*J. Functional Programming* **1** (1): 1–000, January 1993 © 1993 Cambridge University Press 1

# FUNCTIONAL PEARLS

## *Back to Basics: Deriving Representation Changers Functionally*

Graham Hutton and Erik Meijer †

*Department of Computer Science, University of Utrecht,*
*PO Box 80.089, 3508 TB Utrecht, The Netherlands.*

**Abstract**

A representation changer is a function that converts a concrete representation of an abstract value into a different concrete representation of that value. Many useful functions can be recognised as representation changers; examples include compilers, and arithmetic functions such as addition and multiplication. Functions that can be specified as the right inverse of other functions are special cases of representation changers.

In recent years, a number of authors have used a relational calculus to derive representation changers from their specifications. In this paper we show that the generality of relations is not essential, and representation changers can be derived within the more basic setting of functional programming. We illustrate our point by deriving a carry-save adder and a base-converter, two functions which have previously been derived relationally.

## 1 Introduction

In the *calculational* approach to programming the aim is to derive programs from their specifications by a process of formal 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, experience has shown that many algorithms can in fact be derived from their specifications in a smooth and simple way.

In this paper we are concerned with deriving *representation changers*, a widely occurring kind of functional program. Many useful functions can be recognised as representation changers; examples include compilers, and arithmetic functions such as addition and multiplication. Functions that can be specified as the right inverse of other functions are special cases of representation changers.

For the last few years, representation changers have been a major topic of study within the relational calculus Ruby (Sheeran, 1986; Jones and Sheeran, 1990). We show that the generality of relations is not essential, and representation changers can

† Part of this work was completed while at Department of Computer Sciences, Chalmers University of Technology, S-412 96 Gothenburg, Sweden.

be derived within the more basic setting of functional programming. We illustrate our point by deriving a carry-save adder and a base-converter, two examples which have previously been derived relationally.

## 2 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 typical example of a representation changer is a base-conversion function 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 $f : C1 \to A$ and $g : C2 \to A$ (we assume throughout that all functions are total) that convert concrete values of types $C1$ and $C2$ to abstract values of type $A$, a representation changer $h : C1 \to C2$ 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 the function $h$ 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 precisely when the range of $g$ is at least the range of $f$. A sufficient condition for $h$ to exist is that the function $g : C2 \to A$ be surjective, which is often the case in practice. In other cases, one can try strengthening the specification by adding extra requirements on $h$, in the manner of (Runciman and Jagger, 1990).

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

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

We can also express (1) in the form of a relational inclusion (4), by observing that $h\ x = y$ iff $x\ h\ y$ and that $f\ x = g\ y$ iff $x\ (g^{-1} \circ f)\ y$. Here functions are implicitly viewed as relations—by taking their graph—, and relational composition $\circ$ and relational converse $\_^{-1}$ are the evident generalisations of the composition and inverse operators on functions (Ross and Wright, 1992).

$$h \subseteq g^{-1} \circ f. \tag{4}$$

It is sometimes more 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 $g^{-1} \circ f$ has an intuitive operational reading: first use $f$ to convert a concrete value to an abstract value, then use $g^{-1}$ to convert the result into another concrete value. In general $g^{-1} \circ f$ is a true relation, which implies that there may be more than one function $h$ that satisfies the specification $h \subseteq g^{-1} \circ f$. Constructing a function $h$ that satisfies (4) usually proceeds by transforming the term $g^{-1} \circ f$ using laws of a relational calculus (Jones and Sheeran, 1991; Jones and Sheeran, 1992; Hutton, 1992).

In this paper we don't follow the relational route, but rather show how representation changers can be derived functionally. Starting with a specification $f = g \circ h$ we synthesize a function $h$ by constructing a pointwise proof that the equation holds, aiming to end up with assumptions that give a definition for $h$. This is a well-established technique for deriving functional programs, but the application to deriving representation changers appears to be new.

## 3 Example: carry-save addition

Our first example concerns a representation of numbers which is much-used in digital circuits (Davio *et al.*, 1983). A *carry-save* number is like a binary number in that the $i$th 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 $(0+1).2^0 + (1+1).2^1 + (1+0).2^2 = 9$. 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:

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

Note that *ceval* expects the least significant bit-pair first.

