Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 177 (2007) 59–73
www.elsevier.com/locate/entcs

Reporting Failures in Functional Logic Programs[1]

Michael Hanus[2]

Institut für Informatik
Christian-Albrechts-Universität Kiel
D-24098 Kiel, Germany

Abstract

Computing with failures is a typical programming technique in functional logic programs. However, there are also situations where a program should not fail (e.g., in a deterministic top-level computation) but the evaluation fails accidentally, e.g., due to missing pattern combinations in an operation defined by pattern matching. In this case, the programmer is interested in the context of the failed program point in order to analyze the reason of the failure. Therefore, this paper discusses techniques for reporting failures and proposes a new one that has been integrated in a Prolog-based compiler for the declarative multi-paradigm language Curry. Our new technique supports separate compilation of modules, i.e., the compilation of modules has not taken into account whether failures should be reported or not. The failure reporting is only considered in some linking code for modules. In contrast to previous approaches, the execution of programs in the failure reporting mode causes only a small overhead so that it can be also used in larger applications.

Keywords: Functional logic programming, debugging, implementation

1 Motivation

Functional logic languages (see [17] for a survey) integrate the most important features of functional and logic languages to provide a variety of programming concepts to the programmer. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming are combined with logic programming features like computing with partial information (logic variables), constraint solving, and nondeterministic search for solutions. This combination, supported by optimal evaluation strategies [3] and new design patterns [5], leads to better abstractions in application programs such as implementing graphical user interfaces [20] or programming dynamic web pages [21,23].

[1] This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-1.
[2] Email: mh@informatik.uni-kiel.de

doi:10.1016/j.entcs.2007.01.002

Since functional logic languages, like Curry [19,28] or Toy [31], support both functional and logic programming styles, functional logic programs often contain parts that are evaluated in a purely deterministic manner and other parts where search for solutions is involved. These parts are sometimes combined by encapsulating search [10,27] so that the search results are processed by a deterministic part of the program. Thus, computing with failure is a typical programming technique in nondeterministic computations whereas a failure in the deterministic parts is a programming error in most cases. In the latter situation, the programmer is interested to see the failed function call as well as the context of the failure in order to analyze its reason.

For instance, consider the Curry program defining functions to compute the head, tail and the length of a list, and to select an element in a list at a particular position (`nth`) or at the end (`last`):

```
head (x:_) = x

tail (_:xs) = xs

length []     = 0
length (_:xs) = 1 + length xs

nth xs n = if n==0 then head xs
                   else nth (tail xs) (n-1)

last xs = nth xs (length xs)
```

If we evaluate the expression "`last [1,2]`" w.r.t. these definitions, we do not get a value: for instance, the PAKCS programming environment for Curry [24] prints "No more solutions" instead of a value. Since this is a purely deterministic computation without any logic variables or nondeterministic operations involved, the result is not intended so that some function call accidentally fails. In a deterministic computation, this must be the last function call in the sequence of reductions. In this example, it is the function call "`head []`". Although this is the only failing function call in the evaluation of the initial expression, the partial definition of the function `head` is obviously not the reason of this failure. Therefore, one is interested to see the context of this failure in order to understand why `head` is applied to the empty list. One possibility is to show the sequence of unevaluated calls from the initial expression to the failed function call. This corresponds to the call stack which is shown by the tool presented in this paper as follows:

```
5: last [1,2]
4: nth [1,2] (length [1,2])
3: nth (tail [1,2]) (2-1)
2: nth (tail (tail [1,2])) (1-1)
1: head (tail (tail [1,2]))
head: failed for argument: []
```

Now one can see that the reason of this failure is not the definition of `head` but one superfluous application of `tail` which can be avoided by the following correct

definition of `last`:

```
last xs = nth xs (length xs - 1)
```

Note that the call stack contains only those function calls from the initial expression to the failed call that are not yet evaluated at the moment when the failure occurs. In particular, intermediate calls, like "`if length [1,2]==0 then...else...`" or "`length [1,2]==0`" are not shown since they have already been evaluated.

Although one can discuss whether the visualization of such a call sequence or another representation of the context of the failure is better to spot the reason of the failure, it is clear that information about the failed call and its context is quite useful for debugging.

