

# Relative specification and transformational reuse of functional programs

Colin Runciman  
Nigel Jagger  
Department of Computer Science  
University of York  
YORK YO1 5DD  
United Kingdom

## Abstract

A *relative specification* is a collection of laws relating the behaviour of a required new program to that of one or more existing programs. A two stage method for transforming such relative specifications into effective functional programs is described and illustrated. The *inversion* stage re-arranges the specifying laws to obtain a collection of *partial definitions* for each unknown function, typically involving non-deterministic operators. The subsequent *fusion* stage combines each set of partial definitions into a single complete definition, thereby eliminating non-deterministic operators.

Keywords: semantics and reasoning about programs, program transformations, specifications.

## 1 Introduction

The first assumption of this paper is a preference for mathematical precision in the development of computer programs. Ideal program development begins with a formal specification of required behaviour, and proceeds by way of conjectures, laws, proofs, transformations, calculi and the like until a satisfactory final program is reached. The second assumption is that re-use and adaptation of programs is desirable, mainly to avoid wasted resources and needless cost but also because requirements for a new program are often first thought of in terms of the capabilities or deficiencies of existing programs.

It follows that we need ways of combining transformational development from an exact specification with the re-use of existing programs. We need ways of accomplishing *transformational re-use*. This in turn requires exact *relative specification* of new programs in terms of old.

This paper describes a transformational approach to the re-use of functional programs, starting from relative specifications that are equational laws. The deductive mechanism used is a form of *narrowing*—a technique first applied in the realm of theorem proving [8], and more recently adopted as the computational model for *equational programming* [2]. A distinctive feature of the method proposed is the introduction of intermediate definitions involving non-determinism.

Section 2 briefly describes the functional programming context. Section 3 explains the form of relative specifications and introduces an example that is used to illustrate subsequent sections.

---

<sup>o</sup>Supported by Science and Engineering Research Council, Grant GR/E16571

The transformation scheme is first presented in outline only, in section 4, and then in more detail in sections 5 (non-determinism), 6 (inversion stage) and 7 (fusion stage). Section 8 offers some comparisons with other work, and discusses some issues concerning practical application of the method by way of conclusion.

## 2 Programming framework

The programming framework adopted here is purely functional. The notation used is that of the Glide system [6], in which the programming language is similar in many respects to Miranda [10].

All data types have values generated by a small number of *constructors*, which may be constants or functions. For example, the natural numbers have 0 as a constant constructor and +1 as a functional constructor. Any natural number can be constructed from these. Lists have [], as a constant constructor for the empty list, and : (colon) as a binary functional constructor for non-empty lists. Applications of : are written infix, with the first list item (the *head*) as left argument, and the list of remaining items (the *tail*) as right argument. Two values are equal exactly if they have the same construction. That is, construction is *free* in all cases. There are no *laws*, such as can be defined in Miranda [9], equating different constructions.

Programs are collections of function definitions in the recursion equation style. In comparison with general equations between applicative terms, there are various restrictions on the the defining clauses of a program. In particular, left hand sides must take the form of function applications in which the function is a name being defined and the arguments are patterns formed only from variables and constructors. Each occurrence of a variable on a left hand side is a *binding* occurrence, for which a substitution is determined according to the form of supplied arguments and *applied* to all free occurrences of the same variable on the right hand-side. Left hand sides must be *linear*, not containing more than one occurrence of the same variable.

### Examples: the functions length and member

The function `length` computes the number of items in its list argument. The function `member` yields a truth value indicating whether its first argument occurs as an item in its second argument, a list. They can be defined as follows.

```
length [] -> 0
length (h:t) -> 1 + length t
    and
member e [] -> False
member e (h:t) -> (e = h) | member e t
```

Identifiers beginning with upper case letters always denote constructors: here `False` is one of the two constant constructors for truth-values. The infix symbol `|` is used for applications of the logical or function.

## 3 Relative specification

Definition by a collection of characteristic equational laws is a standard technique in mathematics. Equational specification, often referred to as *algebraic* specification [3], has also become well-

established as a technique for the precise characterisation of software requirements. In the context of functional programming, equations are a particularly natural form of specification, because they can be viewed as defining clauses generalised by lifting the restrictions on left-hand sides and the directed binding of variables.

