The *Lazy* Functional Side of Logic Programming*

Sandro Etalle[1] and Jon Mountjoy[2]

[1] Universiteit Maastricht `ealle@cs.unimaas.nl`
[2] University of Amsterdam

1 Introduction

The possibility of translating logic programs into functional ones has long been a subject of investigation. Among the different proposals we find [9, 10, 8, 12, 13, 15]. Common to all the approaches mentioned is that the original logic program, in order to be translated, needs to be *well-moded* and this has led to the common understanding that these programs can be considered to be the "functional part" of logic programs. This is confirmed by the following statement in [10]: "... the class of functionally moded (well-moded and simply moded) programs can be rightly considered *the* functional core of logic programs".

Well-moded programs have, among other features, a straightforward left-to-right dataflow model (see [2]) and prohibit the use of *logical variables* to their full potential such as in complex logical data structures like difference-lists. As a consequence of this it is now widely accepted that "complex" logical variables, the possibility of a dynamic selection rule, and general properties of non-well-moded programs are exclusive features of logic programs.

This is not quite right. At least, not to the extent that one is brought to believe.

In this paper we show, among other things, that logical structures such as difference lists have a natural counterpart in *lazy* functional programs; i.e. that most programs using difference-lists are functional in nature. This shows immediately that many common non-well-moded programs are functional in nature and that well-modedness is thus not a necessary attribute of those logic programs behaving functionally. We do this by employing a straightforward – literal – translation of moded logic programs into Haskell, a lazy functional language.

Furthermore, we use the same translation system to show that some programs requiring a dynamic scheduling mechanism are also intrinsically functional.

Summarizing, in this paper we readdress the old question of what features are exclusive to the logic programming paradigm and demonstrate that the current circumscription is unreasonably restrictive.

* Extended Abstracts of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

2 Preliminaries

Due to space constraints we omit preliminaries and assume that the reader is acquainted with the terminology and the main results of logic programming theory (see [1]). We use over-lined characters to indicate (a possibly empty) sequence of objects, so $\overline{t}$ can denote a sequence $t_1, \ldots, t_n$ of terms, $\overline{x}$ a sequence of variables and $\overline{\mathtt{A}}$ a sequence of atoms (i.e. a query).

Haskell Programs Our translation system maps logic programs into lazy functional programs, which are written in (a subset of) Haskell [6]. The subset we use includes the proposed extension of *pattern guards*[1]. The programs we are going to generate are built as sets of *equations*, each of the following form:

$$
\begin{array}{lll}
\mathsf{f}\ \mathsf{s}_1\ \ldots\ \mathsf{s}_j & \mid \mathsf{guard}_{1,1},\ \ldots,\ \mathsf{guard}_{1,j} & = \mathsf{result}_1 \\
 & \vdots & \\
 & \mid \mathsf{guard}_{n,1},\ \ldots,\ \mathsf{guard}_{n,m} & = \mathsf{result}_{n,1} \\
 & \mid \mathsf{otherwise} & = \mathsf{result}_{n+1}
\end{array}
$$

where f is a function symbol, $\mathsf{s}_1\ \ldots\ \mathsf{s}_j$ are *parameters* and $\mathsf{guard}_{\mathsf{x,y}}$, $\mathsf{otherwise}$ are *guard qualifiers*. The '|' introduces a guard, and the ',' acts as a logical conjunctive. Pattern matching may take place on the parameters. Without pattern guards, the guard qualifiers would have to be boolean expressions; *patterns guards* can also contain *let-expressions* (which are defined as usual) and *pattern-matching expressions* which are expressions of the form $\mathsf{pattern} \leftarrow \mathsf{term}$, and whose semantics is the following: if term matches with $\mathsf{pattern}$ then the variables in $\mathsf{pattern}$ are appropriately instantiated, and the pattern-matching guard returns true, otherwise it return false.

3 A Translation System

Moded Logic Programs Even though the programs we are going to translate are *not* necessarily well-moded (after all we want to show that non-well-moded programs might be functional), we have to make some mode-based restrictions on the initial logic programs (in the Appendix we also report the definition of well-moded logic programs).