Unfortunately, most publications related to the implementation of functional logic languages concentrate on the efficient implementation of successful implementations rather than reporting failures (see, for instance, the survey in [17] or more recent works like [4,6,7,12,26,31,32]). Approaches to report failures can be found in the context of debugging declarative languages. For instance, Gill [16] proposed the debugger Hood for Haskell that is based on the idea that the user annotates expressions in the source program where the evaluation of those expressions is shown after the entire execution of a program. This idea has been extended to Curry with the COOSy tool [8]. Since these observation debuggers require the annotation of "relevant" expressions by the programmer before the program's execution, it is not very helpful in order to spot failures at some unknown position in the program.

Alternative approaches are tracers that record complete information about the evaluation steps during the execution and present them to the programmer with browsing facilities after the execution (e.g., see [15] for Haskell or [11] for Curry). Although these tools could be quite useful, since they present the evaluation in a different and better comprehensible order, they have also a disadvantage from a practical point of view. Since the complete trace, containing all reduction steps, variable bindings etc., must be stored, the amount of data to be stored can be huge so that these tracers have problems to deal with larger applications. Although there are approaches to improve their efficiency (e.g., [9]), the current implementations are not mature enough to be applied for larger applications. Therefore, we propose in this paper a simpler approach that can be implemented in a much more efficient way but still provides useful information to the programmer. Moreover, we show the integration of this approach in a Prolog-based compiler for Curry that supports a separate compilation of modules: no specific compilation is required when modules are executed such that failures are reported (e.g., in contrast to [9,11,15]). The latter property is important for the usability of this debugging method.

In the next section, we review the general structure of Curry programs in order to understand the subsequent development. A standard scheme to compile Curry programs into Prolog programs is reviewed in Section 3. Section 4 discusses existing approaches to report failures and proposes our new approach that enables a more efficient implementation. Finally, Section 5 contains our conclusions.

2 Curry Programs

We review in this section some aspects of Curry programs that are necessary to understand the contents of this paper. More details about Curry's computation model and a complete description of all language features can be found in [1,19,28].

Curry is a declarative multi-paradigm language combining in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program extended by the possible inclusion of free (logic) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [33], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of a function f to an argument e is denoted by juxtaposition ("$f\ e$").

A *Curry program* consists of the definition of functions and the data types on which the functions operate. Functions are defined by conditional equations with constraints in the conditions. They are evaluated lazily and can be called with partially instantiated arguments. Function calls with free variables are evaluated by a possibly nondeterministic instantiation of demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule (this evaluation mechanism is often called "*narrowing*"). In order to support concurrent programming (in a style that is also known as "*residuation*"), there is a primitive to define general "suspension" combinators for concurrent programming: the predefined operation `ensureNotFree` returns its argument evaluated to head normal form but suspends as long as the result is a free variable.

Example 2.1 The following program defines the types of Boolean values and polymorphic lists and functions to concatenate lists and to compute the last element of a list in a logic programming style:

```
data Bool   = True | False
data List a = []   | a : List a

conc :: [a] -> [a] -> [a]
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last :: [a] -> a
last xs | conc ys [x] =:= xs  = x  where x,ys free
```

The `data` type declarations define `True` and `False` as the Boolean constants and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell). The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output

list, where all list elements are of the same (unspecified) type. [3]

In general, functions are defined by (*conditional*) *rules* of the form

$$f\,t_1 \ldots t_n \;\texttt{|}\, c \;\texttt{=}\; e \quad \texttt{where}\; vs \;\texttt{free}$$

with f being a function, $t_1, \ldots, t_n$ *patterns* (i.e., expressions without defined functions) without multiple occurrences of a variable, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, lambda abstractions etc, and vs is the list of *free variables* that occur in c and e but not in $t_1, \ldots, t_n$. [4] The condition and the `where` parts can be omitted if c and vs are empty, respectively. The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable.

A *constraint* is any expression of the built-in type `Success`. For instance, the trivial constraint `success` is an expression of type `Success` that denotes the always satisfiable constraint. "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2, i.e., this expression is evaluated by proving both argument constraints concurrently. Each Curry system provides at least *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable patterns. However, specific Curry systems also support more powerful constraint structures, like arithmetic constraints on real numbers, Boolean constraints, finite domain constraints, or constraint handling rules [22], as in the PAKCS implementation [25].