In this paper, specifying equations have left and right hand sides that are *typed* expressions of the functional programming language, quantified over non-functional types only. The distinguishing attribute of a *relative* specification is that most of the functions occurring in the equations are either primitives of the functional programming language or else already have a programmed definition. The remaining functions are the true subjects of the specification.

### An example: relative specification of the rep function

Consider the function `rep` such that `rep n x` yields a list of `n x`'s. Intuitively, the length of `rep`'s result should be `n` and all its members should be `x`, but there is no way that `length` and `member` can be *applied directly as functions* to form a programmed definition of `rep`.

The following pair of laws constitute a relative specification for `rep` in terms of the functions `length` and `member` defined in the previous section. They formalise the intuitive requirement observed above.

```
length (rep n x) == n
member e (rep n x) => (e = x)
```

(`==` denotes semantic equality as distinct from `=` which is just ordinary programmed equality.) The second law can be treated as the left hand side of an equation completed by `== True`.

## 4 Transformational re-use in outline

The problem of transformational re-use, broadly expressed, is this: given some defining clauses from existing programs and a relative specification of one or more new functions, obtain acceptable program definitions for each of the new functions.

This problem can be tackled in two stages. In the first stage, *inversion*, specifying equations are re-arranged into a collection of *partial definitions* for each unknown function. The partial nature of these definitions is manifest in the form of non-deterministic operators. In the second stage, *fusion*, each set of partial definitions is combined into a single complete definition. In the process of combination, non-deterministic operators are eliminated.

### Main transformation stages for the rep example

The inversion stage can be performed on each equation independently. Inversion of the first equation for `rep` results in a partial definition specifying only the length of the result, and not the actual values of items it contains.

```

length (rep n x) == n
  and
length [] -> 0
length (h:t) -> 1 + length t
  ↓
rep n x -> oflength n
oflength 0 -> []
oflength (n+1) -> ? : oflength n

```

The name `oflength` is formally arbitrary but chosen to be suitable for explanatory purposes. Non-determinism is introduced in the second defining clause for `oflength`. In this case the non-determinism takes an extreme form: the symbol `?` means *any* value of the appropriate type. So the intuitive reading of the second defining clause for `oflength` is that a list of length `n+1` is *any* first item followed by a list of length `n`.

Inversion of the second specifying equation yields another partial definition complementary to the first. It defines only what items the result of `rep` may contain, and says nothing about how many of them there are.

```

member e (rep n x) => (e = x)
  and
member e [] -> False
member e (h:t) -> (e = h) | member e t
  ↓
rep n x -> containsonly x
containsonly x -> [] <> x : containsonly x

```

This time the non-determinism is of a different kind, involving the operator `<>` which indicates a binary choice. Intuitively, a list containing only items with the value `x` is *either* empty *or* has `x` as its head and a similarly constrained list as its tail.

The fusion stage combines the two partial definitions of `rep`, derived from the two specifying equations, into a single programmed definition.

```

rep n x -> oflength n >< containsonly x
  and
oflength 0 -> []
oflength (n+1) -> ? : oflength n
containsonly x -> [] <> x : containsonly x
  ↓
rep 0 x -> []
rep (n+1) x -> x : rep n x

```

The infix operator `><` denotes a *splice*. An intuitive reading of the `rep` defining clause containing `><` is that the result is *both* a list of length `n` *and* a list that contains only `x`. The twin clause definition of `rep` that emerges from the fusion stage is the finished product, suitable for incorporation in a program.

## 5 Non-determinism

To prepare the way for a more detailed look at inversion and fusion, this section describes the non-deterministic expressions that inversion introduces and fusion eliminates.

The use of non-determinism here is purely as a specification device, providing a means of expression for the intermediate forms of partial definition. Although it is convenient to think of the non-deterministic operators as specifying a legitimate choice of values, their actual use is to specify acceptable expressions and definitions in a program by constraining the values that these must yield with respect to other definitions and expressions. A collection of non-deterministic definitions is regarded as a representation of a class of deterministic programs and a specification for members of that class. Non-deterministic operators do not occur in final programs.