Definition 1 (Mode). *Consider an n-ary relation symbol* $\mathtt{p}$. *By a* mode *for* $\mathtt{p}$ *we mean a function* $\mathtt{m_p}$ *from* $\{1, \ldots, n\}$ *to the set* $\{\mathtt{In}, \mathtt{Out}\}$. *If* $\mathtt{m_p}(\mathtt{i}) = \mathtt{In}$, *we call i an* input position *of p and if* $\mathtt{m_p}(\mathtt{i}) = \mathtt{Out}$, *we call i an* output position *of* $\mathtt{p}$ *(both w.r.t.* $\mathtt{m_p}$*).* □

[1] Pattern guards are described in http://www.dcs.gla.ac.uk/˜simonpj/. It is important to remark that we could also do without pattern guards – at the price of less elegant translation. Thus all the statements we are going to give in the sequel are true regardless of the availability of a pattern guard construct in the target language.

An n-ary relation $\mathtt{p}$ with a mode $\mathtt{m_p}$ will be denoted by $\mathtt{p}(\mathtt{m_p}(\mathtt{1}),\ldots,\mathtt{m_p}(\mathtt{n}))$ (e.g., `append(In,In,Out)`). We assume that to each relation symbol is associated a unique mode. Here and in the sequel, when writing an atom as $\mathtt{p}(\overline{\mathtt{u}},\ \overline{\mathtt{v}})$ we assume that $\overline{\mathtt{u}}$ is the sequence of terms filling in its input positions and $\overline{\mathtt{v}}$ is the sequence of terms filling in its output positions.

Now, we require the programs and the queries to be *plain* and *consistent* wrt the chosen mode. Let us call *producing* the input position of the head and the output positions of the body atoms, and *consuming* the other positions of a clause. Recall that, a set of terms is called *linear* if every variable occurs at most once in it.

– A clause (query) is *consistent* (wrt the mode) iff every variable occurs in at least one producing position.
– A clause $\mathtt{p_0}(\overline{\mathtt{t}}_0,\ \overline{\mathtt{s}}_{\mathtt{n+1}}) \leftarrow \mathtt{p_1}(\overline{\mathtt{t}}_1,\ \overline{\mathtt{s}}_1),\ldots,\mathtt{p_n}(\overline{\mathtt{t}}_\mathtt{n},\ \overline{\mathtt{s}}_\mathtt{n})$ is called *plain* if: $\overline{\mathtt{s}}_1,\ldots,\overline{\mathtt{s}}_n$ is a linear family of variables, and $\overline{\mathtt{t}}_0$ is linear.
– Similarly, a query $\overline{\mathtt{Q}}$ is called *plain* iff the clause $\mathtt{q} \leftarrow \overline{\mathtt{Q}}$ is plain, where $\mathtt{q}$ is any atom of zero arity, and a program is *plain* if every clause of it is plain.

Our translation requires programs to be consistent and plain. This is far less restrictive than well-modedness plus simple-modedness, and allows us to capture a broader segment of functional behaviour found in logic programs. It can be shown that most programs are plain, and that virtually all consistent programs are either plain or safely translatable into a plain form, we refer to the temporary full version [7] for the details.

Test and Non-Test Predicates An important tool we employ for the translation is a *partitioning* of the predicates into *test* and *non-test* ones: In logic programming, queries can succeed, loop or fail. This third possibility is of crucial importance. Nevertheless, relations which are "not supposed to fail" are quite common. We say that a relation is "not supposed to fail" if – when called in a "correct" way – produces at least one answer (e.g., `sort`, `flatten` and `append`) The ubiquity of such predicates is confirmed by [4, 11, 5] and the fact that Mercury [14] employs a mode system which actually subsume our partitioning.

A *partitioning* is a map from the set of predicate symbols into the set $\{$*test*, *non-test*$\}$. We also say that the program $\mathtt{P}$ is *correctly partitioned* wrt the query $\overline{\mathtt{A}}$ iff every time that a non-test atom $\mathtt{B}$ is selected in a LD-derivation of $\overline{\mathtt{A}}$ in $\mathtt{P}$ then $\mathtt{B}$ has at least one non-failing LD-derivation. Thus every program is correct wrt every query when the partitioning is the trivial one in which all predicates are *test*. Using a non-trivial partitioning has the advantage of undercovering the *"lazy aspects"* of logic programming, which is what we want to do. Checking correctness is orthogonal to the purposes of this paper, but we should mention that it can be done either using abstract interpretation [5] or on modes and types [4, 11].

