Back to Basics:

Deriving Representations Changers Without Relations

G. Hutton and E. Meijer

UU-CS-1994-04
January 1994

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Back to Basics:

Deriving Representations Changers Without Relations

G. Hutton and E. Meijer

Technical Report UU-CS-1994-04
January 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Back to Basics: Deriving Representation Changers Without Relations

Graham Hutton[*] Erik Meijer[†]

Abstract

A representation changer is a function that can be specified in a particular way in terms of two other functions. Examples of representation changers include binary addition and multiplication, base conversion, and compilers. There has been much recent work in using a relational language, namely Jones and Sheerans' *Ruby*, to derive representation changers from their specifications using equational reasoning. In this paper we show that by beginning with a slightly less intuitive form of specification, the use of relations can be avoided, and representation changers can be derived within the simpler framework of functional programming. Moreover, our techniques can be applied to derive a carry-save adder, a representation changer that has not yet been derived in Ruby without the aid of informal reasoning.

1 Introduction

In the *calculational* approach to programming the aim is to formally derive (or synthesize) programs from their specifications by using equational reasoning. Programs so derived require no post-hoc proof of correctness; rather they are "correct by construction". Despite the fact that program derivations can often be viewed as correctness proofs turned upside down,

[*] Department of Computer Sciences, Chalmers University of Technology, and University of Göteborg, S-412 96 Göteborg, Sweden. Email: graham@cs.chalmers.se.

[†] Department of Computer Science, University of Utrecht, PO Box 80.089, NL-3508 TB Utrecht, The Netherlands. Email: erik@cs.ruu.nl.

experience has shown that many algorithms can in fact be derived from their specifications in a smooth and simple way. Typically there are only a few key steps where experience and creativity is needed in a derivation, the remaining steps being largely mechanical.

In this paper we are concerned with deriving *representation changers*, a widely occuring kind of functional program. For the last few years, representation changers have been the major topic of study within the relational language Ruby. The relational paradigm is a powerful framework for specifying and deriving programs, but arguably one of the most tricky to learn and use. We show that the use of relations can be avoided, and representation changers can be derived within the simpler setting of functional programming. Moreover, our techniques can be applied to derive a carry-save adder, a representation changer that has not yet been derived in Ruby without the aid of informal reasoning.

2 Preliminaries

We write $R : A \leftrightarrow B$ to mean that R is a (binary) relation between the sets A and B, i.e. that R is a subset of the cartesian product $A \times B$. We write $f : A \rightarrow B$ to mean that f is a total function between sets A and B. Functions can be viewed as relations in the evident way. If R is a relation, then $a \; R \; b$ means $(a, b) \in R$, and the *domain* and *range* of R are the sets defined by $dom(R) = \{a \mid \exists b.\ a \; R \; b\}$ and $rng(R) = \{b \mid \exists a.\ a \; R \; b\}$. Inclusion ($\subseteq$) and equality ($=$) of relations is defined just as for any other sets. Note that if $f, g : A \rightarrow B$ are functions, then $f \subseteq g$ iff $f = g$.

Given two relations $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$, the *composite* relation $S \circ R : A \leftrightarrow C$ is defined by $a \; (S \circ R) \; c$ iff there exists a b such that $a \; R \; b$ and $b \; S \; c$. Composition on relations generalises composition on functions: if f and g are functions, then $g \circ f$ is just ordinary function composition. The *converse* (or reciprocal) $R^c : B \leftrightarrow A$ of a relation $R : A \leftrightarrow B$ is defined by $b \; (R^c) \; a$ iff $a \; R \; b$. Converse on relations generalises inverse on functions: if $f : A \rightarrow B$ has inverse $f^{-1} : B \rightarrow A$, then $f^{-1} = f^c$.

3 Representation changers

A *representation changer* is a function that converts a concrete representation of an abstract value into a different concrete representation of that value. A simple example of a representation changer is a base-conversion function *conv* that converts a number in base m to a number in base n. In this case, abstract values are natural numbers, and concrete values are numbers in base m and base n respectively.