The intermediate definition of `rep` involved just three non-deterministic operators, but these three belong to a class numbering seven in all. There is a direct correspondence between this class and symbols of the predicate logic. Indeed, the meaning of each operator can be formalised by an axiom or inference rule employing the corresponding logical symbol.

<i>expression</i>	<i>informal meaning</i>	<i>logical analogue</i>
<code>non e</code>	not a value of $e$	$\neg$
<code>e&lt;&gt;e'</code>	a value of $e$ or of $e'$	$\vee$
<code>e.v&gt;&gt;d</code>	a value of $e$ for some value $v$ of $d$	$\exists$
<code>?</code>	any value	<i>True</i>
<code>e&gt;&lt;e'</code>	a value of $e$ or of $e'$	$\wedge$
<code>e.v&lt;&lt;d</code>	a value of $e$ for every value $v$ of $d$	$\forall$
<code>!</code>	no value	<i>False</i>

One non-deterministic expression  $e$  *fulfills* another  $e$  if every possible value of  $e$  is also a value of  $e'$ . In the special case that  $e$  is in fact deterministic, it *implements*  $e'$ .

### Examples of non-deterministic definitions and their implementation

A function that extracts an item from a list might be specified and implemented as follows. The implementation is not the only one possible; but it happens to be the simplest.

```
itemof [] -> !
itemof (h:t) -> h <> itemof t
    implemented by
itemof -> head
```

The next example is a function that forms a subsequence of the items of its list argument. (Here subsequence is used as in mathematics where the primes are a subsequence of the naturals—the items do *not* have to be consecutive in the original list.)

```

subseq [] -> []
subseq (h:t) -> (h:s <> s where s -> subseq t)
    implemented by
subseq -> id
    or perhaps by
subseq -> k []

```

Once again the implementations shown are merely illustrative. They are the outcomes of a simplistic approach to the implementation of `<>` in which either the left or right expression is dropped altogether. In the context of a larger transformation, such implementations would perhaps be ruled out by other constraints on `subseq`.

Lastly, an example illustrating the use of *quantified non-determinism* in the form of the `<<` operator analogous to  $\forall$  in predicate logic. Let the `inplace` function be specified as follows.

```

inplace 0 x -> x : ?
inplace (n+1) x -> ? : inplace n x

```

Informally, the result of `inplace p i` is a list that includes `i` as an item at the numeric position indicated by `p`, numbering positions from zero. What other items the resulting list contains, and where, is not specified. Now consider the following definition of the natural numbers.

```

naturals -> inplace n n . n << ?
    implemented by
naturals -> gen 0 (+1)
gen x f -> x : gen (f x) f

```

## Laws about non-deterministic operators

The correspondence with predicate logic provides a source of many laws that can be incorporated into a calculus of non-determinism. In addition, free construction provides the basis for an induction principle. This is illustrated in the derivation of `rep` given in Section 6.

## 6 Inversion

For every function `f` a *family* of inverses can be specified—one for each argument position of `f`. The inverse for the `i`th argument position, denoted `fi`, takes as its `i`th argument a result of `f` and, as the other arguments, the arguments of `f` corresponding to that result. The result of `fi` is a possible argument value for `f` at position `i`.

### Examples of simple inverses

Consider the family of two inverses for logical implication. These can be defined as follows.

```

=>1 True True -> ?
=>1 True False -> False
=>1 False q -> True
    and
=>2 True True -> True
=>2 False True -> ?
=>2 p False -> False

```

## Deriving inverses

The method used to derive definitions of inverses is based on the technique of *narrowing*. To express the workings of this method, *equational expressions* of the form

```
e . lhs == rhs
```

are useful. Here, as with the quantified forms  $\ll$  and  $\gg$ , the `.` operator may be pronounced “such that” and is left associative. Informally the possible values of an equational expression are values of the expression on the left after substitution for variables in such a way that the equation on the right is true.