The Translation Clearly, we need to capture the fact that a predicate might succeed (returning a computed answer), or fail. To do so, we introduce the datatype Result in our Haskell programs by:

$\mathsf{data\ Result}\ \alpha = \mathsf{Suc}\ \alpha \mid \mathsf{Fail}$

That is, the datatype Result has two constructors, Fail and Suc, the latter of which can be applied to some term. Our translation will transform non-test predicates as ordinary functions, but transform test predicates into functions returning something of the type Result α, allowing us to indicate failure. Let $\mathtt{p}$ be a predicate symbol, which mode indicates that it has i input and j output positions. Then $\mathtt{p}$ can naturally be translated into a function of type

$$\begin{array}{lll} \mathsf{p} : \mathsf{T}_1 \times \cdots \times \mathsf{T}_i & \to & (\mathsf{S}_1 \times \cdots \times \mathsf{S}_j) \quad \text{if } \mathtt{p} \text{ is a } \textit{non-test} \text{ predicate} \\ \mathsf{p} : \mathsf{T}_1 \times \cdots \times \mathsf{T}_i & \to & \mathsf{Result}(\mathsf{S}_1 \times \cdots \times \mathsf{S}_j) \text{ if } \mathtt{p} \text{ is a } \textit{test} \text{ predicate} \end{array}$$

Where T_i and S_i are appropriate Haskell types. The actual translation method requires the program to be translated to be *consistent*, *plain* and correctly partitioned. Let $\mathtt{P}$ be a logic program, and

$$\begin{array}{l} \mathtt{p}(\bar{\mathtt{t}}_1, \bar{\mathtt{s}}_1) \leftarrow \mathtt{p}_{1,1}(\bar{\mathtt{i}}_{1,1}, \bar{\mathtt{o}}_{1,1}), \ldots, \mathtt{p}_{1,k_1}(\bar{\mathtt{i}}_{1,k_1}, \bar{\mathtt{o}}_{1,k_1}),\ \mathtt{q}_{1,1}(\bar{\mathtt{u}}_{1,1}, \bar{\mathtt{v}}_{1,1}), \ldots, \mathtt{q}_{1,l_1}(\bar{\mathtt{u}}_{1,l_1}, \bar{\mathtt{v}}_{1,l_1}). \\ \quad\vdots \\ \mathtt{p}(\bar{\mathtt{t}}_n, \bar{\mathtt{s}}_n) \leftarrow \mathtt{p}_{n,1}(\bar{\mathtt{i}}_{n,1}, \bar{\mathtt{o}}_{n,1}), \ldots, \mathtt{p}_{n,k_n}(\bar{\mathtt{i}}_{n,k_n}, \bar{\mathtt{o}}_{n,k_n}),\ \mathtt{q}_{n,1}(\bar{\mathtt{u}}_{n,1}, \bar{\mathtt{v}}_{n,1}), \ldots, \mathtt{q}_{n,l_n}(\bar{\mathtt{u}}_{n,l_n}, \bar{\mathtt{v}}_{n,l_n}). \end{array}$$

be the set of clauses of $\mathtt{P}$ defining predicate $\mathtt{p}$, where the predicates $\mathtt{p}_{\mathtt{i},\mathtt{j}}$ are test predicates and the predicates $\mathtt{q}_{\mathtt{i},\mathtt{j}}$ are the non-test ones. Here we assume that the clauses had been renamed apart, i.e., that they share no variables. If $\mathtt{p}$ is a *test* predicate, then the translation of the above section into Haskell is the following script (for the moment the underlined parts have to be treated as if the underline was not there):

