6] Traces contain a huge amount of data, so a trace that appears too complex can actually hide the information the user wants. We do not elaborate the details of the trusting mechanism here.

5 Multiple Views from a Single Trace

Having outlined a unifying trace structure, we must now demonstrate that it can satisfy the needs of the Redex Trail, EDT, and Observation views. We have built three separate viewers which mimic the user interface behaviour of the three previous systems (Freja, HOOD, and the old Hat). In this section, we describe how the required information is reconstructed from the new ART trace.

5.1 A Redex Trail View: hat-trail

The original Redex Trail structure can be recovered by following mainly parent pointers. The result pointer chain is used to show a subexpression in its most evaluated form. The original Hat browser has been adapted to use the new ART trace, and is now called `hat-trail`. The viewer starts with program output or an error message, and enables the user to interactively explore a computation backwards from effect to cause by revealing the parent (origin) of any selected subexpression.

5.2 A Static Call Stack: hat-stack

One special-case use of the parent pointers is to show a static call-stack back-trace from any error message. This does not represent the real lazy evaluation stack — often sadly incomprehensible. Instead the backtrace gives the virtual

[6] It would of course be possible to implement a trusting mechanism in the viewing tool itself, rather than omitting the data from the trace altogether.

stack showing how an eager evaluation model would have arrived at the same result. In our system, this tool is `hat-stack`.

5.3 Algorithmic Debugging: hat-detect

The tool `hat-detect` provides Algorithmic Debugging, by extracting a virtual EDT 'by need' from the ART trace structure.[7] It can be seen from Figure 4 that we need three kinds of information from the trace: first, the EDT's root node; second, an EDT node's label, where a label is an equation containing an application and its result; and third, the children of an EDT node.

The root node of the EDT is always the *main* CAF, found at the beginning of the trace file.

Each EDT label is an equation: the LHS is an application or CAF itself, and the RHS is its result in its most fully evaluated form. When an *App* or *Const* has a *status* of *Completed*, we can follow the *result* pointer to determine the eventual value. The immediately referenced node might in turn be *Completed*, so the *result* chain must be followed iteratively until we find a node with an *Applied*, *Blackholed* or *Value* status. An *Applied* node was unevaluated, therefore it cannot have influenced the execution. It is presented to the user as a '_' symbol. A status of *Blackholed* is similarly displayed as $\perp$. Only a *Value* status represents a genuine result, either a simple value or a complex structure, and can be printed as a normal expression.

To determine the children of an EDT node, we must find all fully-evaluated applications on which the evaluation of the current node depended. The first child of a EDT node p, may be found by following p's ART result pointer, but the referenced node q is a child only if its *status* is *Completed* or *Blackholed*. (Only with one of these status annotations does the node describe an application or CAF whose result was actually demanded.) Further children can be found if q is *Completed*, *Blackholed* or a *Value*. In these cases the argument pointers of q are considered. If an argument's ART parent is also p, provided the argument itself is *Completed* or *Blackholed*, it is also a child of p. More children can be found recursively by the same method.

In this way, all the information necessary to define a computation's EDT can be retrieved from an ART trace file.

Only applications of top-level identifiers are considered by `hat-detect`. A locally defined function may depend on the values of free variables bound in an enclosing scope. To decide whether an application appears to be correct, the programmer needs to know the values of the free variables, yet the ART trace does not record any direct link to these variables. Program errors found by our tool therefore always refer to the top-level function; computation within local definitions is attributed upwards to its enclosing top-level definition.

[7] Although we describe the reconstruction of an EDT as if performed in one pass, the implementation need never build the whole structure - it can be constructed and traversed piecemeal.

The dialogue presented by `hat-detect` is straightforward to arrange, following the standard debugging algorithm. It starts at the root of the EDT, the *main* CAF. If the programmer answers that a node label is erroneous, he is asked about the correctness of its children, but children of nodes identified as correct need not be considered. An erroneous node with only correct children, or no children at all, is the location of the bug.

5.4 Observation of Functions: hat-observe

Our tool `hat-observe` displays all function applications of a given identifier within a computation. Unlike HOOD, no annotations are needed in the program's source code. As the observed identifier is chosen independently of the program run, it is easy to make a number of successive observations without modifying or rerunning the computation.

The tool observes a function by searching sequentially through the trace file. First, the identifier itself is found as a *Const* node containing an *Id* atom with the name. (See structures in Figures 7 and 3). Then every application node is checked for a reference to the given *Const* in the function position.

To deal with partial applications we must search the ART trace not only for references to the original *Const* node, but for references to any application which in turn references the *Const*, and so on recursively. If the function involved in an application is a reducible expression (with a function as result) we must follow this expression's forward *result* link, to see whether it is the desired function, or a partial application of it. The cost of such searching from application nodes to determine the associated function turns out to be low, as the relevant expressions are usually found close to the original application node. In particular, an additional file access is very rarely needed, as these expressions are usually within the file's buffer. Linearisation ensures that the function reference in an application node can only refer to an earlier node in the trace[8], so a single linear scan through the trace is sufficient to collect all applications of a specified function.

Not all applications or CAFs have results – they may be unevaluated, or an application may be partial – but where a result is available, the RHS of the equation can be determined as described in Section 5.3. All applications or CAFs with results are displayed as a list of equations.

To avoid redundant output, equivalent or less general applications of the identifier can be omitted in the display. One application of an identifier is considered more general than another if all its arguments are less defined (due to lazy evaluation). To avoid problems with local functions capturing free variables, as described in Section 5.3, we again only permit observations of top-level functions.

Our tool shows all applications of the function throughout the program, whereas HOOD observes a specific function application at one point in the

[8] All *Ref*s in the ART structure, apart from the *result*, refer to earlier nodes.

source code. However, because the source code position is recorded in the ART trace, an equivalent feature could be achieved by a different interface, perhaps a source code browser allowing the user to select expressions to be observed.

6 Trace Generation

The developers of Freja, Hat and HOOD made different choices about the architectural level at which they implemented the creation of the trace. For instance, in HOOD the trace is created by the combinator *observe* defined in a high-level Haskell library, which uses side-effecting I/O to record the information. In Freja the trace is created in the heap by low-level variants of the graph reduction machine instructions [5].

To generate the new ART trace [11] we took the old Redex Trail approach, but adapted to write traces to file instead of constructing them in heap memory. First, the original program is transformed into a new program that computes its trace in addition to its normal result. Second, the transformed program is compiled. Third, the compiled program is run. The computation writes a trace to file in addition to any normal I/O of the original program. Fourth, the trace is viewed.

Currently the program transformation is performed by an early phase of the Haskell compiler nhc98. However, we intend to separate the transformation from the compiler, so that the transformed program can be compiled with all Haskell compilers. The Augmented Redex Trail approach is then potentially as portable as the HOOD implementation, in contrast to Freja. The principle of using an automatic source-to-source transformation, coupled with a library of combinators written in standard Haskell, permits the possibility of using any Haskell compiler system to generate an ART trace.

6.1 The Program Transformation

The transformation wraps every expression of the original program into the R data type, which is defined as follows:

$$\mathbf{data}\, R\, \alpha \;=\; R\, \alpha\, \mathit{Ref}$$

The *Ref* is a reference to an *Expr* node of the trace in file. The pairing assures that an expression and its description “travel together” throughout the computation, so that when expressions are plumbed together by application, the corresponding descriptions in the trace can be plumbed together by creating an *App* node at the same time. Trace nodes are written to file by side-effects which are triggered when certain expressions are evaluated. All the plumbing and writing of trace nodes is performed by combinators which are defined in a library.

The program transformation introduces numerous calls of the combinators into the program. For example, here is the original *oneTrue* definition, together with its transformed version.

```
oneTrue          :: [Bool] → Bool
oneTrue []       = False
oneTrue (x : xs) = xor x (oneTrue xs)
```

```
oneTrue :: SrcPos → Ref → R (Ref → (R [Bool]) → R Bool)
oneTrue sr p = fun1 (mkAtomId "oneTrue" 7) oneTrueW sr p
  where
    oneTrueW :: Ref → R [Bool] → R Bool
    oneTrueW p′ (R [] _) =
        con0 (mkSrcPos 2) p′ False (mkAtomId "False" 6)
    oneTrueW p′ (R (x : xs) _) =
        rap2 (mkSrcPos 3) p′ (xor (mkSrcPos 3) p′) x
            (ap1 (mkSrcPos 4) p′ (oneTrue (mkSrcPos 4) p′) xs)
```

In this example the combinators *fun*1, *con*0, *ap*1, *rap*2, *mkAtomId*, and *mkSrcPos* are used. The combinator *fun*1 wraps the function *oneTrueW*, which does the actual work, with *R* constructors. The combinator *con*0 wraps the constructor *False*. The combinators *ap*1 and *rap*2 assure the correct plumbing of applications. The combinators *mkAtomId* and *mkSrcPos* build references to detailed information about the identifier *oneTrue*, the constructor *False* and various source references. Numeric arguments are indexes to tables that contain the detailed information.

Very similar combinators were used in the old Hat system. The most important difference is that the new Hat combinators now record the trace nodes directly to file.

6.2 Writing with Updating

The main technical obstacle is that the trace is a (usually cyclic) graph which is continuously modified during generation. These modifications were no problem in main memory but for efficient writing to file updates have to be minimised.

We assume that writing nodes to file has much better performance if it can be achieved sequentially. However, even a cursory examination of the ART structure tells us that after writing an *Expr* node to file, it is highly likely that we will need to return to it to update the result pointer. Although some expressions remain completely unevaluated throughout the computation, the vast majority of intermediate expressions are indeed entered and evaluated to their reduct.

What is more, in our scheme there are *two* possible updates for each *Expr*, one on entering the expression (*Applied* → *Blackholed*), and another on its completion (*Blackholed* → *Completed*).

However, we observe that a *Blackholed* expression is almost always transient. The only situation in which such an annotation can remain in the final trace is when the program's overall result is undefined, such as an error or interruption. We also observe that the order in which trace expressions are entered and then completed follows a strict stack discipline, mirroring the evaluation stack of the underlying abstract machine. Hence, we do not update *Applied* to *Blackholed* on entry, but only write the remaining stack of 'blackholes' at the end of the computation should it fail.

We also try to avoid interspersing the final update of each *Expr* with the sequential generation of nodes. This is easily achieved by storing a large queue of updates that are performed all at once.

7 Future Work

The practical questions that interest most people are about time and space. How large is the trace? And how long does it take to produce, relative to the original computation?

An ART trace is undoubtedly big, to be measured in megabytes for a computation of any significant size. We estimate about 40–50 bytes are required per reduction. The largest trace we have yet generated is 240Mb in size, for a computation of around 6 million reductions (a chess end-game solver). Traced computations also take about 50 times longer than normal computations.

If Hat is to be used for substantial computations, we have to reduce the slow-down factor for traced computations. The fact that only 10% of traced computation time is spent on actually writing to file demonstrates that the implementation of trace generation can be improved. Since the computation of a transformed program spends most of its time evaluating the combinators, efficient definitions of the combinators are vital. We will also separate the program transformation from nhc98, so that a transformed program can be compiled by an optimising Haskell compiler such as ghc. Not only would this improve absolute runtimes, but aggressive optimisation may also reduce the relative slow-down. Furthermore, the computation of trusted function definitions is not yet much faster than that of untrusted definitions. We intend to investigate how transformed modules can be combined with trusted *untransformed* modules. Such a scheme, not requiring access to the sources of trusted modules, would also aid portability.

Other issues we want to address include:

- Currently I/O actions are traced in a rather ad hoc way that works well only with some views for simple I/O only. We aim to develop a general method for tracing I/O actions.

- We want to solve the problem that none of the views copes well with programs that make substantial use of higher-order combinators, for example in monadic or continuation-passing style.
- We plan to extend `hat-observe` to observe values at any program point. We could also add information about free variables to expressions in the trace, so that `hat-detect` and `hat-observe` can show a fuller trace of local computation. It may even be desirable to switch levels of detail within a view.
- There is scope for new viewing tools. For instance, the evaluation order of the computation is stored implicitly in the sequence in which *Expr* nodes are written to file. Hence, the computation, or selected parts of it, could be shown as an "animation", perhaps in the style of GHood[9]. We could also offer a "stories" view in the style outlined in [1]. A more specialised tool could isolate the circular self-dependency that evokes a "blackhole" error.
- We have begun to integrate `hat-detect` and `hat-observe` into a single tool. Eventually we hope for a full integration allowing the programmer to switch between views at any point during the exploration of a computation.
- How can we evaluate the useability of Hat in practice and gain information to improve it?

More generally, we intend to study the properties of the ART trace further. Is the trace complete with respect to information, such as recorded reductions, intermediate unevaluated expressions and values, and with respect to distinctions and relationships, such as sharing and evaluation order? How conveniently and efficiently can one access the trace to obtain a specific snippet of information? Can we claim some sort of "universality" for the trace structure, in terms of the range of queries it can support? How are all these properties affected by trusting? Does the exclusion of trusted redexes from the trace compromise the reachability of individual trace nodes from designated entry points?

There should be a close relationship between tracing and operational semantics, both of which aim to describe the relationship between a program and the observed actions of a computation of the program. We have begun work to define the ART trace and specific views through conservative extensions of an operational semantics of a program. Different kinds of formal semantics may suggest new views for tracing. For instance, the evaluation dependency tree of algorithmic debugging is closely related to a big-step structured operational semantics; the Redex Trail view was inspired by graph-rewriting machines; the observation of values recalls denotational semantics, especially the view of functional values as (finite) mappings ('minimal function graphs' [4]).

In principle, a semantics defines all the answers a tracer could give for the computation of a particular program with particular input. A tracer

[9] `http://www.cs.ukc.ac.uk/people/staff/cr3/toolbox/haskell/GHood/`

makes this information available. A tracer avoids providing its own semantics but hooks on to a compiler instead. A program transformation provides this "hook" in a portable way. Even more important than the information in a trace is its accessibility.

8 Conclusions

We have presented the new modular architecture of our Haskell tracer Hat. At its heart lies the new Augmented Redex Trail trace structure, designed on the one hand to be written to file while performing the traced computation, and on the other hand to provide data sufficient for multiple views.

As an immediate result, we have widened the applicability of the new Hat considerably. Initial experiences confirm the usefulness of generating a trace only once and viewing it in several different ways.

The new modularity improves the understanding of the tracing process. The new architecture has also prompted us to ask some more general questions, such as those in the Future Work section.

9 Acknowledgements

Many thanks to Amanda Clare, Henrik Nilsson, John O'Donnell, Claus Reinke, and Jan Sparud who each took time to travel to York and play with various different tracing systems. The feedback from these experiments was extremely valuable.

We also thank the referees for their constructive comments.

The work reported in this paper was supported by the UK's Engineering and Physical Sciences Research Council under grant number GR/M81953.

References

[1] Simon P Booth and Simon B Jones. Walk backwards to happiness — debugging by time travel. Technical Report CSM-143, University of Stirling, 1997.

[2] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000*, LNCS 2011, pages 176–193. Springer, 2001.

[3] Andy Gill. Debugging Haskell by observing intermediate data structures. In *2000 ACM SIGPLAN Haskell Workshop*, 2000. Technical Report NOTTCS-TR-00-1, University of Nottingham.

[4] N. D. Jones and A. Mycroft. Data Flow Analysis of Applicative Programs using Minimal Function Graphs. In *Proc. 13th Annual Symposium on the Principles of Programming Languages* (POPL'86), pages 296–306, ACM Press, January 1986.

[5] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.

[6] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering: An International Journal*, 4(2):121–150, April 1997.

[7] Claus Reinke. GHood — Graphical visualisation and animation of Haskell object observations. *Proceedings of ACM Sigplan Haskell Workshop 2001*, September 2001.

[8] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[9] Jan Sparud. *Tracing and Debugging Lazy Functional Computations.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1999.

[10] Jan Sparud and Colin Runciman. Complete and partial redex trails of functional computations. In C. Clack, K. Hammond, and T. Davie, editors, *Selected papers from 9th Intl. Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS Vol. 1467, September 1997.

[11] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS Vol. 1292, September 1997.

FUNCTIONAL PEARL
Parsing Permutation Phrases

Arthur Baars, Andres Löh, S. Doaitse Swierstra

Institute of Information and Computing Sciences,
Utrecht University, Utrecht, The Netherlands

Abstract

A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. We show a way to extend a parser combinator library with support for parsing such free-order constructs. A user of the library can easily write parsers for permutation phrases and does not need to care about checking and reordering the recognised elements. Possible applications include the generation of parsers for attributes of XML tags and Haskell's record syntax.

1 Introduction

Parser combinator libraries for functional programming languages are well-known and subject to active research. Higher-order functions and the possibility to define new infix operators allow parsers to be expressed in a concise and natural notation that closely resembles the syntax of EBNF grammars. At the same time, the user has the full abstraction power of the underlying programming language at hand. Complex, often recurring patterns can be expressed in terms of higher-level combinators.

A specific parsing problem is the recognition of permutation phrases. A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. Since permutation phrases are not easily expressed by a context-free grammar, the usual approach is to tackle this problem in two steps: first parse a relaxed version of the grammar, then check whether the recognised elements form a permutation of the expected elements. This method, however, has a number of disadvantages. Dealing with a permutation of typed values is quite cumbersome, and the problem is often avoided by encoding the values in a universal representation, thus adding an extra level of interpretation. Furthermore, because of the two steps involved, error messages cannot be produced until a larger part of the input has been consumed, and special care has to be taken to make them point to the right position in the code.

