# Inversion of Functions

*Reinhold Heckmann*

FB 10 - Informatik
Universität des Saarlandes
D-6600 Saarbrücken
Bundesrepublik Deutschland

1989-06-01

*ABSTRACT*

Finding an algorithm for the inverse of an explicitly given function is important for program and data type development. We informally present a method to derive an algorithm for the inverse from algorithms for given functions.

**Footnote:** This research was supported by the Commission of the European Communities under Esprit Project Ref. No. 390 (PROSPECTRA).

# 1 Introduction

Assume we are given a function $f : t_1 \times \cdots \times t_n \to t_0$ by an algorithm in a functional language. We want to derive from it an algorithm for the inverse function $f^{-1} : t_0 \to t_1 \times \cdots \times t_n$, or for partial inverses

$$f_{i_1 \ldots i_k} : T_{j_1} \times \cdots \times T_{j_r} \to T_{i_1} \times \cdots \times T_{i_k}$$

where $\{j_1, \ldots, j_r\} = \{1, \ldots, n, 0\} \setminus \{i_1, \ldots, i_k\}$.

All functions are potentially multi-valued, i.e. they are relations when considered exactly. $f : A \to B$ means: $f$ may be applied to arguments of type $A$, and returns zero, one, or many results of type $B$. Thus, the inverse function always exists: $f^{-1} : B \to A$, $f^{-1} y = \{x \in A \mid y \in f x\}$.

The capability to invert functions allows for the usage of arbitrary functions in patterns. Classically, i.e. in functional languages such as Standard ML [9], Hope [3], and Miranda [13], only constructors may be used in patterns, e.g. $f\,(cons\,(h,\, t)) = \ldots$. In the experimental language TrafoLa-H [7, 8], a few predefined functions, e.g. concatenation, could also be used in patterns, since the inverses of these functions were statically included into the interpreter.

An example with arbitrary functions would be

$$\begin{array}{lll}
power\,(b,\, 0) & \to & 1 \\
power\,(b,\, 2 * n) & \to & square\,(power\,(b,\, n)) \\
power\,(b,\, n+1) & \to & b * power\,(b,\, n)
\end{array}$$

A second application is in the area of implementing abstract data types [5, 6]. Let $A$ be an abstract data type and $I$ its implementation with abstraction $\alpha : I \to A$. Let $f$ be an abstract operation and $g$ its implementation. Then the following diagram commutes:

$$\begin{array}{ccc}
A & \xrightarrow{f} & A \\
\uparrow \alpha & & \uparrow \alpha \\
I & \xrightarrow{g} & I
\end{array}$$

Thus, $g$ may be obtained as $g = \alpha^{-1} \circ f \circ \alpha$.

The presentation in the remainder of this article is to be understood as the description of some experimental methods and their practical results. It is by no means a thorough theoretical analysis, and it is not claimed that the methods are completely correct. Thus, the overall flavor of this presentation is similar to [12] where Prolog programs are inverted. In a later paper, we shall develop some sound semantical background for proofs of correctness and further investigation.

# 2 The functional source language $\mathcal{F}$

As mentioned above, the language supports multi-valued functions. They are strongly typed by means of user defined data types. This implies that there is a distinction between the constructors of the data types and the names of user defined functions. A term is said to be in normal form iff it contains no function symbols, i.e. it only consists of constructors.

Functions are defined by a set of rules whose left hand side is headed by the function symbol to be defined. In contrast to classical functional languages, the subterms of the left hand side are not restricted to constructor terms. Overlapping left hand sides are allowed; they are – together with inversion of non-injective functions – a source of non-determinism.

A function application is reduced by first reducing all its arguments (call by value), and then applying all possible rules for the function. If no rule is applicable, a failure results and there is no normal form of the term. On the other hand, there may be many different normal forms resulting from different rules being applicable.

**Example 1**

$$\textbf{type}\ t\ ==\ a\ |\ b\ |\ c(t)$$

Data type $t$ has three constructors: two nullary ones ($a$ and $b$) and the unary constructor $c$.

$$f : t \to t, \qquad \begin{array}{lcl} fa & \to & a \\ fa & \to & b \\ f(c(x)) & \to & fx \end{array}$$

where $x$ is a variable of type $t$. □

Sometimes, we shall use an abbreviation for rules with the same left hand side, e.g. $fa \ \to \ a \ | \ b$ instead of the first two rules above.

Possible reduction sequences with the rules of example 1:

$$f(c(c(a))) \to f(c(a)) \to f(a) \begin{array}{l} \nearrow a \\ \searrow b \end{array}$$

$$f(c(b)) \to f(b) \dashv$$

where '$\dashv$' means failure.

## 3 The relational internal form $\mathcal{R}$

Inversion of a function given in the language $\mathcal{F}$ is done by first translating it into a relational internal form (language $\mathcal{R}$). The definition of the inverse function in $\mathcal{R}$ is derived by some transformations. It may be translated back into $\mathcal{F}$ or directly executed.

$$\begin{array}{llcl}
\mathcal{F} & \text{functional definition of } f & & \text{functional definition of } f^{-1} \\
 & \downarrow & & \uparrow \\
\mathcal{R} & \text{relational form for } f & \longrightarrow & \text{relational form for } f^{-1} \\
 & & & \downarrow \\
 & & & \text{execution}
\end{array}$$

We shall not formally introduce the mappings from $\mathcal{F}$ to $\mathcal{R}$ and vice versa, but explain them by examples.

Consider first the addition of Peano-style natural numbers.

**Example 2**

$$\textbf{type}\ nat\ ==\ 0\ |\ s\,(nat)$$

where '$s$' denotes the successor on naturals.

$$\begin{array}{lcccl}
\text{'+'}:\ nat & \times & nat & \to & nat \\
a & + & 0 & \to & a \\
a & + & s\,(b') & \to & s\,(a+b')
\end{array}$$

Relational form:

$$\begin{array}{lcl}
(a+b=c) & == & b=0,\ \ c=a \\
 & | & s\,(b')=b,\ \ a+b'=c',\ \ s\,(c')=c \qquad \Box
\end{array}$$

We assume for the moment that there are no 'logical' variables, i.e. the language internally works with ground terms not containing free variables. Thus, each variable in the clause above is either not (yet) bound, or bound to a ground term. This distinction is important to understand the algorithm described by the clause. With the normal usage of addition, the arguments $a$ and $b$ are bound, whereas the result $c$ is unbound in the beginning. The goal of the algorithm is to bind $c$.

In the sequel, we shall adopt the convention to mark unbound variables by a question mark '?':

$$\begin{array}{lcl}
(a+b=c?) & == & b=0,\ \ c?=a \\
 & | & s\,(b'?)=b,\ \ a+b'=c'?,\ \ s\,(c')=c?
\end{array}$$

Interpretation:

How to add $a$ and $b$ giving $c$:
  Check whether $b$ equals 0
  If so, assign (the value bound to) $a$ to $c$
or: Check whether $b$ is the successor of some number $b'$ and compute this number
  If so, recursively add $a$ and $b'$ giving $c'$
  (If this was successful) let $c$ be the successor of $c'$

This is exactly the originally given functional algorithm, i.e. the translation into $\mathcal{R}$ was correct.

Now, we shall consider the various kinds of literals that may occur in clauses.

Let $c$ be an $n$-ary constructor of some data type. There are two kinds of literals involving $c$:

$c\,(x_1, \ldots, x_n) = y?$ :
: $y$ is constructed from the arguments. Construction cannot fail and corresponds to an occurrence of the constructor in the right hand side of a rule in the functional definition.

$c\,(x_1?, \ldots, x_n?) = y$ :
: It is checked whether $y$ is really constructed with $c$, and if so, the variables $x_i$ are bound to the subterms of $y$. Thus, this literal may fail. It corresponds to an occurrence of the constructor in the left hand side of a rule.

In the special case of a nullary constructor $null$, $null = y?$ means 'let $y$ be $null$', and $null = y$ means 'check whether $y$ equals $null$'.

Further literals may be equality checks $x = y$, assignments $x? = y$, function calls $f\,(x_1, \ldots, x_n) = y?$, and inverse function calls where some arguments are marked by '?' and/or the result is not marked.

# 4 Inverses of addition

## 4.1 The original form

Consider the relational definition of addition again:

$$\begin{array}{rcl} (a + b = c?) & == & b = 0, \;\; c? = a \\ & | & s\,(b'?) = b, \;\; a + b' = c'?, \;\; s\,(c') = c? \end{array}$$

It is easy to see that this algorithm terminates, since the recursive call is done with argument $b'$, and $b'$ is – as term – smaller than the original argument $b$. The algorithm is deterministic, because the two cases are disjoint – $b = 0$ and $b = s\,(b')$ contradict each other – and all used predicates are deterministic.

## 4.2 Looking for the second summand

We now assume that $a$ and $c$ are known and $b$ is wanted. By simple relabeling, we obtain the following algorithm:

$$\begin{array}{lcl} (a + b? = c) & == & b? = 0, \quad c = a \\ & | & s\,(b'?) = b?, \quad a + b' = c'?, \quad s\,(c') = c \end{array}$$

Interpretation:

Let $b$ be zero
But this is a result only if $c$ equals $a$
or: Build all $b$, $b'$ such that $s\,(b') = b$
Add $a$ to $b'$ giving $c'$ with the original algorithm in subsection 4.1
The obtained values for $b$ are final results only if $s\,(c')$ equals $c$

This is by no means a good algorithm, since first many – even infinitely many – values are produced for $b$, and most of them have to be discarded afterwards. A much better algorithm is obtained by reordering the literals:

$$\begin{array}{lcl} (a + b? = c) & == & c = a, \quad b? = 0 \\ & | & s\,(c'?) = c, \quad a + b'? = c', \quad s\,(b') = b? \end{array}$$

Interpretation:

Does $a$ equal $c$?
If so, let $b$ be zero
or: Check whether $c$ is successor of some number $c'$
Determine $b'$ recursively from the equation $a + b' = c'$
Let $b$ be the successor of $b'$

The algorithm terminates, since the recursive call is performed with $c'$ and this is a smaller term than $c$. It cannot be recognized to be deterministic, since the two alternatives are not perceptibly disjoint. They are disjoint, but this is revealed only by some mathematic reasoning.