Given functions $f : C1 \rightarrow A$ and $g : C2 \rightarrow A$ that convert concrete values of types $C1$ and $C2$ to abstract values of type A, a representation changer $h : C1 \rightarrow C2$ can be specified by the requirement that if h maps concrete value $x \in C1$ to concrete value $y \in C2$, then x and y must represent the same abstract value:

$$h\ x = y \quad \Rightarrow \quad f\ x = g\ y. \tag{1}$$

Since g need not be injective, there may be more than one choice of such a y for each x, and hence there may be more than one solution for h. An equivalent specification then is that h maps a concrete value x to any concrete value y that represents the same abstract value as x:

$$h\ x \in \{y \mid f\ x = g\ y\}. \tag{2}$$

If for some value of x there is no y for which $f\ x = g\ y$ then there exists no total function h that satisfies the specification. An h exists iff the range of g is at least the range of f, i.e. $rng(g) \supseteq rng(f)$. A sufficient condition for this inclusion to hold is that the function $g : C2 \rightarrow A$ be surjective, which is often the case in practice.

Substituting $y = h\ x$ in (1) gives an equivalent functional equality:

$$f = g \circ h. \tag{3}$$

Observing in (1) that $h\ x = y$ iff $x\ h\ y$ and that $f\ x = g\ y$ iff $x\ (g^c \circ f)\ y$, we obtain a relational inclusion equivalent to (3):

$$h \subseteq g^c \circ f. \tag{4}$$

It is often natural to specify representation changers in this form. Using specification (4) has the advantage that h is in some sense the subject of the formula, and the term $g^c \circ f$ has an intuitive operational reading: first use f to convert a concrete value to an abstract value, then use g^c to convert the result into another concrete value. Moreover, $g^c \circ f$ is in general a relation, which emphasises the fact that there may be more than one function h that satisfies the specification $h \subseteq g^c \circ f$.

For example, the base-conversion function *conv* can be specified by the requirement that $conv \subseteq (eval_n)^c \circ eval_m$, where the function $eval_b$ converts a number in base b to the corresponding natural number:

$$\begin{array}{lcll} eval_b & [\,] & = & 0, \\ eval_b & (x:xs) & = & x + b * (eval_b\ xs). \end{array}$$

Note that $eval_b$ expects the digits in the opposite order to which they are normally written; for example, $eval_2\ [1,0,1,1] = 13$. The specification for *conv* expresses that a number in base m can be converted to a number in base n by first evaluating the base-m number, and then converting the resulting natural number to base-n. Since $eval_b$ is surjective (every natural number can be represented in base b), a solution exists for *conv*. Since $eval_b$ is not injective (for example the lists $[0,1]$, $[0,1,0]$ and $[0,1,0,0]$ all represent the natural number 2 under $eval_2$ — numbers can have trailing zeros) the specification does not have a unique solution for *conv*.

Functional programmers might recognise that $eval_b$ could be expressed using the operator *foldr* of Bird and Wadler [5]. When we come to do induction over the argument of $eval_b$, however, it is more convenient to have $eval_b$ defined using explicit recursion, rather than using *foldr*. Modern calculational programmers avoid the labour of inductive proofs by using the so-called 'unique extension property' (UEP) of *foldr* [19]. In the present situation, the effort of setting things up so that the UEP can be used is no less bother than an induction. Moreover, one of the aims of this paper is to make the techniques available to a wide audience, so we don't want to use a perhaps unfamiliar technique like UEP when it doesn't buy very much.

In the literature, relations R which can be expressed as $R = g^c \circ f$ (i.e. in the form of the right-hand side of a representation changer specification) are called *difunctional* relations [20, 12]. Any relation can be expressed in the dual form $R = f \circ g^c$, a result which underlies much of the recent categorical work on relations [10, 7, 9]. A natural generalisation of the paradigm of representation changers adopted in this paper is to allow the functions f, g, h in a specification $h \subseteq g^c \circ f$ to be in fact difunctional relations. This generalisation is addressed in [15, 11].

4 Satisfying the specification

Typically for representation changers it is easy to define programs $f : C1 \rightarrow A$ and $g : C2 \rightarrow A$ directly, while defining a program $h : C1 \rightarrow C2$ that satisfies $h \subseteq g^c \circ f$ requires some creative effort. Rather than defining an h and proving after-the-fact that is satisfies the specification, in this note we are concerned with deriving (or synthesising) an h that is guaranteed to satisfy the specification.