Permutation phrases have been proposed by Cameron [1] as an extension to EBNF grammars, not aiming at greater expressive power, but at more clarity. Cameron also presents a pseudo-code algorithm to parse permutation phrases with optional elements efficiently in an imperative setting. It fails, however, to address the types of the constituents.

We show a way to extend any existing parser combinator library with support for parsing permutations of a number of typed, potentially optional elements. Our approach uses Haskell, relying essentially on existentially quantified types, that are used to encode reordering information that permutes the recognised elements to a canonical order. Existential types are not part of the Haskell 98 standard [6], but are, for example, implemented in GHC and Hugs. Additionally, we utilise lazy evaluation to make the resulting implementation efficient. The administrative part of parsing permutation phrases has a quadratic time complexity in the number of permutable elements. The size of the code, however, is linear in the number of permutable elements.

Possible applications include the implementation of Haskell's *read* function where it is desirable to parse the fields of data types with labelled fields in any permutation, the parsing of XML tags which have large sets of potentially optional attributes that may occur in any order, and the decomposition of a query in a URI, consisting of a number of permutable key-value pairs.

The paper is organised as follows: Section 2 explains the parser combinators we build upon. Section 3 presents the basic idea of dealing with permutations in terms of permutation trees and explains how trees are built and converted into parsers. Section 4 shows how to extend the mechanism in order to handle optional elements. In Section 5, we take a brief look at two of the applications mentioned above, the parsing of data types with labelled fields and the parsing of XML attribute sets. Section 6 concludes.

2 Parsing using combinator libraries

The use of a combinator library for describing parsers instead of writing them by hand or generating them from a separate formalism is a well-known technique in functional programming. As a result, there are several excellent libraries around. For this reason we just briefly present the interface we will assume in subsequent sections of this paper, but do not go into the details of the implementation. However, we want to stress that our extension is not tied to any specific library.

We make use of a simple arrow-style [3,9] interface that is parametrised by the result type of the parsers and assumes a list of characters as input. It can easily be implemented by straightforward list-of-successes parsers [2,10], but we also have a version based on the fast, error-correcting parser combinators of Swierstra [7,8]. A mapping to monadic-style parser combinators [4,5] or

```
infixl 3 <|>
infixl 4 <*>, <$>
class Parser p where
  pFail           :: p a
  pSucceed        :: a → p a
  pSym            :: Char → p Char
  (<*>)           :: p (a → b) → p a → p b
  (<|>)           :: p a → p a → p a
  (<$>)           :: (a → b) → p a → p b
  f <$> p         =  pSucceed f <*> p
  parse           :: p a → String → Maybe a
```

Fig. 1. Type class for parser combinators

```
infixl 4 <$ , *>, <*
(<$ )            :: Parser p ⇒ a → p b → p a
f <$  p          =  const f <$> p
(<* )            :: Parser p ⇒ p a → p b → p a
p <*  q          =  const <$> p <*> q
( *>)            :: Parser p ⇒ p a → p b → p b
p  *> q          =  flip const <$> p <*> q
pParens          :: Parser p ⇒ p a → p a
pParens p        =  pSym '('  *> p <*  pSym ')'
```

Fig. 2. Some useful parser combinators

abstracting from the type of the input tokens is possible without difficulties.

The parser interface used here is given as a type class declaration in Figure 1. The function *pFail* represents the parser that always fails, whereas *pSucceed* never consumes any input and always returns the given result value. The parser *pSym* accepts solely the given character as input. If this character is encountered, *pSym* consumes and returns this character, otherwise it fails. The <*> operator denotes the sequential composition of two parsers, where the result of the first parser is applied to the result of the second. The operator <|> expresses a choice between two parsers. Finally, the application operator <$> is a parser transformer that can be used to apply a semantic function to a parse result. It can be defined in terms of *pSucceed* and <*>.

Many useful higher-level combinators can be built on top of these basic ones. A small selection that we will use later in this paper is presented in Figure 2. These parser combinators are useful if one wants to combine parsers and is interested in the result of only some of the constituents.

3 Permutation trees

3.1 Data types

Before explaining permutation parsers, we investigate how to represent permutation phrases. We decide to store the permutations of a set of elements in a rose tree.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Perms}\ p\ a & = & \mathit{Choice}\ [\mathit{Branch}\ p\ a] \\
 & | & \mathit{Empty}\ a \\
\textbf{data}\ \mathit{Branch}\ p\ a & = & \forall\ x.\ \mathit{Br}\ (\mathit{Perms}\ p\ (x \to a))\ (p\ x)
\end{array}
$$

The data types are parametrised by a type constructor p (e.g. the parser type) and a result type a.

Each path from the root to a leaf in the tree represents a particular permutation. Figure 3 illustrates this idea for three elements a, b, c. If the permutations are grouped in such a way that different permutations with a common prefix share the same subtree, the number of choices in each node will be limited by the number of permutable elements.

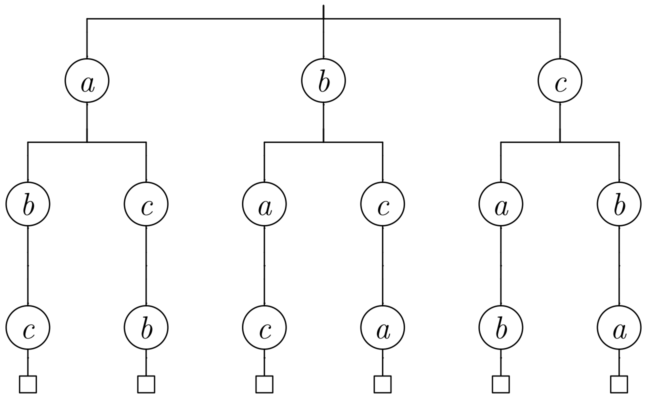


Fig. 3. A permutation tree containing three elements

A value of type *Branch* stores a subtree together with an element. The subtree returns a function that, applied to the element, computes a value of the required result type. Thus, the existentially quantified type of the element stored in a branch is used to hide the order in which the types in a subtree occur; all subtrees in a *Choice* node share a common type because all correspond to a permutation of the same set of elements. To show that reordering is almost completely determined by the type of the components, we use the convention that values, parsers and permutation trees are named v, p, and t, respectively, indexed by their type.

The idea that each path in the tree represents the parser for one of the possible permutations is reflected by the following simple conversion function from permutation trees to parsers.

$$\begin{array}{lcl}
\mathit{pPerms} & :: & \mathit{Parser}\ p \Rightarrow \mathit{Perms}\ p\ a \rightarrow p\ a \\
\mathit{pPerms}\ (\mathit{Empty}\ v_a) & = & \mathit{pSucceed}\ v_a \\
\mathit{pPerms}\ (\mathit{Choice}\ \mathit{chs}) & = & \mathit{foldr}\ (\lhd\!\rhd)\ \mathit{pFail}\ (\mathit{map}\ \mathit{pars}\ \mathit{chs}) \\
\quad \mathbf{where}\ \mathit{pars}\ (\mathit{Br}\ t_{x\rightarrow a}\ p_x) & = & \mathit{flip}\ (\$) \mathbin{<\!\$\!>} p_x \mathbin{<\!*\!>} \mathit{pPerms}\ t_{x\rightarrow a}
\end{array}$$

It might be surprising at first sight that the last line does not read:

$$\begin{array}{lcl}
\quad \mathbf{where}\ \mathit{pars}\ (\mathit{Br}\ t_{x\rightarrow a}\ p_x) & = & \mathit{pPerms}\ t_{x\rightarrow a} \mathbin{<\!*\!>} p_x
\end{array}$$

But using this more obvious definition, the elements at the leaves of the permutation tree would be recognised first by the constructed parser. Therefore, the permutation tree would have to be unfolded completely before the first element could be parsed. This would result in $O(n!)$ memory usage where n is the number of permutable elements.

Fortunately, because we are parsing a permutation, reversing the order of the constituents when constructing the parser does not change the semantics. In the "flipped" variant, lazy evaluation ensures that only the path corresponding to the recognised permutation is unfolded.

3.2 Building a permutation tree

Permutation trees are created by adding the elements of the permutation one by one to an initially empty tree.

$$\begin{array}{lcl}
\mathit{add} & :: & \mathit{Perms}\ p\ (x \rightarrow a) \rightarrow p\ x \rightarrow \mathit{Perms}\ p\ a \\
\mathit{add}\ t_{x\rightarrow a}@(\mathit{Empty}\ \_)\ p_x & = & \mathit{Choice}\ [\mathit{Br}\ t_{x\rightarrow a}\ p_x] \\
\mathit{add}\ t_{x\rightarrow a}@(\mathit{Choice}\ \mathit{chs})\ p_x & = & \mathit{Choice}\ (\mathit{first} : \mathit{others}) \\
\quad \mathbf{where}\ \mathit{first} & = & \mathit{Br}\ t_{x\rightarrow a}\ p_x \\
\qquad\qquad \mathit{others} & = & \mathit{map}\ \mathit{ins}\ \mathit{chs} \\
\qquad\qquad \mathit{ins}\ (\mathit{Br}\ t_{y\rightarrow x\rightarrow a}\ p_y) & = & \mathit{Br}\ (\mathit{add}\ (\mathit{mapPerms}\ \mathit{flip}\ t_{y\rightarrow x\rightarrow a})\ p_x)\ p_y
\end{array}$$

If we already have constructed a non-empty permutation tree, we can add a new element p_x by inserting it in all possible positions to every permutation in the tree. The function *add* explicitly constructs the tree that represents the permutation in which p_x is the top element; for each branch, the top element is left unchanged, and p_x is inserted everywhere (by a recursive call to *add*) in the subtree. Because the new element and the top element of the branch are now swapped, the function resulting from the subtree of the branch gets its arguments passed in the wrong order, which is repaired by applying *flip* to that function.

The function *mapPerms* is a map on permutation trees. In a branch, $v_{a\rightarrow b}$ is composed with the function that is resulting from the subtree.

$$\begin{array}{lcl}
\mathit{mapPerms} & :: & (a \rightarrow b) \rightarrow \mathit{Perms}\ p\ a \rightarrow \mathit{Perms}\ p\ b \\
\mathit{mapPerms}\ v_{a\rightarrow b}\ (\mathit{Empty}\ v_a) & = & \mathit{Empty}\ (v_{a\rightarrow b}\ v_a) \\
\mathit{mapPerms}\ v_{a\rightarrow b}\ (\mathit{Choice}\ \mathit{chs}) & = & \mathit{Choice}\ (\mathit{map}\ (\mathit{mapBranch}\ v_{a\rightarrow b})\ \mathit{chs}) \\
\\
\mathit{mapBranch} & :: & (a \rightarrow b) \rightarrow \mathit{Branch}\ p\ a \rightarrow \mathit{Branch}\ p\ b \\
\mathit{mapBranch}\ v_{a\rightarrow b}\ (\mathit{Br}\ t_{x\rightarrow a}\ p_x) & = & \mathit{Br}\ (\mathit{mapPerms}\ (v_{a\rightarrow b}\circ)\ t_{x\rightarrow a})\ p_x
\end{array}$$

By defining the following combinators for constructing permutation parsers we can use a similar notation for permutation parsers as for normal parsers.

```
succeedPerms    :: a → Perms p a
succeedPerms x  =  Empty x

(<<*>>)         :: Parser p ⇒ Perms p (a → b) → p a → Perms p b
perms <<*>> p   =  add perms p

(<<$>>)         :: Parser p ⇒ (a → b) → p a → Perms p b
f <<$>> p       =  succeedPerms f <<*>> p
```

An example with three permutable elements, corresponding to the tree in Figure 3, can now be realised by:

```
pPerms ((,,) <<$>> pInt <<*>> pChar <<*>> pBool)
```

Suppose *pInt*, *pChar*, and *pBool* are parsers for literals of type *Int*, *Char*, and *Bool*, respectively. Then all permutations of an integer, a character and a boolean are accepted, and the result of a successful parse will always be of type (*Int*, *Char*, *Bool*).

3.3 Separators

Often the permutable elements are separated by symbols that do not carry meaning—typically commas or semicolons. Consider extending the three-element example to the Haskell tuple syntax: not just the elements, but also the parentheses and the commas should be parsed. Since there is one separation symbol less than there are permutable elements, our current variant of *pPerms* cannot handle this problem.

Therefore we define *pPermsSep* as a generalisation of *pPerms* that accepts an additional parser for the separator as an argument. The semantics of the separators are ignored for the result.

```
pPermsSep                :: Parser p ⇒ p b → Perms p a → p a
pPermsSep sep perm       =  p2p (pSucceed ()) sep perm
```

The function *p2p* now converts a permutation tree into a parser almost in the same way as the former *pPerms*, except that before each permutable element a separator is parsed. To prevent that a separator is expected before the first permutable element, we make use of the following simple trick. The *p2p* function expects two extra arguments: the first one will be parsed immediately before the first element, and the second will be used subsequently. Using *pSucceed* () as first extra argument in *pPermsSep* leads to the desired result.

```
p2p                      :: Parser p ⇒ p c → p b
                         → Perms p a → p a
p2p _ _ (Empty v_a)      =  pSucceed v_a
p2p fsep sep (Choice chs) = foldr (<|>) pFail (map pars chs)
  where pars (Br t_x→a p_x) = flip ($) <$ fsep <*> p_x <*> p2p sep sep t_x→a
```

The *pPerms* function can now be implemented in terms of *pPermsSep*.

```
pPerms                 :: Parser p => Perms p a -> p a
pPerms                 =  pPermsSep (pSucceed ())
```

To return to the small example, triples of an integer, a character, and a boolean—in any order—are parsed by:

```
pParens (pPermsSep (pSym ',') ((,,) <<$>> pInt <<*>> pChar <<*>> pBool))
```

4 Adding optional elements

This section shows how the permutation parsing mechanism can be extended such that it can deal with optional elements. Optional elements can be represented by parsers that can recognise the empty string and return a default value for this element. Calling the *pPermsSep* function on a permutation tree that contains optional elements leads to ambiguous parsers. Consider, for example, the tree in Figure 3 containing all permutations of a, b and c. Suppose b can be empty and we want to recognise ac. This can be done in three different ways since the empty b can be recognised before a, after a or after c. Fortunately, it is irrelevant for the result of a parse where exactly the empty b is derived, since order is not important. This allows us to use a strategy similar to the one proposed by Cameron [1]: parse nonempty constituents as they are seen and allow the parser to stop if all remaining elements are optional. When the parser stops the default values are returned for all optional elements that have not been recognised.

To implement this strategy we need to be able to determine whether a parser can derive the empty string and split it into its default value and its non-empty part, i.e. a parser that behaves the same except that it does not recognise the empty string. The splitting of parsers is represented by the *ParserSplit* class that is an extension of the normal *Parser* class. Most parser combinator libraries can be easily adapted to cover this extension.

```
class Parser p      => ParserSplit p where
  pEmpty            :: p a -> Maybe a
  pNonempty         :: p a -> Maybe (p a)
```

In the solution that does not deal with optional elements a parser for a permutation follows a path from the root of a permutation tree to a leaf, i.e. an *Empty* node. In the presence of optional elements, however, a parser may stop in any node that stores only optional elements. We adapt the *Perms* data type to incorporate this additional information. If all elements stored in a tree are optional then their default values are stored in *defaults*, otherwise *defaults* is *Nothing*. The parser stored in each *Branch* is not allowed to derive the empty string. Note that we do not need an *Empty* constructor anymore, since its semantics can be represented as a *Choice* node with an empty list of branches.

$$\textbf{data}\ \mathit{Perms}\ p\ a = \mathit{Choice}\ \{\mathit{defaults} :: \mathit{Maybe}\ a, \mathit{branches} :: [\mathit{Branch}\ p\ a]\}$$

The function *p2p* constructs a parser out of a permutation tree. If there are default values stored in *defaults* then the constructed parser can derive the empty string, returning those values.

$$\begin{array}{lll}
\mathit{p2p} & :: & \mathit{Parser}\ p \\
 & \Rightarrow & p\ c \to p\ b \to \mathit{Perms}\ p\ a \to p\ a \\
\mathit{p2p}\ \mathit{fsep}\ \mathit{sep}\ t_{a\to b} & = & \mathit{foldr}\ (\mathbin{<\!|\!>})\ \mathit{empty}\ \mathit{nonempties} \\
\quad \textbf{where}\ \mathit{empty} & = & \mathit{maybe}\ \mathit{pFail}\ \mathit{pSucceed}\ (\mathit{defaults}\ t_{a\to b}) \\
\qquad\qquad \mathit{nonempties} & = & \mathit{map}\ \mathit{pars}\ (\mathit{branches}\ t_{a\to b}) \\
\qquad\qquad \mathit{pars}\ (\mathit{Br}\ t_{x\to a}\ p_x) & = & \mathit{flip}\ (\$) \mathbin{<\$}\ \mathit{fsep} \mathbin{<\!*\!>} p_x \\
 & & \qquad\qquad \mathbin{<\!*\!>} \mathit{p2p}\ \mathit{sep}\ \mathit{sep}\ t_{x\to a}
\end{array}$$