The operational semantics of Curry [1,19] is based on an optimal evaluation strategy [3] for functional logic evaluations. It is a conservative extension of lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Due to its demand-driven behavior, it provides optimal evaluation (e.g., shortest derivation sequences, minimal solution sets) on well-defined classes of programs (see [3] for details). Curry also offers the standard features of functional languages, like higher-order functions or monadic I/O [35].

3 Compilation into Prolog

In this section we discuss standard high-level implementation techniques of functional logic languages by compilation into Prolog. This is the basis of our proposal to integrate failure reporting presented in the subsequent section.

The main extensions of functional logic languages compared to purely functional languages are the coverage of logic variables and nondeterministic search. Since these features are directly supported in Prolog [34], it is a natural idea to translate

[3] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[4] The explicit declaration of free variables is sometimes redundant (it is not redundant in case of nested scopes introduced by lambda abstractions or local definitions) but still useful to provide some consistency checks by the compiler.

functional logic programs into Prolog programs in order to exploit the implementation technology available for Prolog. Actually, there are various approaches to implement functional logic languages with demand-driven evaluation strategies in Prolog (e.g., [2,4,14,18,29,30]). Since modern functional logic languages are based on the non-strict lazy evaluation of functions [1], the main challenge of Prolog-based implementations are efficient techniques to obtain this behavior. Since the computation to a *head normal form* (i.e., a constructor-rooted term or a variable) is the central task of lazy evaluation, a common idea of such implementations is the translation of source operations into predicates that compute only the head normal form of a call to this operation. Thus, an n-ary operation could be translated into a predicate with $n+1$ arguments where the last argument contains the head normal form of the evaluated call. For instance, the list concatenation `conc` defined in Example 2.1 and the function `head` defined in Section 1 can be translated into the following Prolog predicates:

```
conc(Xs,Ys,H) :- hnf(Xs,HXs), conc_1(HXs,Ys,H).

conc_1([],Ys,H) :- hnf(Ys,H).
conc_1([X|Xs],Ys,[X|conc(Xs,Ys)]).

head(Xs,H) :- hnf(Xs,HXs), head_1(HXs,H).
head_1([X|Xs],H) :- hnf(X,H).
```

Since `conc` is defined by a case distinction on the first argument, the value is needed and, hence, computed by the predicate `hnf` before it is passed to `conc_1` implementing the pattern matching on the first argument. Since the right-hand side of the second rule of `conc` is already in head normal form, no further evaluation is necessary. In the first rule of `conc_1`, it is unknown at compile time whether the second argument `Ys` is already in head normal form. Therefore, the evaluation to head normal form is enforced by the predicate `hnf`. The goal `hnf(`t`,`h`)` evaluates any term t to its head normal form h. Some of the clauses defining `hnf` are:

```
hnf(V,V) :- var(V), !.
hnf([],[]).
hnf([X|Xs],[X|Xs]).
...
hnf(conc(Xs,Ys),H) :- conc(Xs,Ys,H).
hnf(head(Xs),H)    :- head(Xs,H).
...
```

Variables and constructor-rooted terms are already in head normal form (first three clauses). For each call to a defined function, there is a clause that calls the corresponding predicate implementing the evaluation of this function. Using this scheme, there is a straightforward transformation of Curry programs into Prolog. Furthermore, the predefined equational constraint "`=:=`" can be implemented by a predicate `constrEq` which computes the head normal form of its arguments and performs a variable binding if one of the arguments is a variable (following the scheme presented in [30]):

```
constrEq(A,B,H) :- hnf(A,HA), hnf(B,HB), constrEqHnf(HA,HB,H).

constrEqHnf(A,B,H) :- var(A), !, bind(A,B,H).
constrEqHnf(A,B,H) :- var(B), !, bind(B,A,H).
constrEqHnf(A,B,success) :- number(A), !, A=B.
constrEqHnf(c(X1,...,Xn),c(Y1,...,Yn),H) :- !,
   hnf((X1=:=Y1)&...&(Xn=:=Yn),H).  % ∀n-ary constructors c

bind(X,Y,success) :- var(Y), !, X=Y.
bind(X,Y,success) :- number(Y), !, X=Y.
bind(X,c(Y1,...,Yn),H) :- !,  % ∀n-ary constructors c
   occursNot(X,Y1),..., occursNot(X,Yn), X=c(X1,...,Xn),
   hnf(Y1,HY1), bind(X1,HY1,H),
   ...
   hnf(Yn,HYn), bind(Xn,HYn,H).
```

