Derivation of logic programs by functional methods

A. Bijlsma*

June, 1991

Keywords: logic programming, functional programming, program derivation.

1 Introduction

In this note we present a method for the calculational derivation of logic programs, employing techniques recently developed for the derivation of functional programs.

It has been proposed [10] that the process of synthesizing logic programs should begin with a specification that is itself a (possibly inefficient) logic program; subsequently transformations might be applied to obtain from this a logic program with more desirable properties. The difficulty with this point of view is that some problem descriptions that one might like to consider are not trivially equivalent to logic programs. In these cases, important design decisions are involved in their reformulation; hence, this phase should be considered part of the programming process. Moreover, if the initial specification were required to be a logic program itself, it would in many cases be less than clear whether the specification really describes the problem we are interested in. Finally, such a requirement tends to load the dice in favour of solutions whose structure resembles that of the specification, making them much easier to derive than all other ones.

Therefore, we take a different approach. The predicates occurring in logic programs may be interpreted as boolean functions. Starting with a specification of arbitrary form, we may derive, for these boolean functions, functional programs. This may be done by calculational techniques, such as the ones developed by Burstall and Darlington [4], Bird [2, section 5.5] [1], and Hoogerwoord [9]. In the next section, we shall prove a theorem that states precisely when and how these functional programs may be transformed into logic programs.

Several other methods for the synthesis of logic programs contain intermediate results that are, in effect, functional programs. In particular, the 'logic descriptions' of Deville [6] and the 'tracts' of Bundy et al. [3] may be so regarded.

*Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

2 The theorem and its proof

Predicates are viewed as boolean functions on some anonymous space. Notation and terminology for predicate calculus are taken from [7]; in particular, enclosing a formula in square brackets denotes universal quantification over the domain space, and implication and equivalence are considered less binding than conjunction and disjunction.

Let a, b, c be predicates and let h be a predicate transformer. We are interested in equation

$$q: \quad [a \Rightarrow q] \ \& \ [b \Rightarrow \neg q] \ \& \ [c \Rightarrow (q \equiv h.q)] \quad , \tag{1}$$

because the denotational semantics of functional programs [14] [12, section 10.4] defines the meaning of a recursive functional program (of type boolean) as the least defined solution of an equation of the form (1). In particular, if the functional program is complete in the sense that it defines a total function on the domain space, equation (1) has a unique solution. We are also interested in equation

$$q: \quad [a \Rightarrow q] \ \& \ [c \wedge h.q \Rightarrow q] \quad , \tag{2}$$

because the declarative semantics of logic programs [8] [11, section 6] defines the meaning of a logic program as an interpretation (the *least Herbrand model*) which maps every predicate symbol occurring in the program to a predicate obtained as strongest solution of an equation of the form (2). We now proceed to investigate the relation between (1) and (2).

The final conjunct of (1) may be rewritten as follows: for any q,

$$\begin{array}{ll}
 & [c \Rightarrow (q \equiv h.q)] \\
\equiv & \quad \{\text{elimination of } \equiv\} \\
 & [c \Rightarrow (h.q \Rightarrow q) \wedge (\neg h.q \Rightarrow \neg q)] \\
\equiv & \quad \{\text{distributing}\} \\
 & [c \Rightarrow (h.q \Rightarrow q)] \ \& \ [c \Rightarrow (\neg h.q \Rightarrow \neg q)] \\
\equiv & \quad \{\text{shunting}\} \\
 & [c \wedge h.q \Rightarrow q] \ \& \ [c \wedge \neg h.q \Rightarrow \neg q] \quad .
\end{array}$$

Therefore, for any q,

$$\begin{array}{ll}
 & [a \Rightarrow q] \ \& \ [b \Rightarrow \neg q] \ \& \ [c \Rightarrow (q \equiv h.q)] \\
\equiv & \quad \{\text{previous derivation}\} \\
 & [a \Rightarrow q] \ \& \ [b \Rightarrow \neg q] \ \& \ [c \wedge h.q \Rightarrow q] \ \& \ [c \wedge \neg h.q \Rightarrow \neg q] \\
\equiv & \quad \{\text{distributing}\} \\
 & [a \Rightarrow q] \ \& \ [c \wedge h.q \Rightarrow q] \ \& \ [b \vee (c \wedge \neg h.q) \Rightarrow \neg q] \quad .
\end{array}$$