## Details of inversion for the rep example

As a unary function, `length` has just one inverse, `length1`, for which the earlier alias `oflength` will continue to be used. The re-arrangement of the first specification law into the form of a partial definition for `rep` is immediate, assuming the availability of this inverse.

```

length (rep n x) == n
  ↓
rep n x -> oflength n

```

Given the definition of `length` and the availability of equational expressions, it is also straightforward to formulate *initial* defining clauses for `oflength`.

```

length [] -> 0
length (h:t) -> 1 + length t
  ↓
oflength 0 -> ([] . 0 == 0) <> (?:t . 0 == 1 + length t)
oflength (n+1) -> ([] . n+1 == 0) <> (?:t . n+1 == 1 + length t)

```

The left hand sides are determined by case analysis of constructors, and the right hand sides are choices based on `length`'s arguments. The choice branches are constrained by equating a possible argument of `oflength` with the appropriate right hand side in the relevant defining clause for `length`.

Let's proceed to derive a simpler `oflength` definition, not involving equational expressions. Consider first the zero case.

```

oflength 0
-> ([] . 0 == 0) <> (?:t . 0 == 1 + length t)
-> [] <> !
-> []

```

(from above)  
(free construction)  
(! is identity of <>)

Now the non-zero case.

```

oflength (n+1)
-> ([] . n+1 == 0) <> (? : t . n+1 == 1 + length t)           (from above)
-> ! <> (? : t . n == length t)                                 (+ commutative, free construction)
-> (? : t . n == length t)                                     (! identity of <>)
-> (? : t . t == oflength n)                                  (induction)
-> ? : oflength n                                           (substitution)

```

Turning now to the second specification law for `rep`, this can be re-arranged to obtain a partial definition as follows.

```

member e (rep n x) => (e = x) == True
  ↓
rep n x -> member2 e (=>1 True (e = x)) . e << ?

```

The quantifier `<<` is needed because there is no binding occurrence of `e` on the left hand side. In keeping with the earlier presentation of `rep`, the auxiliary `containsonly` will be employed.

```

rep n x -> containsonly x
  and
containsonly x -> member2 e (=>1 True (e = x)) . e << ?

```

Synthesis of `containsonly` requires the inverse `member2`.

```

member2 e False -> [] <> (non e : member2 e False)
member2 e True  -> (e : member2 e ?) <> (non e : member2 e True)

```

Also required is the following lemma regarding `member2`.

```

∀ e . member2 e ? -> ?

```

The derivation for `member2` and a proof of the lemma can be found in the Appendix. Returning now to the partial definition of `containsonly`, synthesis proceeds as follows.

```

containsonly x
-> member2 e (=>1 True (e = x)) . e << ?           (from above)
-> member2 e (=>1 True (e = x)) . e << (x <> non x) (law non)
-> (member2 e (=>1 True (e = x)) . e << x) ><     (range splitting)
    (member2 e (=>1 True (e = x)) . e << non x)
-> member2 x (=>1 True (x = x)) ><                 (singleton range)
    (member2 e (=>1 True (e = x)) . e << non x)
-> member2 x (=>1 True True) ><                     (law =)
    (member2 e (=>1 True False) . e << non x)
-> member2 x ? >< (member2 e False . e << non x) (defn. =>1)
-> ? >< (member2 e False . e << non x)           (lemma)
-> member2 e False . e << non x                  (? identity of ><)
-> ([] <> (non e : member2 e False)) . e << non x (defn. member2)
-> [] <> ((non e . e << non x) :                  (distributivity)
    (member2 e False . e << non x))
-> [] <> (x : (member2 e False . e << non x))     (law non)
-> [] <> (x : containsonly x)                     (induction)

```

## 7 Fusion

### The rep example: details of fusion

Input to the fusion stage comprises a splice definition of `rep` along with definitions of the inverse functions that are applied in the two splice arguments.

```
rep n x -> oflength n >< containsonly x
      and
oflength 0 -> []
oflength (n+1) -> ? : oflength n
containsonly x -> [] <> x : containsonly x
```