One approach to deriving representation changers is that as used in *Ruby*, the relational calculus developed by Jones and Sheeran for designing programs that represent hardware circuits [21, 13]. In Ruby we define $f : C1 \rightarrow A$ and $g : C2 \rightarrow A$ as relational terms, and synthesise a term $h : C1 \rightarrow C2$ that represents a circuit and satisfies $h \subseteq g^c \circ f$ by transforming the term $g^c \circ f$ using relational laws [14, 15, 11]. There are several reasons why Ruby is based upon relations rather than functions. Relational languages offer a rich set of operators and laws for combining and transforming programs, and a natural treatment of non-determinism in specifications. Furthermore, many methods for combining circuits (viewed as networks of functions) are unified if the distinction between input and output is removed [21].

But there are a number of disadvantages to working with relations rather than with functions. Functional programs are typically defined using bound variables, and manipulated using the powerful tools of beta-reduction (application) and eta-reduction (extentionality) [6, 3, 17]. With relational programs however, pointwise manipulations quickly become tedious, and are avoided by eliminating bound variables completely and building programs using combinators. Such relational programs then become cluttered with plumbing relations whose only goal is to route arguments to the right place. Moreover, familiarity with a large number of—often non-intuitive—laws is needed to manipulate relational programs.

5 Avoiding relations

In this section we present a simple method by which we can derive representation changers using a functional language. Starting with the specification $h \subseteq g^c \circ f$ (an inclusion of relational terms) we immediately convert it into the equivalent but less intuitive form $f = g \circ h$ (an equality of functional terms). Then we synthesize a program h that satisfies $f = g \circ h$ by constructing a pointwise proof that the equation holds, aiming to end up with assumptions that give a definition for h. This is similar to the way recursive programs are constructed in type theory. The technique is also known to functional programmers; many of the programs in Bird and Wadler [5] are synthesized in a similar fashion, but the application to representation changers in this paper is new.

Let us consider an example: a function *add* that takes a binary number (represented as a list of bits) and a bit (0 or 1), and adds them together to give a binary number [15, 11]. For any bit b the function *add* $\_$ b is a representation changer, specified by the requirement that

$$add \mathbin{\_} b \quad \subseteq \quad (eval_2)^c \circ (+b) \circ eval_2. \qquad (5)$$

The specification expresses that we can add a bit b to a binary number by first converting the binary number to a natural number, adding b, and then converting the result back to binary. For the remainder of this section, we abbreviate $eval_2$ by $eval$.

Since $rng\ eval \supseteq rng\ ((+b) \circ eval)$ for any bit b, the specification (5) has a solution for $add \mathbin{\_} b$. Since $(+b) \circ eval$ is not injective, the specification has many solutions. Different solutions give different numbers of trailing 0's in the result list.

The first step is to re-write the specification:

$$\begin{array}{ll}
 & add \mathbin{\_} b \subseteq eval^c \circ (+b) \circ eval \\
\Leftrightarrow & \qquad h \subseteq g^c \circ f \text{ iff } f = g \circ h \\
 & eval \circ (add \mathbin{\_} b) = (+b) \circ eval \\
\Leftrightarrow & \qquad \text{extensionality} \\
 & eval\ (add\ xs\ b) = (eval\ xs) + b.
\end{array}$$

The second step is to verify the final equation above, which we do by induction on xs. In the base-case $xs = [\,]$, we end up with an assumption that gives a definition for $add\ [\,]\ b$. In the inductive-case $xs = x : xs$, we end up with an assumption that gives a recursive definition for $add\ (x : xs)\ b$ in terms of $add\ xs\ b$.