Due to the lazy semantics of the language, the binding is performed incrementally. We use an auxiliary predicate, `bind`, which performs an occur check (implemented by `occursNot`) followed by an incremental binding of the goal variable and the binding of the arguments.

Note that the scheme presented so far does not implement sharing (where it is required that each function call should be evaluated at most once) or residuation (i.e., the suspension of operations where arguments are required to be bound to a non-variable value). Both features can be covered by extending this scheme: sharing can be supported by introducing "share structures" for arguments with multiple occurrences in the right-hand side of rules, and residuation can be supported by additional arguments in each predicate to control the potential suspension of function calls and exploiting coroutining facilities of Prolog implementations (see [4] for details). Since these extensions complicates the presentation and are independent of the design of our approach to report failures, we omit the implementation of sharing and residuation in the following.

4 Prolog-based Failure Reporting

Before we present our new proposal to report failures, we discuss existing approaches with a similar objective.

4.1 Report Failures by Backtracking

A failure occurring in a Curry program compiled into Prolog with the approach sketched in the previous section causes a failure of the corresponding Prolog program. Thus, standard Prolog implementations just report "no" which is clearly not very helpful to locate a bug. Therefore, Prolog implementations usually support source-level tracing of the Prolog program under execution following Byrd's box model [13]. However, tracing the compiled Curry program on the level of Prolog is also not helpful since basic reduction steps are implemented by a sequence of predicate calls and the Curry programmer should not know the compilation model

in order to find a bug in his program. Thus, it is better to hide the implementation level of Prolog to the programmer but add features that report the failures on the level of the source language Curry.

It is a well known technique in logic programming to enhance meta-interpreters with features for tracing or debugging [34]. Similar techniques can also be used when compiling other languages into Prolog. For instance, to report a failure in a function call, one can modify the generation of clauses for the predicate `hnf` so that each evaluation of the corresponding predicate has an alternative goal that reports the failure:

```
...
hnf(conc(Xs,Ys),H) :- conc(Xs,Ys,H) ; failprint(cons(Xs,Ys)).
hnf(head(Xs),H)    :- head(Xs,H)    ; failprint(head(Xs)).
...
```

For instance, in case of a call to `head`, the goal `failprint(head(Xs))` is executed only if the evaluation of the goal `head(Xs,H)` failed. In this case, `failprint` just prints its argument (in the Curry syntax format) and also fails:

```
failprint(Exp) :-
        write('Failure due to irreducible expression: '),
        writeCurry(Exp), nl,
        !, fail.
```

Since failures are printed for each failed function call, this implementation reports all failures up to the main expression. Due to its striking simplicity, this technique has been integrated for a long time in the PAKCS programming environment [24].

Although this approach is easy to implement and reports the complete context of a failure, it has also a serious drawback that makes it impractical for larger programs. For each function call, a choice point is created in order to report the potential failure by `failprint`. Thus, as long as the computation proceeds without a failure, a huge number of choice points is created without ever discarding them. Since the creation of choice points is one of the most expensive operation in Prolog implementations and requires a considerable amount of memory [36], larger computations are often terminated due to insufficient memory before reaching the failure point which we want to analyze. This demands for another implementation technique that causes execution costs only in the case of a failed computation. Our solution to this problem will be described next.