Translated back into the functional language $\mathcal{F}$, the algorithm becomes

$$\begin{array}{lcl} a - a & \to & 0 \\ s\,(c') - a & \to & s\,(c' - a) \end{array}$$

Examples:

$$\begin{array}{lllll} 2 - 3 & \to & s\,(1 - 3) & \to & s\,(s\,(0 - 3)) \dashv \\ 3 - 2 & \to & s\,(2 - 2) & \to & s\,(0) = 1 \\ & & & \searrow & s\,(1 - 2) \quad \to \quad s\,(0 - 2) \dashv \end{array}$$

## 4.3 Looking for the first summand

Now, $b$ and $c$ are known and $a$ is searched. Relabeling yields

$$\begin{array}{rcl} (a? + b = c) & == & b = 0, \;\; c = a? \\ & | & s\,(b'?) = b, \;\; a? + b' = c'?, \;\; s\,(c') = c \end{array}$$

Again, the algorithm is improved by rearranging the literals:

$$\begin{array}{rcl} (a? + b = c) & == & b = 0, \;\; c = a? \\ & | & s\,(b'?) = b, \;\; s\,(c'?) = c, \;\; a? + b' = c' \end{array}$$

Interpretation:

Does $b$ equal zero?
If so, let $a$ be $c$
or: Check whether $b$ is successor of some number $b'$
Check whether $c$ is successor of some number $c'$
Determine $a$ recursively from the equation $a + b' = c'$

The algorithm terminates, since the term $b'$ is smaller than $b$ (and also $c'$ is smaller than $c$). It is deterministic, since the alternatives are disjoint. The corresponding algorithm in $\mathcal{F}$ is as follows:

$$\begin{array}{rcl} c \;\; - \;\; 0 & \rightarrow & c \\ s\,(c') - s\,(b') & \rightarrow & c' - b' \end{array}$$

Examples: $\begin{array}{l} 2-3 \;\rightarrow\; 1-2 \;\rightarrow\; 0-1 \dashv \\ 3-2 \;\rightarrow\; 2-1 \;\rightarrow\; 1-0 \;\rightarrow\; 1 \end{array}$

Remarks:

- Instead of $s\,(b'?) = b, \; s\,(c'?) = c$, we could also have written $s\,(c'?) = c, \; s\,(b'?) = b$ . With this example, there is no difference, but generally the question arises how to detect the optimal order of literals.

- The second algorithm for subtraction coming from $? + b = c$ is clearly more efficient than the first one derived from $a + ? = c$. But since the system does not know that addition is commutative, it cannot always use the better algorithm.

### 4.4 Searching for both summands

We give immediately the final algorithm with relabeled and rearranged literals:

$$
\begin{array}{lcl}
(a? + b? = c) & == & b? = 0, \;\; c = a? \\
& | & s\,(c'?) = c, \;\; a? + b'? = c', \;\; s\,(b') = b?
\end{array}
$$

The algorithm terminates, since $c'$ is smaller than $c$. The alternatives are not recognizably disjoint, and in fact, they are not, i.e. there may be more than one solution. But termination ensures that the number of solutions is finite.

Interpretation:

Let $b$ be zero and $a$ be $c$
or: Match $s\,(c')$ against $c$
Compute recursively all solutions $(a,\, b')$ of $a + b' = c$
Final results are the pairs $(a,\, b)$ where $b = s\,(b')$.

After translating back to $\mathcal{F}$, we obtain:

$$
\begin{array}{lcl}
f\,(c) & \to & (c,\, 0) \\
f\,(s\,(c')) & \to & \textbf{let } (a,\, b') = f\,(c') \textbf{ in } (a,\, s\,(b'))
\end{array}
$$

Note that both rules have equal rights.

Examples:

$$
\begin{array}{lcl}
f\,(0) & \to & (0,\, 0) \\
f\,(1) & \to & (1,\, 0) \\
& \searrow & (a,\, s\,(b')) \text{ where } (a,\, b') = f\,(0) \;\; \to \;\; (0,\, s\,(0)) = (0,\, 1) \\
f\,(2) & \to & (2,\, 0) \\
& \searrow & (a,\, s\,(b')) \text{ where } (a,\, b') = f\,(1) \;\; \to \;\; (1,\, 1) \,|\, (0,\, 2)
\end{array}
$$

### 4.5 Only the first summand is known

$$
\begin{array}{lcl}
(a + b? = c?) & == & b? = 0, \;\; c? = a \\
& | & a + b'? = c'?, \;\; s\,(b') = b?, \;\; s\,(c') = c?
\end{array}
$$

The recursive call is performed with the same argument as the original one, and the alternatives are not recognizably disjoint. Thus, the system can't say anything about this algorithm. There are indeed many solutions, namely infinitely many, and thus, the algorithm does not terminate. It may be safely executed by means of lazy evaluation. The call $a + ? = ?$ is defined by itself and produces an infinite stream of solutions.

$$
f\,(a) \;\; \to \;\; (0,\, a) \;\; | \;\; \textbf{let } (b',\, c') = f\,(a) \textbf{ in } (s\,(b'),\, s\,(c'))
$$

## 4.6 Only the second summand is known

$$\begin{aligned}(a? + b = c?) \quad == \quad & b = 0, \quad c? = a? \\ \mid \quad & s\,(b'?) = b, \quad a? + b' = c'?, \quad s\,(c') = c?\end{aligned}$$

This clause contains the nasty literal $c? = a?$. It has to enumerate all pairs $c = a = 0$, $c = a = 1$, etc. . Thus, it does not terminate and is not deterministic. This makes the whole algorithm non-deterministic and non-terminating, although the recursion decreases and the alternatives are disjoint.

## 4.7 Nothing is known

$$\begin{aligned}(a? + b? = c?) \quad == \quad & b? = 0, \quad c? = a? \\ \mid \quad & a? + b'? = c'?, \quad s\,(b') = b?, \quad s\,(c') = c?\end{aligned}$$

Here, two infinite streams have to be combined – one from $c? = a?$ and one from the non-decreasing recursion. The implementation should ensure fair evaluation and merging of the streams.

Remark:

> Alternatively, generators for natural numbers and normal addition could be used:
>
> $$(a? + b? = c?) \;==\; a? = nat\,(), \quad b? = nat\,(), \quad a + b = c?$$
>
> We guess that it is difficult to determine whether this is better or worse than the previous solution.
>
> Generators such as *nat* will be considered more closely in subsection 5.1 below.

## 4.8 All is known

Given three numbers $a$, $b$, and $c$, the question is: does $a + b$ equal $c$?

$$\begin{aligned}(a + b = c) \quad == \quad & b = 0, \quad c = a \\ \mid \quad & s\,(b'?) = b, \quad s\,(c'?) = c, \quad a + b' = c'\end{aligned}$$

The algorithm terminates and is deterministic.

If this algorithm is translated back into $\mathcal{F}$, what should it return? If the test fails, there is no result; otherwise, the result should be an empty tuple () in analogy to the question $a? + b? = c$ where a pair is returned, and $a + b = c?$ where the result is just one number.

The functional form of the algorithm would look like

$$\begin{aligned}&addcheck\,(a, \quad 0 \quad , \quad a \quad ) \;\rightarrow\; () \\ &addcheck\,(a, \; s\,(b'), \; s\,(c')) \;\rightarrow\; addcheck\,(a, \, b', \, c')\end{aligned}$$

Remark:

This algorithm often performs more efficient than the obvious algorithm that computes $a + b$ and compares the result with $c$. For instance, $addcheck\,(2, 200, 0)$ immediately fails without a recursive call whereas the computation of $2 + 200$ needs 200 recursive calls.

# 5 Some derived algorithms

## 5.1 Value generators

Remember the definition of type $nat$:

$$\textbf{type}\ nat\ ==\ 0\ \mid\ s\,(nat)$$

This may also be understood as definition of a function:

$$\begin{array}{lcl} nat\,() & \rightarrow & 0 \\ nat\,() & \rightarrow & s\,(nat\,()) \end{array}$$

This function produces the results 0, 1, 2, ... when applied to (). In $\mathcal{R}$, the definition looks like

$$(nat\,() = n?)\ ==\ n? = 0\ \mid\ nat\,() = n'?,\ \ s\,(n') = n?$$

This is a non-terminating, non-deterministic algorithm producing an infinite stream of results. The inverse is computed by the following algorithm:

$$(nat\,() = n)\ ==\ n = 0\ \mid\ s\,(n'?) = n,\ \ nat\,() = n'$$

This algorithm terminates and is deterministic. It checks whether the input is really a natural number. In a well-typed application, the answer must always be affirmative such that the test is useless.

## 5.2 From addition to comparison, first kind

Assume we define $a \leq b$ iff there is a number $x$ such that $a + x = b$. In $\mathcal{F}$, this would be written

$$a\ \leq\ (a\ +\ \_)\quad \rightarrow\quad ()$$

using the pattern $a\ +\ \_$ that inverts addition. '_' is a wild card in patterns, or an anonymous variable. It is used if we are only interested in the existence of a value, but not in the value itself.

In $\mathcal{R}$, the definition looks like

$$(a \leq c)\ ==\ a\ +\ \_? = b$$

This may be left as it is, or improved by partial evaluation:

$$\begin{aligned}(a \le c) \;&==\; a + x? = c \\ &==\; a = c,\;\; x? = 0 \\ &\;\;\;\mid\;\; s\,(c'?) = c,\;\; a + x'? = c',\;\; s\,(x') = x?\end{aligned}$$

by unfolding addition.

The actual value of $x$ is not important. Hence, all literals computing $x$ may be dropped, if one is sure that they cannot fail. This is true for $x? = 0$ and $s\,(x') = x?$. If these two literals are removed, one can see that the value of $x'$ is not needed, too. We obtain

$$\begin{aligned}&==\; a = c \\ &\;\;\;\mid\;\; s\,(c'?) = c,\;\; a\, +\, \_? = c'\end{aligned}$$

whence by folding

$$(a \le c) \;==\; a = c \;\mid\; s\,(c'?) = c,\;\; a \le c'$$

This algorithm terminates. Determinism is not recognizable, but it holds.