$$\begin{array}{lll} \mathsf{p}\ (\bar{\mathsf{x}}) & \mid & (\bar{\mathsf{t}}_1) \leftarrow (\bar{\mathsf{x}}), \\ & & \mathsf{Suc}\ (\bar{\mathsf{o}}_{1,1}) \leftarrow \mathsf{p}_{1,1}(\bar{\mathsf{i}}_{1,1}),\ \ldots, \mathsf{Suc}(\bar{\mathsf{o}}_{1,k_1}) \leftarrow \mathsf{p}_{1,k_1}(\bar{\mathsf{i}}_{1,k_1}), \\ & & \mathsf{let}\ (\bar{\mathsf{v}}_{1,1}) = \mathsf{q}_{1,1}(\bar{\mathsf{u}}_{1,1}),\ \ldots, \mathsf{let}\ (\bar{\mathsf{v}}_{1,l_1}) = \mathsf{q}_{1,l_1}(\bar{\mathsf{u}}_{1,l_1}) \quad = \underline{\mathsf{Suc}}(\bar{\mathsf{s}}_1) \\ & & \vdots \\ & \mid & (\bar{\mathsf{t}}_n) \leftarrow (\bar{\mathsf{x}}), \\ & & \mathsf{Suc}\ (\bar{\mathsf{o}}_{n,1}) \leftarrow \mathsf{p}_{n,1}(\bar{\mathsf{i}}_{n,1}),\ \ldots, \mathsf{Suc}(\bar{\mathsf{o}}_{n,k_n}) \leftarrow \mathsf{p}_{n,k_n}(\bar{\mathsf{i}}_{n,k_n}), \\ & & \mathsf{let}\ (\bar{\mathsf{v}}_{n,1}) = \mathsf{q}_{n,1}(\bar{\mathsf{u}}_{n,1}),\ \ldots, \mathsf{let}\ (\bar{\mathsf{v}}_{n,l_n}) = \mathsf{q}_{n,l_n}(\bar{\mathsf{u}}_{n,l_n}) \quad = \underline{\mathsf{Suc}}(\bar{\mathsf{s}}_n) \\ & \underline{\mid} & \underline{\mathsf{otherwise} = \mathsf{Fail}} \end{array}$$

If one of the clauses of the above section is a unit clause, (i.e. $\mathtt{p}(\bar{\mathtt{t}}, \bar{\mathtt{s}}_\mathtt{j})$.) or if its body contains no test predicates then the corresponding line has the trivial guard True. Note that sequences may be empty, so if the predicate had no output positions, then $\bar{\mathsf{s}}$ would be the term ( ).

If $\mathtt{p}$ is a *non-test* predicate, then translation of the above section corresponds to the above script *after removal* of the underlined parts; namely we have to eliminate from it the otherwise statement and the Suc's from the return values.

Clearly, list constructions and built-in predicates need to be handled separately. The electronically available technical report [7] reports more examples.

4 Logic Programs in a Lazy Functional Language

We are at last in a position to demonstrate our thesis that the set of logic programs considered as functional needs to be expanded. As issues, we consider logic variables and dynamic scheduling.

Logical Variables vs. Lazy Evaluation *Logical variables* are one of the peculiarities of logic programming. Most of the time, they are used just as variables in an imperative language – this is the case when the program is well-moded and simply moded. Nevertheless there are important situations in which logical variables are exploited in all their power. A typical such case is in the presence of *difference structures* such as *difference lists*, as in the following Polish Flag Problem example (a simplified version of Dijkstra's Dutch Flag Problem), which reads as follows: given a list of objects which are either red or white, rearrange it in such a way that the red elements appear first and the white ones appear after them.

```
polish(InList, RedWhites) ←
   distribute(InList, RedWhites, Whites, Whites).
distribute([ ], Reds, Reds, []).
distribute([X|Xs], [X|RedsHead], RedsTail, Whites) ←red(X),
   distribute(Xs, RedsHead, RedsTail, Whites).
distribute([X|Xs], RedsHead, RedsTail, [X|Whites]) ←white(X),
   distribute(Xs, RedsHead, RedsTail, Whites).
```