Now let us assume that h is monotonic. Then the Knaster-Tarski theorem [7, (8,25)] tells us that (2), or, equivalently,

$$q: \quad [a \vee (c \wedge h.q) \Rightarrow q] \quad ,$$

has a strongest solution, which is also a solution of

$$q: \quad [a \vee (c \wedge h.q) \equiv q] \quad . \tag{3}$$

Hence, if we succeed in showing that every solution of (3) satisfies

$$[b \vee (c \wedge \neg h.q) \Rightarrow \neg q] \quad ,$$

we may conclude that (1) also has a strongest solution, and that it has the same strongest solution as (2). To this end, we observe

$$
\begin{array}{ll}
 & [a \vee (c \wedge h.q) \equiv q] \;\Rightarrow\; [b \vee (c \wedge \neg h.q) \Rightarrow \neg q] \\
\Leftarrow & \quad \{\text{Leibniz}\} \\
 & [b \vee (c \wedge \neg h.q) \;\Rightarrow\; \neg(a \vee (c \wedge h.q))] \\
\equiv & \quad \{\text{de Morgan}\} \\
 & [b \vee (c \wedge \neg h.q) \;\Rightarrow\; \neg a \wedge (\neg c \vee \neg h.q)] \\
\equiv & \quad \{\text{distributing}\} \\
 & [b \Rightarrow \neg a] \;\&\; [b \Rightarrow \neg c \vee \neg h.q] \;\&\; [c \wedge \neg h.q \Rightarrow \neg a] \;\&\; [c \wedge \neg h.q \Rightarrow \neg c \vee \neg h.q] \\
\Leftarrow & \quad \{\text{weakening antecedents and strengthening consequents}\} \\
 & [b \Rightarrow \neg a] \;\&\; [b \Rightarrow \neg c] \;\&\; [c \Rightarrow \neg a] \\
\equiv & \quad \{\text{elimination of } \Rightarrow\} \\
 & [\neg(a \wedge b)] \;\&\; [\neg(b \wedge c)] \;\&\; [\neg(c \wedge a)] \quad .
\end{array}
$$

The property in the final line will be expressed by calling a, b, and c *disjoint*. We have now obtained the following theorem:

Theorem *Let* a, b, c *be disjoint predicates and let* h *be a monotonic predicate transformer. Then equations*

$$q: \quad [a \Rightarrow q] \;\&\; [b \Rightarrow \neg q] \;\&\; [c \Rightarrow (q \equiv h.q)]$$

and

$$q: \quad [a \Rightarrow q] \;\&\; [c \wedge h.q \Rightarrow q]$$

have the same strongest solution.

3 An example derivation

For comparison purposes, we illustrate our methods by a very well-known problem [5, sec. 7.5] [13, program 7.4] [6]: write a program for the predicate p such that $p.d.x.y$ holds if and only if y is the list obtained by removing the first occurrence of d in the list x.

As a formal specification for this problem we choose

$$p.d.x.y \equiv (\exists u, v : n.d.u : x = u +\!\!+ [d] +\!\!+ v \;\&\; y = u +\!\!+ v) \quad ,$$

where

$$n.d.u \equiv (\forall s, t :: u \neq s +\!\!+ [d] +\!\!+ t)$$

and where $+\!\!+$ denotes concatenation of lists. This specification is not a logic program, nor trivially equivalent to one; it has, however, the more important virtue of obviously describing the given problem.

For the moment, we ignore matters of type: we shall assume silently that s, t, u, v, w, x, y, z denote lists. Relation p may be regarded as a boolean function. For this function we now derive a functional program. We use induction on the length of x. We let $e;x$ denote the list with head e and tail x, i.e. $[e] +\!\!+ x$.

The base of the induction is

$$
\begin{array}{ll}
 & p.d.[].y \\
\equiv & \quad \{\text{specification to be satisfied}\} \\
 & (\exists u, v : n.d.u : [] = u +\!\!+ [d] +\!\!+ v \;\&\; y = u +\!\!+ v) \\
\equiv & \quad \{\} \\
 & \textit{false} \quad .
\end{array}
$$

For the step of the induction, we treat the cases $y = []$ and $y \neq []$ separately. First the case $y = []$:

$$
\begin{array}{ll}
 & p.d.(e;x).[] \\
\equiv & \quad \{\text{specification to be satisfied}\} \\
 & (\exists u, v : n.d.u : e;x = u +\!\!+ [d] +\!\!+ v \ \ \& \ \ [] = u +\!\!+ v) \\
\equiv & \quad \{\ [] = u +\!\!+ v \ \equiv \ u = [] \ \ \& \ \ v = []\ \} \\
 & n.d.[] \ \ \& \ \ e;x = [d] \\
\equiv & \quad \{\text{recursive definition of } n\text{, see below}\} \\
 & e;x = [d] \\
\equiv & \quad \{\text{list equality}\} \\
 & e = d \ \ \& \ \ x = [] \quad .
\end{array}
$$