Various inverses:

$$\begin{aligned}(a? \le c) \;&==\; a? = c \;\mid\; s\,(c'?) = c,\;\; a? \le c' \\ (a \le c?) \;&==\; a = c? \;\mid\; a \le c'?,\;\; s\,(c') = c? \\ (a? \le c?) \;&==\; a? = c? \;\mid\; a? \le c'?,\;\; s\,(c') = c?\end{aligned}$$

The first algorithm terminates, but is not deterministic; the second one does not terminate and produces an infinite stream of results; and the last one produces two streams to be merged.

## 5.3 Comparison, second kind

Now we define $b \le c$ iff there is a number $x$ such that $x + b = c$. Proceeding as above, we obtain

$$\begin{aligned}(b \le c) \;&==\; x? + b = c \\ &==\; b = 0,\;\; c = x? \\ &\;\;\;\mid\;\; s\,(b'?) = b,\;\; s\,(c'?) = c,\;\; x? + b' = c'\end{aligned}$$

Only $c = x?$ may be dropped. Folding results in

$$(b \le c) \;==\; b = 0 \;\mid\; s\,(b'?) = b,\;\; s\,(c'?) = c,\;\; b' \le c'$$

This algorithm terminates and is deterministic. Some inverses:

$$\begin{aligned}(b? \le c) \;&==\; b? = 0 \;\mid\; s\,(c'?) = c,\;\; b'? \le c',\;\; s\,(b') = b? \\ (b \le c?) \;&==\; b = 0 \;\mid\; s\,(b'?) = b,\;\; b' \le c'?,\;\; s\,(c') = c?\end{aligned}$$

The first algorithm terminates, but is not deterministic. At first glance, the second one seems to be terminating – $b'$ is smaller than $b$ – and deterministic – the alternatives are disjoint. But we know that there are infinitely many solutions. Where do they come from? Note that the variable $c$ is left unbound by the first alternative. In order to bind $c$, a value generator $c? = nat\,()$ must be added that produces an infinite stream of solutions. Hence, the final outcome is

$$(b \le c?) \quad == \quad b = 0,\ \ c? = nat\,() \ \mid\ s\,(b'?) = b,\ \ b' \le c'?,\ \ s\,(c') = c?$$

and analogously

$$(b? \le c?) \ == \ b? = 0,\ \ c? = nat\,() \ \mid\ b'? \le c'?,\ \ s\,(b') = b?,\ \ s\,(c') = c?$$

# 6 Multiplication

## 6.1 Normal case: both factors are known

Functional definition:

$$\begin{array}{lcl} a * \ 0 & \rightarrow & 0 \\ a * s\,(b) & \rightarrow & a * b + a \end{array}$$

Note that the variable $a$ does not occur in the right hand side of the first rule, and occurs twice in the second one. With addition, all variables occurred exactly once on either side of the rules.

Translation into $\mathcal{R}$:

$$\begin{array}{lcl} (a * b = c?) & == & b = 0,\ \ c? = 0 \\ & \mid & s\,(b'?) = b,\ \ a * b' = c'?,\ \ c' + a = c? \end{array}$$

This algorithm is terminating and deterministic since the recursion decreases, the cases are disjoint, and the mode of addition used here is deterministically terminating.

## 6.2 Searching for the second factor

We immediately show the result of relabeling and reordering the algorithm above:

$$\begin{array}{lcl} (a * b? = c) & == & c = 0,\ \ b? = 0 \\ & \mid & c'? + a = c,\ \ a * b'? = c',\ \ s\,(b') = b? \end{array}$$

We know that the mode of addition used here is terminating and deterministic such that this algorithm is not too bad so far. But neither termination nor determinism are directly recognizable, and indeed they are not given.

The functional form of this algorithm is as follows:

$$\begin{array}{lcl} 0\,/\,a & \rightarrow & 0 \\ c\,/\,a & \rightarrow & s\,((c - a)/a) \end{array}$$

Examples:

$$\begin{array}{lllllllll} 12/4 & \to & s\,(8/4) & \to & s^2\,(4/4) & \to & s^3\,(0/4) & \to & s^3\,(0) = 3 \\ & & & & & & & \searrow & s^4\,((0-4)/4) \dashv \end{array}$$

12/4 terminates and returns 3.

$$5/4 \;\to\; s\,(1/4) \;\to\; s^2\,((1-4)/4) \dashv$$

5/4 terminates and gives no solution.

$$\begin{array}{lllllllll} 0/0 & \to & 0 & & & & & & \\ & \searrow & s\,(((0-0)/0) & \to & s\,(0/0) & \to & s\,(0) = 1 & & \\ & & & & & \searrow & s^2\,((0-0/0) & \to & \cdots \end{array}$$

0/0 returns an infinite stream of solutions 0, 1, 2, ...

$$1/0 \;\to\; s\,((1-0)/0) \;\to\; s\,(1/0) \;\to\; \cdots$$

This is the worst behavior possible for a process: it does not terminate, but never produces any result.

## 6.3 Searching for the first factor

$$\begin{array}{lll} (a? * b = c) & == & b = 0,\;\; c = 0,\;\; a? = nat\,() \quad \text{to bind } a \\ & | & s\,(b'?) = b,\;\; a? * b' = c'?,\;\; c' + a = c \end{array}$$

or ordered the other way round:

$$\begin{array}{lll} (a? * b = c) & == & b = 0,\;\; c = 0,\;\; a? = nat\,() \quad \text{to bind } a \\ & | & s\,(b'?) = b,\;\; c'? + a? = c,\;\; a * b' = c' \end{array}$$

Both algorithms are not really recursive, since multiplication occurs in a different mode in the second alternative. Both share the drawback that first many results are generated by the second literal which are mostly rejected by the third literal. The second algorithm should be preferred, since $c'? + a? = c$ is known to terminate, i.e. to produce only a finite number of results. Rejecting much of a finite number is inefficient, but rejecting much of an infinite number is dangerous if only a finite number of results remains, since the algorithm does not terminate after producing the last result.

Algorithm translated back into $\mathcal{F}$:

$$\begin{array}{ll} 0\,/\,0 & \to\; nat\,() \\ c\,/\,s\,(b') & \to\; \textbf{let}\;\; c' + a = c\;\; \textbf{in if}\;\; a * b' = c'\;\; \textbf{then}\;\; a\;\; \textbf{else fail} \end{array}$$

This algorithm looks a bit strange, and in fact, it is:

$$
\begin{array}{lllll}
12/4 & \rightarrow & b' = 3; & c' = 0 \text{ and } a = 12; & 12 * 3 = 0? \text{ No} \\
 & & | & c' = 1 \text{ and } a = 11; & 11 * 3 = 1? \text{ No} \\
 & & \ldots | & c' = 8 \text{ and } a = 4; & 4 * 3 = 8? \text{ No} \\
 & & | & c' = 9 \text{ and } a = 3; & 3 * 3 = 9? \text{ Yes } \Rightarrow \text{Result } 3 \\
 & & | & c' = 10 \text{ and } a = 2; & 2 * 3 = 10? \text{ No} \\
 & & | & c' = 11 \text{ and } a = 1; & 1 * 3 = 11? \text{ No} \\
 & & | & c' = 12 \text{ and } a = 0; & 0 * 3 = 12? \text{ No}
\end{array}
$$

$$
\begin{array}{lllll}
5/4 & \rightarrow & b' = 3; & c' = 0 \text{ and } a = 5; & 5 * 3 = 0? \text{ No} \\
 & & | & c' = 1 \text{ and } a = 4; & 4 * 3 = 1? \text{ No} \\
 & & | & c' = 2 \text{ and } a = 3; & 3 * 3 = 2? \text{ No} \\
 & & | & c' = 3 \text{ and } a = 2; & 2 * 3 = 3? \text{ No} \\
 & & | & c' = 4 \text{ and } a = 1; & 1 * 3 = 4? \text{ No} \\
 & & | & c' = 5 \text{ and } a = 0; & 0 * 3 = 5? \text{ No } \Rightarrow \text{no solution}
\end{array}
$$

$$0/0 \ \rightarrow \ nat() \ \rightarrow \ 0 \ | \ 1 \ | \ 2 \ | \ \ldots$$

$$1/0 \dashv \ \text{(failure)}$$

The inverse searching for the first factor presented here is in most cases much less efficient than searching for the second factor. The only exception is with equations $0 * ? = c$ resp. $? * 0 = c$ where $c \neq 0$. Then, the search for the first factor immediately fails, whereas the search for the second factor keeps on searching infinitely without finding anything.

## 6.4 Does $a * b = c$ really hold?

Here, we assume that $a$, $b$, and $c$ are all known, and we want to check whether $a * b$ equals $c$. Again, there are two possibilities to order the literals in the clause:

$$
\begin{array}{lll}
(a * b = c) & == & b = 0, \ \ c = 0 \\
 & | & s(b'?) = b, \ \ a * b' = c'?, \ \ c' + a = c \\
(a * b = c) & == & b = 0, \ \ c = 0 \\
 & | & s(b'?) = b, \ \ c'? + a = c, \ \ a * b' = c'
\end{array}
$$

Both algorithms terminate and are deterministic. Only the second one is really recursive.

Examples:

$$
\begin{array}{llllll}
(1) & 2*3=6? & \rightarrow & b'=2,\;\; c'=2*2=4,\;\; 4+2=6?\;\; \text{Yes} \\
(2) & 2*3=6? & \rightarrow & b'=2,\;\; c'=6-2=4,\;\; 2*2=4? \\
 & & \rightarrow & b'=1,\;\; c'=4-2=2,\;\; 2*1=2? \\
 & & \rightarrow & b'=0,\;\; c'=2-2=0,\;\; 2*0=0?\;\; \text{Yes} \\
(1) & 2*3=4? & \rightarrow & b'=2, c'=2*2=4, 4+2=4?\;\; \text{No} \\
(2) & 2*3=4? & \rightarrow & 2*2=2? \;\rightarrow\; 2*1=0? \;\rightarrow\; c'?+2=0 \dashv
\end{array}
$$

It is impossible to see which algorithm is better without a thorough analysis. There is even a third algorithm:

$$(a*b=c) \;==\; a*b=c'?,\;\; c=c'$$

i.e. the product of $a$ and $b$ is computed and afterwards compared with $c$.