A tuple, that represents a parser split into its empty part and its non-empty part, can describe four different kinds of parsers, as depicted in the following table:

empty part	non-empty part	
Nothing	*Nothing*	*pFail*
Just	*Nothing*	*pSucceed*
Nothing	*Just*	required element
Just	*Just*	optional element

The new definition of *add* reflects the four different cases. In the first case the resulting tree represents a failing permutation parser, i.e. it has no default values and no branches. In the second case the value stored in the empty part of the parser is pushed into the tree, only modifying the semantics of the tree but keeping its structure. In the cases where an element is added the non-empty part of the element is inserted in the tree in the same way as in the original definition of *add*. For an optional element the default value is combined with the *defaults* of the permutation tree.

$$\begin{array}{lll}
\mathit{add} & :: & \mathit{Perms}\ p\ (a \to b) \\
 & \to & (\mathit{Maybe}\ a, \mathit{Maybe}\ (p\ a)) \\
 & \to & \mathit{Perms}\ p\ b \\
\multicolumn{3}{l}{\mathit{add}\ t_{a\to b}@(\mathit{Choice}\ d_{a\to b}\ \mathit{bs}_{a\to b})\ \mathit{mp}_a = \textbf{case}\ \mathit{mp}_a\ \textbf{of}} \\
\quad (\mathit{Nothing}, \mathit{Nothing}) & \to & \mathit{Choice}\ \mathit{Nothing}\ [\,] \\
\quad (\mathit{Just}\ v_a, \mathit{Nothing}) & \to & \mathit{Choice}\ (\mathit{fmap}\ (\$v_a)\ d_{a\to b})\ (\mathit{appSem}\ v_a) \\
\quad (\mathit{Nothing}, \mathit{Just}\ p_a) & \to & \mathit{Choice}\ \mathit{Nothing}\ (\mathit{insert}\ p_a) \\
\quad (\mathit{Just}\ v_a, \mathit{Just}\ p_a) & \to & \mathit{Choice}\ (\mathit{fmap}\ (\$v_a)\ d_{a\to b})\ (\mathit{insert}\ p_a) \\
\quad \textbf{where}\ \mathit{insert}\ p_a & = & \mathit{Br}\ t_{a\to b}\ p_a : \mathit{map}\ \mathit{ins}\ \mathit{bs}_{a\to b} \\
\qquad\qquad \mathit{ins}\ (\mathit{Br}\ t_{x\to a\to b}\ p_x) & = & \mathit{Br}\ (\mathit{add}\ (\mathit{mapPerms}\ \mathit{flip}\ t_{x\to a\to b})\ \mathit{mp}_a)\ p_x \\
\qquad\qquad \mathit{appSem}\ v_a & = & \mathit{map}\ (\mathit{mapBranch}\ (\$v_a))\ \mathit{bs}_{a\to b}
\end{array}$$

The function *mapPerms* for the new *Perms* data type is defined as follows:

$$\begin{array}{lll}
\mathit{mapPerms} & :: & (a \to b) \to \mathit{Perms}\ p\ a \to \mathit{Perms}\ p\ b \\
\mathit{mapPerms}\ v_{a\to b}\ t_a & = & \mathit{Choice}\ (\mathit{fmap}\ v_{a\to b}\ (\mathit{defaults}\ t_a)) \\
 & & \qquad\qquad (\mathit{map}\ (\mathit{mapBranch}\ v_{a\to b})\ (\mathit{branches}\ t_a))
\end{array}$$

Since we no longer have an *Empty* constructor the function *succeedPerms* is now defined as:

```
succeedPerms        :: a -> Perms p a
succeedPerms x      =  Choice (Just x) []
```

Using the functions from the *ParserSplit* class we can straightforwardly define a new sequence operator for permutation parsers.

```
(<||>)              :: ParserSplit p
                    => Perms p (a -> b) -> p a -> Perms p b
perms <||> p        =  add perms (pEmpty p, pNonempty p)
```

5 Applications

5.1 XML attributes

We will now demonstrate the use of the permutation parsers by showing how to parse XML tags with attributes. For simplicity, we just consider one tag (the `img` tag of XHTML) and only deal with a subset of the attributes allowed. In a Haskell program, this tag might be represented by the following data type.

```
data XHTML          =  Img { src        :: URI
                           , alt        :: Text
                           , longdesc   :: Maybe URI
                           , height     :: Maybe Length
                           , width      :: Maybe Length
                           }
                    |  ...
```

Our variant of the `img` tag has five attributes of three different types. We use Haskell's record syntax to keep track of the names. The first two attributes are mandatory whereas the others are optional. We choose the *Maybe* variant of their types to reflect this optionality. Our parser should be able to parse the attributes in any order, where any of the optional arguments may be omitted. For the parsing process, we ignore whitespace and assume that there is a parser *pTok* that consumes just the given token and fails on any other input.

Using the *pPerms* combinator, writing the parser for the `img` tag is easy:

```
pImgTag     :: ParserSplit p => p XHTML
pImgTag     =  pTok "<" *> pTok "img" *> attrs <* pTok "/>"
  where
  attrs     =  pPerms (Img <$$> pField    "src"      pURI
                           <||> pField    "alt"      pText
                           <||> pOptField "longdesc" pURI
                           <||> pOptField "height"   pLength
                           <||> pOptField "width"    pLength
                      )
```

The order in which we denote the attributes determines the order in which the results are returned. Therefore, we can apply the *Img* constructor to form a value of the *XHTML* data type. The two helper functions *pField* and *pOptField* are used to parse a mandatory and an optional argument, respectively.

```
pField            :: Parser p ⇒ String → p a → p a
pField f p        =  pTok f  *> pSym '='  *> p

pOptField         :: Parser p ⇒ String → p a → p (Maybe a)
pOptField f p     =  Just <$> pField f p
                  <|> pSucceed Nothing
```

5.2 Haskell's record syntax

Haskell allows data types to contain labelled fields. If one wants to construct a value of that data type, one can make use of these names. The advantage is that the user does not need to remember the order in which the fields of the constructor have been defined. Furthermore, all fields are considered as optional. If a field is not explicitly set to a value, it is silently assumed to be ⊥.

Whereas compilers implement these features as a syntax for constructing values inside of Haskell programs, the *read* function that both GHC and Hugs generate with help of the **deriving** construct lacks this functionality. Although this behaviour is permitted by the Haskell Report, the resulting asymmetry is unfortunate.

We show here that the code that would do the job is easy to write or generate using the *pPermsSep* combinator.

```
pImg        :: ParserSplit p ⇒ p XHTML
pImg        =  pTok "Img"  *> pTok "{"  *> fields <*  pTok "}"
  where
  fields    =  pPermsSep (pSym ',')
                 (Img <<$>> pRecField "src"        pURI
                      <<*>> pRecField "alt"        pText
                      <<*>> pRecField "longdesc"   (pMaybe pURI)
                      <<*>> pRecField "height"     (pMaybe pLength)
                      <<*>> pRecField "width"      (pMaybe pLength)
                 )
```

The *pMaybe* combinator just parses the *Maybe* variant of a data type, and the *pRecField* makes each field optional.

```
pRecField          :: Parser p ⇒ String → p a → p a
pRecField f p      =  pField f p
                   <|> pSucceed ⊥
```

6 Conclusion

We have presented a way to extend a parser combinator library with the functionality to parse free-order constructs. It can be placed on top of any combinator library that implements the *Parser* interface. A user of the library can easily write parsers for free-order constructs and does not need to care about checking and reordering the parsed elements. Due to the use of existentially quantified types the implementation of reordering is type safe and hidden from the user.

The underlying parser combinators can be used to handle errors, such as missing or duplicate elements, since the extension inherits their error-reporting or error-repairing properties. Figure 4 shows an example GHCi session that demonstrates error recovery with the UU_Parsing [8] library.

We have shown how our extension can be used to parse XML attributes and Haskell records. Other interesting examples mentioned by Cameron [1] include citation fields in BibTEX bibliographies and attribute specifiers in C declarations. Their pseudo-code algorithm uses a similar strategy. It does not show, however, how to maintain type safety by undoing the change in semantics resulting from reordering, nor can it deal with the presence of separators between free-order constituents.

```
UU_Parsing_Demo> let pOptSym x = pSym x <|> pSucceed '_'
UU_Parsing_Demo> let ptest = pPerms $ (,,) <<$>> pList (pSym 'a')
                                           <<|>> pSym 'b'
                                           <<|>> pOptSym 'c'
                                :: Parser Char (String, Char, Char)
UU_Parsing_Demo> t ptest "acb"
Result:
("a",'b','c')
UU_Parsing_Demo> t ptest ""
Symbol 'b' inserted at end of file; 'b' or 'c' or ('a')* expected.
Result:
("",'b','_')
UU_Parsing_Demo> t ptest "cdaa"
Errors:
Symbol 'd' before 'a' deleted; 'b' or ('a')* expected.
Symbol 'b' inserted at end of file; 'a' or 'b' expected.
Result:
("aa",'b','c')
UU_Parsing_Demo> t ptest "abd"
Errors:
Symbol 'd' at end of file deleted; 'c' or eof expected.
Result:
("a",'b','_')
```

Fig. 4. Example GHCi session (line breaks added for readability)

References

[1] Cameron, R. D., *Extending context-free grammars with permutation phrases*, ACM Letters on Programming Languages and Systems **2** (1993), pp. 85–94.
URL `http://www.acm.org/pubs/toc/Abstracts/1057-4514/176490.html`

[2] Fokker, J., *Functional parsers*, in: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*, LNCS **925** (1995), pp. 1–23.
URL `http://www.cs.uu.nl/~jeroen/article/parsers/parsers.ps`

[3] Hughes, J., *Generalising monads to arrows*, Science of Computer Programming **37** (2000), pp. 67–111.

[4] Hutton, G. and H. Meijer, *Monadic parser combinators*, Journal of Functional Programming **8** (1988), pp. 437–444.

[5] Leijen, D., *Parsec, a fast combinator parser* (2001).
URL `http://www.cs.uu.nl/~daan/parsec.html`

[6] Peyton-Jones, S. and J. Hughes, editors, "Report on the Programming Language Haskell 98," 1999.
URL `http://www.haskell.org/onlinereport`

[7] Swierstra, S. D., *Parser combinators: from toys to tools*, Haskell Workshop, 2000.
URL `http://www.cs.uu.nl/~doaitse/Papers/2000/HaskellWorkshop.pdf`

[8] Swierstra, S. D., *Fast, error repairing parser combinators* (2001).
URL `http://www.cs.uu.nl/groups/ST/Software/UU_Parsing`

[9] Swierstra, S. D. and L. Duponcheel, *Deterministic, error correcting combinator parsers*, in: *Advanced Functional Programming, Second International Summer School on Advanced Functional Programming Techniques*, LNCS **1129** (1996), pp. 184–207.
URL `http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/AFP2.ps`

[10] Wadler, P., *How to replace failure with a list of successes*, in: *Functional Programming Languages and Computer Architecture*, LNCS **201** (1985), pp. 113–128.

FUNCTIONAL PEARL
Pretty Printing with Lazy Dequeues

Olaf Chitil

University of York, UK

Abstract

There are several Haskell libraries for converting tree structured data into indented text, but they all make use of some backtracking. Over twenty years ago Oppen published a more efficient imperative implementation of a pretty printer without backtracking. We show that the same efficiency is also obtainable without destructive updates by developing a similar but purely functional Haskell implementation with the same complexity bounds. At its heart lie two lazy double ended queues.

1 Pretty Printing

Pretty printing is the task of converting tree structured data into text, such that the indentation of lines reflects the tree structure. Furthermore, to minimise the number of lines of the text, substructures are put on a single line as far as possible within a given line-width limit. Here is the result of pretty printing an expression within a width of 35 characters:

```
if True
   then if True then True else True
   else
      if False
         then False
         else False
```

John Hughes [2], Simon Peyton Jones [3], Phil Wadler [7], and Pablo Azero and Doaitse Swierstra [1] have all developed pretty printing libraries for Haskell. A pretty printing library implements the functionality common to a large class of pretty printers. The layout of a subtree does not only depend on its form but also on its context, the remaining tree. A pretty printing library provides functions for *compositionally* defining a transformation of a tree data structure to an abstract document. Finally, such a library has one function to transform a document into the desired text. For example, Wadler's library [7] provides the following functions:

```
text   :: String -> Doc
line   :: Doc
(<>)   :: Doc -> Doc -> Doc
nest   :: Int -> Doc -> Doc
group  :: Doc -> Doc
pretty :: Int -> Doc -> String
```

The function `text` converts a string to an atomic document, the document `line` denotes a (potential) line break, and `<>` concatenates two documents. The function `nest` increases the indentation for all line breaks within its document argument. The function `group` marks the document as a unit to be printed on a single line by converting all its line breaks into single spaces, if this is possible without exceeding the line-width limit. Hence a document describes a set of texts with the same content but different layouts. Finally, the function `pretty` yields the text with the minimal number of lines that does not exceed the given line-width limit.

To obtain the pretty printed expression shown at the beginning of the paper, we only have to compositionally define a function that transforms an abstract expression into a document:

```
data Exp = ETrue | EFalse | If Exp Exp Exp

toDoc :: Exp -> Doc
toDoc ETrue = text "True"
toDoc EFalse = text "False"
toDoc (If e1 e2 e3) =
  group (nest 3 (
    group (nest 3 (text "if" <> line <> toDoc e1)) <> line
    group (nest 3 (text "then" <> line <> toDoc e2)) <> line
    group (nest 3 (text "else" <> line <> toDoc e3))))
```

All previous implementations of Haskell pretty printing libraries use backtracking to determine the optimal layout. They limit backtracking to achieve reasonable efficiency, but their time complexity is not just linear in the size of the input. However, back in 1980 Dereck Oppen published an imperative algorithm for a pretty printer with linear time complexity [6]. At the heart of his algorithm lies an array that is updated in a complex pattern. Are destructive updates necessary to achieve efficiency? No, but the proof is not straightforward. We develop here, starting with a simple but inefficient implementation, step by step, guided by Oppen's algorithm, a similar but purely functional implementation in Haskell.

We implement Wadler's pretty printing interface. The interfaces of Hughes' and Peyton Jones' libraries are more complex, but extending our implementation to support them seems straightforward.[1] Supporting the interface of

[1] Hughes' and Peyton Jones' implementations do not always yield the optimal layout with respect to Hughes' semantics. The layouts that the implementations can produce seem

```
pretty width d = fst (inter False width d)
  where
  inter :: Bool -- in fitting group
        -> Int  -- remaining space on current line
        -> Doc  -- input document
        -> (String   -- formatted output
           ,Int)     -- remaining space on last line
  inter f r (Group d)   = inter (f || fits d r) r d
  inter f r (Text z)    = (z,r - length s)
  inter f r (d1 :<> d2) = (s1 ++ s2,r2)
    where
    (s1,r1) = inter f r  d1
    (s2,r2) = inter f r1 d2
  inter True  r Line    = (" ",r-1)
  inter False r Line    = ("\n",width)
```

Fig. 1. Recursive Implementation using `fits`

Azero's and Swierstra's library is impossible, because it is considerably more expressive than the others.

2 Recursive Implementations

We can define a document simply as an algebraic data type with a constructor for each function that yields a document:

```
data Doc = Text String
         | Line
         | Doc :<> Doc
         | Group Doc

text  = Text
line  = Line
(<>)  = (:<>)
group = Group
```

For simplicity we ignore the function `nest` for the moment. We will see in Section 6 how it can easily be added to the final implementation [2].

We define the function `pretty` as an interpreter of documents that implements the whole functionality. The only slightly difficult case is the formatting of a group. We take Oppen's approach: a group is formatted in a single line if and only if it fits on the remaining space of the line. Previous Haskell libraries

to coincide with those of Wadler's library, except for a difference in nesting discussed in Section 6.

[2] Therefore it is fortunate that separate functions `nest` and `group` exist, although for most applications a single function combining the two is useful.

take a slightly different approach which we discuss in Section 5.

The implementation is given in Figure 1. The function `inter` has to pass some state information: first, if the interpreter is within a group that fits in the remaining space on the current line; second, the size of the remaining space on the current line.

For a `Group` document the interpreter has to determine if the group fits. It obviously fits if the group is within another group that fits. Otherwise, a function `fits` is used to determine if the document `d` fits within the remaining space `r`.

A naïve implementation of `fits` evaluates the width of the document `d` (with a `Line` equal to a single space) and compares the result with the remaining space `r`.

```
fits :: Doc -> Int -> Bool
fits d r = width d <= r

width :: Doc -> Int
width (Group d)   = width d
width Line        = 1
width (Text z)    = length z
width (d1 :<> d2) = width d1 + width d2
```

Unfortunately, the additional traversals of the sub-documents to determine their widths cause the function `pretty` to require exponential time for formatting some documents with nested groups.