Transformation proceeds by structural case analysis of the argument `n`, a natural number for which 0 and +1 are the two possible constructors. (Recall that specifying equations are assumed to be typed.)

```
rep 0 x
-> oflength 0 >< containsonly x                                     (defn. rep)
-> [] >< ([] <> x : containsonly x)                                (defn. oflength, containsonly)
-> [] >< []                                                         (free construction)
-> []                                                               (>< idempotence)

rep (n+1) x
-> oflength (n+1) >< containsonly x                                 (defn. rep)
-> ? : oflength n >< containsonly x                                 (defn. oflength)
-> ? : oflength n >< ([] <> x : containsonly x)                    (defn. containsonly)
-> ? : oflength n >< x : containsonly x                             (free construction)
-> (? >< x) : (oflength n >< containsonly x)                       (distributivity)
-> x : rep n x                                                       (? is >< identity, defn. rep)
```

## 8 Discussion and conclusions

### Transformation strategy

One potential hindrance in the way of transformational methods is that a small specification change might require a substantial part of the program derivation to be re-worked. Any discussion of strategy must address the need for *incremental* working, *revision* of the specifying equations and *re-use* of previous derivation. From this perspective, the ideal transformation system maximises the extent to which useful derivations can be made from individual equations or operations, and minimises the work needed to combine and complete these derivations. The introduction of non-deterministic operators in order to give a closed expression to intermediate forms is an important means towards this end. There are many subtle issues of strategy to be resolved, particularly those relating to multiple unknown functions and the avoidance of problems of scale.

### Incomplete/inconsistent specification

If the required function is not completely specified by the given equations, some non-determinism will still remain after the fusion stage. This might be resolved by additional laws of specification,

inverted and fused with the non-deterministic solution. The possibility of such incremental working is a consequence of being able to express incomplete solutions appropriately using non-determinism. However, the incompleteness of a specification may reflect a wish to avoid being over-specific, in which case residual non-determinism might be transformed algorithmically to obtain a program that enumerates a set of solutions [11]. This shift from non-determinacy to sets of solutions is one that others prefer to make earlier in a transformational development [1].

On the other hand, a fusion step may fail to produce anything other than ! (no-value expression), clearly indicating inconsistency in the specification.

## Automation

Work is in progress to provide automated assistance with transformational re-use. The approach being taken generally assumes that the programmer should be involved closely at all stages, especially in strategic decisions such as fusion order, even where the support system is able to generate a plan for itself. Even so, some large steps, such as the synthesis of a particular inverse or transformation to a set enumerator, are tasks to be performed at the press of a button.

## Related Work

Darlington and co-workers at Imperial College have added *absolute set abstraction* (ASA) to their *Hope* functional programming system, and seek to transform definitions using this to equivalents in the unadorned system [5]. ASA characterises the elements of a required set by a collection of equations which they must satisfy. But there are equational specifications with no equivalent ASA (the `rep` example is a case in point) because ASA allows only a limited form of implicit quantification and, in effect, only a single unknown function. The collection of all solutions in a set imposes a fixed method of resolving nondeterminism that seems less flexible than the fusion approach. The kind of transformation system being developed for ASA is global, not incremental, and full automation is the ultimate aim, rather than programmer participation.

The Munich CIP project [4] also addresses transformation from equational specifications to functional definition. But very particular constraints are made on the form of equations and although both inverses and non-determinism are exploited in CIP, non-deterministic operators are more limited than those proposed here and are used in a different way.

Although various previous approaches to program derivation use non-deterministic choice, few pay much attention to the kind of fusion operators applied here. Sanderson [7] even says “...it is hard to see much practical significance in the intersection operation.  $A \cap B$  produces a result ... only if both A and B produce this result.” One could hardly disagree more with this observation.

In a *logic programming* language, auxiliaries such as `length` and `member` are characterised as relations, not functions. In principle, this greatly increases the possibilities for their direct re-use. In practice, a fully relational view is rarely achieved. The usual definitions of `length` and `member` in Prolog, for example, cannot be used straightforwardly to define `rep`.

## References

- [1] R. S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming, North Holland*, 6:159 – 189, 1986.