4.2 Report Failures without Backtracking

The main idea of our approach is to treat a failing computation not as a failed computation in the Prolog program but as a computation that returns a specific value containing some information about the source of the failure. For this purpose, we assume a predefined function `failure` that wraps its argument into the distinguished constructor `FAIL` that is not accessible to standard Curry programs. Although this function is predefined (since it is not typable w.r.t. the standard type

system of Curry), for the moment we assume the following definition of `failure`:

```
failure x = FAIL [x]
```

Thus, `failure` puts its argument into a list which will later be stepwise extended to the list of all failed function calls from the main expression to the innermost failed call. The function `failure` is explicitly used whenever some function call might fail due to missing pattern combinations. [5] For instance, the operation `head` is not defined on empty lists. Therefore, we complete the definition of `head` with a call to `failure` in case of an empty list as argument so that we obtain the following extended definition of the predicate `head_1`:

```
head_1([X|Xs],H) :- hnf(X,H).
head_1([],H)     :- hnf(failure(head([])),H).
```

Note that all predicates implementing pattern matching in source programs can be automatically completed in this way due to the typed nature of the source language Curry. [6] However, this code contains a slight problem. Since Curry is a functional *logic* language, `head` can be also called with a logic variable as argument. In the new implementation, the logic variable can be bound to `[X|Xs]` as well as to `[]`. In the latter case, `failure` is called to report a failure although this was not present in the original program.

In order to avoid such unintended instantiations of logic variables, we introduce in front of the new `failure` clauses a single clause which tries to bind the argument to a new constructor (here: `varcut`) that does not occur in regular computations. If this binding is successful, we know that the argument must be a free variable so that we can safely ignore the remaining clauses by a cut/fail combination:

```
head_1([X|Xs],H) :- hnf(X,H).
head_1(varcut,H) :- !, fail. % ignore further clauses
head_1([],H)     :- hnf(failure(head([])),H).
```

Note that we could have also implemented the second `varcut` clause by a call to the Prolog meta-predicate `var` in order to check the freeness of the argument:

```
head_1(X,H) :- var(X), !, fail. % ignore further clauses
```

However, this would destroy the standard Prolog indexing scheme on the first argument and creates additional choice points. Our proposed translation scheme has (almost) no influence on the execution time but only extends the program size a little bit (around 10% in our larger examples).

As mentioned above, `FAIL` is a new constructor to pass the information about failing computations. Therefore, it is a new value that must be considered in all pattern matchings, i.e., if the actual argument is a `FAIL` value, it is directly returned to the caller. Thus, we obtain the following final code that implements the pattern matching of `head`:

[5] `failure` is also used whenever a call to the equational constraint "`=:=`" fails, see Section 4.3.

[6] The completion of functions with many missing patterns could be optimized if the implementation supports some sort of "default cases." In our case, the definition of predicates with a complete set of patterns is more efficient due to Prolog's specific support for argument indexing.

```
head_1([X|Xs],H) :- hnf(X,H).
head_1(varcut,H) :- !, fail. % ignore further clauses
head_1([],H)     :- hnf(failure(head([])),H).
head_1('FAIL'(A),'FAIL'(A)).
```

Note that all pattern matching predicates must be extended by a `FAIL` clause. However, the `varcut` clause needs only be inserted in case of partially defined functions. Our extended translation of pattern matching for failure reporting causes only a slight increase in the code size but has no negative influence on the execution of these predicates. Thus, we can compile all Curry modules in this extended way independent of the fact whether we want to report failures or not. This property is important to support separate compilation, e.g., usually system libraries cannot be recompiled by individual users of an installed Curry system. All the logic about the treatment of `FAIL` values is contained in the implementation of the predefined operation `failure` and `hnf` clauses which we discuss next.

Translated Curry programs can be executed in a standard mode or a "failure reporting" mode. The mode can be selected in the PAKCS environment which stores it in the predicate `reportFailure`. The Prolog implementation of the primitive `failure` is as follows:

```
failure(_,_) :- reportFailure(no), !, fail. % no reporting required
failure(FailedCall,'FAIL'([FailedCall]). % return FAIL value
```

Hence, if the user do not want to see the failed calls, `failure` just fails (first clause) so that the behavior is identical to the standard execution. Otherwise, the `FAIL` value containing the failed call is returned (second clause).