The implementation is more efficient if `fits` traverses the document `d` at most up to the width `r`. When that point is reached, it is clear that the document does not fit.

```
fits :: Doc -> Int -> Bool
fits d r = isJust (remaining d r)
  where
  remaining :: Doc -> Int -> Maybe Int
  remaining (Group d) r = remaining d r
  remaining Line      r = r ‘natMinus‘ 1
  remaining (Text z)  r = r ‘natMinus‘ length z
  remaining (d1 :<> d2) r = do
    r1 <- remaining d1 r
    remaining d2 r1

natMinus :: Int -> Int -> Maybe Int
natMinus n1 n2 = if n1 >=n2 then Just (n1-n2) else Nothing
```

This pruning method is similar to Wadler's method of pruning backtracking and hence we obtain the same time complexity: In the worst case it is $O(n \cdot w)$, where n is the size of the input and w the line-width limit. This pruning method has, however, a major drawback. We want to obtain $O(n)$

```
pretty width d = fst3 (inter False width 0 d)
  where
  inter :: Bool -- in fitting group
        -> Int  -- remaining space on current line
        -> Int  -- absolute start position
        -> Doc
        -> (String
           ,Int     -- remaining space on last line
           ,Int)    -- next absolute start position
  inter f r p (Group d) = (s,r',p')
    where
    (s,r',p') = inter (f || p'-p <= r) r p d
  inter f r p (Text z) = (z,r-l,p+l)
    where
    l = length z
  inter f r p (d1 :<> d2) = (s1 ++ s2,r2,p2)
    where
    (s1,r1,p1) = inter f r  p  d1
    (s2,r2,p2) = inter f r1 p1 d2
  inter f r p Line = (o,r',p+1)
    where
    (o,r') = if f then (" ",r-1) else ("\n",width)

fst3 :: (a,b,c) -> a
fst3 (x,_,_) = x
```

Fig. 2. Recursive Implementation that Returns the Next Start Position

time complexity, independent of w, but a further optimisation is not in sight. The optimisation leads into a cul-de-sac.

On the other hand, we can obtain a linear implementation[3] from the naïve definition by applying the tupling transformation: instead of a separate function that traverses a document to determine its width, the interpreter `inter` can determine the width in addition to its other tasks.

Because groups can be nested, it is not obvious how `inter` should be defined to return the width of its input document. The solution is to introduce an absolute measure of a document's position. The absolute position gives the column in which the document would start, if the *whole* document that is passed to `pretty` was formatted on a single line. The function `inter` receives the start position as argument and returns the next start position which is

[3] The use of `(++)` for formatting a document `d1 :<> d2` actually leads to quadratic time complexity. However, we can use the same optimisation as is used in the Haskell class `Show` to assemble the result string in linear time: `inter` has to return a value of type `String -> String` instead of just a `String` and list concatenation becomes function composition. We do not apply this optimisation here to not to distract from the main issues.

free after its input document. The difference between the two positions is the width of the input document.

Figure 3 shows the new implementation. It takes advantage of the lazy evaluation of the recursive call of `inter`: the result position `p'` is passed as part of the first argument. The implementation has linear time complexity, because a computation spends only constant time on each document constructor.

3 Iterative Implementations

Unfortunately our current implementation has a major drawback: only after the full traversal of a group it is known if the group fits on the remaining line. Hence a computation produces most of the output string for a group only after it has traversed the whole group. The time delay may not be a problem in practice, but the delayed computation of the output uses memory space linear in the size of the group. We would like our pretty printer to use only a small amount of space which is independent of the formatted document. Because the document will usually be constructed lazily or even be read sequentially from a file, pretty printing a document element should also only require a limited look-ahead into the remaining document.

The recursive implementation with pruning has the desired space behaviour, but it is not obvious how it can be married with the time efficient tuppled implementation. The problem is that pruning at a certain width and the tree structured recursion of `inter` do not fit together. Hence we move from tree structured recursion to sequential iteration. Following Oppen we represent the document as a list of tokens:

```
type Doc = [Token]

data Token = Text String | Line | Open | Close
```

A group is represented as the sequence of an `Open` token, the sequence of the grouped document and a final `Close` token. Translation from the old document data type to the new one is straightforward. Alternatively, an efficient direct construction can be defined in continuation-passing style. We assume in the following that documents are well-formed, that is, the `Open` and `Close` tokens are well-bracketed.

We redefine the tuppled implementation for the token sequence. Because we no longer use recursion that follows the structure of the document, we have to make the nesting structure of the groups explicit by using stacks. For every group the interpreter has to determine the next absolute start position. Hence it has to return a stack of positions — represented by a list. At the end of a group the interpreter needs to know if there is a surrounding fitting group. For this purpose we could pass a stack of booleans, but a natural number which states how deep the interpreter is in fitting groups is simpler. The interpreter no longer needs to return the size of the space that remains on the last line,

```
pretty width d = fst (inter 0 width 0 d)
  where
  inter :: Int  -- depth of fitting groups
        -> Int  -- remaining space on current line
        -> Int  -- absolute start position
        -> [Token]
        -> (String
           ,[Int])  -- next absolute start positions
  inter f r p (Open:ts) = (s,es')
    where
    e':es' = es
    (s,es) = inter (if f>0 then succ f
                           else (if (e'-p)<=r then 1 else 0))
              r p ts
  inter f r p (Close:ts) = (s,p:es)
    where
    (s,es) = inter (pred f) r p ts
  inter f r p (Text z : ts) = (z ++ s,es)
    where
    (s,es) = inter f (r-l) (p+l) ts
    l = length z
  inter f r p (Line:ts) = (o:s,es)
    where
    (o,r') = if f>0 then (' ',r-1) else ('\n',width)
    (s,es) = inter f r' (p+1) ts
  inter f r p [] = ("",[])
```

Fig. 3. Iterative Implementation that Returns Next Start Positions

because the document constructor `(:<>)` has vanished. Figure 3 shows the implementation.

To see how the implementation works, consider a simple example. The following table shows the values of most variables for each iteration step. The document is given in the top row. We assume that the strings of the `Text` tokens have length 1. The line-width limit is 3. The arrows indicate in which directions values are passed. The inner group fits on a single line whereas the outer one does not.

	`Open`		`Text`		`Line`		`Open`		`Text`		`Line`		`Text`		`Close`		`Close`
`f`	0	→	0	→	0	→	0	→	1	→	1	→	1	→	1	→	0	→	0
`r`	3	→	3	→	2	→	3	→	3	→	2	→	1	→	0	→	0	→	0
`p`	0	→	0	→	1	→	2	→	2	→	3	→	4	→	5	→	5	→	5
`es`	[]	←	[5]	←	[5]	←	[5]	←	[5,5]	←	[5,5]	←	[5,5]	←	[5,5]	←	[5]	←	[]

Laziness leads to a kind of co-routine computation. The applications of `inter` to `Close` tokens can be identified with a *front* process and the other applications of `inter`, especially to `Open` tokens, as a *back* process. The front process determines the position of each `Close` token and passes this information in the variable `es` backwards to the back process. The back process determines the position of each `Open` token and the remaining space on the line. Together with the end position obtained from the front process it can determine if the group still fits. Despite this image, however, there is no simple implementation of the interpreter through two real processes, because the communication channel between them is a stack.

This iterative implementation has the same time and space behaviour as our last recursive implementation.[4] However, in the example we can already see that the outer group does not fit, when we reach the absolute position 4 (`p` = `4`). The group does not fit, because it starts at position 0 and the line space remaining for it is 3 (the values of `p` and `r` at the first `Open` token). Unfortunately, the interpreter does not know these values 0 and 3 when it reaches position 4.

Therefore we introduce an additional argument that is passed from left to right: a stack which holds for each `Open` token of a group the sum of the absolute position and the space remaining on the line; we call this sum the group's maximal end position. We also simplify the resulting stack of end positions to a stack of booleans instead. At a `Close` token we just have to take the top position from the maximal end positions stack, compare it with the current position, and push the result on the stack of booleans which we return. Figure 4 shows the modified implementation and the following table shows the values of the new variables for our example document.

	`Open`	`Text`	`Line`	`Open`	`Text`	`Line`	`Text`	`Close`	`Close`
`f`	0 →	0 →	0 →	0 →	1 →	1 →	1 →	1 →	0 → 0
`r`	3 →	3 →	2 →	3 →	3 →	2 →	1 →	0 →	0 → 0
`p`	0 →	0 →	1 →	2 →	2 →	3 →	4 →	5 →	5 → 5
`ps`	[] →	[3] →	[3] →	[3] →	[5,3] →	[5,3] →	[5,3] →	[5,3] →	[3] → []
`fs`	[] ←	[F] ←	[F] ←	[F] ←	[T,F] ←	[T,F] ←	[T,F] ←	[T,F] ←	[F] ← []

Now for the optimisation. At position 4 the information for determining that the outer group does not fit is now available. The bottom of the stack `ps` contains the maximal end position of the outermost group. Because it is 3, smaller than the current position, the outermost group cannot fit. Hence

[4] In contrast to the recursive implementation the use of `(++)` does not lead to quadratic time complexity here, because the first argument of `(++)` is not constructed by a recursive call of `inter`. No optimisation of list concatenation is necessary. Compare for Footnote 3.

```
pretty :: Int -> Doc -> String
pretty width d = fst (inter 0 width 0 [] d)
  where
  inter :: Int  -- depth of fitting groups
        -> Int  -- remaining space on current line
        -> Int  -- absolute start position
        -> [Int]  -- maximal end positions
        -> [Token]
        -> (String
           ,[Bool])  -- fitting infos
  inter f r p ps (Open:ts) = (s,fs')
    where
    f':fs' = fs
    (s,fs) = inter (if f>0 then succ f
                           else (if f' then 1 else 0))
                 r p (r+p:ps) ts
  inter f r p ps (Close:ts) = (s,(p<=p'):fs)
    where
    p':ps' = ps
    (s,fs) = inter (pred f) r p ps' ts
  inter f r p ps (Text z : ts) = (z ++ s,fs)
    where
    (s,fs) = inter f (r-l) (p+l) ps ts
    l = length z
  inter f r p ps (Line:ts) = (o:s,fs)
    where
    (o,r') = if f>0 then (' ',r-1) else ('\n',width)
    (s,fs) = inter f r' (p+1) ps ts
  inter f r p [] [] = ("",[])
```

Fig. 4. Iterative Implementation that Returns Fitting Information in a Stack

at this point we can already remove the end position from the bottom of the "stack" `ps` and add `False` (F) to the bottom of the boolean "stack" `fs`:

$$\begin{array}{lccccccccccccccccccc}
 & & \texttt{Open} & & \texttt{Text} & & \texttt{Line} & & \texttt{Open} & & \texttt{Text} & & \texttt{Line} & & \texttt{Text} & & \texttt{Close} & & \texttt{Close} & \\
\texttt{f} & 0 & \rightarrow & 0 & \rightarrow & 0 & \rightarrow & 0 & \rightarrow & 1 & \rightarrow & 1 & \rightarrow & 1 & \rightarrow & 1 & \rightarrow & 0 & \rightarrow & 0 \\
\texttt{r} & 3 & \rightarrow & 3 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 3 & \rightarrow & 2 & \rightarrow & 1 & \rightarrow & 0 & \rightarrow & 0 & \rightarrow & 0 \\
\texttt{p} & 0 & \rightarrow & 0 & \rightarrow & 1 & \rightarrow & 2 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 & \rightarrow & 5 & \rightarrow & 5 & \rightarrow & 5 \\
\texttt{ps} & [] & \rightarrow & [3] & \rightarrow & [3] & \rightarrow & [3] & \rightarrow & [5,3] & \rightarrow & [5,3] & \rightarrow & \underline{[5]} & \rightarrow & \underline{[5]} & \rightarrow & \underline{[]} & \rightarrow & [] \\
\texttt{fs} & [] & \leftarrow & [F] & \leftarrow & [F] & \leftarrow & [F] & \leftarrow & [T,F] & \leftarrow & [T,F] & \leftarrow & \underline{[T]} & \leftarrow & \underline{[T]} & \leftarrow & \underline{[]} & \leftarrow & []
\end{array}$$

4 Lazy Dequeues

Obviously we no longer simply use `ps` and `fs` as stacks but as double ended queues. Fortunately we find in Okasaki's book [4] the Haskell implementation of the banker's dequeue. If used in a single threaded manner as here, each operation runs in $O(1)$ amortized time. Hence our optimised iterative implementation still has linear time complexity.

There is a problem left: the intention of our optimisation is to enable the interpreter to determine with a limited look-ahead if a group fits. By looking at the bottom of the end positions dequeue, the decision can be made with a look-ahead of at most the line-width limit. However, the interpreter adds this information to the *bottom* of the dequeue `fs` and removes it from the *top* of the dequeue `fs` at the `Open` token. To avoid further look-ahead these operations must work without fully evaluating the dequeue `fs`. That means in the example that the interpreter must be able to add and remove `False` (F) without ever "touching" the `True` (T).

We can add and remove elements from a *list* without "touching" the remaining list, but can we do the same for *dequeues*? Yes, within the special context of our pretty printer we can.

The two dequeues `ps` and `fs` are accessed in perfect synchrony: Each time an operation is performed on one dequeue, exactly the inverse operation is performed on the other dequeue. Hence we combine the operations on the two dequeues. So

```
cons :: a -> Q1 a -> Q2 b -> (Q1 a, b, Q2 b)
```

adds an element to the front of the first dequeue and splits the second dequeue into its front element and the tail dequeue;

```
rview :: Q1 a -> Q2 b -> b -> (Q1 a, a, Q2 b)
```

splits the first dequeue into an initial dequeue and a rear element and adds an element to the rear of the second dequeue;

```
lview :: Q1 a -> b -> Q2 b -> (a, Q1 a, Q2 b)
```

splits the first dequeue into its front element and its tail dequeue and adds an element to the front of the second dequeue.

The dequeue `ps` is passed from left to right and the dequeue `fs` is passed from right to left. Furthermore, both dequeues are empty at the beginning and at the end of interpreting the token sequence. Hence the internal structures of the two dequeues are the same at each interpretation step. So we can use the knowledge about the internal structure of `ps`, which may be fully evaluated, to apply an operation to `fs` without evaluating any part of `fs`.

Because we use the structure of `fs` to manipulate `ps`, the operations `cons` and `lview` are not identical up to swapping of arguments and result elements but have different strictness properties. We define separate types for the two dequeues to stress the asymmetry and enable a minor optimisation.

```
pretty width d = fst (inter 0 width 0 empty1 d)
  where
  inter :: Int  -- depth of fitting groups
        -> Int  -- remaining space on current line
        -> Int  -- absolute start position
        -> Q1 Int  -- maximal end positions
        -> [Token]
        -> (String
           ,Q2 Bool)  -- fitting infos
  inter f r p ps (Open:ts) = (s,fs')
    where
    (ps',f',fs') = cons (r+p) ps fs
    (s,fs) = inter (if f>0 then succ f
                           else (if f' then 1 else 0))
               r p ps' ts
  inter f r p ps (Close:ts)
    | isEmpty1 ps = inter (pred f) r p ps ts
    | otherwise   = (s,fs')
    where
    (_,ps',fs') = lview ps True fs
    (s,fs) = inter (pred f) r p ps' ts
  inter f r p ps (Text z : ts) = (z ++ s,fs)
    where
    (s,fs) = prune f (r-l) (p+l) ps ts
    l = length z
  inter f r p ps (Line : ts) = (o : s,fs)
    where
    (o,r') = if f>0 then (' ',r-1) else ('\n',width)
    (s,fs) = prune f r' (p+1) ps ts
  inter _ _ _ _ [] = ("",empty2)

  prune :: Int -> Int -> Int -> Q1 Int -> [Token]
        -> (String,Q2 Bool)
  prune f r p ps ts
    | isEmpty1 ps || p <= p' = inter f r p ps ts
    | otherwise              = (s,fs')
    where
    (ps',p',fs') = rview ps fs False
    (s,fs) = prune f r p ps' ts
```

Fig. 5. Iterative Implementation with Lazy Dequeues

Without going into details of the implementation we note that a banker's dequeue is represented by two lists. One holds the top elements and the other the bottom elements of the dequeue. An invariant requires that the lengths of the lists are not too far apart. When addition or removal of an element threatens to invalidate the invariant, list elements are moved from one list to the other. The only operations applied to the two lists are `reverse`, `(++)` and `splitAt`. We can easily define lazy variants of these standard list functions which — given the length of a list argument or a result — construct the list structure of the result without demanding evaluation of any of its list arguments. Only demanding some list element of the result will lead to more demand of the arguments. Here, for example, is the lazy variant of `(++)`:

```
lappend :: Int -> [a] -> [a] -> [a]

lappend 0 _ zs = zs
lappend n xs zs = y : lappend (n-1) ys zs
  where
  y:ys = xs
```

Using the lazy variants to implement the dequeue operations `cons`, `rview` and `lview`, we obtain the required lazy dequeues. The full implementation is given in Appendix B.

With these dequeues we can finally define our time and space efficient pretty printer. Figure 5 shows the implementation. For each `Text` and `Line` token the function `prune` tests if some surrounding groups do not fit. Hence, if when reaching a `Close` token the maximal positions dequeue is non-empty, then the group certainly fits.

5 Overfull Lines

We took Oppen's approach to formatting a group: a group is formatted in a single line if and only if it fits on the remaining space of the line. Unfortunately this approach may yield layouts with lines wider than the width limit, although a fitting layout exists. A group that still fits on a line may be followed by further text without a separating `line`. Because there is no `line`, the text has to be added to the current line, even if does not fit. Breaking the group may have avoided the problem.