## 6.5 Looking for both factors

$$
\begin{array}{lll}
(a?*b?=c) & == & c=0,\;\; b?=0,\;\; a?=nat\,() \\
 & | & c'?+a?=c,\;\; a*b'?=c',\;\; s\,(b')=b?
\end{array}
$$

This algorithm is not really recursive. If the multiplicative literal preceded the additive one, the clause would contain $a?*b'?=c'?$, $c'+a=c$ what looks worse. The order we choose has also the advantage that $c'?+a?=c$ is known to terminate.

Examples:

$$
\begin{array}{l}
a * b = 6 \quad \rightarrow \quad c' + a = 6 \quad \rightarrow \\
\quad \rightarrow \; c' = 0 \text{ and } a = 6 \;\rightarrow\; b = s\,(0\,/\,6) = 1 \\
\quad\;\; \mid \; c' = 1 \text{ and } a = 5 \;\rightarrow\; b = s\,(1\,/\,5) \qquad \text{-} \\
\quad\;\; \mid \; c' = 2 \text{ and } a = 4 \;\rightarrow\; b = s\,(2\,/\,4) \qquad \text{-} \\
\quad\;\; \mid \; c' = 3 \text{ and } a = 3 \;\rightarrow\; b = s\,(3\,/\,3) = 2 \\
\quad\;\; \mid \; c' = 4 \text{ and } a = 2 \;\rightarrow\; b = s\,(4\,/\,2) = 3 \\
\quad\;\; \mid \; c' = 5 \text{ and } a = 1 \;\rightarrow\; b = s\,(5\,/\,1) = 6 \\
\quad\;\; \mid \; c' = 6 \text{ and } a = 0 \;\rightarrow\; b = s\,(6\,/\,0) \quad \text{non-terminating}
\end{array}
$$

Solutions: (6, 1) (3, 2) (2, 3) (1, 6) $<$*looping*$>$

$$
\begin{array}{l}
a * b = 0 \;\rightarrow\; b = 0,\; a = nat\,() \;\rightarrow\; (0,\,0)\,(1,\,0)\,(2,\,0)\,\ldots \\
\qquad\qquad \mid \; c' = 0 \text{ and } a = 0 \;\rightarrow\; b = s\,(0/0) \;\rightarrow\; (0,\,1)\,(0,\,2)\,\ldots
\end{array}
$$

### 6.6 Other modes

$$
\begin{array}{lcl}
(a? * b = c?) & == & b = 0,\; c? = 0,\; a? = nat\,() \\
 & \mid & s\,(b'?) = b,\; a? * b' = c'?,\; c' + a = c? \\
(a * b? = c?) & == & b? = 0,\; c? = 0 \\
 & \mid & a * b'? = c'?,\; s\,(b') = b?,\; c' + a = c? \\
(a? * b? = c?) & == & b? = 0,\; c? = 0,\; a? = nat\,() \\
 & \mid & a? * b'? = c'?,\; s\,(b') = b?,\; c' + a = c?
\end{array}
$$

### 6.7 Summary of multiplication

The double occurrence of $a$ in the right hand side of the second rule of multiplication causes the '+' and the '$*$' literal in the relational form to be doubly linked: once via the intermediate result $c'$ and once via $a$ itself. This makes the task difficult to find a good order for these two literals. Variable $c'$ is never bound in the first literal, since it does not occur outside; $a$ might be bound as well as unbound.

Another problem is the non-termination of $0 * ? = 1$. It is not clear whether and how it may be avoided by transformations.

## 7 The factorial function

### 7.1 The simple recursive definition

$$
\begin{array}{lcl}
fac(0) & \rightarrow & 1 \\
fac(s\,(n)) & \rightarrow & s\,(n) * fac\,(n)
\end{array}
$$

Translation into $\mathcal{R}$ results in

$$\begin{array}{lll} (fac(n) = f?) & == & n = 0,\ \ f? = s(0) \\ & | & s(n'?) = n,\ \ s(n') = n''?,\ \ fac(n') = f'?,\ \ n'' * f' = f? \end{array}$$

The clause might be slightly improved. From $s(n') = n$ and $s(n') = n''$, $n = n''$ follows. Thus, we may omit the construction of $n''$:

$$\begin{array}{lll} (fac(n) = f?) & == & n = 0,\ \ f? = s(0) \\ & | & s(n'?) = n,\ \ fac(n') = f'?,\ \ n * f' = f? \end{array}$$

This improvement is also possible in $\mathcal{F}$ by use of a so-called multi-level pattern:

$$fac(n \,\&\, s(n')) \ \ \rightarrow \ \ n * fac(n')$$

Algorithm for the inverse:

$$\begin{array}{lll} (fac(n?) = f) & == & f = s(0),\ \ n? = 0 \\ & | & n? * f'? = f,\ \ s(n'?) = n,\ \ fac(n') = f' \end{array}$$

This is a poor algorithm, since first many potential solutions are produced that are mostly dropped at the end by the test $fac(n') = f'$. Worse, the algorithm of subsection 6.5 to solve the equation $? * ? = f$ did not terminate although there are only a finite number of results. This implies that the inverse factorial never terminates, even if there are no solutions.

Factorial test: does $fac(n) = f$ hold?

$$\begin{array}{lll} (fac(n) = f) & == & n = 0,\ \ f = s(0) \\ & | & s(n'?) = n,\ \ n * f'? = f,\ \ fac(n') = f' \end{array}$$

or

$$\begin{array}{lll} (fac(n) = f) & == & n = 0,\ \ f = s(0) \\ & | & s(n'?) = n,\ \ fac(n') = f'?,\ \ n * f' = f \end{array}$$

Which one is better? We guess that the first algorithm will be more efficient in case of failure, e.g. $fac(8) = 9$ fails quickly since $9/8$ fails. The second one seems to be better if the test will succeed. Since failure is more likely in case of the factorial function, the first method is preferable. Note that this an informal argument difficult to be stated by an automated system.

## 7.2 Tail-recursive definition of the factorial function

$$\begin{array}{lll} fac(n) & \rightarrow & fac'(n,\ s(0)) \\ fac'(0,\ x) & \rightarrow & x \\ fac'(s(n'),\ x) & \rightarrow & fac'(n',\ s(n') * x) \end{array}$$

With a similar improvement as in the previous subsection, we obtain as relational clause:

$$\begin{array}{lcl} (fac(n) = f?) & == & x? = s(0),\ \ fac'(n, x) = f? \\ (fac'(n, x) = f?) & == & n = 0,\ \ f? = x \\ & | & s(n'?) = n,\ \ n * x = x'?,\ \ fac'(n', x') = f? \end{array}$$

For the inversion, the question arises whether the accumulating parameter $x$ should be assumed as known or not. One first guesses that it should be known.

$$\begin{array}{lcl} (fac'(n?, x) = f) & == & f = x,\ \ n? = 0 \\ & | & n? * x = x'?,\ \ s(n'?) = n,\ \ fac'(n', x') = f \end{array}$$

This is bad, since $n? * x = x'?$ produces an infinite amount of proposals that are discarded by the last literal. Other orders neither lead to a truly recursive clause nor a clause with more suitable properties.

Now let's try what happens if the accumulating $x$ is assumed to be unknown:

$$\begin{array}{lcl} (fac'(n?, x?) = f) & == & n? = 0,\ \ f = x? \\ & | & fac'(n'?, x'?) = f,\ \ s(n') = n?,\ \ n * x? = x' \end{array}$$

This looks quite nice and is truly recursive. There is no pure test at the end. Since the recursion does not decrease, the result is a stream of $(n, x)$ pairs.

The definition for the inverse factorial that belongs to the clause above is as follows:

$$(fac(n?) = f) \ == \ fac'(n?, x?) = f,\ \ x = s(0)$$

The $fac'$ literal must come first to ensure that $x$ is not yet bound.

This looks not too nice, since a value generator is followed by a test.

Functionally, the algorithm may be written

$$invfac(f) \ \rightarrow \ \mathbf{let}\ (n, x) = invfac'(f)\ \mathbf{in\ if}\ x = 1\ \mathbf{then}\ n\ \mathbf{else\ fail}$$

or shorter

$$invfac(f) \ \rightarrow \ \mathbf{let}\ (n, 1) = invfac'(f)\ \mathbf{in}\ n$$

$$\begin{array}{lcl} invfac'(f) & \rightarrow & (0, f) \\ invfac'(f) & \rightarrow & \mathbf{let}\ (n', x') = invfac'(f)\ \mathbf{in\ let}\ n = s(n')\ \mathbf{in}\ (n, x'/n) \end{array}$$

i.e. $invfac'(f)$ creates a stream of $(n, x)$ pairs that is investigated for a pair where $x$ equals 1.

Examples:

$fac(n) = 120$ :

$invfac'(120)$ generates the stream
$(0, 120),\ (1, 120/1) = (1, 120),\ (2, 120/2) = (2, 60),\ (3, 60/3) = (3, 20),$
$(4, 20/4) = (4, 5),\ (5, 5/5) = (5, 1),\ (6, 1/6) = failure,$
i.e. the stream is finite. The rule for $g$ browses through the stream and finds the only solution 5.

$fac(n) = 100$ :

The stream resulting from $invfac'(100)$ is (0, 100), (1, 100), (2, 50). The construction of the next item – (3, 50/3) – fails. There is no solution.

$fac(n) = 1$ :

The stream has two items: (0, 1) and (1, 1). (2, 1/2) fails. Solutions are both 0 and 1.

$fac(n) = 0$ :

In this case, $invfac'(0)$ generates an infinite stream (0, 0), (1, 0), (2, 0) etc. Examination of the stream gives no results. Thus, the algorithm does not terminate, but fails to produce any solution.

Summarizing, this algorithm for the inverse factorial is quite good – except when applied to zero.

# 8 List functions

We shall now leave the natural numbers and consider some functions on lists.

## 8.1 The data type of lists

We shall write '$nil$' for the empty list and use the symbol '::' as right associative infix constructor of lists. Lists are constructed polymorphically over the item type $\alpha$.

$$\textbf{type}\ list(\alpha) \ == \ nil \ | \ \alpha :: list(\alpha)$$

For concrete lists in examples, the abbreviation $[x_1, \ldots, x_n]$ is used instead of $x_1 :: \cdots :: x_n :: nil$.