Where we assume that predicates `red` and `white` are appropriately defined elsewhere in the program and have mode `red(In)`:test. The other predicates are non-test and have mode `polish(In,Out)` and `distribute(In,Out,In,Out)`. This program is plain, consistent and correctly partitioned and by translating it we obtain

```
polish inlist     | t ← inlist,
                    let (redwhites, whites) = distr(t, whites)   = redwhites
distr(is, rstail) | ([ ],rs) ← (is,rstail)                       = (rs, [ ])
                  | (x:xs,rs) ← (is,rstail),
                    let (rshead, whites) = distr(xs, rs),
                    Suc () ← red x                               = (x:rshead, whites)
                  | (x:xs,rs) ← (is,rstail),
                    let (rshead, whites) = distr(xs, rs),
                    Suc () ← white x                             = (rshead, x:whites)
```

This program runs perfectly well. Notice that the definition of polish employs a *circular data structure* [3]: in fact the variable whites appears both on the left hand side and on the right hand side of the expression let (redwhites, whites) = distr(t, whites), in the guard. It is also important to notice that the original logic program employs logical variables in a highly non-trivial way dute to the double occurrence of the variable `Whites` in the body of the first clause. In fact, it uses difference-lists. This is confirmed by the fact that the program is not well-moded. Considering the above translation, we can say that the difference list part was taken care of by the circular data structure. We dare to generalize this and say that difference lists have a natural counterpart in the circular structures of *lazy* functional programming (circular structures do not generally function within *strict* functional language). Of course, one can contrive an example of a program using difference-lists which would not work in Haskell (some programs cannot be translated using our systemsee e.g., the `SEQUENCE` example reported in the technical report of this extended abstract [7]). We don't want to deny this, on the contrary: here we are interested in how programs are *usually* used, and in pointing out that some standard methodologies which are normally considered as applicable only to LP, are actually not so.

Dynamic scheduling vs. Lazy Evaluation Another prominent property of logic programming is the possibility of having a dynamic selection rule, possibly guided by appropriate *delay declarations*. Let us consider the following example, which, given the list `Xs` of integer values, `del_max(Xs,Zs)` produces the list `Zs` by deleting all the occurrences of its maximum element. It does so by calling `find_max_and_del`, which reads: `find_max_and_del(InList,El,OutList,Max) :- Max` is the maximum element of `InList`, and `OutList` is obtained from `Inlist` by deleting all the occurrences of `El` from it.

```
del_max(Xs, Zs) ← find_max_and_del(Xs, Max, Zs, Max).
find_max_and_del([ ], _, [ ], 0).
find_max_and_del([X | Xs], El, Ys, Max) ←
   find_max_and_del(Xs, El, Zs, Max'),
   sup(X, Max', Max),
   del_if_first([X | Zs], El, Ys).
del_if_first([EL | Zs], El, Zs).
del_if_first([X | Zs], El, [X|Zs]) ← X ≠ El.
```

Together with the following mode: `del_max(In,Out)`, `sup(In,In,Out)` (the definition of this is not reported, but standard), `find_max_and_del(In,In,Out,Out)`, `del_if_first(In,In,Out)`, and `≠(In,In)`, and where all predicates except for ≠ are declared as *non-test*. It is worth noticing that the program uses logical variables in a nontrivial way. This is confirmed by the fact that it is not well-moded[2]. Furthermore, the program requires *dynamic scheduling*. In fact, when run with a standard left-to-right selection rule, the query `del_max(ts, Zs)` (`ts` being a list of natural numbers) usually leads to a run-time error (or to an incorrect answer), and, provided that we fix this problem, to a very inefficient computation (with quadratic instead of linear complexity). Both problems can be solved by employing a *dynamic selection rule* using the following *delay declarations*:

```
delay sup(X, Y, _) until ground(X)  ∧  ground(Y).
```

[2] Specifically, the variable `Max` in the first clause is used as an asynchronous communication channel between processes, as the atom `find_max_and_del(Xs, Max, Max, Zs)` uses `Max` as input value that it has to produce itself

```
delay ≠(X,Y) until ground(X)  ∧  ground(Y).
delay del_if_first([X|Xs], El, _) until ground(X)  ∧  ground(El)
```