In order to extend `FAIL` values with outermost function calls up to the main expression, we modify the definition of the predicate `hnf`. Note that, due to separate compilation implemented in PAKCS, the definition of the predicate `hnf` is generated for each program loaded into PAKCS: since `hnf` transfers the evaluation of each function call to the predicate implementing this function, it can be considered as the "linking code" that glues the separately compiled modules. Therefore, generating a new definition of `hnf` is a minor task compared to the compilation of a module (usually performed in a few milliseconds) so that this is usually not recognized by the PAKCS user when he switches to the failure reporting mode.

The definition of `hnf` for failure reporting adds a goal to check for `FAIL` values after each predicate call so that it has the following structure:

```
hnf(V,V) :- var(V), !.
hnf([],[]).
hnf([X|Xs],[X|Xs]).
...
hnf(conc(Xs,Ys),H) :- conc(Xs,Ys,HF),
                      checkFailValue(conc(Xs,Ys),HF,H).
hnf(head(Xs),H)    :- head(Xs,HF), checkFailValue(head(Xs),HF,H).
...
```

Note that this change causes only a small overhead due to the call to

`checkFailValue` but does not introduce new choice points. Hence, this scheme is compatible with the execution of large applications. The predicate `checkFailValue` checks whether its second argument is a `FAIL` value. If this is the case, it extends the argument by the current function call which is passed as the first argument, otherwise it just returns the second argument:

```
checkFailValue(Call,Value,Result) :-
        (nonvar(Value), Value='FAIL'(FailStack))
         -> Result='FAIL'([Call|FailStack])
          ; Result=Value.
```

Due to this implementation scheme, a failed computation returns the complete call stack from the outermost main function call to the innermost failed call in a list structure. This list structure can be processed in the PAKCS environment in different ways according to the current settings:

- Show only the innermost failed function call or the list of all failed calls (as shown in Section 1).
- Enter an interactive mode for failure tracing. This mode is useful if the complete trace is too large to show it on a screen. In this mode the programmer can explore different regions of the complete trace, show calls with arguments up to some depth (useful for large argument terms) etc.
- Write the list of all failed calls into some file. This is useful to explore failures occurring in non-interactive applications like dynamic web pages executed by an HTTP server [21,23].

Our implementation scheme causes only a small overhead in case of non-failing computations and has a behavior substantially different from the standard execution only if a failure occurs. Although the returned structure can be large if considered as a term, it fits well into main memory even for larger applications since most parts of the structure (in particular, the arguments of the function calls) are already created in the heap when the failure occurs. Thus, our implementation is a viable and more generally applicable alternative than failure reporting based on backtracking (Section 4.2) or tracing the complete execution [9,11,15].

4.3 Failures in Equational Constraints

In functional logic languages, failures cannot only occur in user-defined operations but also in equational constraints due to non-unifiable terms. For instance, the constraints "`True=:=False`" and "`x=:=1:x`" are not solvable (the former due to incompatible constructors and the latter due to the occur check) and, thus, fail. Since equational constraints are predefined with a specific implementation (see Section 3), one cannot use the implementation scheme to report failures in user-defined functions as presented in the previous section. Thus, in order to report failures in equational constraints, we extend the definition of `constrEqHnf` and `bind` in the implementation scheme for "`=:=`" presented in Section 3 as follows:

```
constrEqHnf(A,B,H) :- var(A), !, bind(A,B,H).
```

```
constrEqHnf(A,B,H) :- var(B), !, bind(B,A,H).
constrEqHnf('FAIL'(A),B,'FAIL'(A)) :- !.
constrEqHnf(A,'FAIL'(B),'FAIL'(B)) :- !.
constrEqHnf(A,B,H) :- number(A), !,
                      (A=B -> H=success ; hnf(failure(A=:=B),H)).
constrEqHnf(c(X1,...,Xn),c(Y1,...,Yn),H) :- !,
   hnf((X1=:=Y1)&...&(Xn=:=Yn),H).  % ∀n-ary constructors c
constrEqHnf(A,B,H) :- hnf(failure(A=:=B),H).

bind(X,Y,success) :- var(Y), !, X=Y.
bind(X,Y,success) :- number(Y), !, X=Y.
bind(X,'FAIL'(Y),'FAIL'(Y)) :- !.
bind(X,c(Y1,...,Yn),H) :-   % ∀n-ary constructors c
   occursNot(X,Y1),..., occursNot(X,Yn), !, X=c(X1,...,Xn),
   hnf(Y1,HY1), bind(X1,HY1,H1),
   ((nonvar(H1), H1='FAIL'(S)) -> H=H1 ;
     ...
      hnf(Yn,HYn), bind(Xn,HYn,H)...).
bind(X,Y,H) :- hnf(failure(X=:=Y),H).
```