Our solution is to normalise the token list with respect to the following two rewriting rules before applying `pretty`:

```
Close, Text s  ⇒  Text s, Close
Open, Text s   ⇒  Text s, Open
```

The normalised token list has the property that between a `Close` token and the next `Text` token there is always a `Line` token. Hence the aforementioned problem can no longer occur. Like Wadler's pretty printer ours always produces a fitting layout if it exists. Note that rewriting only moves `Text` tokens in and out of groups. Hence the set of `lines` "belonging" to each group, which are either all formatted as new lines or all as spaces, is unchanged. So rewriting does not change the set of texts described by a document.

Normalisation can be implemented by a linear, straightforward traversal of the token list, keeping track of the number of currently opened and closed groups. Note that analogous normalisation of the tree structured documents, which we used in Section 2, is hard to implement efficiently.

6 Indentation

To complete the library we still have to implement the function `nest`. There are different interpretations of the expression `nest` n. In Wadler's library it increases the current left margin by n columns whereas in Oppen's pretty printer (and other libraries) it sets the left margin to the current column position plus n. We can easily implement either of these variants by introducing two new tokens

```
data Token =  ... | NestOpen Int | NestClose
```

and interpreting them appropriately in the function `inter` which also acquires a stack of current left margins as additional argument. Alternatively, we can implement Wadler's variant just as he does by a preceding transformation which moves the indentation information to every `Line` token.

7 Conclusions

We have developed a purely functional pretty printer that only requires time linear in the size of the input/output and space linear in the line-width limit. It demonstrates that we do not need updateable data structures to achieve the same efficiency as Oppen's imperative algorithm and also throws some light on this rather monolithic algorithm. Oppen's algorithm consists of two parts which also work together in a co-routine like fashion. For communication between the two processes an array is used as dequeue. The difference is that dequeue elements are updated where our implementation uses a second, synchronous, lazy dequeue.

We have obtained a useful library. An extended version is part of the distribution of the Haskell compiler nhc98[5]. The compiler itself uses the library to provide pretty printing of the abstract syntax tree after any compiler phase.

On a general level the derivation of our pretty printing implementation demonstrates two points in algorithm design: First, defining a function recursively along the structure of the main data type may not lead to the best solution. We sometimes have to leave the limits of an implicit recursive control

[5] http://www.cs.york.ac.uk/fp/nhc98

structure by making it explicit as data structure. A data structure can be replaced by a more flexible one (here a stack by a dequeue). [6] Second, there are useful lazy variants of non-inductively defined abstract data structures such as dequeues.

Acknowledgements

I thank Colin Runciman and the anonymous referees for many suggestions for improving this paper.

The work reported in this paper was supported by the Engineering and Physical Sciences Research Council of the United Kingdom under grant number GR/M81953.

References

[1] Pablo Azero and Doaitse Swierstra. Optimal pretty-printing combinators. http://www.cs.uu.nl/groups/ST/Software/PP/, 1998.

[2] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925. Springer Verlag, 1995.

[3] Simon Peyton Jones. A pretty printer library in Haskell. Available from http://research.microsoft.com/Users/simonpj/downloads/pretty-printer/pretty.html, 1997.

[4] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, Cambridge, UK, 1998.

[5] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *International Conference on Functional Programming*, pages 131–136, 2000.

[6] Dereck C Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[7] Philip Wadler. A prettier printer. Available from http://cm.bell-labs.com/cm/cs/who/wadler/topics/language-design.html, 1998.

[6] A related well-known example is breadth-first traversal of a tree: Depth-first traversal can be implemented easily by direct recursion. However, instead of recursion we can also use a stack. Replacing the stack by a queue we obtain breadth-first traversal of a tree [5].

A The Complete Pretty Printing Library

```
module Pretty (Doc,text,line,(<>),group,nestInc,nestSet,pretty)
  where

import LazyDequeue

-- Public interface:

-- precondition: string contains no formatting characters
-- \n \t etc.
text   :: String -> Doc
text s = Doc (Text s :)

line   :: Doc
line = Doc (Line :)

(<>)   :: Doc -> Doc -> Doc
Doc l1 <> Doc l2 = Doc (l1 . l2)

group  :: Doc -> Doc
group (Doc l) = Doc ((Open :) . l . (Close :))

-- increment indentation
-- the delta (i) may be any integer,
-- but runtime error if indentation becomes negative
nestInc :: Int -> Doc -> Doc
nestInc i (Doc l) =
  Doc ((NestIncOpen i :) . l . (NestIncClose :))

-- set indentation to current column plus given increment
nestSet :: Int -> Doc -> Doc
nestSet i (Doc l) =
  Doc ((NestSetOpen i :) . l . (NestSetClose :))

pretty :: Int -> Doc -> String
pretty width (Doc l) =
  fst (inter width [0] 0 width 0 empty1
        (normalise 0 0 (l [])))
```

```
-- Internal parts:

newtype Doc = Doc ([Token] -> [Token])
data Token = Text String
           | Line
           | Open
           | Close
           | NestIncOpen Int
           | NestIncClose
           | NestSetOpen Int
           | NestSetClose

-- normalise the stream of tokens with respect to the rules
--  Open, Close  ==>
--  Open, t      ==>  t, Open
--  Close, t     ==>  t, Close
-- for all tokens t except Line, Open and Close
normalise :: Int  -- number of deferred opening brackets
          -> Int  -- number of deferred closing brackets
          -> [Token]
          -> [Token]
normalise o c [] = replicate c Close
  -- there should be no deferred opening brackets
normalise o c (Open : ts) = normalise (o+1) c ts
normalise o c (Close : ts)
  | o == 0    = normalise o (c+1) ts
  | otherwise = normalise (o-1) c ts
normalise o c (t@(NestIncOpen _) : ts) = t : normalise o c ts
normalise o c (t@NestIncClose : ts) = t : normalise o c ts
normalise o c (t@(NestSetOpen _) : ts) = t : normalise o c ts
normalise o c (t@NestSetClose : ts) = t : normalise o c ts
normalise o c (t@(Text _) : ts) = t : normalise o c ts
normalise o c (t@Line : ts) =
  rep c Close . rep o Open . (t :) . normalise 0 0 $ ts

inter :: Int  -- width
      -> [Int] -- left margins (current on top)
      -> Int   -- depth of fitting groups
      -> Int   -- remaining space on current line
      -> Int   -- absolute start position
      -> Q1 Int  -- maximal end positions
      -> [Token]
      -> (String
         ,Q2 Bool)  -- fitting infos
```

```
inter _ _ _ _ _ _ _ [] = ("",empty2)
inter width ms f r p ps (Open:ts) = (s,fs')
  where
  (ps',f',fs') = cons (r+p) ps fs
  (s,fs) = inter width ms (if f>0 then succ f
                                else (if f' then 1 else 0))
             r p ps' ts
inter width ms f r p ps (Close:ts)
  | isEmpty1 ps = inter width ms (pred f) r p ps ts
  | otherwise   = (s,fs')
  where
  (_,ps',fs') = lview ps True fs
  (s,fs) = inter width ms (pred f) r p ps' ts
inter width ms f r p ps (Text t : ts) = (t ++ s,fs)
  where
  (s,fs) = prune width ms f (r-l) (p+l) ps ts
  l = length t
inter width ms@(m:_) f r p ps (Line : ts) = (o s,fs)
  where
  (o,r') = if f>0 then ((' ':),r-1)
                  else (('\n':) . rep m ' ', width-m)
  (s,fs) = prune width ms f r' (p+1) ps ts
inter width ms@(m:_) f r p ps (NestIncOpen i : ts) =
  inter width (m+i : ms) f r p ps ts
inter width (_:ms) f r p ps (NestIncClose : ts) =
  inter width ms f r p ps ts
inter width ms@(m:_) f r p ps (NestSetOpen i : ts) =
  inter width (width-r+i : ms) f r p ps ts
inter width (_:ms) f r p ps (NestSetClose : ts) =
  inter width ms f r p ps ts

prune :: Int -> [Int] -> Int -> Int -> Int -> Q1 Int -> [Token]
      -> (String,Q2 Bool)
prune width ms f r p ps ts
  | isEmpty1 ps || p <= p' = inter width ms f r p ps ts
    -- note: to evaluate p' fs does not need to be evaluated
  | otherwise              = (s,fs')
  where
  (ps',p',fs') = rview ps fs False
  (s,fs) = prune width ms f r p ps' ts

-- continuation style variant of ‘replicate’
rep :: Int -> a -> [a] -> [a]
rep n x rs = if n <= 0 then rs else x : rep (n-1) x rs
```

B Implementation of the Lazy Dequeues

```
module LazyDequeue(Q1,Q2,empty1,empty2,isEmpty1
                  ,cons,rview,lview) where

-- q12List (Q1 _ f _ r) = f ++ reverse r
-- Q1 also contains the lengths of the two lists
-- Q2 does not contain lengths
data Q1 a = Q1 !Int [a] !Int [a]
data Q2 a = Q2 [a] [a]

reverse1 :: Q1 a -> Q1 a
reverse1 (Q1 lenf f lenr r) = Q1 lenr r lenf f

reverse2 :: Q2 a -> Q2 a
reverse2 (Q2 f r) = Q2 r f

empty1 = Q1 0 [] 0 []
empty2 = Q2 [] []

isEmpty1 (Q1 lenf _ lenr _) = lenf + lenr == 0

-- Keep lengths of the two lists in balance
check :: Int -> [a] -> Int -> [a] -> Q2 b -> (Q1 a, [b], [b])
check lenf f lenr r q2 =
  if lenf > balanceConstant * lenr + 1 then
    let
      len = lenf + lenr
      lenf' = len `div` 2
      lenr' = len - lenf'
      (f', rf') = splitAt lenf' f
      (r2, rf2) = lsplitAt lenr r2'
    in (Q1 lenf' f' lenr' (r ++ reverse rf')
       ,lappend lenf' f2' (lreverse (lenr'-lenr) rf2)
       ,r2)
   else
    (Q1 lenf f lenr r, f2', r2')
  where
  Q2 f2' r2' = q2
  balanceConstant = 3 :: Int
```

```
cons :: a -> Q1 a -> Q2 b -> (Q1 a, b, Q2 b)
cons x (Q1 lenf f lenr r) q2' = (q', head f2, Q2 (tail f2) r2)
  where
  (q', f2, r2) = check (lenf+1) (x:f) lenr r q2'

rview :: Q1 a -> Q2 b -> b -> (Q1 a, a, Q2 b)
rview (Q1 _ (x:_) _ []) q2' y = (empty1, x, Q2 [y] [])
rview (Q1 _ [] _ []) _ _ = error "empty dequeue"
rview (Q1 lenf f lenr (x:r)) q2' y = (q', x, Q2 f2 (y:r2))
  where
  (q', f2, r2) = check lenf f (lenr-1) r q2'

lview :: Q1 a -> b -> Q2 b -> (a, Q1 a, Q2 b)
lview q1 y q2 = (x, reverse1 q1', reverse2 q2')
  where
  (q1', x, q2') = rview (reverse1 q1) (reverse2 q2) y

-- The lazy variants of standard list functions:

-- The first argument gives the length
-- of the argument/result list.
lreverse :: Int -> [a] -> [a]
lreverse n xs = lreverseAcc n xs []
  where
  lreverseAcc 0 _ acc = acc
  lreverseAcc n xs acc = lreverseAcc (n-1) ys (y:acc)
    where
    y:ys = xs

-- The first argument gives the length of the second argument.
lappend :: Int -> [a] -> [a] -> [a]
lappend 0 _ zs = zs
lappend n xs zs = y : lappend (n-1) ys zs
  where
  y:ys = xs

-- The first argument gives the position at which the input list
-- shall be split. The list must be at least that long.
lsplitAt :: Int -> [a] -> ([a], [a])
lsplitAt 0 xs = ([],xs)
lsplitAt n ys = (x:xs',xs'')
  where
  x:xs = ys
  (xs',xs'') = lsplitAt (n-1) xs
```

***

Playing by the Rules: Rewriting as a practical optimisation technique in GHC

Simon Peyton Jones [a,1] Andrew Tolmach [b,2,3] Tony Hoare [a,4]

[a] *Microsoft Research Ltd, St George House, 1 Guildhall St, Cambridge CB2 3NH, England*

[b] *Dept. of Computer Science, Portland State University, P.O. Box 751, Portland, OR 97207, USA*

Abstract

We describe a facility for improving optimization of Haskell programs using rewrite rules. Library authors can use rules to express domain-specific optimizations that the compiler cannot discover for itself. The compiler can also generate rules internally to propagate information obtained from automated analyses. The rewrite mechanism is fully implemented in the released Glasgow Haskell Compiler.

Our system is very simple, but can be effective in optimizing real programs. We describe two practical applications involving short-cut deforestation, for lists and for rose trees, and document substantial performance improvements on a range of programs.

1 Introduction

Optimising compilers perform program transformations that improve the efficiency of the program. However, a compiler can only use relatively shallow reasoning to guarantee the correctness of its optimisations. In contrast, the programmer has much deeper information about the program and its intended behaviour. For example, a programmer may know that

```
integerToInt (intToInteger x) = x
```

(where `Integer` is the type of infinite-precision integers, and `Int` is 32-bit integers), but the compiler has little chance of working this out for itself. While programmers are unlikely to write such expressions themselves, they can

[1] Email: `simonpj@microsoft.com`
[2] Work performed in part while visiting Microsoft Research Ltd.
[3] Email: `apt@cs.pdx.edu`
[4] Email: `thoare@microsoft.com`

easily appear when aggressive inlining brings together code that was written separately.

In this paper we explore a very simple idea: *allow the programmer to specify program properties that the compiler can use to improve performance, by treating each property as a rewrite rule.* In effect, we give the programmer the ability to extend the compiler with domain-specific optimisations, giving it specialised knowledge about the particular vocabulary of functions that are used heavily in a particular program. Our setting is that of the purely functional language Haskell, because the lack of side effects makes it possible to state many properties simply, without complex side conditions, and to exploit them using only local information.

We make the following contributions:

- We describe a concrete design, which is fully implemented in the released Glasgow Haskell Compiler, an optimising compiler for Haskell (Section 2; Section 4).
- We describe two practical applications of the technique, one to perform list fusion in the Haskell standard Prelude (Section 3) and other to perform tree fusion in an application-specific library (Section 6).
- We show that rewrite rules can also be generated automatically as a result of compiler analyses, and then constitute a useful way to exploit specialised versions of functions (Section 5).

The idea of allowing the programmer to specify domain-specific compiler extensions is not new (Section 7), but it has not yet been widely successful. Our principal selling point is *simplicity*. Rewrite rules are expressed declaratively using the syntax of Haskell itself, and not in a separate meta-language. They use very simple first-order pattern matching, have no side conditions, and are applied using a trivial strategy. Yet they are effective in real programs, assuming some cooperation from library writers.

We currently make no attempt to verify that programmer-specified rules are consistent with the underlying function definitions that they purport to describe. Rather, a programmer who adds a rule implicitly incurs a proof obligation, in much the same way as a user of GHC's `unsafePerformIO`. In addition, a rule's effect on program performance can be tricky to predict. For these reasons, the rules mechanism in its present form is primarily intended for use by "expert" programmers and library authors, who understand GHC's optimization behavior.

Having the rules explicitly codified does, however, raise the possibility of feeding the same program into a theorem prover, and having it prove that the rules are consistent with the implementation, perhaps with some human assistance — although we have not explored this avenue so far.

Adding explicit equational properties to programs has already been advocated for other purposes. They can serve to document the intended behavior of the program, independently of the implementation, and have been used to explore efficient algorithms and as a design methodology that reduces the incidence of programming error [4]. Another advantage may be reaped in testing and debugging of programs, where they can play the role of a test oracle [8]. Perhaps the additional incentive of efficiency gains in compilation will help convince the world that equational specification is a worthwhile part of the programming process.

2 The basic idea

Consider the familiar `map` function that applies a function to each element of a list. Written in Haskell, `map` looks like this:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Now suppose that the compiler encounters the following call of `map`:

```
map f (map g xs)
```

We know that this expression is equivalent to

```
map (f . g) xs
```

(where "`.`" is function composition), and we know that the latter expression is more efficient than the former because there is no intermediate list. But the compiler has no such knowledge.

One possible rejoinder is that the compiler should be smarter — but the expert programmer will always know things that the compiler cannot figure out. Another suggestion is this: allow the programmer to communicate such knowledge directly to the compiler. That is the direction we explore here.

The Glasgow Haskell Compiler (GHC) allows the programmer to add a *rule* to the program thus:

```
{-# RULES
  "map/map" forall f g xs.
            map f (map g xs) = map (f . g) xs
#-}
```

The "`{-# ... #-}`" brackets enclose a *pragma*, which is ignored by a non-optimising compiler. The `RULES` keyword identifies the pragma as defining a rewrite rule. The `"map/map"` part is an arbitrary string that names the rule; this name is used when reporting which rules fired during a compilation run in diagnostic mode. The body of the rule expresses the identity that

```
map f (map g xs) = map (f . g) xs
```

while the `forall` part identifies which of the variables in the rule body are universally quantified (`f`, `g`, and `xs` in this case), and which are constants bound elsewhere (`map` in this case).