For instance, the first declaration will suspend any call to `sup(t,s,v).` until `t` and `s` are ground terms. Now, let us for a moment not bother about the delay declarations and translate this program into Haskell. We obtain the following script.

```
del_max as                 | xs  ← as,
                             let (zs, maxel) = find_max_and_del (xs,maxel)   = zs
find_max_and_del (x1,x2)   | ([ ], el)  ← (x1, x2)                           = ([ ], 0)
                           | (x:xs, el)  ← (x1,x2),
                             let (zs, maxel') = find_max_and_del (xs,el),
                             let maxel = max x maxel',
                             let ys = del_if_first (x:zs, el)                = (ys, maxel)
del_if_first (x1, x2)      | ([ ], el)  ← (x1, x2)                           = [ ]
                           | (x:xs, el)  ← (x1,x2),
                             x == el                                         = zs
                           | (x:xs, el)  ← (x1,x2),
                             x /= el                                         = x:zs
```

This program works fine, and its runtime complexity is linear in the size of the input. We can therefore state that the lazy computational mechanism compensates for the lack of control over dynamic scheduling, without which the above logic program could not be run or would have a quadratic complexity. The fact that lazy evaluation here plays a crucial role is confirmed by the fact that in a strict language the translated program would not function properly. Thus again we are in presence of a program exploiting logical variables in a complex way which nevertheless has a natural counterpart in lazy functional programming..

Backtracking and Nondeterminism Another crucial feature of logic programs is their backtracking mechanism, which virtually implements a *don't know* nondeterministic system. In the light of the above examples, we believe that nondeterminism is by far the most important and the mostly used peculiar feature of the logic programming paradigm. We don't want to challenge this, on the contrary. Our translation works only for those programs which don't employ backtracking in a crucial way. For a discussion on this we refer to [7].

5 Conclusions

The goal of our research was to investigate to which extent some features considered peculiar of the logic programming paradigm are really so. For this purpose we have devised a simple – literal – system which enabled us to translate logic programs into the lazy functional language Haskell. It is known (see also [9, 8, 12, 15]) that *if we restrict our attention to non-failing, non-backtracking computation* then well-moded simply moded programs have a natural counterpart in a functional language. The properties of being well-moded and simply moded indicates a manner of use of variables in logic programming which is undoubtedly "functional". To this statement we want to add that well- and simply moded programs can be considered as *strictly* functional, as they can be safely translated into a *strict* functional language.

In this paper we have shown that in a *lazy* functional language, this picture broadens significantly, and some of the features that were – in the light of the results above – commonly considered as exclusive of the logic programming paradigm, can be naturally found in a lazy functional language such as Haskell. In particular, we have shown that the use of complex logical variables in data structures such as difference lists (or such as in program in `del_max`) find a natural counterpart in the circular structures [3] of lazy functional programs. We could then attempt a rough classification of logic programs according to the level of complexity at which they employ their variables (backtracking and nondeterminism is not considered here). We then have the following division: (i) *Strictly Functional* programs which use variables in a standard (imperative-like) way. These are characterized by being *well-moded.* (ii) *Lazy Functional* programs which admit a safe translation into Haskell. (iii) *Intrinsically Logical* programs which do not admit a safe translation into Haskell with our translation scheme.

Acknowledgments We would like to thank the anonymous referees for their useful suggestions.

References

1. K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
2. K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, Lecture Notes in Computer Science, pages 1–19, Berlin, 1993. Springer-Verlag.
3. R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.
4. A. Bossi and N. Cocco. Programs without failures. In R. Fuchs, editor, *Proc. Seventh Workshop on Logic Program Synthesis and Transformation*, volume 1463 of *Lecture Notes in Computer Science*, pages 28–48. Springer-Verlag, Berlin, 1998.
5. S. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming*, pages 48–62, Cambridge, 1997. MIT Press.
6. P. Hudak (ed), S. L. Peyton Jones (ed), and P. L. Wadler (ed). Report on the programming language Haskell – a non-strict purely functional language, version 1.2. *ACM SIGPLAN notices*, 27(5):1–162, May 1992.
7. S. Etalle and J. Mountjoy. The lazy functional side of logic programming. Technical Report CS 00-02, Universiteit Maastricht, 2000. Available at the CoRR at the address http://arXiv.org/abs/cs/0003070.