Now the case $y \neq []$, say $y = f;z$:

$$
\begin{array}{ll}
 & p.d.(e;x).(f;z) \\
\equiv & \quad \{\text{specification to be satisfied}\} \\
 & (\exists u, v : n.d.u : e;x = u +\!\!+ [d] +\!\!+ v \ \ \& \ \ f;z = u +\!\!+ v) \\
\equiv & \quad \{\text{domain split, } u = [] \ \vee \ (\exists g, w :: u = g;w)\ \} \\
 & (\exists v : n.d.[] : e;x = [d] +\!\!+ v \ \ \& \ \ f;z = v) \\
 & \vee\ (\exists g, w, v : n.d.(g;w) : e;x = g;w +\!\!+ [d] +\!\!+ v \ \ \& \ \ f;z = g;w +\!\!+ v) \\
\equiv & \quad \{\text{list equality, elimination of } v \text{ and } g \text{ respectively}\} \\
 & (n.d.[] \ \ \& \ \ e = d \ \ \& \ \ x = f;z) \\
 & \vee\ (\exists w, v : n.d.(e;w) : e = f \ \ \& \ \ x = w +\!\!+ [d] +\!\!+ v \ \ \& \ \ z = w +\!\!+ v) \\
\equiv & \quad \{\text{recursive definition of } n\text{, see below}\} \\
 & (e = d \ \ \& \ \ x = f;z) \\
 & \vee\ (\exists w, v : e \neq d \ \ \& \ \ n.d.w : e = f \ \ \& \ \ x = w +\!\!+ [d] +\!\!+ v \ \ \& \ \ z = w +\!\!+ v) \\
\equiv & \quad \{\text{distribution of } \wedge \text{ over } \exists\ \} \\
 & (e = d \ \ \& \ \ x = f;z) \\
 & \vee\ (e \neq d \ \ \& \ \ e = f \ \ \& \ \ (\exists w, v : n.d.w : x = w +\!\!+ [d] +\!\!+ v \ \ \& \ \ z = w +\!\!+ v) \\
\equiv & \quad \{\text{induction hypothesis}\} \\
 & (e = d \ \ \& \ \ x = f;z) \ \vee \ (e \neq d \ \ \& \ \ e = f \ \ \& \ \ p.d.x.z) \quad .
\end{array}
$$

This shows that

$$
\begin{aligned}
p.d.[].y &\equiv \mathit{false} \quad , && (4)\\
p.d.(e;x).[] &\equiv e = d \ \ \& \ \ x = [] \quad , && (5)\\
p.d.(e;x).(f;z) &\equiv (e = d \ \ \& \ \ x = f;z) \ \vee \ (e \neq d \ \ \& \ \ e = f \ \ \& \ \ p.d.x.z) \quad , && (6)
\end{aligned}
$$

provided n satisfies

$$
\begin{aligned}
n.d.[] &\equiv \mathit{true} \quad , && (7)\\
n.d.(e;w) &\equiv e \neq d \ \ \& \ \ n.d.w \quad . && (8)
\end{aligned}
$$

To prove (7), we observe

$$
\begin{array}{ll}
 & n.d.[] \\
\equiv & \quad \{\text{specification to be satisfied}\} \\
 & (\forall s, t :: [] \neq s +\!\!+ [d] +\!\!+ t) \\
\equiv & \quad \{\} \\
 & \mathit{true} \quad ,
\end{array}
$$

and for (8) we have

$$
\begin{array}{ll}
 & n.d.(e;w) \\
\equiv & \quad \{\text{specification to be satisfied}\} \\
 & (\forall s,t :: e;w \neq s +\!\!+ [d] +\!\!+ t) \\
\equiv & \quad \{\text{domain split, } s = [] \ \vee \ (\exists g,v :: s = g;v)\ \} \\
 & (\forall t :: e;w \neq [d] +\!\!+ t) \ \ \& \ \ (\forall g,v,t :: e;w \neq g;v +\!\!+ [d] +\!\!+ t) \\
\equiv & \quad \{\text{list equality}\} \\
 & e \neq d \ \ \& \ \ (\forall g,v,t :: e \neq g \ \vee \ w \neq v +\!\!+ [d] +\!\!+ t) \\
\equiv & \quad \{\text{trading, one-point rule}\} \\
 & e \neq d \ \ \& \ \ (\forall v,t :: w \neq v +\!\!+ [d] +\!\!+ t) \\
\equiv & \quad \{\text{induction hypothesis}\} \\
 & e \neq d \ \ \& \ \ n.d.w \quad .
\end{array}
$$