The general form of a programmer-specified rule is

$$\texttt{"}\mathit{name}\texttt{" forall } (v_1\texttt{::}t_1) \ \ldots\ (v_m\texttt{::}t_m)\texttt{.} f \ e_1 \ \ldots \ e_n \texttt{ = } e$$

for $m, n \geq 0$. Here *name* is a string identifying the rule, as described above. The v_i are bound variables with associated types t_i; the types can be omitted unless required to make the rule type-check, and the entire `forall` clause can be omitted if there are no bound variables. f is an *unquantified* function or constant identifier (i.e., not one of the `forall`'d variables), and the e_i are arbitrary Haskell expressions. A `RULES` pragma can occur only at the top level of the program, and all the free variables of the rule, on both sides of the equation, must be in scope.

One can regard the rules for a function as extra (redundant) equations defining the function, thus:

```
map f []          = []
map f (x:xs)      = f x : map f xs
map f (map g xs) = map (f . g) xs
```

Unlike ordinary defining equations, of course, rules are not restricted to having constructors in the patterns on the left hand side.

Rewrite rules express identities that the programmer knows to be true, but GHC also assumes that they are *oriented*, so that the right hand side is preferable to the left. Throughout compilation, GHC tries to spot instances of the left hand side of a rule, and rewrite that call to the right hand side.

2.1 Assumptions

The ability to add rewrite rules to a program is a pretty powerful weapon, and raises a host of issues. In particular:

- GHC makes no attempt to verify that the rule is consistent with the underlying function definitions, apart from ensuring that the left and right hand sides of the rule have the same type. The whole point is that the rule asserts something that GHC is not smart enough to work out for itself! Moreover, if rule and implementation disagree, the implementation is just as likely to be wrong as the rule, perhaps even more so.

  Indeed, we might not even want the rule to be "true" in a concrete sense! For example, consider an abstract data type for sets. It is sound to give a rule expressing the fact that `union` on sets is commutative. But suppose our implementation represents a set by an unordered list. Then the concrete representation of `(a ‘union‘ b)` may differ from `(b ‘union‘ a)`, even though they represent the same sets.
- GHC makes no attempt to ensure that the right hand side is more "efficient" than the left hand side. One might like to say "simply write down some true properties, and the compiler will use them to optimise the program", but that is well beyond what we offer. Instead, as we discuss in Section 4, we

rely on the (fallible) programmer to specify oriented rewrite rules, and even a simple rewrite strategy. Using rules effectively therefore requires some understanding of how GHC works.

- GHC makes no attempt to ensure that the set of rules is confluent, or even terminating. For example, the following rule will send GHC into an infinite loop if it encounters a call to `foo`.

  ```
  {-# RULES
      "commute"  forall x y. foo x y = foo y x
  #-}
  ```

  There is a considerable literature on proving the confluence or termination of sets of rewrite rules; in particular, commutativity and associativity have received special study [3]. However, for us matters are seriously complicated by the other automatic rewrites that the compiler performs (beta reduction, inlining, case switching, let-floating, etc. [32]), so we are not able to take direct advantage of this work.

  For an optimising compiler, confluence seems too strong, since that would implausibly suggest a canonical optimised form for a program. Termination is certainly important, but has not proved to be a problem in practice.

2.2 Restrictions

As noted above, the pattern on the left hand side of a rule must be a function application (for some fixed function) or a constant. Here, for example, is a plausible rule that we cannot write:

```
{-# RULES
  "let/let"  forall x y e1 e2 e3.   -- ILLEGAL!
        let { x = let { y = e1 } in e2 } in e3
        = let { y = e1 } in let { x = e2 } in e3
#-}
```

The rule is illegal because the left hand side is not a function application. This restriction has two advantages. First, it underpins the idea introduced above, that a rewrite rule is simply an extra (redundant) equation defining a function. Second, it makes rule matching much more efficient, because the rules can be indexed by the function on the left hand side. At each call of `f`, GHC need only check matches for rules for `f`. If the left hand side of a rule could instead be an arbitrary expression, matching would likely be much less efficient.

The function-application restriction does mean that rules cannot be used to replace many of GHC's built-in transformations. Inlining, let-floating, beta reduction, case swapping, case elimination, and so on are all too complex to explain using our restricted language of rules. There are, however, some compiler transformations – such as specialisation – for which rules do prove directly useful, as we discuss in Section 5.

2.3 Library writers and library clients

Reading these assumptions and restrictions, one might reasonably ask: are rewrite rules going to be of practical use? It is certainly easy to shoot oneself in the foot.

For this reason, we regard a set of rewrite rules as something much more like *a domain-specific compiler extension* than a general programming paradigm. We expect rewrite rules to be written mainly by the author of a library. Such authors often go to great lengths to craft efficient data structures and algorithms. Rewrite rules give them the ability to explain deep truths about their code to the compiler, and thereby extend its ability to optimise client programs. We assume also a willingness to cooperate in the optimisation, to the extent of adapting library code to take advantage of the optimisation rules, as well as the other way round. In return, we hope to preserve a level of simplicity, in which the correctness of the optimisation rules (but not their effectiveness, unfortunately) is as easy to establish as that of all the other clauses in a declarative program.

In GHC the rewrite rules defined in a module are embedded in the compiler-readable meta-data (its "`.hi` file") that accompanies the module's object code. The client of the library never sees the rules, but GHC can nevertheless use them to optimise compositions of calls to functions supplied by the library. Rules are not explicitly exported or imported. Instead, when compiling module `M`, GHC can "see" all the rules given in any module imported by `M`, or in any module imported by these imports, and so on transitively. (Haskell's `instance` declarations have exactly the same property.)

A rule is *not* required to be in the same module as the function whose definition it extends. For example the `"map/map"` rule does not have to be given in the module that defined `map`. So rules can incrementally extend a function's definition. This is important, because a rule may describe the interaction of an imported function with one defined locally. Rules can also be given for a class member function, in which case they work on the corresponding function in each class instance.

Rewrite rules make perfect sense even if the library is written in another language, in which case the rules express facts about the foreign library. For example, in Reid's graphics library for Haskell he provides a whole section of the user manual devoted to algebraic optimisation laws that are satisfied by the library interface [33].

3 Rules in practice

In the rest of the paper we report on our experience of applying rewrite rules in practice. We have found two main classes of applications:

- Programmer-written rules in library code. This was our initial motivation, and we have used it to achieve list fusion (this section) and more ambitious tree fusion (Section 6).

- Automatically-generated rules, derived from some kind of program analysis, invisibly to the programmer (Section 5). This was an unexpected, but very persuasive, practical benefit of implementing the rewrite-rule technology.

3.1 Short-cut Deforestation

Our initial motivating example for adding rewrite rules was the case of list fusion. In earlier work, Gill, Launchbury, and Peyton Jones described so-called *short-cut deforestation*, a technique for eliminating intermediate lists from programs [16]. At the centre of the method is the single rewrite rule `"foldr/build"`:

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr k z [] = z
foldr k z (x:xs) = k x (foldr k z xs)

build :: (forall b. (a->b->b) -> b -> b) -> [a]
build g = g (:) []

{-# RULES
"foldr/build"
  forall k z (g::forall b.(a->b->b) -> b -> b) .
  foldr k z (build g) = g k z
#-}
```

The definition of `foldr` is conventional. The function `build` takes a "list" `g`, functionally abstracted over its cons and nil constructors, and applies `g` to the ordinary list constructors `(:)` and `[]` to return an ordinary list. (`g`'s type is a rank-2 polymorphic type, as discussed in [16]; we must specify it explicitly in order to make the rule type-check.) The rule states that when `foldr` consumes the result of a call to `build`, one can eliminate the intermediate list by applying `g` directly to `k` and `z`.

To give an example of applying this rule we must write list-consuming and producing functions using `foldr` and `build` respectively. For example:

```
-- (sum [5,4,3,2,1]) = 15
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

-- (down 5) = [5,4,3,2,1]
down :: Int -> [Int]
down v = build (\c n -> down' v c n)

down' 0 cons nil = nil
down' v cons nil = cons v (down' (v-1) cons nil)
```

Again, the definition of `sum` in terms of `foldr` is conventional. The function `down` returns a list of integers, from its argument down to 1. We express it as a call to `build`, using an auxiliary function `down'` which is abstracted over the functions it uses to construct its result. (We have called these functions `cons` and `nil` for old times' sake, but they are simply the formal parameters to `down'` and their names are insignificant.) It is somewhat inconvenient to write `sum` and `down` in this way, but that is the task of the author of the `List` library.

Now we can try fusion on the call `(sum (down 5))`:

```
sum (down 5)
=    {inline sum and down}
  foldr (+) 0 (build (down' 5))
=    {apply the foldr/build rule}
  down' 5 (+) 0
```

The intermediate list has been eliminated; instead `down'` does the arithmetic directly.

3.2 A real (albeit small) example

List fusion works well when the programmer does "bulk" operations over lists, and then it can be stunningly effective. Here is an example taken verbatim from the `paraffins` code [29], a small program that computes a list of all the hydrocarbon paraffins of a given size:

```
three_partitions :: Int -> [(Int,Int,Int)]
three_partitions m
  = [ (i,j,k) | i <- [0..(m `div` 3)],
                j <- [i..(m-i `div` 2)],
                let k = m - (i+j)
    ]

-- A test harness
main = print (length (three_partitions 4000))
```

The form `[0..n]` is Haskell's notation for the list of integers between `0` and `n`. The list comprehension builds the list of all triples `(i,j,k)` where `i` is drawn from the list `[0..(m `div` 3)]`, and `j` is drawn from a similar list, and `k` is computed directly from `i` and `j`. Finally, the test harness prints the length applying `three_partitions` to 4000.

GHC translates range notation, `[0..n]`, into an application of `build`, much as we did for `down` above. It translates a list comprehension into a `build`, using `foldr` to consume the sub-lists. Finally, the Prelude library function `length` is implemented using a `foldr`.

So in this program, all the intermediate lists are removed, leading to a dramatic drop in allocation. Without fusion, this program allocates 188 Mbytes;

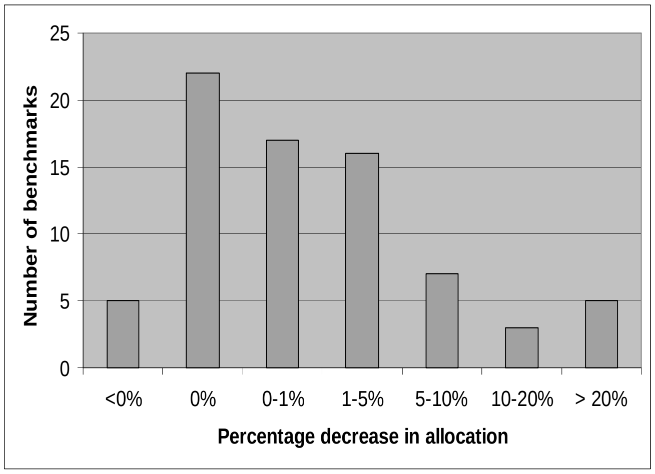


Fig. 1. Distribution of fusion effects on programs in "real" and "spectral" divisions of `nofib` benchmark suite, under ghc4.08.2.

when fusion is enabled, it allocates only 16 Mbytes. (Most of the allocation for the fused version is used for the stack, because the length computation is not properly tail-recursive, so the stack grows 1.3M activation records.)

3.3 Benchmark Results

Over a broader range of programs from the `nofib` benchmark set [29] the effect of enabling list fusion is very patchy, as Figure 1 shows. Fusion has no measurable effect on most programs but it gives a useful 5-25% reduction in allocation for a few. Only a very few programs are made worse, and the worst of these by less than 4%. One program, a parser called `parstof`, shows a 96% reduction; this turns to be because fusion transforms the (artificial) outer loop of the benchmark, causing the sample text input to be parsed once instead of 40 times!

The geometric mean improvement, about 5% if we omit `parstof`, seems disappointingly low, but we are undismayed. Compiler optimisations are like therapeutic drugs. Some, like antibiotics, are effective on many programs; such optimisations tend to be built into a compiler. Others are targeted at particular "diseases", on which they are devastatingly effective, but have no effect at all on most other programs. The rules mechanism allows library authors to add targeted, domain-specific optimizations without modifying the internals of the compiler.

We also hope that programmers may adopt a more modular programming style if they expect fusion to take place. For example, it is clearer to write

```
concat (map f xs)
```

than it is to write

```
foldr ((++) . f) [] xs
```

Yet programmers will sometimes write the latter form because it does not build an intermediate list. Section 6 gives an extended example of the way in which fusion can make modular programming practically efficient.

Finally, note that our measurements relate to un-modified benchmark programs. None of the functions in these programs use `build`, so fusion only occurs for compositions of functions from the Standard Prelude, whose functions we re-implemented using `foldr` and `build`. If the compiler were to transform user-written functions to use `foldr` and `build` we might see greater benefits — but that is beyond the scope of this paper, and in any case certainly would require compiler modification [25].

4 From theory to practice: the sticky details

So far we have implied that one simply needs to add one rewrite rule, and re-implement some key functions using `foldr` and `build`. In practice, though, we encountered a number of obstacles, which we discuss in this section, after first explaining our implementation of rewriting.

4.1 Implementation

The implementation of the rule rewriting mechanism within GHC is largely straightforward. The front-end has been extended to handle rule parsing, type checking, and translation into the `Core` intermediate language. The GHC optimiser is structured as a number of separate passes over `Core` expressions [32,31]. The most fundamental pass – iterated many times – is the *simplifier*, which performs inlining, case simplification, and eta-expansion in the course of a single top-to-bottom traversal of the program. To support rewriting, we just modified the simplifier to check each function application it encounters against a list of active rules; if the application matches the rule LHS pattern, it is replaced by a suitably instantiated version of the RHS. Matching is performed modulo eta-reduction, so that, e.g., an application of `\x -> f x` matches a rule with head `f`. We need to take a little care to make sure that the rule remains attached to the right function if alpha-renaming takes place.

Including rules adds a modest overhead to GHC compilation time. For example, using the list fusion rules described in Section 3 increases compilation times an average of 5% over the `nofib` benchmark suite. Some of this increase is probably due to performing conventional optimisations that are enabled by rule-based rewrites. In any case, we have made no serious attempt to analyse or optimise this aspect of compiler performance, so it can probably be sped up should this prove important.

4.2 Phases

The first obstacle faced when defining rules is a subtle interaction between function inlining — a transformation that GHC does aggressively [31] — and rule application. Returning to our `sum/down` example, we can see:

- `sum` and `down` must both be inlined before the rule can fire.
- On the other hand `foldr` and `build` must *not* be inlined. For example, inlining `build` before firing the rule would give

  ```
  foldr (+) 0 (down' 5 (:) [])
  ```

  and we have lost the fusion opportunity.
- However, once we have run out of opportunities to use the `foldr/build` rule, we *should* inline `build`. Recall that its definition is both small and higher-order:

  ```
   build g = g (:) []
  ```

  Inlining a function like this is very beneficial (`g` is often an explicit lambda).

These considerations led us initially to the following two-phase strategy:

(i) "Black list" any function that appears on the left hand side of a rule. Run the simplifier, applying rewrite rules, but refraining from inlining any black-listed functions.

(ii) Empty the black list, and repeat the exercise, so that previously-blacklisted functions will now be inlined.

Alas, two phases are not necessarily enough. In general, a program uses many layers of abstract data types, each implemented using the layer below. First we want to apply rewrite rules for the top-level ADT; then we want to expose its implementation (only to the compiler, of course) by inlining, and apply rewrite rules for the next layer; then we want to inline that layer and apply rewrite rules for the layer below; and so on.

Organising rules into phases is a form of *rewriting strategy*, a subject that has received considerable attention [36,22,9,37] However, one of the merits of rewrite rules is their simple, declarative nature: "here is a true fact: please use it whenever possible". We resist polluting this story with elaborate rewrite strategies. Nevertheless, it seems that some very simple strategy, such as a phase organisation is necessary. To gain experience, we have implemented the following very simple scheme.

The compiler runs the simplifier repeatedly, each run having a smaller [5] *phase number* than the previous one. A function may have an *inline pragma*, and this pragma can tell the compiler which phase to inline the function in. For example:

[5] Our present implementation uses increasing phase numbers, but we plan to reverse this shortly.