8. H. Ganzinger and U. Waldmann. Termination Proofs of Well-Moded Logic Programs via Conditional Rewrite Systems. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *Conditional Term Rewriting Systems, Third International Workshop*, LNCS 656, pages 430–437. Springer-Verlag, 1992.
9. M. Marchiori. Logic Programs as Term Rewriting Systems. In Giorgio Levi and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming, 4th International Conference, ALP'94*, volume 850 of *lncs*, pages 223–241. Springer-Verlag, 1994.
10. M. Marchiori. The Functional Side of Logic Programming. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 55–65, La Jolla, California, 1995. ACM Press.
11. D. Pedreschi and S. Ruggieri. On logic programs that do not fail. In S. Etalle and J-G. Smaus, editors, *Proc. ICLP99 Workshop on Verification of Logic Programs*, volume 30 of *Electronic Notes in Theoretical Computer Science*, 1999. http://www.elsevier.nl/locate/entcs/.
12. M. R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. Transformational Methodology for Proving Termination of Logic Programs. *Journal of Logic Programming*, 34(1):1–41, 1998.
13. U. S. Reddy. Transformation of logic programs into functional programs. In *International Symposium on Logic Programming*, pages 187–198, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.
14. Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
15. F. van Raamsdonk. Translating Logic Programs into Conditional Rewriting Systems. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming*, pages 168–182. MIT Press, 1997.

A Well-Moded Programs

For the sake of self containedness, we now report the definition of well-moded program and of simply-moded program.

- A clause $\mathtt{p}_0(\overline{\mathtt{s}}_0, \overline{\mathtt{t}}_{\mathtt{n+1}}) \leftarrow \mathtt{p}_1(\overline{\mathtt{t}}_1, \overline{\mathtt{s}}_1), \ldots, \mathtt{p}_\mathtt{n}(\overline{\mathtt{t}}_\mathtt{n}, \overline{\mathtt{s}}_\mathtt{n})$ is called *well-moded* if for $i \in [1, n+1]$, $\mathit{Var}(\overline{\mathtt{t}}_\mathtt{i}) \subseteq \bigcup_{\mathtt{j}=0}^{\mathtt{i}-1} \mathit{Var}(\overline{\mathtt{s}}_\mathtt{j})$. A query $\overline{\mathtt{A}}$ is called *well-moded* iff the clause $\mathtt{q} \leftarrow \overline{\mathtt{A}}$ is, where $\mathtt{q}$ is any (dummy) atom of zero arity.
- A clause $\mathtt{p}_0(\overline{\mathtt{t}}_0, \overline{\mathtt{s}}_{\mathtt{n+1}}) \leftarrow \mathtt{p}_1(\overline{\mathtt{t}}_1, \overline{\mathtt{s}}_1), \ldots, \mathtt{p}_\mathtt{n}(\overline{\mathtt{t}}_\mathtt{n}, \overline{\mathtt{s}}_\mathtt{n})$ is called *simply moded* if if $\overline{\mathtt{s}}_1, \ldots, \overline{\mathtt{s}}_\mathtt{n}$ is a linear family of variables and for $i \in [1, n]$, $\mathit{Var}(\overline{\mathtt{s}}_\mathtt{i}) \cap (\bigcup_{\mathtt{j}=0}^{\mathtt{i}} \mathit{Var}(\overline{\mathtt{t}}_\mathtt{j})) = \emptyset$. A query $\overline{\mathtt{A}}$ is called *simply moded* iff the clause $\mathtt{q} \leftarrow \overline{\mathtt{A}}$ is, where $\mathtt{q}$ is any (dummy) atom of zero arity.

Both notions are persisent under PROLOG's resolution: An LD-resolvent of a well-moded goal and a well-moded clause that is variable-disjoint with it, is well-moded. The same applies to simply-moded goals and clauses. By known results, in presence of well-moded programs and queries and of the leftmost selection rule, in every selected atom in every possible derivation, the input position are filled by ground terms. This shows that well-moded programs have a straightforward left-to-right data-flow.