This completes the proof that p satisfies (4) through (6). These equations in effect constitute a functional program for p, expressible, for instance, in the Miranda script

```
p d [] y = False
p d (e:x) [] = (e = d) & (x = [])
p d (e:x)(f:z) = ((e = d) & (x = f:z)) \/ ((e ~= d) & (e = f) & p d x z)
```

Apply the theorem in the previous section with a, b, c defined by

$$
\begin{array}{rcl}
a.d.[].y & \equiv & \textit{false} \quad , \\
a.d.(e;x).[] & \equiv & e = d \ \ \& \ \ x = [] \quad , \\
a.d.(e;x).(f;z) & \equiv & e = d \ \ \& \ \ x = f;z \quad , \\
b.d.[].y & \equiv & \textit{true} \quad , \\
b.d.(e;x).[] & \equiv & e \neq d \ \vee \ x \neq [] \quad , \\
b.d.(e;x).(f;z) & \equiv & (e \neq d \ \vee \ x \neq f;z) \ \ \& \ \ (e = d \ \vee \ e \neq f) \quad , \\
c.d.[].y & \equiv & \textit{false} \quad , \\
c.d.(e;x).[] & \equiv & \textit{false} \quad , \\
c.d.(e;x).(f;z) & \equiv & e \neq d \ \ \& \ \ e = f
\end{array}
$$

and with h defined by

$$h.q.d.(e;x).(f;z) \ \equiv \ q.d.x.z \quad .$$

It is a trivial matter to verify that a, b, c, h satisfy the conditions of the theorem and that the conjunction of (4) through (6) is equivalent to (1). The theorem now yields that p is also the strongest solution of (2). Substitution of the definitions of a, c, h in the conjuncts of (2) gives

$$
\begin{array}{ll}
 & [a \Rightarrow q] \\
\equiv & \quad \{\text{definition of } a\ \} \\
 & (\forall d,e,x :: e = d \ \ \& \ \ x = [] \ \Rightarrow \ q.d.(e;x).[]) \\
 & \wedge\ (\forall d,e,x,f,z :: e = d \ \ \& \ \ x = f;z \ \Rightarrow \ q.d.(e;x).(f;z)) \\
\equiv & \quad \{\text{domain merge, } y = [] \ \vee \ (\exists f,z :: y = f;z)\ \} \\
 & (\forall d,e,x,y :: e = d \ \wedge \ x = y \ \Rightarrow \ q.d.(e;x).y) \\
\equiv & \quad \{\text{one-point rule, elimination of } e \text{ and } y\ \} \\
 & (\forall d,x :: q.d.(d;x).x)
\end{array}
$$

and

$$
\begin{array}{ll}
 & [c \wedge h.q \Rightarrow q] \\
\equiv & \quad \{\text{definitions of } c \text{ and } h\} \\
 & (\forall d, e, x, f, z :: e \neq d \ \wedge \ e = f \ \wedge \ q.d.x.z \ \Rightarrow \ q.d.(e;x).(f;z)) \\
\equiv & \quad \{\text{one-point rule, elimination of } f\} \\
 & (\forall d, e, x, z :: e \neq d \ \wedge \ q.d.x.z \ \Rightarrow \ q.d.(e;x).(e;z)) \quad .
\end{array}
$$

We conclude that p is the strongest predicate such that, for all d, e, x, z,

$$
\begin{cases}
p.d.(d;x).x \quad , \\
e \neq d \ \wedge \ p.d.x.z \ \Rightarrow \ p.d.(e;x).(e;z) \quad .
\end{cases}
$$

p is therefore described by the logic program

```
p(D, [D|X], X).
p(D, [E|X], [E|Z]):- E ≠ D, p(D, X, Z).
```

This is, in fact, program 7.4 of [13].

4 Remarks

1. We have followed the convention from [13] in assuming the operator $\neq$ to be provided by the system and defined by a sufficiently large explicit table. However, if we make some assumption concerning the type of the arguments, it is not difficult to derive an explicit program for $\neq$.