Consider the function *cadd* that takes a bit and carry-save number, and adds them together to give a carry-save number. For any bit $b$, the function *cadd* $b$ is a representation changer, specified by the requirement that

$$\mathit{cadd}\ b \quad \subseteq \quad \mathit{ceval}^{-1} \circ (b+) \circ \mathit{ceval}. \tag{5}$$

The specification expresses that we can add a bit $b$ to a carry-save number by first converting the carry-save number to its natural-number representation, adding $b$, and then converting the result back to a carry-save representation.

Since the range of *ceval* is at least the range of $(b+) \circ \mathit{ceval}$ for any bit $b$, the specification (5) has a solution for *cadd* $b$. Since $(b+) \circ \mathit{ceval}$ is not injective, the specification has many solutions. Different solutions can, for example, give different numbers of trailing $(0,0)$ pairs in the result list.

Expressing the relational specification (5) in the functional form of (3), and then using extensionality, gives our working specification:

$$\mathit{ceval}\ (\mathit{cadd}\ b\ xs) \quad = \quad b + \mathit{ceval}\ xs.$$

Our task now is to find a definition for *cadd* that satisfies this equation. We do this by a constructive induction on $xs$. In the base-case $xs = [\,]$, we end up with an assumption that gives a definition for *cadd* $b$ $[\,]$. In the inductive-case $xs = (x,y) : xs$, we get an assumption that gives a recursive definition for *cadd* $b$ $((x,y):xs)$ in terms of *cadd* $b'$ $xs$, where $b'$ is a bit computed from $b$ and $(x,y)$.

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

$$
\begin{array}{ll}
 & ceval\ (cadd\ b\ [\,]) = b + ceval\ [\,] \\
\Leftrightarrow & \qquad \text{unfolding } ceval \\
 & ceval\ (cadd\ b\ [\,]) = b + 0 \\
\Leftrightarrow & \qquad \text{folding } ceval \\
 & ceval\ (cadd\ b\ [\,]) = ceval\ [(b,0)] \\
\Leftarrow & \qquad \text{Leibniz law: } f\ x = f\ y \Leftarrow x = y \\
 & cadd\ b\ [\,] = [(b,0)].
\end{array}
$$

We conclude that the definition $cadd\ b\ [\,] = [(b,0)]$ satisfies the specification in the base-case. Note that in the "folding *ceval*" step above, replacing $b$ by $ceval\ [(b,0)]$ is not the only possibility: $ceval\ [(0,b)]$, $ceval\ [(b,0),(0,0)]$, etc., are equally valid. Choosing $eval\ [(b,0)]$ means that the result list produced by $cadd\ b\ xs$ will have no trailing $(0,0)$ pairs.

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

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

Now, because the expression $b{+}x{+}y$ can have the value 3, which cannot be expressed as a sum of two bits, it is not possible to fold *ceval* at this point. We proceed in fact by splitting the value $x+y$ into two bits: $(x+y)\ \mathbf{mod}\ 2$ and $(x+y)\ \mathbf{div}\ 2$. The first bit will be paired with the incoming carry $b$ to form a bit-pair in the output carry-save number, and the second will become the propogated carry-bit. Splitting in this way avoids a "rippling carry" in the final program.

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

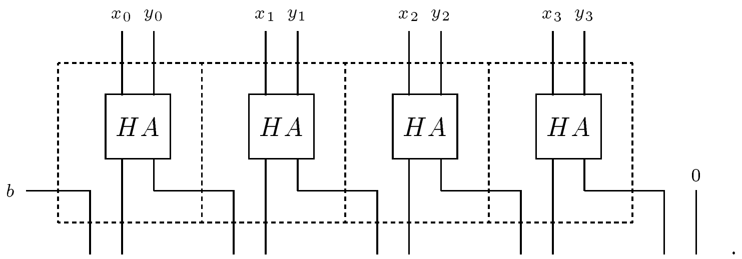
The final term above is of the form *ceval exp*, so we are finished and conclude with the definition $cadd\ b\ ((x,y):xs) = (b,(x+y)\ \mathbf{mod}\ 2) : cadd\ ((x+y)\ \mathbf{div}\ 2)\ xs$.