A function such as $nat : \{()\} \to nat$ does not exist for lists, but

$$\begin{array}{lcl} list : \alpha & \to & list(\alpha) \\ list\ x & \to & nil \\ list\ x & \to & x :: list(x) \end{array}$$

e.g. $$\begin{array}{lcl} list(5) & \to & [\,] \ | \ [5] \ | \ [5, 5] \ | \ [5, 5, 5] \ | \ \ldots \\ list(nat()) & \to & \text{all lists of natural numbers} \end{array}$$

Unfortunately, this function cannot be used as value generator, since its argument is generally unknown. Thus, there are modes of list functions that must be forbidden, if it is not allowed to leave variables unbound.

## 8.2 The append function

### 8.2.1 Functional usage

$$\begin{aligned} app(\ nil\ , x) &\rightarrow x \\ app(h :: t, x) &\rightarrow h :: app(t, x) \end{aligned}$$

Translated into $\mathcal{R}$:

$$\begin{aligned} (app(a, b) = c?) \quad == &\quad a = nil, \ \ b = c? \\ | &\quad h? :: t? = a, \ \ app(t, b) = z?, \ \ h :: z = c? \end{aligned}$$

This is the usual algorithm. It is terminating and deterministic.

### 8.2.2 Computing the first argument

$$\begin{aligned} (app(a?, b) = c) \quad == &\quad b = c, \ \ a? = nil \\ | &\quad h? :: z? = c, \ \ app(t?, b) = z, \ \ h :: t = a? \end{aligned}$$

This algorithm terminates, since $z$ is smaller than $c$. It is deterministic, but this is not easy to see. It is not too efficient, since $c$ must be completely decomposed before a failure may occur.

### 8.2.3 Computing the second argument

$$\begin{aligned} (app(a, b?) = c) \quad == &\quad a = nil, \ \ b? = c \\ | &\quad h? :: t? = a, \ \ h :: z? = c, \ \ app(t, b?) = z \end{aligned}$$

This algorithm is deterministic and terminating. If you do not like the mode of the second '::' in the second alternative, you may write $h? :: t? = a$, $h'? :: z? = c$, $h = h'$, ... The algorithm is quite efficient; failure is often recognized early.

### 8.2.4 Computing both arguments

$$\begin{aligned} (app(a?, b?) = c) \quad == &\quad a? = nil, \ \ b? = c \\ | &\quad h? :: z? = c, \ \ app(t?, b?) = z, \ \ h :: t = a? \end{aligned}$$

The algorithm terminates, i.e. there is only a finite number of solutions. Functionally written, it often occurs in the literature [10, 4]:

$$\begin{aligned} split(c) &\rightarrow (nil, c) \\ split(h :: z) &\rightarrow \textbf{let}\ (t, b) = split(z)\ \textbf{in}\ (h :: t, b) \end{aligned}$$

### 8.2.5 Only the second argument is known

$$\begin{array}{rcl} (app\,(a?,\ b) = c?) & == & a? = nil,\quad b = c? \\ & | & app\,(t?,\ b) = z?,\quad h? :: z = c?,\quad h :: t = a? \end{array}$$

The literal $h? :: z = c?$ is not operationally executable, since $h$ cannot be computed. Even the type of $h$ is unknown. Hence, this is not a legal algorithm in our sense. If $app$ is used monomorphically, e.g. only for lists of natural numbers, then a value generator might be used.

$$\ldots app\,(t?,\ b) = z?,\quad h? = nat\,(),\quad h :: z = c?,\quad h :: t = a?$$

### 8.2.6 Only the first argument is known

$$\begin{array}{rcl} (app\,(a,\ b?) = c?) & == & a = nil,\quad b? = c? \\ & | & h? :: t? = a,\quad app\,(t,\ b?) = z?,\quad h :: z = c? \end{array}$$

This is illegal, too, because the literal $b? = c?$ is not effective. Restricting to e.g. $list\,(nat)$, the literal may be avoided:

$$\ldots a = nil,\quad n? = nat\,(),\quad b? = list\,(n),\quad b = c? \ldots$$

where $b$ is bound to all lists of natural numbers, i.e. $b = list\,(nat\,())$.

### 8.2.7 The test: all is known

$$\begin{array}{rcl} (app\,(a,\ b) = c) & == & a = nil,\quad b = c \\ & | & h? :: t? = a,\quad h? :: z? = c,\quad app\,(t,\ b) = z \end{array}$$

Failure is detected early, whence this algorithm is more efficient than just computing and comparing, i.e. $app\,(a,\ b) = c'?,\quad c = c'$.

## 8.3 List construction from the back end

We now consider an operation dual to '::' – attaching a new item to the end of a list.

$$dcons\,(a,\ b) \quad\rightarrow\quad app\,(a,\ b :: nil)$$

Relationally:

$$(dcons\,(a,\ b) = c?) \ == \ n? = nil,\quad bl? = b :: n,\quad app\,(a,\ bl) = c?$$

The inverse of this function is interesting, since it returns the last item of a sequence.

$$(dcons\,(a?,\ b?) = c) \ == \ app\,(a?,\ bl?) = c,\quad b? :: n? = bl,\quad n = nil$$

This is terminating, but poor since first many proposals are generated by the inverse append, that are mostly omitted by the test at the end. Note that there is always at most one final result.

Example:

$$
\begin{array}{lllll}
dcons\,(a?,\, b?) = [1,\, 2,\, 3] & \rightarrow & a = [\,],\; bl = [1,\, 2,\, 3], & b = 1,\; n = [2,\, 3] & \Rightarrow \text{No} \\
 & \mid & a = [1],\quad bl = [2,\, 3], & b = 2,\; n = [3] & \Rightarrow \text{No} \\
 & \mid & a = [1,\, 2],\quad bl = [3], & b = 3,\; n = [\,] & \Rightarrow \text{Yes} \\
 & \mid & a = [1,\, 2,\, 3],\; bl = [\,], & Failure &
\end{array}
$$

Hence, the solution is $a = [1,\, 2],\; b = 3$.

A much better algorithm may be obtained by transformations:

$$
\begin{array}{ll}
\multicolumn{2}{l}{(dcons\,(a,\, b) = c) \;==\; n = nil,\;\; bl = b :: n,\;\; app\,(a,\, bl) = c} \\
\multicolumn{2}{l}{\text{Unfolding } app\text{:}} \\
== & n = nil,\;\; bl = b :: n, \\
 & \quad (a = nil,\;\; bl = c \\
 & \quad |h :: t = a,\;\; app\,(t,\, bl) = z,\;\; h :: z = c) \\
== & n = nil,\;\; b :: n = bl,\;\; a = nil,\;\; bl = c \qquad (1) \\
\mid & h :: t = a,\;\; n = nil,\;\; b :: n = bl,\;\; app\,(t,\, bl) = z,\;\; h :: z = c \qquad (2) \\
== & n = nil,\;\; a = nil,\;\; b :: n = c \\
\mid & h :: t = a,\;\; dcons\,(t,\, b) = z,\;\; h :: z = c
\end{array}
$$

where (1) was simplified by using the equality $bl = c$ and the fact that $bl$ does not occur outside, (2) by folding the three central literals back into *dcons*.

For the positive usage, we obtain

$$
\begin{array}{lll}
(dcons\,(a,\, b) = c?) & == & a = nil,\;\; n? = nil,\;\; b :: n = c? \\
 & \mid & h? :: t? = a,\;\; dcons\,(t,\, b) = z?,\;\; h :: z = c?
\end{array}
$$

and for the inverse

$$
\begin{array}{lll}
(dcons\,(a?,\, b?) = c) & == & b? :: n? = c,\;\; n = nil,\;\; a? = nil \\
 & \mid & h? :: z? = c,\;\; dcons\,(t?,\, b?) = z,\;\; h :: t = a?
\end{array}
$$

After renaming of local variables, the first literal may be factored out:

$$
\begin{array}{llll}
(dcons\,(a?,\, b?) = c) & == & h? :: z? = c, & (\;\; z = nil,\;\; a? = nil,\;\; b? = h \\
 & & & \mid\;\; dcons\,(t?,\, b?) = z,\;\; h :: t = a?\;\; )
\end{array}
$$

The algorithm is deterministic, since the two alternatives are disjoint. The recursive call of the inverse *dcons* starts by $? :: ? = z$.

Example:

$$\begin{array}{llll}
dcons\,(a_1,\ b_1) = [1,\ 2,\ 3] \rightarrow h_1 = 1,\ z_1 = [2,\ 3]. & z_1 \neq nil & \Rightarrow & 2^{nd} \text{ alt.} \\
dcons\,(a_2,\ b_2) = [2,\ 3] \rightarrow h_2 = 2,\ z_2 = [3]. & z_2 \neq nil & \Rightarrow & 2^{nd} \text{ alt.} \\
dcons\,(a_3,\ b_3) = [3] \rightarrow h_3 = 3,\ z_3 = [\,]. & z_3 = nil & \Rightarrow & 1^{st} \text{ or } 2^{nd} \text{ alt.} \\
dcons\,(a_4,\ b_4) = [\,] \rightarrow \text{Failure} & & &
\end{array}$$

Result: $b_1 = b_2 = b_3 = h_3 = 3, \quad a_1 = h_1 :: a_2 = h_1 :: h_2 :: a_3 = [1,\ 2].$

## 8.4 Last item of a list

Finding the last item of a list is easy using *dcons*:

$$last\,(dcons\,(\_,\ b)) \ \rightarrow \ b$$

Some improving transformations of the relational form:

$$\begin{array}{lll}
(last\,(c) = b?) & == & dcons\,(_?,\ b?) = c \\
 & == & h? :: z? = c, \quad ( \ z = nil, \ \ _? = nil, \ \ b? = h \\
 & & \qquad\qquad\qquad\ \ \mid \ dcons\,(t?,\ b?) = z, \ \ h :: t = \ _? \ )
\end{array}$$

Constructions never fail. They may be omitted if the result is not interesting.

$$\begin{array}{lll}
 & == & h? :: z? = c, \quad ( \ z = nil, \ \ b? = h \\
 & & \qquad\qquad\qquad\ \ \mid \ dcons\,(_?,\ b?) = z \ ) \\
 & == & h? :: z? = c, \quad ( \ z = nil, \ \ b? = h \ \ \mid \ \ last\,(z) = b? \ )
\end{array}$$

Note that this algorithm is tail-recursive, whereas the inverse *dcons* was not so, such that we really got a gain in efficiency by the transformations.