2. The program has been derived under the convention that x and z automatically denote lists. Within the framework of logic programming such a convention is not usually present and type information must be explicitly programmed. With the program as given, the query `?- p(5, X, 3)` will succeed with the meaningless answer `X = [5|3]`, rather than failing as it should. A better program would therefore be

```
p(D, [D|X], X):- list(X).
p(D, [E|X], [E|Z]):- E ≠ D, p(D, X, Z).
list([]).
list([D|X]):- list(X).
```

3. It must be emphasized that this paper is about the derivation of logic programs as characterized by their declarative semantics, not about Prolog programs. For instance, it says nothing about the ordering of goals and clauses, nor of metalogical and extralogical predicates. The method described above constructs programs whose declarative semantics is a given predicate; it is not guaranteed that the corresponding sets of terms will be enumerated by a given deterministic search strategy.

4. It is not true *in general* that equations (1) and (2) have a strongest solution. For instance, if we take the integers as domain space, define a, b, c to be *false*, *false*, *true* respectively and let h be defined by $h.q.n \equiv \neg q.(n+1)$ for all n, equation (1) has precisely two solutions (characterizing the even and odd integers, respectively), which are not comparable. Hence (1) has no strongest solution. As the two solutions of (1) also solve (2) but their conjunction, *false*, does not, (2) has no strongest solution either.

5. We remark that the disjointness of a, b, and c cannot be omitted from the theorem, as is demonstrated by the following example. Let a be a nonconstant predicate, let c equal a, let b be the constant predicate *false*, and let h be defined by $[h.q \equiv [q]]$. Then h is monotonic and (1) has the unique solution *true*, as the reader may easily verify. On the other hand, the strongest solution of (2) is in this case a.

6. Similarly, the monotonicity of h cannot be omitted. To see this, choose *false* for a and b, and *true* for c; let h be defined by

$$[h.q \equiv \neg[q \equiv x]] \quad ,$$

where x is some fixed nonconstant predicate. Then (1) has the unique solution *true*, whereas the strongest solution of (2) is now x.

7. In case h is finitely conjunctive, the conjunct $[\neg(b \wedge c)]$ may be omitted from the disjointness condition (we omit the proof of this). In case h is finitely disjunctive, the conjunct $[\neg(a \wedge c)]$ may be omitted. The conjunct $[\neg(a \wedge b)]$ is obviously necessary for the existence of solutions of (1).

Acknowledgements

I wish to thank M.E. van Dijk and E. de Kogel for some very useful comments concerning a previous version of this paper.

References

[1] R.S. Bird, 'Lectures on constructive functional programming', in: M. Broy (ed.), *Constructive methods in computing science*. NATO ASI series F55. Springer, Berlin, 1989; pp. 151–216.

[2] R.S. Bird & P. Wadler, *Introduction to functional programming*. Prentice-Hall, London, 1988.

[3] A. Bundy, A. Smaill, & G. Wiggins, 'The synthesis of logic programs from inductive proofs', in: J.W. Lloyd (ed.), *Computational logic*. Springer, Berlin, 1990.

[4] R.M. Burstall & J. Darlington, 'A transformation system for developing recursive programs'. *J. ACM* **24** (1977), 44–67.

[5] W.F. Clocksin & C.S. Mellish, *Programming in Prolog*. 3rd edition. Springer, Berlin, 1986.

[6] Y. Deville, *Logic programming*. Addison-Wesley, Wokingham, 1990.

[7] E.W. Dijkstra & C.S. Scholten, *Predicate calculus and program semantics*. Springer, New York, 1990.

[8] M.H. van Emden & R.A. Kowalski, 'The semantics of predicate logic as a programming language'. *J. ACM* **23** (1976), 733–742.

[9] R.R. Hoogerwoord, *The design of functional programs: a calculational approach*. Ph.D. thesis, Eindhoven University of Technology, 1989.

[10] R.A. Kowalski, 'The relation between logic programming and logic specification', in: C.A.R. Hoare & J.C. Shepherdson (eds.), *Mathematical logic and programming languages*. Prentice-Hall, London, 1985; pp. 11–24.

[11] J.W. Lloyd, *Foundations of logic programming*. 2nd edition. Springer, Berlin, 1987.

[12] C. Reade, *Elements of functional programming*. Addison-Wesley, Wokingham, 1989.

[13] L. Sterling & E. Shapiro, *The art of Prolog*. MIT Press, Cambridge MA, 1986.

[14] J.E. Stoy, 'Some mathematical aspects of functional programming', in: J. Darlington, P. Henderson, D.A. Turner (eds.), *Functional programming and its applications: an advanced course*. Cambridge University Press, 1982; pp. 217–252.