In summary, we have derived a functional program

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

that satisfies the specification $cadd\ b \subseteq ceval^{-1} \circ (b+) \circ ceval$.

Picturing an instance of *cadd*, for example $cadd\ b\ [(x_0,y_0),\ (x_1,y_1),\ (x_2,y_2),\ (x_3,y_3)]$, illustrates that the carry-save adder has no rippling carry, and hence the addition can be done in parallel in constant time:




The component $HA\ (x,y) = ((x+y)\ \mathbf{mod}\ 2), (x+y)\ \mathbf{div}\ 2))$ above is usually called a *half-adder*. If a large number of binary additions are to be done (such as in a multiplier circuit), a considerable speed-up can be obtained by first converting to carry-save numbers, and then doing all the additions using carry-save adders. Using this technique, the carry propogation which occurs when the final carry-save number is converted back to binary is amortized over many additions.

The carry-save adder is an example which has proved difficult to derive fully-formally using the relational calculus Ruby (Jones and Sheeran, 1992).

## 4 A two-stage example: base conversion

For our second example we turn to the problem of converting numbers from one base to another. This example turns out to be particularly interesting because in the process of its derivation we construct an auxiliary representation changer.

A function *conv* that converts a number represented in base $m$ to a number represented in base $n$ can be specified by the requirement that

$$conv \subseteq (eval_n)^{-1} \circ eval_m,$$

where $eval_b$ converts a number in base $b$ to the corresponding natural number:

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

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 a solution exists for *conv*; since it is not injective, there may more than one solution.

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

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

As for the carry-save adder, we synthesize *conv* by a constructive induction on $xs$. In the base-case $xs = [\,]$, unfolding $eval_m$ immediately results in the definition $conv\ [\,] = [\,]$. For the inductive case $xs = x : xs$, we calculate as follows:

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

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

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

This equation expresses that $convd\ ys$ is *itself* a representation changer, which takes a digit in base $m$ and yields a number in base $n$. The auxiliary function *convd* is constructed by a double induction on its two arguments. A simple calculation gives the base-case: $convd\ [\,]\ 0 = [\,]$. For the first inductive case, $ys = [\,]$ and $x \neq 0$, manipulating the left-hand side of equation (6) results in the definition $convd\ [\,]\ x = x \mathbf{\ mod\ } n : convd\ [\,]\ (x \mathbf{\ div\ } n)$:

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

For the induction hypothesis to be applicable above, we must assume $n > 1$. For the second inductive case, $ys = y : ys$, splitting $x + m * y$ into $(x + m * y) \mathbf{\ mod\ } n$

and $(x + m * y)\ \mathbf{div}\ n$ ensures that the output list contains only base-$n$ digits as required. Such a splitting was also the essential step in establishing the previous induction step.

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

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

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

The auxiliary function *convd* is defined as follows:

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

This base conversion program can be readily implemented as a circuit (Davio *et al.*, 1983). In our opinion, it would be a non-trivial exercise to arrive at this program without the use of formal reasoning. The reader might like to compare the above functional derivation with the corresponding relational derivation in (Hutton, 1992).


## *Acknowledgements*

We would like to thank Jeroen Fokker, Maarten Fokkinga, Johan Jeuring, Lambert Meertens, and the JFP referees, for their comments and suggestions.

## References


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

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

Jones, G. and Sheeran, M. 1990. Circuit design in Ruby. In Staunstrup, editor, *Formal Methods for VLSI Design*, Amsterdam. Elsevier Science Publications.

Jones, G. and Sheeran, M. 1991. Relations and refinement in circuit design. In Morgan, editor, *Proc. BCS FACS Workshop on Refinement*, Workshops in Computing. Springer-Verlag.

Jones, G. and Sheeran, M. 1992. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer-Verlag.

Ross, K.A. and Wright, C.R.B. 1992. *Discrete Mathematics*. Prentice-Hall, New Jersey.

Runciman, C. and Jagger, N. 1990. Relative specification and transformational re-use of functional programs. *Lisp and Symbolic Computation*, 3:21–37.

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