First the base-case, $xs = [\,]$:

$$\begin{array}{ll}
 & eval\ (add\ [\,]\ b) = (eval\ [\,]) + b \\
\Leftrightarrow & \qquad \text{unfolding } eval \\
 & eval\ (add\ [\,]\ b) = b \\
\Leftrightarrow & \qquad \text{folding } eval \\
 & eval\ (add\ [\,]\ b) = eval\ [b] \\
\Leftarrow & \qquad \text{application} \\
 & add\ [\,]\ b = [b].
\end{array}$$

We conclude that the definition $add\ [\,]\ b = [b]$ satisfies the specification in the $xs = [\,]$ case. Note that in the "folding *eval*" step above, replacing b by

$eval\ [b]$ is not the only possibility; $eval\ [b,0]$, $eval\ [b,0,0]$, etc., are equally valid. Choosing $eval\ [b]$ means that the result list produced by $add\ xs\ b$ will have no trailing 0's.

Now for the inductive-case, $xs = (x : xs)$. Rather than manipulating the equation $eval\ (add\ (x : xs)\ b) = (eval\ (x : xs)) + b$ as a whole, we work only with the right-side, aiming (just as in the $xs = [\,]$ case) to express it in the form $eval\ exp$ for some expression exp, from which we can conclude that the definition $add\ (x : xs)\ b = exp$ satisfies the specification in the $xs = (x : xs)$ case. We begin by unfolding:

$$
\begin{array}{ll}
 & eval\ (x : xs) + b \\
= & \qquad \text{unfolding } eval \\
 & 2 * (eval\ xs) + x + b.
\end{array}
$$

One might think that by folding using $eval$ now to give $eval\ ((x + b) : xs)$ we are finished, but this step is not valid, because $eval$ expects a list of 0's and 1's, but the expression $x + b$ can have the value 2. We proceed in fact by splitting the value $x + b$ into two parts: $(x + b)$ **mod** 2 and $(x + b)$ **div** 2. This solves the problem because both parts are bits; the first can be viewed as a *sum*-bit, and the second as a *carry*-bit.

$$
\begin{array}{ll}
 & 2 * (eval\ xs) + x + b \\
= & \qquad \text{splitting } x + b \\
 & 2 * (eval\ xs) + 2 * ((x + b)\ \mathbf{div}\ 2) + ((x + b)\ \mathbf{mod}\ 2) \\
= & \qquad \text{arithmetic} \\
 & 2 * (eval\ xs + ((x + b)\ \mathbf{div}\ 2)) + ((x + b)\ \mathbf{mod}\ 2) \\
= & \qquad \text{induction hypothesis} \\
 & 2 * (eval\ (add\ xs\ ((x + b)\ \mathbf{div}\ 2))) + ((x + b)\ \mathbf{mod}\ 2) \\
= & \qquad \text{folding } eval \\
 & eval\ (((x + b)\ \mathbf{mod}\ 2) : add\ xs\ ((x + b)\ \mathbf{div}\ 2)).
\end{array}
$$

The final term above is of the form $eval\ exp$, so we are finished and conclude with the definition $add\ (x : xs)\ b = ((x + b)\ \mathbf{mod}\ 2) : add\ xs\ ((x + b)\ \mathbf{div}\ 2)$.

In summary, we have derived a functional program

$$
\begin{array}{llcl}
add\ \ [\,] & b & = & [b], \\
add\ \ (x : xs) & b & = & ((x + b)\ \mathbf{mod}\ 2) : add\ xs\ ((x + b)\ \mathbf{div}\ 2).
\end{array}
$$

that satisfies the specification $add \_ b \subseteq eval^c \circ (+b) \circ eval$.

Deriving a program which adds two binary numbers and a carry-bit (rather than a single binary number and a carry-bit) is no more complicated. The only trick is to begin with a specification in which the two binary numbers are zipped together to form a list of pairs of bits.

Our second example is given by a function *cadd* that takes a carry-save number and a bit, and adds them together to give a carry-save number. The *cadd* function is an example which has proved difficult to derive fully formally using Ruby [15]. A *carry-save* number is like a binary number in that the ith digit has weight 2^i, but different in that digits range over $\{0, 1, 2\}$, with each digit being represented by a pair of bits whose sum is that digit. For example, $[(0,1),(1,1),(1,0)]$ is a carry-save representation of the natural number 9, because $(0+1).2^0 + (1+1).2^1 + (0+1).2^2 = 9$. A natural number can have many carry-save representations; for example, $[(1,0),(0,0),(1,1)]$ also represents the number 9. The function *ceval* converts a carry-save number to the corresponding natural number:

$$\begin{array}{lcl} ceval \quad [] & = & 0, \\ ceval \quad ((x,y):xs) & = & x + y + 2 * (ceval\ xs). \end{array}$$