```
{-# INLINE 1 build #-}
build g = g (:) []
```

means "inline `build` in phase 1 (or smaller), *even if it appears on the left-hand-side of a rule*".

This scheme is clearly very crude. It requires the programmer to know something about GHC's phases, which is undesirable; and assigning phase numbers is not modular, requiring a global view of the program. Another question is whether it is better to annotate `INLINE` pragmas or the rules themselves. Various more elaborate schemes have occurred to us — using the module hierarchy, for example — but we have taken the view that we should refine the scheme in the light of practical experience, rather than implement an over-elaborate scheme right away.

4.3 Backing out

Suppose fusion does not take place. That is, suppose we have an isolated call `(down 34)`. It would be bad to actually implement `down` using `build` and `down'`, because doing so involves much more run-time function-passing than a straightforward implementation of `down`. It is unacceptable for programs to run slower in the (common) places when fusion fails than using the original library.

One solution is to rewrite `down'` to be non-recursive, and inline vigorously:

```
down :: Int -> [Int]
down v = build (\c n -> down' v c n)

down' v cons nil = go v
                 where
                   go 0 = nil
                   go v = cons v (go (v-1))
```

Now suppose we have inlined `down` at a call `(down 34)`, but alas it has not fused with a `foldr`. We can now inline as follows:

```
build (\c n -> down' 34 c n)  -- Did not fuse
=          { Inline build }
  down' 34 (:) []
=          { Inline down' }
  (go 34) where
            go 0 = []
            go v = v : go (v-1)
```

This code is as good as the original, straightforward implementation of `down` — because it *is* the original, straightforward implementation of `down`! The trouble is that we have effectively made a complete copy of the straightforward code at every call site. While this is acceptable for a function as small as `down`, it would be quite undesirable for larger functions.

An alternative solution, and the one we generally adopt, is to have the library author add a new definition and rewrite rule:

```
downList :: Int -> [Int]
downList 0 = []
downList v = v : downList (v-1)

{-# RULES "downList"
    forall v. down' v (:) [] = downList v #-}
```

An isolated call to `(down 34)` would now transform as follows:

```
down 34
=         {Inline down}
  build (down' 34)
=         {Inline build}
  down' 34 (:) []
=         {Apply "downList" rule}
  downList 34
```

The `"downList"` rule spots the special case in which `down'` is applied the standard list constructors, and transforms the call to use the directly-coded `downList` function.

4.4 One-shot lambdas

Here is the definition of `map` in terms of `foldr` and `build`:

```
map f xs = build (\c n -> foldr (c . f) n xs)
```

Now, suppose we find an application `(map f (build g))`. We want to transform the call like this:

```
map f (build g)
=     {Inline map}    DANGER!
  build (\c n -> foldr (c . f) n (build g))
=     {Apply foldr/build rule}
  build (\c n -> g (c . f) n)
```

The difficulty is in the step marked `DANGER!`. Here we substitute `(build g)` for `xs` in the body of `map`, but this occurrence of `xs` is under a lambda abstraction. In general, one can make a program run arbitrarily more slowly by substituting a redex inside a lambda abstraction, so GHC usually does something more conservative:

```
map f (build g)
=     {Inline map}    SAFE!
  let xs = build g
  in build (\c n -> foldr (c . f) n xs)
```

Alas now the `foldr/build` rule cannot fire!

The solution is to observe that the abstraction `(\c n -> ...)` is a *one-shot lambda*; that is, it is a function that is only called once. Why? Because it is the argument to `build`, and `build` simply calls its argument, passing `(:)` and `[]`. Substituting inside one-shot lambdas is perfectly safe.

The Right Thing To Do is to analyse the program for one-shot lambdas and act accordingly. A type-based analysis that achieves this (among other things) is described by Wansbrough [38], but it is not yet fully implemented in GHC. Instead we have a temporary hack that spots the special case of an application of `build`.

4.5 Sharing

Consider this function

```
f x = sum (filter (> x) [1..10])
```

One might expect all intermediate lists to be eliminated from this function, but GHC correctly spots that the expression `[1..10]` can be floated out:

```
one_to_ten = [1..10]
f x = sum (filter (> x) one_to_ten)
```

Alas, now the filter consumer cannot fuse with the `[1..10]` producer. Floating out `one_to_ten` would be a good transformation if the producer — in this case `[1..10]` — were more expensive. It would be worth losing the fusion, in order to share the computation of `one_to_ten` among all calls to `f`. But in the case of `[1..10]`, it would be better to lose sharing to gain fusion.

This problem turned out to be central when Elliott *et al.* tried to use rewrite rules to optimise Pan programs [12]. In Pan, it is crucial to inline absolutely everything, caring nothing for sharing, apply rewrite rules, and then do aggressive common sub-expression and code-motion transformations to make up for the loss.

This is a problem that is unlikely to have a cut-and-dried solution, but we are exploring the idea of using *virtual data types*. The programmer declares some data types as *virtual*, meaning that all data structures of virtual type should be eliminated. In particular, the compiler can ignore loss of sharing when considering inlining a value of virtual type. It remains to be seen how usable such a feature would be.

5 Dynamically-generated Rules

Thus far we have concentrated on rewrite rules that are written by the programmer, but we have found that it is often useful for the compiler itself to generate rewrite rules dynamically. We give three examples in this section.

5.1 Specialisation

Haskell's type classes give rise to overloaded functions with types like this:

```
invert :: Num elt => Matrix elt -> Matrix elt
```

Such overloaded functions are somewhat inefficient: `invert` takes a tuple (or "dictionary") of functions as an extra argument, which give the arithmetic operations over values of type `elt`. Optimising compilers for Haskell allow the programmer to write a `SPECIALISE` pragma, thus:

```
{-# SPECIALISE
      invert :: Matrix Int -> Matrix Int
#-}
```

This pragma encourages the compiler to build a specialised version of `invert`, in which the matrix elements are known to be of type `Int`, giving much more efficient code. (GHC will also infer such pragmas from the types at which `invert` is called, but only within a single module.)

Suppose, then, that the compiler has constructed the specialised function, and called it (say) `invert_Int`. The next task is to make sure that suitable calls to `invert` are replaced by calls to `invert_Int`. This is where rules come in. The compiler dynamically generates a rewrite rule like this:

```
{-# RULES
 "invert/Int" forall d::Num Int.
              invert @ Int d = invert_Int
#-}
```

Unlike our earlier, programmer-specified rules, this rule is written in GHC's explicitly-typed intermediate language, called "`Core`". In `Core`, every binder has an explicit type, and polymorphism is expressed using explicit type abstraction and application. The rules written by the user in the (implicitly-typed) Haskell source code are translated into the `Core` language by the type-checker (which adds type information) followed by the desugarer (which converts Haskell's rich syntax into `Core`'s much more limited forms).

In this case `invert` is polymorphic, and so takes a type argument, indicated by the "`@ Int`" on the left hand side of the rule. It also takes an argument corresponding to the `Num elt` constraint, namely the tuple of arithmetic operations referred to earlier. So the rule simply says that a call to `invert` applied to type `Int` and tuple `d` can be rewritten to `invert_Int`. Haskell's type system ensures that there is only one possible value for the tuple of methods `d::Num Int` — namely the numeric operations on `Int` values — and these methods are "baked into" `invert_Int`, so the rule can simply discard the argument `d`.

5.2 Evaluated arguments

In array-intensive code, one often encounters a loop like this:

```
f :: Int -> Int -> Int
f x y = if x == 0 then 0
        else y + f (x-1) (y+1)
```

GHC represents values of type `Int` using the following data type:

```
data Int = I# Int#
```

where `Int#` is the type of unboxed, 32-bit integers. GHC will compile `f` thus:

```
f :: Int -> Int -> Int
f x y = case x of { I# xv -> fw xv y }

fw :: Int# -> Int -> Int
fw xv y
  = if (xv ==# 0#) then I# 0#
    else
      case y of { I# yv ->
      case fw (xv -# 1#) (I# (yv +# 1#)) of { I# rv ->
      I# (yv +# rv) }}
```

`f` has turned into a mere "wrapper" that evaluates `x` before calling the "worker", `fw` [30]. It can do this because `f` is sure to evaluate `x`. However, `f` is not certain to evaluate `y`, so the evaluation of `y` must be in the `else` branch of the conditional in the worker, `fw`. That means that the worker must re-box `y` before calling itself ("`I# (yv +# 1#)`"), and in the common case, `y` will immediately be un-boxed again. This is bad.

What can be done? Again, it is a matter of specialisation. Recognising that there is a recursive call to `fw` in which the second argument is a constructor application, GHC can make a specialised version of `fw`, and generate an appropriate rule, thus:

```
fw1 :: Int# -> Int# -> Int
fw1 xv yv = let y = I# yv
            in ...original RHS of fw....

{-# RULES "fwV"  forall xv yv.
                 fw xv (I# yv) = fw1 xv yv
#-}
```

After simplifying the right hand side of `fw1`, using the rule, we get just what we want:

```
fw1 :: Int# -> Int# -> Int
fw1 xv yv
  = if (xv ==# 0#) then I# 0#
    else
      case fw1 (xv -# 1#) (yv +# 1#) of { I# rv ->
      I# (yv +# rv) }
```

`fw` remains as an "impedance matcher" embodying the first iteration of the loop, before calling `fw1`. However the rule remains to transform any call of `f` with an already-evaluated second argument into a call to `fw1`.

All of this is done invisibly by the compiler — the programmer is not involved at all. The transformation is fully implemented in GHC, enabled by "`-O2`". The analysis, generation of specialised code, and generation of the rewrite rule, takes only 225 lines of Haskell. The rewrite-rule infrastructure automatically takes care of applying the rule when it is relevant, and propagating the rule across separate compilation boundaries.

5.3 Usage types

We are exploring another example of the same pattern. Wansbrough's work on usage types suggests that considerable efficiency gains can be made by specialising functions based on their usage patterns [38]. For example, consider `map` again:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

If `map` is called in a context in which the result list is consumed at most once, then the thunks for `f x` and `map f xs` do not need to be self-updating; instead the updates can be omitted. To express this, GHC adds extra usage-type arguments to `map`, both at its definition and at its call sites. Once this is done, a specialised version of `map` can be compiled for the case when the usage-type argument is "`once`", and a rule generated to match such calls, in exactly the same way as for specialising overloading.

5.4 Summary

In each example, we can discern the same pattern:

- Based on pragmas or program analysis, perform a local transformation (e.g., generating the specialised version of `invert`).
- Generate a rule that explains how that transformation can be useful to the rest of the program. In some cases the rule looks at the type arguments, in others at value arguments.
- Apply the rule throughout the rest of the program.

This may not sound like much, but it is extremely helpful to have a single, consistent way to propagate the benefits of a transformation to the rest of the program. For example, it is not enough for the specialiser to generate specialised versions of a function *and* find all appropriate call sites for the specialised function. There may not *be* any calls to `invert` at type `Int` when the specialiser runs. Such calls may only show up after some other inlinings have exposed them. Or they may be in other modules altogether, so the rule must be propagated across module boundaries (which is relatively easily done).

Programmer-defined `RULES` pragmas are only allowed at top level, but this is a purely syntactic restriction. Rewrite rules make perfect sense for nested functions bound by a local `let` or `letrec`, and GHC will indeed generate dynamic rules using the ideas of this section for local functions. This is important in practice, because inlining generates many nested function definitions.

6 Application: Constraint Satisfaction Problems

Next we give an example user application — solving constraint satisfaction problems (CSPs) — in which rewrite rules help support high-level, modular programming style. The added rules, which describe short-cut deforestation on rose trees, are confined to a library, and they make a representative kernel of the application run three times faster, by eliminating essentially all the overhead due to the modular style.

6.1 Modular search

Many interesting algorithms for solving CSPs are conceptually based on trees, whose nodes represent states in the search space. Solutions to the search problem are found by locating nodes that represent complete, consistent states. In a conventional imperative recursive implementation, these search trees are merely notional; they correspond to the tree of procedure activation histories. In Haskell, one can make the state tree into an explicit (lazy) data structure instead [19,5]. This approach permits search algorithms to be modularized into separate functions (really coroutines) that communicate via a lazily-constructed tree labeled with consistency information. The component functions perform generation of all possible states, consistency labeling, pruning of inconsistent states, and collection of solutions. In earlier work, Nordin and Tolmach showed that a large variety of useful algorithms — which look quite different from one another when written imperatively — can be obtained in the lazy framework just by varying the labeling and pruning functions [28].

The underlying algorithm is a simple composition of functions, where all the intermediate results are trees or lists.

```
solver :: Labeler a -> Pruner a -> CSP -> [State]
solver labeler pruner csp =
  (filter (complete csp) . map fst . leaves .
     prune pruner . (labeler csp) .
     mkSearchTree) csp
```

Here `CSP` is a type describing instances of constraint satisfaction problems; for example, we might have a function

```
queens :: Int -> CSP
```

to generate instances of the familiar n-queens problem. `State` is the type of partial solutions. Function

```
mkSearchTree :: CSP -> Tree State
```

constructs a tree of all possible partial solutions to a given CSP. Here `Tree` is the type of ordinary "rose trees," in which each node has a value and an arbitrary number of children. The `labeler` argument to `solver` has this type:

```
type Labeler a =
  CSP -> Tree State -> Tree (State, a)
```

It specifies how to attach consistency annotations to each node in the tree. The `pruner` argument, of type

```
type Pruner a = (State,a) -> Bool
```

says how to inspect the annotations to determine whether the node is consistent; `prune` removes subtrees rooted at inconsistent nodes. `leaves` returns the leaves of the tree as a list in left-to-right order. The subsequent list operations throw away the annotations and weed out nodes representing incomplete solutions.

To obtain simple back-tracking search, we can provide a `Labeler` that checks the consistency of each node individually, and annotates the node with the boolean result of the check.

```
labelInconsistencies :: CSP -> Tree State -> Tree (State,Bool)
labelInconsistencies csp = mapTree f
   where f s = (s,not (consistent csp s))

btsolver :: CSP -> [State]
btsolver = solver labelInconsistencies snd
```

More sophisticated algorithms use labelers that may look at more than one node at a time or store more information in the annotations. For example, a well-known algorithm called forward checking can be implemented by a labeler that stores a (lazily constructed) cache table of consistency information at each node.

```
labelCSCache :: CSP -> Tree State ->
                       Tree (State,Cache ConflictSet)
extractConflict :: (State,Cache ConflictSet) -> Bool

fcsolver :: CSP -> [State]
fcsolver = solver labelCSCache extractConflict
```

Interesting new combinations of algorithms can be obtained by appropriate composition of labeling functions, giving us a "mix and match" approach to algorithm construction. The modular algorithms that result are much simpler to read, write, and modify than their imperative counterparts, and have the same asymptotic behavior (in both space and time).

However, the modular Haskell code *is* much slower than equivalent C code, if only by a constant factor. We measured performance of a representative kernel of code that implements standard backtracking search on the n-queens problem and counts the number of solutions found. The modular version of this function is written

```
qsolns :: Int -> Int
qsolns n = length (btsolver (queens n))
```

On the 11-queens problem, `qsolns` runs about 30 times slower than a conventional recursive C algorithm that doesn't use trees at all. More strikingly, perhaps, it is almost four times slower than a non-modular Haskell transliteration of the C algorithm. This difference suggests that we try to fuse the tree traversals to avoid building the nodes of the several intermediate trees.

In the remainder of this section, we describe short-cut deforestation for rose trees, and discuss our experience in using rules with this application. Full code for the kernel modular code and the corresponding monolithic function are given in the Appendix.

6.2 Fusion on rose trees

We treat rose trees as an abstract data type, with public functions `initTree`, `mapTree`, `prune`, and `leaves`. The internal representation data type and `foldTree` operation are standard:

```
data Tree a = T a [Tree a]

foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f t = go t
  where go (T a ts) = f a (map go ts)
```

We introduce a `buildTree` analogous to `build` on lists, and the corresponding fusion rule:

```
buildTree :: forall a.
                (forall b. (a -> [b] -> b) -> b) -> Tree a
buildTree g = g T

{-# RULES
"foldTree/buildTree"
  forall k (g::forall b.(a->[b]->b) -> b) .
    foldTree k (buildTree g) = g k
#-}
```

Now we must take care that all tree-producing functions use `buildTree`, and all tree-consuming functions use `foldTree`. Since `Tree` is as ADT, we don't need to worry about client code using the `T` constructor directly.

Function `initTree` generates a tree from a function that computes the children of a node [19]; `mapTree` is the analogue of the familiar functions on lists.

```
initTree :: (a -> [a]) -> a -> Tree a
initTree f a =  buildTree g
  where g n = go a
              where go a = n a (map go (f a))

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f t = buildTree g
  where g n = foldTree h t
              where h a ts = n (f a) ts
```

`prune` *p t* removes every subtree of *t* whose root value matches predicate *p*. Since we cannot represent empty trees, we require that *p* always return `False` on the root node of the entire tree, which is always appropriate in our applications.

```
prune :: (a -> Bool) -> Tree a -> Tree a
prune p t = buildTree g
  where
  g n = head (foldTree f t)
      where f a ts | p a = []
                   | otherwise = [n a (concat ts)]
```

Finally, `leaves` extracts the values at the leaves of a tree into a list in left-to-right order.