Comparing '$\leq$' with '+' and '*last*' with '*dcons*', one notes that each argument and result position of a function might have three modes: it may have a fixed value – denoted as $x$ –, or an unknown value that is to be determined – written $x?$ –, or an unknown value whose existence must be proven without the need to explicitly construct the value – $\_?$.

## 8.5 Reverse in its simple form

Functional definition:

$$\begin{array}{lll}
rev\,(nil) & \rightarrow & nil \\
rev\,(h :: t) & \rightarrow & dcons\,(rev\,(t),\ h)
\end{array}$$

Positive relational form:

$$\begin{array}{lll}
(rev\,(a) = b?) & == & a = nil, \ \ b? = nil \\
 & \mid & h? :: t? = a, \ \ rev\,(t) = t'?, \ \ dcons\,(t',\ h) = b?
\end{array}$$

Inverse form:

$$\begin{array}{lll}
(rev\,(a?) = b) & == & b = nil, \ \ a? = nil \\
 & \mid & dcons\,(h?,\ t'?) = b, \ \ rev\,(t?) = t', \ \ h :: t = a?
\end{array}$$

The algorithm terminates. It can be shown to be deterministic, since the inverse *dcons* is deterministic and disjoint to *nil*. Although the algorithms for positive and inverted *rev* are different, we think that their efficiency is equal because *dcons* and the inverse optimized *dcons* have about the same complexity.

### 8.6 Tail-recursive definition of reverse

Functional definition:

$$\begin{array}{lll} rev\,(a) & \rightarrow & rev'\,(a,\ nil) \\ rev'\,(nil,\quad x) & \rightarrow & x \\ rev'\,(h :: t,\ x) & \rightarrow & rev'\,(t,\ h :: x) \end{array}$$

Relational form:

$$\begin{array}{lll} (rev\,(a) = b?) & == & x? = nil,\ \ rev'\,(a,\ x) = b? \\ (rev'\,(a,\ x) = b?) & == & a = nil,\ \ b? = x \\ & \mid & h? :: t? = a,\ \ h :: x = hx?,\ \ rev'\,(t,\ hx) = b? \end{array}$$

Similarly to the factorial function, the accumulating parameter $x$ should be considered unknown for the inverse:

$$\begin{array}{lll} (rev\,(a?) = b) & == & rev'\,(a?,\ x?) = b,\ \ x = nil \\ (rev'\,(a?,\ x?) = b) & == & a? = nil,\ \ b = x? \\ & \mid & rev'\,(t?,\ hx?) = b,\ \ h? :: x? = hx,\ \ h :: t = a? \end{array}$$

Again, a stream is generated that is examined later with the condition $x = nil$. The stream is finite, since the term $x$ is smaller than $hx$, and thus, one component of the stream items decreases in size. This implies the termination of the inverse $rev'$.

Although the algorithm for the inverse *rev* looks quite different from the original one, it operates similarly:

Example:

$$\begin{array}{l} rev\,[1,\ 2,\ 3] \ \rightarrow\ rev'\,([1,\ 2,\ 3],\ [\,]) \ \rightarrow\ rev'\,([2,\ 3],\ [1]) \\ \quad \rightarrow\ rev'\,([3],\ [2,\ 1]) \ \rightarrow\ rev'\,([\,],\ [3,\ 2,\ 1]) \ \rightarrow\ [3,\ 2,\ 1] \end{array}$$

$$\begin{array}{l} invrev\,[1,\ 2,\ 3] \ \rightarrow \\ \quad \textbf{let}\ (a,\ x) = invrev'\,[1,\ 2,\ 3]\ \textbf{in if}\ x = nil\ \textbf{then}\ a\ \textbf{else fail} \\ invrev'\,[1,\ 2,\ 3] \ \rightarrow\ ([\,],\ [1,\ 2,\ 3]) \mid ([1],\ [2,\ 3]) \mid ([2,\ 1],\ [3]) \mid ([3,\ 2,\ 1],\ [\,]) \\ \quad \Rightarrow\ \text{Answer } [3,\ 2,\ 1] \end{array}$$

## 9 Examples for successful optimizations by transformations

The strategy *unfold/simplify/fold* is quite successful in improving function definitions [2, 4, 1]. We had given some examples previously, and two larger ones will follow.

## 9.1 Membership in a list

| | |
|---|---|
| We functionally define | $x\ in\ app(\_,\ x :: \_) \ \rightarrow\ ()$ |
| Internal form: | $(x\ in\ l)\ \ ==\ \ app(\_,\ l') = l,\ \ x :: \_ \ = l'$ |
| with the modes | |
| test for membership | $(x\ in\ l)\ \ ==\ \ app(\_?,\ l'?) = l,\ \ x :: \_? \ = l'$ |
| enumeration of the items | $(x?\ in\ l)\ \ ==\ \ app(\_?,\ l'?) = l,\ \ x? :: \_? \ = l'$ |

Other modes don't make sense. The algorithms derived above are quite poor; especially the first one generates many proposals that are omitted afterwards.

We shall now apply the standard strategy unfold/simplify/fold to improve the algorithms. We shall work 'neutrally' within the transformations, i.e. without annotated question marks.

Often, two alternatives will not be written separated by ' | ', but one above the other enclosed by large braces { }.

$$\begin{aligned}(x\ in\ l)\ &==\ app(\_,\ l') = l,\ \ x :: \_ \ = l' \qquad\qquad \text{unfolding } app\\ &== \left\{\begin{array}{c} \_ \ = nil,\ \ l' = l \\ h :: t = \_,\ \ app(t,\ l') = z,\ \ h :: z = l \end{array}\right\},\ \ x :: \_ \ = l'\end{aligned}$$

Now we distribute the right literal over the alternatives. The literals $\_ \ = nil$ and $h :: t = \_$ are always successful and may be omitted.

$$\begin{aligned}==\ \ & l' = l,\ \ x :: \_ \ = l' && (1)\\ \mid\ \ & app(t,\ l') = z,\ \ h :: z = l,\ \ x :: \_ \ = l' && (2)\end{aligned}$$

In (1), $l'$ is not needed and may be replaced by the equal $l$. In (2), $t$ is not used and may be replaced by $\_$. Then, the literal with $app$ and the second with :: may be folded to $in$.

$$==\ \ x :: \_ \ = l\ \mid\ x\ in\ z,\ \ h :: z = l$$

Supplied with annotations, we obtain for the test:

$$(x\ in\ l)\ \ ==\ \ x :: \_ \ = l\ \mid\ h? :: z? = l,\ \ x\ in\ z$$

The first literal in either alternative may be factored to the left after suitable renaming:

$$(x\ in\ l)\ \ ==\ \ h? :: z? = l,\ \ (\ x = h\ \mid\ x\ in\ z\ )$$

This is a quite good algorithm; functionally, it looks like

$$x\ in\ (h :: z)\ \ \rightarrow\ \ (x = h)\ \mid\ x\ in\ z$$

The only drawback is that the arguments of the operator ' | ' have equal precedence, i.e. the computation still goes on after $x = h$ was satisfied. Considering the result type $\{()\}$, there cannot be any other results. If one is only interested in success or failure, one can stop the computation after the first occurrence of $x$ in the list was found. The system could do the optimization automatically due to the typing. The result would be

$$x\ in\ (h :: z) \quad \rightarrow \quad (x = h) \setminus x\ in\ z$$

where '$\setminus$' denotes a 'biased' alternative that considers its second argument only when the first one has failed.

The inverse, i.e. the enumeration of the list items, is done by the following algorithm:

$$(x?\ in\ l) \ == \ h? :: z? = l, \ \ (x? = h \ \mid \ x?\ in\ z)$$

or in functional notation:

$$enum\,(h :: t) \rightarrow h \ \mid \ enum\,(t)$$

Note that '$\mid$' must not be replaced by '$\setminus$' here.

## 9.2 Inserting an item into a list

We want to insert a new item into a list at an arbitrary point. The inverse is to select and remove an arbitrary list item.

$\mathcal{F}$: $\quad ins\,(x,\, app\,(a,\, b)) \ \rightarrow \ app\,(a,\, x :: b)$

$\mathcal{R}$: $\quad (ins\,(x,\, l) = l'?) \ == \ app\,(a?,\, b?) = l, \ \ x :: b = b'?, \ \ app\,(a,\, b') = l'$

This algorithm terminates, but it may be improved considerably:

$$(ins\,(x,\, l) = l') \ == \ app\,(a,\, b) = l, \ \ x :: b = b', \ \ app\,(a,\, b') = l'$$

Unfold both literals with *app*. Note that the local variables that are newly introduced by the unfolding must have distinct names.

$$\begin{aligned} == \ & \left\{ \begin{array}{c} a = nil, \ \ b = l \\ h :: t = a, \ \ app\,(t,\, b) = z, \ \ h :: z = l \end{array} \right\}, \ \ x :: b = b', \\ & \left\{ \begin{array}{c} a = nil, \ \ b' = l' \\ h' :: t' = a, \ \ app\,(t',\, b') = z', \ \ h' :: z' = l' \end{array} \right\} \end{aligned}$$

Now distribute to disjunctive normal form. Two of the potential four alternatives are vacuous, since $a = nil$ contradicts $\_ \ :: \ \_ \ = a$.

$$\begin{aligned} == \ \ & a = nil, \ \ b = l, \ \ x :: b = b', \ \ a = nil, \ \ b' = l' \\ \mid \ \ & h :: t = a, \ \ app\,(t,\, b) = z, \ \ h :: z = l, \ \ x :: b = b', \\ & \quad h' :: t' = a, \ \ app\,(t',\, b') = z', \ \ h' :: z' = l' \end{aligned}$$

In the first alternative, both occurrences of $a = nil$ are superfluous, since $a$ occurs nowhere else and $a = nil$ never fails. Similarly, the two constructions of $a$ in the second alternative may be omitted, if the information is kept that the head resp. tail of $a$ is the same.

$$\begin{array}{rl} == & b = l,\ \ b' = l',\ \ x :: b = b' \\ \mid & h = h',\ \ t = t',\ \ app\,(t,\, b) = z,\ \ h :: z = l,\ \ x :: b = b', \\ & \qquad app\,(t',\, b') = z',\ \ h' :: z' = l' \end{array}$$

Remember the head of the clause: $ins\,(x,\, l) = l'$. Only $x$, $l$, and $l'$ occur in it, all other variables are local. Hence, the equalities in the clause may be directly used.

$$\begin{array}{rl} == & x :: l = l' \\ \mid & app\,(t,\, b) = z,\ \ h :: z = l,\ \ x :: b = b',\ \ app\,(t,\, b') = z',\ \ h :: z' = l' \end{array}$$

Now, the two $app$ literals and $x :: b = b'$ may be folded back:

$$\begin{array}{rl} == & x :: l = l' \\ \mid & h :: z = l,\ \ ins\,(x,\, z) = z',\ \ h :: z' = l' \end{array}$$

Now, we got completely rid of append and inverse append. The final clause is only in terms of 'cons'.