Here is our specification for the *cadd* function:

$$cadd \_ b \subseteq ceval^c \circ (+b) \circ ceval.$$

Not only does this specification look the same as that for the binary addition program *add*, using precisely the same derivation pattern (try it!) we can construct the following definition for *cadd*:

$$\begin{array}{llcl} cadd\ \ [] & b & = & [(0,b)], \\ cadd\ \ ((x,y):xs) & b & = & ((x+y)\ \mathbf{mod}\ 2, b) : cadd\ xs\ ((x+y)\ \mathbf{div}\ 2). \end{array}$$

The point to note is that *cadd* is non-strict in the b argument, whereas *add* is strict in b. The effect is that the carry-save adder has no 'rippling carry', and so the addition can be done in parallel in constant time. Carry-save adders are much used in hardware circuits: if a large number of additions are to be done (such as in a multiplier), a considerable speed-up can be obtained by first converting to carry-save numbers, doing all the additions using carry-save adders, and then converting the result back to binary at the end. Using this technique, the only place a rippling carry occurs is when the final carry-save number is converted back to a binary number.

In the *add* example one could have made *eval* injective (and hence invertible) by including a side-condition that there be no trailing zeros in the input number or by working with binary numbers of a specified length. Even with such restrictions, *ceval* is still not invertible, since there can be many carry-save representations of each number. That the carry-save adder proves difficult to derive within the relational framework of Ruby is due to the problem of manipulating type information (types in Ruby are equivalence relations) when inverting the non-injective function *ceval*.

6 A two-stage example

For our final example we return to the base conversion function that was our initial example of a representation changer. The base converter turns out to be particularly interesting because in the process of its derivation we construct an auxilliary representation changer. One might say then that the base converter has a two-stage derivation.

Recall from section 3 that *conv* can be specified by

$$conv \subseteq (eval_n)^c \circ eval_m,$$

Expressing this in the form of equation (3) and then using extensionality gives our working specification:

$$eval_n\ (conv\ xs) = eval_m\ xs.$$

We synthesize *conv* by a constructive induction on xs. In the base-case $xs = [\,]$, unfolding $eval_m$ immediately results in the definition $conv\ [\,] = [\,]$. Like in the binary adder example, in the inductive case $xs = x : xs$ we aim to express the right-hand side of the equation in the form $eval_n\ exp$ for some expression exp, from which we can conclude that $conv\ (x : xs) = exp$:

$$\begin{array}{ll} & eval_m\ (x : xs) \\ = & \qquad \text{unfolding } eval_m \\ & m * (eval_m\ xs) + x \\ = & \qquad \text{induction hypothesis} \\ & m * (eval_n\ (conv\ xs)) + x \\ = & \qquad \text{assumption — see below} \\ & eval_n\ (convd\ (conv\ xs)\ x). \end{array}$$

Hence we make the definition $conv\ (x : xs) = convd\ (conv\ xs)\ x$. In the above derivation, in order to end up in the form $eval_n\ exp$, we had to postulate the existence of a function *convd* satisfying

$$eval_n\ (convd\ ys\ x) \quad = \quad m * eval_n\ ys + x. \tag{6}$$

Re-writing this a little, we observe that *convd ys* is *itself* a representation changer, which takes a digit in base m and yields a number in base n:

$$convd\ ys \subseteq (eval_n)^c \circ ((m * eval_n\ ys)+).$$

The auxiliary function *convd* is constructed by a double induction on its two arguments. A simple calculation gives $convd\ [\,]\ 0 = [\,]$. For the inductive case $ys = [\,]$ and $x \neq 0$, manipulating the right-hand side of equation (6) results in the definition $convd\ [\,]\ x = x \mathbf{\ mod\ } n : convd\ [\,]\ (x \mathbf{\ div\ } n)$:

$$\begin{array}{ll}
 & m * eval_n\ [\,] + x \\
= & \qquad \text{unfolding } eval_n \\
 & x \\
= & \qquad \text{splitting } x \\
 & (x \mathbf{\ mod\ } n) + n * (x \mathbf{\ div\ } n) \\
= & \qquad \text{folding } eval_n \\
 & (x \mathbf{\ mod\ } n) + n * (m * eval_n\ [\,] + (x \mathbf{\ div\ } n)) \\
= & \qquad \text{induction hypothesis} \\
 & (x \mathbf{\ mod\ } n) + n * (eval_n\ (convd\ [\,]\ (x \mathbf{\ div\ } n))) \\
= & \qquad \text{folding } eval_n \\
 & eval_n\ (x \mathbf{\ mod\ } n : convd\ [\,]\ (x \mathbf{\ div\ } n)).
\end{array}$$

For the induction hypothesis to be applicable above, we must assume $n > 1$. The creative part in handling the final case, $ys = y : ys$, is the splitting of $m * y + x$ in order to apply the induction hypothesis. Such a splitting was also the essential step in establishing the previous induction step.

$$\begin{array}{ll}
 & m * (eval_n\ (y : ys)) + x \\
= & \qquad \text{unfolding } eval_n \\
 & m * (y + n * eval_n\ ys) + x \\
= & \qquad \text{arithmetic}
\end{array}$$

$$
\begin{aligned}
& (m * y + x) + m * n * eval_n\ ys \\
= & \qquad \text{splitting } m * y + x \\
& (m * y + x) \ \mathbf{mod}\ n + n * ((m * y + x)\ \mathbf{div}\ n) + m * n * eval_n\ ys \\
= & \qquad \text{arithmetic} \\
& (m * y + x)\ \mathbf{mod}\ n + n * (m * eval_n\ ys + (m * y + x)\ \mathbf{div}\ n) \\
= & \qquad \text{induction hypothesis} \\
& (m * y + x)\ \mathbf{mod}\ n + n * (eval_n\ (convd\ ys\ ((m * y + x)\ \mathbf{div}\ n))) \\
= & \qquad \text{folding } eval_n \\
& eval_n\ ((m * y + x)\ \mathbf{mod}\ n : convd\ ys\ ((m * y + x)\ \mathbf{div}\ n)).
\end{aligned}
$$

We conclude that $convd\ (y : ys)\ x = (m * y + x)\ \mathbf{mod}\ n : convd\ ys\ ((m * y + x)\ \mathbf{div}\ n)$. In summary, we have synthesized the following program:

$$
\begin{array}{lcll}
conv & [\,] & = & [\,], \\
conv & (x : xs) & = & convd\ (conv\ xs)\ x.
\end{array}
$$

The auxilliary function *convd* is defined as follows:

$$
\begin{array}{llcll}
convd & [\,] & 0 & = & [\,], \\
convd & [\,] & x & = & x\ \mathbf{mod}\ n : convd\ [\,]\ (x\ \mathbf{div}\ n), \\
convd & (y : ys) & x & = & (m * y + x)\ \mathbf{mod}\ n : convd\ ys\ ((m * y + x\ \mathbf{div}\ n).
\end{array}
$$

Implementing these functions in Gofer, we can try out some base conversion examples. Taking $m = 2$ and $n = 10$, the following output confirms that 13 is the decimal representation of the binary number 1101 (remember that the digits in the input and output numbers for the program are reversed):

```
Gofer?
conv [1,0,1,1]
[3, 1]
(43 reductions, 92 cells)
```

By taking $m = 10$ and $n = 2$ we can convert the other way around:

```
Gofer?
conv [3,1]
[1, 0, 1, 1]
(35 reductions, 82 cells)
```

The reader might like to compare our derivation of the base conversion program with the corresponding derivation in Ruby [11]. Ours is simple and uses no special techniques, whereas the Ruby version is tricky, being the final and most complicated example in Hutton's thesis. The base conversion algorithm can be implemented as a hardware circuit, as shown in the book "Digital systems, with algorithm implementation" [8]. This book is somewhat novel in hardware circles, making use of algebraic methods in an attempt to explain circuits. Despite this, we still find the explanation in the book lacking in detail and difficult to understand.