```
leaves :: Tree a -> [a]
leaves = foldTree f
  where f leaf [] = [leaf]
        f _    ts = concat ts
```

Ideally, we would like `leaves` to be written as a list `build`, so that it can fuse with list consumers further down the pipeline. Unfortunately, this seems to require doing a higher-order tree fold, which produces an intermediate list of function closures; GHC doesn't handle such lists very effectively, and it proves more efficient to stick with the simple definition shown here.

We mark all the functions to be inlined if possible.

6.3 Short-cut deforestation pays again

Given these definitions, GHC is able to completely fuse away all the rose trees in `qsolns`; i.e., no `T` constructors are applied at all! Indeed, modifying the implementation of our rose tree ADT to perform cheap deforestation improves performance of `(qsolns 11)` by a factor of more than three, bringing it to within 15% of the running time of a hand-fused, non-modular Haskell implementation. Moreover, this improvement comes *without requiring any changes to the search application code itself.*

All is not quite so straightforward as it may seem, however. All the problems we examined in the context of list fusion appear again for trees:

- Effective application of the fusion law requires that GHC inline more enthusiastically than it normally would. For example, our pipeline of tree operations generates many fusion opportunities that require inlining underneath the lambda of a `buildTree` argument. This is, in fact, a safe thing to do, since the lambda is "one shot," but GHC doesn't know this – and since we are thinking of trees as a user-defined library, it would be obviously inappropriate to hack this fact about `buildTree` into the compiler, the way we did for list `build`. As it happens, for the particular kernel of code we show here, GHC can use the fact that the lambda representing the entire program is one shot to deduce – after repeated iteration of inlining – that these `buildTree` lambdas are one shot as well. But in general, we need linearity analysis.
- If fusion fails, the tree library should make sure that the resulting code is not worse than it would have been had fusion never been attempted. As with lists, we must either ensure that inlining `foldTree` produces good code, or provide a "back-out" mechanism, with appropriate attention to phasing of inlining (c.f. Section 4.3).
- For full effectiveness, we need to make sure that inlining of list functions (e.g., on the lists of children in nodes) occurs only *after* inlining of tree functions (c.f. Section 4.2). We can arrange this by attaching an earlier phase number to the tree function inlining directives.
- Most seriously, we might easily write programs for which fusion fails for legitimate reasons, e.g. because there are several consumers for a given producer, or simply because we've made a mistake when writing a rule. But we'll get no feedback from the compiler about such failures. This is clearly a crucial area for further work.

7 Related Work

The basic concepts of our rules system are far from new. There have been a great many attempts to build frameworks for user-directed or application-specific optimization, often by adding additional semantic specifications to functions.

These ideas have been of particular interest in the high-performance computing community. Scientific codes often use well-established, high-level libraries, such as LINPACK or PLAPACK. Because these libraries need to work efficiently over a wide range of machine architectures and data sets, they typically have multiple implementations, each with its own complex interface. For portability and maintainability, client code should be written using portable, high-level library calls, leaving the compiler to determine the appropriate low-level calls to use and optimizing the client code accordingly. To achieve this, library interfaces can be annotated with additional specification information. Systems and proposals along these lines include TAMPR [6], Broadway [17,18], MetaScript [20], and Active Libraries [35].

Another set of systems has developed from the algebraic specification community. For example, the OPAL language [11] combines functional programming and algebraic specification in a uniform framework. OPAL laws are used to justify or guard rewrites of functional code; since laws are first-order predicate formulas over equality of functional expressions, this makes the system very powerful (and of course undecidable). It is unclear to what extent the existing implementation of OPAL supports automated optimization.

Compared to existing systems and proposals, ours is notable primarily for what it leaves out. More precisely, we can identify the following contrasts between our systems and others:

No meta language. Both left-hand and right-hand sides of our rules are just Haskell source expressions. With the exception of TAMPR [6], most of the other tools known to us operate on internal program representations, such as abstract syntax trees or control-flow graphs, and they typically allow right-hand sides to be defined using some kind of meta-programming facility. The choice of a meta-programming language is delicate. A specialized language or notation such as *metal* [13] is concise, but must be learned from scratch by the library author and can be unduly constraining; using a general-purpose programming language, such as LISP (as in early work on Aspect-Oriented Programming [21,27]) is more flexible, but requires the author to take great care to maintain essential invariants.

Simple rewrite strategy We rely on a very simple, built-in strategy, modified by "phases", for determining when and where rules should be applied. As rule sets become more elaborate, authors may need to exercise explicit control over strategy, e.g., as in Stratego [37].

Simple pattern-matching. We rely on the programmer to use high-level operators, such as `foldr`, that encapsulate control flow. Thus we don't need to provide sophisticated contextual pattern matching to identify loops or recursions, unlike systems like OPTRAN [26], Dora/Tess [15], and KHEPERA [14]. Nor do we have to deal with the unpredictability and possible high cost of higher-order matching, as used in MAG [10].

No side conditions. We work with a purely functional language, which means that many useful optimizing transformations are context-independent and don't require elaborate side-conditions. By contrast, most useful transformations on imperative programs must be justified by non-syntactic, and often non-trivial, analysis, e.g., of control flow, dependence, aliasing, etc. Thus many tools for imperative languages focus on specifying analyses in addition to transformations; examples include DFA&OPT-MetaFrame [23], Sharlit [34], Genesis [39], OPTIMIX [2], Intentional Programming [1], and recent work of Lacey and de Moor [24].

No termination guarantees; no AC rewriting. Our rules are all directed, and we cannot easily express commutative laws without causing endless rewriting. In a modern algebraic transformation system like Maude [9], equations are entirely symmetric in their left and right hand sides, which can be arbitrary terms; they can be used for transformation in either direction. Common algebraic properties of an operator can be declared by built-in keywords such as `[assoc]` and `[comm]`; in executing the transformations in a program, all pattern matching is conducted modulo these properties, which makes for shorter and more elegant programs.

In summary, we offer *simplicity* in exchange for more limited functionality. Simplicity is important, both for implementors and library authors. From an implementation point of view, our experience is that simple ideas are seldom easy to implement in a full-scale, optimising compiler, while complex ideas require heroism that is hard to sustain in the long term.

From a programming point of view, too, simplicity is important. Most particularly, the fact that the transformations are expressed entirely in Haskell itself, and not in some (necessarily different, and more indirect) meta-language is a huge advantage. We know of no optimising compiler in widespread use that supports domain-specific extensions; we suspect that this is partly due to the complexity of their meta-programming mechanisms. Of course, GHC's rules are not in widespread use by programmers either — but they are used behind the scenes in every run of GHC, both for list fusion (Section 3) and specialisation (Section 5). It is also possible that our approach is just *too* simple: we do not yet know how the tradeoff between simplicity and expressiveness will play out.

8 Conclusions and further work

We have described a simple, but fully implemented and deployed, way to write domain-specific extensions to a compiler for Haskell, by means of rewrite rules. We have demonstrated that, though simple, rewrite rules are useful in practice. Indeed, the list fusion rules have been deployed in the Prelude of the released GHC compiler for two years. In recent work, Chakravarty and Keller are using GHC's rewrite rules to perform array fusion in their work on nested data-parallel programming [7]; their application is more sophisticated than any we have described here.

The previous section described many directions in which one could imagine making our system more expressive, but we plan to develop more experience of its practical use before elaborating it much further. Indeed, the most pressing area for further work is not even mentioned in Section 7: it is the question of how best to provide feedback to the programmer about which rules have fired and, more especially, which have not and why not. Since rewrites are done on `Core`, which is quite far from Haskell, providing comprehensible feedback is a hard problem.

The status of this paper is as a report of work in progress. We present it in the hope that it will attract the interest of the writers of library packages, and will encourage them to experiment with the feature and report on its inadequacies. For the longer term, we wish to promote the principle that a programmer should supply further declarative information together with the code of the program; and suggest that compilers and other programming tools should take maximum advantage of these declarations.

Acknowledgements

We gratefully thank Manuel Chakravarty, Andy Gill, Oege de Moor, Eelco Visser, and the anonymous referees for their helpful feedback on earlier versions of this paper.

References

[1] Aitken, W., B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter and C. Simonyi, *Transformation in intentional programming*, in: *Proc. 5th International Conference on Software Reuse* (1998).

[2] Assmann, U., *How to uniformly specify program analysis and transformation with graph rewrite systems*, in: *Proc. Compiler Construction 1996*, LNCS **1060**, 1996, pp. 121–135.

[3] Baader, F. and T. Nipkow, "Term rewriting and all that," Cambridge University Press, 1999.

[4] Bird, R. and O. D. Moor, "The Algebra of Programming," Prentice Hall, 1996.

[5] Bird, R. and P. Wadler, "Introduction to Functional Programming," Prentice Hall, 1988.

[6] Boyle, J., T. Harmer and V. Winter, *The TAMPR program transformation system: Design and applications*, in: E. Arge, A. Bruaset and H. Langtangen, editors, *Modern Software Tools for Scientific Computing*, Birkhauser, 1997 .

[7] Chakravarty, M. and G. Keller, *Functional array fusion*, in: *ACM SIGPLAN International Conference on Functional Programming (ICFP'01)* (2001).

[8] Claessen, K. and J. Hughes, *QuickCheck: a lightweight tool for random testing of Haskell programs*, in: *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)* (2000), pp. 268–279.

[9] Clavel, M., S. Eker, P. Lincoln and J. Meseguer, *Principles of Maude*, in: J. Meseguer, editor, *Proceedings of the First International Workshop on*

Rewriting Logic, Electronic Notes in Theoretical Computer Science **4** (1996), pp. 65–89.

[10] de Moor, O. and G. Sittampalam, *Generic program transformation*, in: *Proc. 3rd International Summer School on Advanced Functional Programming*, LNCS **1608**, 1999, pp. 116–149.

[11] Didrich, K., A. Fett, C. Gerke, W. Grieskamp and P. Pepper, *OPAL: Design and Implementation of an Algebraic Programming Language*, in: J. Gutknecht, editor, *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 1994*, LNCS **782** (1994), pp. 228–244.

[12] Elliott, C., S. Finne and O. de Moor, *Compiling embedded languages*, in: *Proc. Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, LNCS **1924**, 2000.

[13] Engler, D., B. Chelf, A. Chou and S. Hallem, *Checking system rules using system-specific, programmer-written compiler extensions*, in: *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 2000.

[14] Faith, R. E., L. S. Nyland and J. F. Prins, *KHEPERA: A system for rapid implementation of domain specific languages*, in: *Proc. USENIX Conference on Domain-Specific Languages*, 1997, pp. 243–255.

[15] Farnum, C. D., "Pattern-Based Languages for Prototyping of Compiler Optimizers," Ph.D. thesis, University of California, Berkeley (1990), technical Report CSD-90-608.

[16] Gill, A., J. Launchbury and S. Peyton Jones, *A short cut to deforestation*, in: *ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, ACM, Cophenhagen, 1993 pp. 223–232.

[17] Guyer, S. Z. and C. Lin, *An annotation language for optimizing software libraries*, in: *Proceedings of the 2nd Conference on Domain-Specific Languages* (1999), pp. 39–52.

[18] Guyer, S. Z. and C. Lin, *Optimizing the use of high performance software libraries*, in: *Proc. 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC00)*, 2000.

[19] Hughes, J., *Why functional programming matters*, The Computer Journal **32** (1989), pp. 98–107.

[20] Kennedy, K., B. Broo, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey and L. Torczon, *Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries*, Journal of Parallel and Distributed Computing (2000), (To Appear).

[21] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: *ECOOP '97 — Object-Oriented Programming 11th European Conference*, LNCS **1241**, 1997, pp. 220–242.

[22] Kirchner, C., H. Kirchner and M. Vittek, *Implementing computational systems with constraints*, in: P. Kanellakis, J.-L. Lassez and V. Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming*, Brown University, Providence R.I., USA, 1993, pp. 166–175.

[23] Klein, M., J. Knoop, D. Koschützki and B. Steffen, *DFA and OPT-METAFrame: A tool kit for program analysis and optimization*, in: *Proc. 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, LNCS **1055**, 1996, pp. 418–421.

[24] Lacey, D. and O. de Moor, *Imperative program transformation by rewriting*, in: *Proc. Compiler Construction 2001*, LNCS **2027**, 2001, pp. 52–68.

[25] Launchbury, J. and T. Sheard, *Warm fusion*, in: *ACM Conference on Functional Programming and Computer Architecture (FPCA'95)*, ACM, La Jolla, California, 1995 pp. 314–323.

[26] Lipps, P., U. Möncke and R. Wilhelm, *OPTRAN - a language/system for the specification of program transformations: System overview and experiences*, in: *Proc 2nd Workshop on Compiler Compilers and High Speed Compilation*, LNCS **371**, 1988, pp. 52–65.

[27] Mendhekar, A., G. Kiczales and J. Lamping, *RG: A case-study for aspect-oriented programming*, Technical Report SPL97-009, Xerox Palo Alto Research Center, Palo Alto, CA, USA (1997).

[28] Nordin, T. and A. Tolmach, *Modular lazy search for constraint satisfaction problems*, Journal of Functional Programming (2001), (To appear.).

[29] Partain, W., *The* `nofib` *benchmark suite of Haskell programs*, in: J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing, Springer Verlag, 1992 pp. 195–202.

[30] Peyton Jones, S. and J. Launchbury, *Unboxed values as first class citizens*, in: *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)* (1991), pp. 636–666.

[31] Peyton Jones, S. and S. Marlow, *Secrets of the Glasgow Haskell Compiler inliner*, in: *Workshop on Implementing Declarative Languages*, Paris, France, 1999.

[32] Peyton Jones, S. and A. Santos, *A transformation-based optimiser for Haskell*, Science of Computer Programming **32** (1998), pp. 3–47.

[33] Reid, A., *The Hugs graphics library*, Technical report, School of Computing, University of Utah (2000).

[34] Tjiang, S. and J. Hennessy, *Sharlit – a tool for building optimizers*, in: *Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, 1992, pp. 82–93.

[35] Veldhuizen, T. L. and D. Gannon, *Active libraries: Rethinking the roles of compilers and libraries*, in: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)* (1998).

[36] Visser, E., *Strategic pattern matching*, in: *Rewriting Techniques and Applications (RTA'99), Trento*, Lecture Notes in Computer Science (1999).

[37] Visser, E., Z.-e.-A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, in: *Proceedings of the International Conference on Functional Programming (ICFP'98)*, 1998, pp. 13–26.

[38] Wansbrough, K. and S. Peyton Jones, *Once upon a polymorphic type*, in: *26th ACM Symposium on Principles of Programming Languages (POPL'99)* (1999), pp. 15–28.

[39] Whitfield, D. and M. L. Soffa, *The design and implementation of Genesis*, Software — Practice and Experience **24** (1994), pp. 307–325.

Appendix: Constraint Satisfaction Problems

Here is the complete code for the constraint satisfaction problem (CSP) search kernel described in Section 6

Problem Definition

A CSP is characterized by a number of variables `vars`, a number of values `vals`, and a consistency relation `rel` between pairs of assignments of values to vars. We represent assignments using an infix constructor `:=`. To solve the CSP, we must assign a value to each variable such that all pairwise combinations of assignments are in `rel`. A well-known example is the n-queens problem, under the standard optimization that we only try to place one queen in each column; this can be modeled as a CSP with n variables (the columns), n values (the rows), and a relation that permits two assignments provided the corresponding positions are on different rows or different diagonals.

```
type Var = Int
type Value = Int

data Assignment = Var := Value

type Relation = Assignment -> Assignment -> Bool

data CSP = C {vars, vals :: Int, rel :: Relation}
```

```
queens :: Int -> CSP
queens n = C{vals=n,vars=n,rel=safe}
  where safe (col1 := row1) (col2 := row2) =
          (row1 /= row2) &&
           abs (col1 - col2) /= abs (row1 - row2)
```

Search States

We model each state in the space of possible solutions as a sequence of assignments, together with the number of the most recently assigned variable. States are built from `emptyState` by repeated use of `extensions`, which takes a state and constructs a list of extended states formed by assigning each possible value to the next variable.

```
data State = S [Assignment] Var

emptyState :: CSP -> State
emptyState C{vars=vars} = S [] 0

extensions :: CSP -> State -> [State]
extensions C{vars=vars,vals=vals} (S as lastvar) =
  [S ((nextvar := val):as) nextvar |
    let nextvar = lastvar+1, nextvar <= vars, val <- [1..vals]]

complete :: CSP -> State -> Bool
complete C{vars=vars} (S _ lastvar) = lastvar == vars

consistent :: CSP -> State -> Bool
consistent _           (S []     _) = True
consistent C{rel=rel} (S (a:as) _) = all (rel a) as
```

A solution is a complete state that is consistent at every level.

Rose Trees

Here is sample library code for rose trees written without concern for fusion. For convenience, we do use `foldTree` in the definition of `prune` and `leaves`.

```
data Tree a = T a [Tree a]

initTree :: (a -> [a]) -> a -> Tree a
initTree f a = go a
  where go a = T a (map go (f a))

foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f t = go t
```

```
  where go (T a ts) = f a (map go ts)

mapTree  :: (a -> b) -> Tree a -> Tree b
mapTree f (T a ts) = T (f a) (map (mapTree f) ts)

prune :: (a -> Bool) -> Tree a -> Tree a
prune p t =
    head (foldTree f t)
      where f a ts | p a = []
                   | otherwise = [T a (concat ts)]

leaves :: Tree a -> [a]
leaves = foldTree f
  where f leaf [] = [leaf]
        f _    ts = concat ts
```

Rose trees supporting fusion

The code for these was shown in Section 6.2 .

Backtracking Search for CSPs

```
mkSearchTree :: CSP -> Tree State
mkSearchTree csp = initTree (extensions csp) (emptyState csp)

type Labeler a = CSP -> Tree State -> Tree (State, a)

type Pruner a = (State,a) -> Bool

labelInconsistencies :: Labeler Bool
labelInconsistencies csp = mapTree f
  where f s = (s,not (consistent csp s))

solver :: Labeler a -> Pruner a -> CSP -> [State]
solver labeler pruner csp =
  (filter (complete csp) . map fst . leaves .
     prune pruner . (labeler csp) .
     mkSearchTree) csp

btsolver :: CSP -> [State]
btsolver = solver labelInconsistencies snd

qsolns :: Int -> Int
qsolns n = length (btsolver (queens n))
```

Hand-fused Code

A hand-fused version of `qsolns` in Haskell:

```
qsolns' :: Int -> Int
qsolns' n = f (emptyState csp)
  where
    csp = queens n
    f state | complete csp state = 1
            | otherwise = g (extensions csp state)
    g [] = 0
    g (s':rest) | consistent csp s' = f s' + g rest
    g (_:rest) = g rest
```

***