With annotations for positive usage, we obtain

$$\begin{array}{lrl} (ins\,(x,\, l) = l'?) & == & x :: l = l'? \\ & \mid & h? :: z? = l,\ \ ins\,(x,\, z) = z'?,\ \ h :: z' = l'? \end{array}$$

The corresponding functional definition is

$$\begin{array}{lll} ins\,(x,\qquad l) & \rightarrow & x :: l \\ ins\,(x,\, h :: z) & \rightarrow & h :: ins\,(x,\, z) \end{array}$$

The inverse mode is

$$\begin{array}{lrl} (ins\,(x?,\, l?) = l') & == & x? :: l? = l' \\ & \mid & h? :: z'? = l',\ \ ins\,(x?,\, z?) = z',\ \ h :: z = l? \\ \text{(Factoring)} & == & h? :: z'? = l',\ \left\{ \begin{array}{c} x? = h,\ \ l? = z' \\ ins\,(x?,\, z?) = z',\ \ h :: z = l? \end{array} \right\} \end{array}$$

The corresponding functional form is

$$f\,(h :: z') \ \rightarrow\ (h,\, z') \ \mid\ \textbf{let}\ (x,\, z) = f\,(z')\ \textbf{in}\ (x,\, h :: z)$$

# 10 Two-sided conditionals and negation

## 10.1 The problem of two-sided conditionals

Remember that there are no Boolean values, but every predicate either succeeds and returns (), or fails and returns nothing. This is true for build-in predicates such as equality '=' as well as for user-defined predicates, e.g. $a \leq b$ and $x\ in\ l$.

Conditionals with one branch in $\mathcal{F}$ – **if** $p$ **then** $t$ **else fail** – may be easily translated into $\mathcal{R}$: $p$, $t$. They are syntactic sugar, and we have already seen some of them in previous examples. Real conditionals with two branches cause some difficulty: the **else** part is to be executed if and only if the condition fails.

A typical function with a two-sided conditional is the maximum function:

$$max\,(a,\,b) \;\rightarrow\; \textbf{if } a \leq b \textbf{ then } b \textbf{ else } a$$

Algorithms for '$\leq$' were derived in subsections 5.2 and 5.3. One might argue that the two-sided 'if' can be avoided by guards:

$$\begin{array}{lcl} max\,(a,\,b) & \rightarrow & \textbf{if } a \leq b \textbf{ then } b \\ max\,(a,\,b) & \rightarrow & \textbf{if } a > b \textbf{ then } a \end{array}$$

but this construction is less efficient since two conditions have to be evaluated instead of one and determinism is not guaranteed.

Nevertheless, the idea of guards is used to translate the conditional into $\mathcal{R}$:

$$\begin{array}{lcl} (max\,(a,\,b) = c?) & == & a \leq b,\;\; c? = b \\ & \mid & not\,(a \leq b),\;\; c? = a \end{array}$$

where $not\,(L)$ is a literal that fails if $L$ succeeds and vice versa. For the moment, we define negation only for literals where all variables are bound, such that the result is just success or failure.

Although the relational form is logically equivalent to the functional definition with two-sided 'if', it is algorithmically less efficient since it implies the condition being evaluated twice. At least, it guarantees determinism since $L$ and $not\,(L)$ are contradictory.

Now let us consider the inverses of the maximum function. Asking for the first argument poses no problem:

$$\begin{array}{lcl} (max\,(a?,\,b) = c) & == & c = b,\;\; a? \leq b \\ & \mid & a? = c,\;\; not\,(a \leq b) \end{array}$$

Looking for the second argument:

$$\begin{array}{lcll} (max\,(a,\,b?) = c) & == & c = b?,\;\; a \leq b & \quad\text{or better: } a \leq c,\;\; c = b? \\ & \mid & a = c,\;\; not\,(a \leq b?) & \end{array}$$

Now, we get into trouble. What does $not\,(a \leq b?)$ mean? We expect it to enumerate all values of $b$ not satisfying the predicate. In our example, this is only a finite number of values such that the literal should terminate. The problem is that we don't have an algorithm at hand to compute these values.

A possible trick is to use a value generator, i.e. to replace the literal by $b? = nat\,(),\;\; not\,(a \leq b)$. This would produce the expected solutions but not terminate. Worse, this trick is only possible for monomorphic types as we know.

A better approach is to try to derive algorithms for negated predicates from the original ones by means of transformations. We shall do so in the next subsection.

## 10.2 Negation

A first idea is to apply de Morgan's laws of negation to the algorithm of a predicate to obtain an algorithm for the negated predicate. We demonstrate this for the algorithm for '$\leq$' derived in subsection 5.3.

$$\begin{array}{lcl}
(b \leq c) & == & b = 0 \;\mid\; s\,(b') = b,\;\; s\,(c') = c,\;\; b' \leq c' \\
not\,(b \leq c) & == & not\,(b = 0),\;\; (\; not\,(s\,(b') = b) \;\mid\; not\,(s\,(c') = c) \;\mid\; not\,(b' \leq c')\;) \\
 & == & not\,(b = 0),\;\; not\,(s\,(c') = c) \;\;\mid\;\; not\,(b = 0),\;\; not\,(b' \leq c')
\end{array}$$

This is complete nonsense. The reason is that local variables are implicitly quantified such that de Morgan's laws are not applicable.

Functionally written, the algorithm for '$\leq$' looks like

$$\begin{array}{lcl}
(\;\; 0 \;\;\leq\;\; c \;\;) & \rightarrow & () \\
(s\,(b') \leq s\,(c')) & \rightarrow & (b' \leq c')
\end{array}$$

i.e. only the succeeding cases are specified, and implicitly it is assumed that all other (well-typed) cases fail. Hence, the solution of our problem is to make failure explicit.

An alternation $A_1 \mid A_2$ fails iff both alternatives fail. Thus, we should transform $not\,(A_1 \mid A_2)$ into $not\,(A_1),\ not\,(A_2)$. In doing so, one must ensure by renaming that the local variables of the two alternatives are distinct.

A conjunction $L_1,\ L_2$ fails if $L_1$ already fails, or $L_1$ succeeds, but $L_2$ fails. Thus, we obtain the following transformations:

$$\begin{array}{lcl}
not\,(A_1 \mid A_2) & \rightarrow & not\,(A_1),\;\; not\,(A_2) \\
not\,(L_1,\; L_2) & \rightarrow & not\,(L_1) \;\;\mid\;\; L_1,\;\; not\,(L_2)
\end{array}$$

The transformation will be demonstrated by some examples. For the sake of brevity, we shall often use the symbols '$\neq$' and '$\not\leq$' instead of the *not* $(L)$ construction.

**Example 1:** The algorithm for '$\leq$' derived in subsection 5.3

$$
\begin{array}{lll}
(a \leq b) & == & a = 0 \\
& | & s\,(a'?) = a, \;\; s\,(b'?) = b, \;\; a' \leq b'
\end{array}
$$

$$
\begin{array}{lllr}
(a \not\leq b) & == & a \neq 0, & \\
& & (\;\; s\,(a'?) \neq a & (1) \\
& & |\;\; s\,(a'?) = a, \;\; s\,(b'?) \neq b & (2) \\
& & |\;\; s\,(a'?) = a, \;\; s\,(b'?) = b, \;\; a' \not\leq b' \;) &
\end{array}
$$

In alternative (1), $a'$ may be replaced by $\_$, since it is not further needed. Similarly, $a'$ and $b'$ may be both replaced by $\_$ in (2). Due to the definition of natural numbers, $a \neq 0$ is equivalent to $a = s\,(\_?)$, and conversely, $a \neq s\,(\_?)$ is the same as $a = 0$. Hence we obtain

$$
\begin{array}{lll}
== & s\,(\_?) = a, & (\;\; a = 0 \\
& & |\;\; s\,(\_?) = a, \;\; b = 0 \\
& & |\;\; s\,(a'?) = a, \;\; s\,(b'?) = b, \;\; a' \not\leq b' \;)
\end{array}
$$

Now distribute to normal form where the first alternative vanishes, and combine the literals $s\,(\ldots?) = a$ into one.

$$
\begin{array}{ll}
== & s\,(\_?) = a, \;\; b = 0 \\
\;\;| & s\,(a'?) = a, \;\; s\,(b'?) = b, \;\; a' \not\leq b' \\
== & s\,(a'?) = a, \;\; (\, b = 0 \;\; | \;\; s\,(b'?) = b, \;\; a' \not\leq b' \,)
\end{array}
$$

Functionally we obtain

$$
\begin{array}{lll}
s\,(\_) \not\leq \;\; 0 & \rightarrow & () \\
s\,(a') \not\leq s\,(b') & \rightarrow & a' \not\leq b'
\end{array}
$$

This is a good algorithm for '$>$'.

Now it is not difficult to complete the example $max\,(a,\, b?) = c$ from the previous subsection. We need $a \not\leq b?$.

$$
(a \not\leq b?) \;\; == \;\; s\,(a'?) = a, \;\; (\, b? = 0 \;\; | \;\; a' \not\leq b'?, \;\; s\,(b') = b? \,)
$$

This algorithm terminates since the term $a$ is bigger than $a'$. Thus, the negating transformation was worth while.