7 Discussion

Many functional programs can be viewed as representation changers and have a natural specification in the form $h \subseteq g^c \circ f$. Once a programming problem is recognised as being an example of a representation changer, deriving a program using our techniques is largely a mechanical process, and requires experience and creativity only in a few key points. It is encouraging to find that the same patterns of transformations are used again and again when deriving representation changers. Certainly for the arithmetic examples given in this paper, one quickly gets a feeling of deja vu.

We conclude with some reflective remarks. Relational programming is gaining favour among calculational programmers, as evidenced by the growing number of systems [1, 13, 4, 22, 2, 16]. Little is being said, however, about the undoubted fact that relational programming is more difficult than functional programming. In this paper we have shown how a number of relational derivations can be reworked using functions. One of the points we would like to make is that one should be frugal in the use of relations, using them only when a simple functional solution can't be found. Similar comments can be made about other trends in calculational programming, including the use of categorical concepts, catamorphisms, UEP rules, point-free reasoning, etc. It is our opinion that calculational programming is losing track of its goals. Too many researchers are striving ahead and using more and more theory (see for example our own theses [11, 18]), without stopping to think if things can be done adequately using standard techniques.

References

[1] Roland Backhouse, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes. In *Proc. STOP Summer School on Constructive Algorithmics*, Ameland, The Netherlands, September 1992.

[2] Rudolf Berghammer. Relational specification of data types and programs. Report 9109, Universitat der Bundeswehr Munchen, September 1991.

[3] Richard Bird. Constructive functional programming. In *Proc. Marktoberdorf International Summer School on Constructive Methods in Computer Science*. Springer-Verlag, 1989.

[4] Richard Bird and Oege de Moor. From dynamic programming to greedy algorithms. In *Proc. STOP Summer School on Constructive Algorithmics*, Ameland, The Netherlands, September 1992.

[5] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.

[6] Rod Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[7] Aurelio Carboni, Stefano Kasangian, and Ross Street. Bicategories of spans and relations. *Journal of Pure and Applied Algebra*, 33:259–267, 1984.

[8] M. Davio, J.-P. Deschamps, and A. Thayse. *Digital Systems, with Algorithm Implementation*. John Wiley & Sons, 1983.

[9] Oege de Moor. *Categories, Relations and Dynamic Programming*. PhD thesis, Oxford University, April 1992. Available as Research Report PRG-98.

[10] Peter Freyd and Andre Scedrov. *Categories, Allegories*. North-Holland, 1990.

[11] Graham Hutton. *Between Functions and Relations in Calculating Programs*. PhD thesis, University of Glasgow, October 1992. Available as Research Report FP-93-5.

[12] A. Jaoua, A. Mili, N. Boudriga, and J. L. Durieux. Regularity of relations: A measure of uniformity. *Theoretical Computer Science*, 79:323–339, 1991.

[13] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Staunstrup, editor, *Formal Methods for VLSI Design*, Amsterdam, 1990. Elsevier Science Publications.

[14] Geraint Jones and Mary Sheeran. Relations and refinement in circuit design. In Morgan, editor, *Proc. BCS FACS Workshop on Refinement*, Workshops in Computing. Springer-Verlag, 1991.

[15] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer-Verlag, 1992. To appear.

[16] Bruce MacLennan. Relational programming. Technical Report NPS52-83-012, Navel postgraduate School, Monterey, CA 93943, September 1983.

[17] Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In *Proc. CWI Symposium*, Centre for Mathematics and Computer Science, Amsterdam, November 1983.

[18] Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, February 1992.

[19] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings FPCA'91*, number 523 in LNCS. Springer-Verlag, 1991.

[20] J. Riguet. Relations binaires, fermetures, correspondances de galois. *Bulletin de la Societé mathématique de France*, 76, 1948.

[21] Mary Sheeran. Describing and reasoning about circuits using relations. In Tucker et al., editors, *Proc. Workshop in Theoretical Aspects of VLSI*, Leeds, 1986.

[22] Paulo Veloso and Armando Haeberer. Partial relations for program derivation. In *Proc. IFIP WG-2.1 Working Conference on Constructing Programs from Specifications*, Pacific Grove, USA, May 1991.