- [2] N. Dershowitz and D. A. Plaisted. Equational programming. In J. E. Hayes D. Michie and J. Richards, editors, *Machine Intelligence*, 1986.
- [3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*. Springer-Verlag, 1985.
- [4] The CIP Language Group. *The Munich Project CIP. Volume 1: The Wide-Spectrum Language CIP-L*. Volume 183 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.
- [5] A. J. Field J. Darlington and H. Pull. The unification of functional and logic languages. In G. Lindstrom, editor, *Logic Programming: relations, functions, and equations*, Prentice-Hall, 1985.
- [6] C. Runciman and I. Toyn. *Notes for glide users (2nd edition)*. Technical Report, University of York, October 1987.
- [7] J. G. Sanderson. *A Relational Theory of Computing*. Volume 82 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.
- [8] J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of Association of Computing Machinery*, 21(4):622 – 642, October 1974.
- [9] S. Thompson. Laws in miranda. In *ACM Symposium on LISP and Functional Programming*, pages 1–12, August 1986.
- [10] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16, Springer-Verlag, September 1985.
- [11] P. Wadler. How to replace failure by a list of successes. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 113–128, Springer-Verlag, September 1985.

# Appendix

## Derivation for member2

Initial defining clauses for member2 are obtained by rearranging the defining clauses of member.

```
member e [] -> False
member e (h:t) -> (e = h) | member e t
  ↓
member2 e False -> ([]. False == False) <>
  (h:t . False == (e = h) | member e t)
member2 e True -> ([]. True == False) <>
  (h:t . True == (e = h) | member e t)
```

These clauses can be simplified to give an acceptable program definition as follows.

```
member2 e False
-> ([]. False == False) <> (from above)
  (h:t . False == (e = h) | member e t)
-> [] <> (h:t . False == (e = h) . False == member e t) (identity, defn. |)
-> [] <> (h:t . h == non e . False == member e t) (law =)
-> [] <> (non e : t . False == member e t) (substitution)
-> [] <> (non e : t . t == member2 e False) (induction)
-> [] <> (non e : member2 e False) (substitution)

member2 e True
-> ([]. True == False) <> (from above)
  (h:t . True == (e = h) | member e t)
-> ! <> (h:t . True == (e = h) | member e t) (free construction)
-> (h:t . True == (e = h) | member e t) (! is identity of <>)
-> (h:t . True == (e = h) . ? == member e t) <> (defn. |)
  (h:t . False == (e = h) . True == member e t)
-> (h:t . h == e . ? == member e t) <> (law =)
  (h:t . h == non e . True == member e t)
-> (e:t . ? == member e t) <> (substitution)
  (non e : t . True == member e t)
-> (e:t . t == member2 e ?) <> (induction)
  (non e : t . t == member2 e True)
-> (e : member2 e ?) <> (non e : member2 e True) (substitution)
```

## Proof of member2 lemma

$\forall e . \text{member2 } e \text{ ?} \rightarrow \text{ ?}$

This lemma can be proved by induction as follows.

<code>member2 e ?</code>	<code>-&gt; member2 e (False &lt;&gt; True)</code>	<i>(? expansion)</i>
<code>-&gt;</code>	<code>member2 e False &lt;&gt; member2 e True</code>	<i>(distributivity)</i>
<code>-&gt;</code>	<code>[] &lt;&gt; (non e : member2 e False) &lt;&gt;</code>	<i>(defn. member2)</i>
	<code>    (e : member2 e ?) &lt;&gt; (non e : member2 e True)</code>	
<code>-&gt;</code>	<code>[] &lt;&gt; (non e : member2 e ?) &lt;&gt; (e : member2 e ?)</code>	<i>(distributivity)</i>
<code>-&gt;</code>	<code>[] &lt;&gt; ((e &lt;&gt; non e) : member2 e ?)</code>	<i>(distributivity)</i>
<code>-&gt;</code>	<code>[] &lt;&gt; (? : member2 e ?)</code>	<i>(law non)</i>
<code>-&gt;</code>	<code>[] &lt;&gt; (? : ?)</code>	<i>(induction)</i>
<code>-&gt;</code>	<code>?</code>	<i>(? contraction)</i>