**Example 2:** Going back, i.e. negating '$\not\leq$'

$$\begin{aligned}
(a \not\leq b) &== s\,(a'?) = a,\ \ (\ b = 0 \ \mid\ s\,(b'?) = b,\ \ a' \not\leq b'\ ) \\
(a \leq b) &== s\,(a'?) \neq a \\
&\quad\ \mid\ s\,(a') = a,\ \ b \neq 0,\ \left\{ \begin{array}{l} s\,(b'?) \neq b \\ s\,(b'?) = b,\ \ a' \leq b' \end{array} \right\} \\
&== a = 0 \\
&\quad\ \mid\ s\,(a'?) = a,\ \ s\,(_?) = b,\ \ (\ b = 0 \ \mid\ s\,(b'?) = b,\ \ a' \leq b'\ ) \\
&== a = 0 \\
&\quad\ \mid\ s\,(a'?) = a,\ \ s\,(b'?) = b,\ \ a' \leq b'
\end{aligned}$$

This is exactly the original algorithm.

**Example 3:** Negation of the less efficient comparison of subsection 5.2

$$\begin{aligned}
(a \leq b) &== a = b \ \mid\ s\,(b'?) = b,\ \ a \leq b' \\
(a \not\leq b) &== a \neq b,\ \ (\ s\,(b'?) \neq b \ \mid\ s\,(b'?) = b,\ \ a \not\leq b'\ ) \\
&== a \neq b,\ \ (\ b = 0 \ \mid\ s\,(b'?) = b,\ \ a \not\leq b'\ )
\end{aligned}$$

This is an algorithm for '$>$'. If the definition is turned into other modes, the literal $a \neq b$ may cause problems. An algorithm for inequality may be obtained by negating the recursive clause for equality depending on the data type in question. For naturals, it is

$$(a = b) == a = 0,\ \ b = 0 \ \mid\ s\,(a'?) = a,\ \ s\,(b'?) = b,\ \ a' = b'$$

**Example 4:** Going back again

$$\begin{aligned}
(a < b) &== a = b \ \mid\ a \neq b,\ \ b \neq 0,\ \left\{ \begin{array}{l} s\,(b'?) \neq b \\ s\,(b'?) = b,\ \ a \leq b' \end{array} \right\} \\
&== a = b \ \mid\ a \neq b,\ \ s\,(_?) = b,\ \left\{ \begin{array}{l} b = 0 \\ s\,(b'?) = b,\ \ a \leq b' \end{array} \right\} \\
&== a = b \ \mid\ a \neq b,\ \ s\,(b'?) = b,\ \ a \leq b'
\end{aligned}$$

This is not exactly the original algorithm. Relationally, it became more complex by the additional literal $a \neq b$. Functionally, it became more efficient, since it is recognizably deterministic, and the two alternatives may be considered the branches of a two-sided conditional. This is not true for other modes where the literal $a \neq b$ might move towards the end of the clause.

# 11 Summary and conclusion

## 11.1 The functional source language

The 'functional' language $\mathcal{F}$ may contain very powerful patterns. They allow for calls of arbitrary invertable functions e.g. $last\,(dcons\,(\_,\,b)) \rightarrow b$.

Names may be locally bound by means of **let** abstraction. It may contain powerful 'declarators' involving invertable functions and predicates other than equality.

| | | |
|---|---|---|
| | **let** $a = 7$ | $a$ is bound to 7 |
| Examples: | **let** $fac\,(n) = 120$ | $n$ is bound to 5 by inverting $fac$ |
| | **let** $a \leq 4$ | $a$ is bound to 0, 1, 2, or 3 |

One-sided conditionals – **if** $c$ **then** $t$ – are related to these **let** constructs. They fail if the condition $c$ fails, otherwise their values are the values of $t$. Two-sided conditionals – **if** $c$ **then** $t$ **else** $e$ – evaluate $e$ if the condition fails, otherwise $t$.

Rules in a function definition have equal precedence. If they overlap, they cause multiple results. There is also an explicit 'or' operator written ' | ' whose arguments have equal rights. A biased alternation written '$a \setminus b$' may be provided with the logical meaning $a \mid not\,(a), b$ i.e. $b$, is only evaluated if $a$ fails.

All expressions in the language are potentially multi-valued, and all functions may return many results. The various alternatives should be investigated in breadth-first order to guarantee completeness.

Up to now, there are only first order functions. Perhaps, this may be extended, but functions of higher order will probably not be invertable.

## 11.2 The relational internal form

A compiler for the functional language would first check syntax and static semantics, especially well-typedness. Then, it would translate the program into a set of relational clauses. The clauses are reordered according to what variables are bound. When a new mode of a function occurs, it will be checked for legality by the trial to derive an algorithm for it. Improved algorithms may be obtained by optimizing and negating transformations.

The relational clauses for a new algorithm may be slightly improved by adding value generators, factoring etc. They are a kind of (abstract) machine code for a function (more exactly a special mode of a relation). They only consist of

| | |
|---|---|
| - assignments | $a? = b$ |
| - tests for equality/inequality | $a = b \quad a \neq b$ |
| - constructions | $c\,(x_1, \ldots, x_n) = y?$ |
| - tests and destructions | $c\,(x_1?, \ldots, x_n?) = y$ |
| - calls of other relations | $r\,(\ldots)$ with any modes of variables |
| - negative tests | $not\,(r\,(x_1, \ldots, x_n))$ with all variables bound |

combined by ',' (sequencing), ' | ' (branching), and as optimization '\' (biased branching), if two alternatives are known to be disjoint.

Furthermore, there could be annotations telling what calls are known to be deterministic or terminating.

## 11.3 Methods to improve algorithms

We presented three methods to improve a given algorithm.

- First, the literals in the clauses may be re-ordered, i.e. the order of evaluation is changed. Normally, one should try to have as much bound variables as possible in a call. An exception are so-called accumulating parameters; they should be left unbound (see $fac$ and $rev$ example). People investigating partial evaluation made the same experience [1]; accumulating parameters should be considered dynamic.

  The re-ordering method is mainly used to derive algorithms for inverse functions. It does not affect partial correctness, but may change termination properties. Hopefully, the rearranged algorithm more often terminates than the original one.

  Limits of this method are cases where no good order can be found. Examples are some modes of multiplication caused by the fact that a variable occurs more than once in one side of a rule (non-linearity). Some modes of polymorphic functions are illegal, since a polymorphic variable cannot be bound by value generators.

- Second, redundant computations may be omitted from algorithms by the well-known unfold/simplify/fold method. It is similar to partial or symbolic evaluation [1]. Success is not guaranteed, since there are examples (not shown in this paper) where the method yields no improvement. It may be dangerous to unfold too many calls or one call to often.

- The third method is the 'negating' transformation to derive an algorithm for the negation of a predicate. Its correctness and limits are not yet known to us, and we do not know a similar method in literature. The examples shown here work fine and correctly. An interesting aspect is that an algorithm is sometimes slightly changed by two negations.

Romanenko [11] proposes two methods to invert functions in the functional language Refal. The transformational approach is similar to unfold/fold; function calls are symbolically 'driven', i.e. unfolded. The configurational analysis is related to reordering. A graph of configurations is created and variables are annotated as known or unknown. Then the graph is rearranged according to some rules.

Harrison's method [6] is radically different from ours. He uses a variable-free FP like language and applies algebraic laws, e.g. $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$, to derive inverses. In his paper, he demonstrates his method with the example of inverse append, i.e. the problem $app\,(x?,\, y?) = z$. Compare his lengthy, tedious derivation with our simple reordering in section 8.2.4. Furthermore, his method is only indirectly able to deal with partial inverses e.g. $app\,(x,\, y?) = z$.

## 11.4 Conclusion

The proposed methods seem a bit too complex for an automated system. Probably, user guidance will be needed to derive good algorithms. A system allowing for the correct application of the transformations involved in the three methods would be nice. Such a system could be used to gain more experience by applying the methods to a wider range of problems without the risk to confuse variable names and known/unknown annotations.

# References


[1] Bondorf, A.: Towards a Self-applicable Partial Evaluator for Term Rewriting Systems. In: Bjørner, D., Ershov, A.P., Jones, N.D., Ed.: Partial Evaluation and Mixed Computation, (Elsevier, North Holland, Amsterdam, 1988) 27-50

[2] Burstall, R., Darlington, J.: A Transformation System for Developing Recursive Programs, Journal of the ACM, Vol. 24, No. 1, (Jan. 1977) 44-67

[3] Burstall, R., MacQueen, D., Sannella, D.: HOPE: An Experimental Applicative Language, Report CSR-62-80, Computer Science Dept., Edinburgh, (1980)

[4] Darlington, J., Field, A.J., Pull, H.: The Unification of Functional and Logic Languages. In: deGroot, Lindstrom, Ed.: Logic Programming, (Prentice Hall, Englewood Cliffs, New Jersey, 1986) 37-70

[5] Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specifications, (Springer, Heidelberg, 1985)

[6] Harrison, P.G.: Function inversion, In: Bjørner, D., Ershov, A.P., Jones, N.D., Ed.: Partial Evaluation and Mixed Computation, (Elsevier, North Holland, Amsterdam, 1988) 153-166

[7] Heckmann, R.: A Functional Language for the Specification of Complex Tree Transformations. In: Ganzinger, H, Ed.: ESOP '88, LNCS 300, (Springer, Heidelberg, 1988) 175-190

[8] Heckmann, R.: Language Reference Manual of TrafoLa-H, PROSPECTRA Note S.1.6-DI-14.0, (Sept. 1988)

[9] Milner, R: The Standard ML Core Language. In: Polymorphism, Vol. II, Number 2, (Oct. 1985)

[10] Reddy, U.: On the Relationship between Logic and Functional Languages. In: deGroot, Lindstrom, Ed.: Logic Programming, (Prentice Hall, Englewood Cliffs, New Jersey, 1986) 3-36

[11] Romanenko, A. Y.: The Generation of Inverse Functions in REFAL, In: Bjørner, D., Ershov, A.P., Jones, N.D., Ed.: Partial Evaluation and Mixed Computation, (Elsevier, North Holland, Amsterdam, 1988) 427-444

[12] Shoham, Y., McDermott, D.: Directed Relations and Inversion of Prolog Programs. In: Proceedings of the international conference on fifth generation computer systems, (ICOT, 1984) 307-315

[13] Turner, D. A.: Miranda: a non-strict Functional Language with Polymorphic Types. In: Functional Programming Languages and Computer Architecture, LNCS 201, (Springer, Heidelberg, 1985) 1-16