The additional third and fourth clause of `constrEqHnf` and the additional third clause of `bind` passes a `FAIL` value of an argument evaluation. The new final clauses of `constrEqHnf` and `bind` report failures due to incompatible constructors and occur check, respectively. Similarly, the `constrEqHnf` clause for numbers must be slightly modified to report failures in case of incompatible numbers.

4.4 Failures in Encapsulated Search

As mentioned at the beginning, computing with failures is a typical programming technique in functional logic programs. However, in practical applications one has to restrict the search for solutions in order to avoid a nondeterministic behavior of the entire program in I/O operations (such a behavior is considered as a run-time error in Curry). Thus, the programmer usually encapsulate nondeterministic search by specific search operators that return the solutions to some constraint [10,27]. Since reporting failures is usually not intended in these parts of the programs, failure reporting is disabled during encapsulated search (which is controlled by a simple flag in the run-time system), i.e., the predicate `failure` always fails inside an encapsulated search. If the programmer is still interested to see failures in these parts of the program, he can just execute these parts at the top-level of PAKCS without the search operators.

5 Conclusions

We have presented a new scheme to compile functional logic programs into Prolog that supports the report of failed computations. The scheme is based on the idea to represent failed computations by a specific failure value containing the sequence of

failed function calls at the end of a failed computation. For this purpose, the missing patterns of each partially defined function are completed with calls to a distinguished failure function that returns a failure value. Furthermore, all functions need to be extended in order to pass failure values as arguments. We have designed this scheme such that the additional clauses for the predicates implementing each function do not cause an execution overhead for standard executions. Thus, the new scheme can be used to compile Curry programs. A specific compilation to report failures is only necessary for the definition of the global predicates to compute the head normal form of any expression and to prove equational constraints. These predicates establish the linking code between the different modules so that their definition is generated before loading each program. Altogether, the proposed scheme can be efficiently implemented and does not cause a substantial execution overhead in contrast to other approaches based on backtracking or storing the complete execution trace. Thus, our scheme can also be used to get information about the context of a failure in larger application programs.

Although we have implemented our approach in a Prolog-based implementation of Curry and carefully designed it in order to exploit the efficiency of current Prolog implementations, it can be also applied to implementations of functional logic languages based on other target languages than Prolog. However, our implementation technique might not be relevant for lower level languages where one has direct access to the call stack and other run-time structures.

For future work, it might be interesting to explore whether it is possible to generate more structural information in case of errors. Since the structure of the call stack is oriented towards the lazy evaluation of expressions, the order of calls might not be the best one for presentation to the programmer. Further practical experience is necessary to develop appropriate presentation structures.

The implementation described in this paper is freely available with the latest distribution of PAKCS [24].

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 795–829, 2005.

[2] S. Antoy. Non-Determinism and Lazy Evaluation in Logic Programming. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'91)*, pp. 318–331. Springer Workshops in Computing, 1991.

[3] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.

[4] S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pp. 171–185. Springer LNCS 1794, 2000.

[5] S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.

[6] S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A Virtual Machine for Functional Logic Computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pp. 108–125. Springer LNCS 3474, 2005.

[7] S. Antoy, M. Hanus, B. Massey, and F. Steiner. An Implementation of Narrowing Strategies. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 207–217. ACM Press, 2001.

[8] B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing Functional Logic Computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pp. 193–208. Springer LNCS 3057, 2004.

[9] B. Brassel, S. Fischer, and F. Huch. A Program Transformation for Tracing Functional Logic Computations. In *Pre-Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pp. 141–157. Technical Report CS-2006-5, Università ca' Foscari di Venezia, 2006.

[10] B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming*, Vol. 2004, No. 6, 2004.

[11] B. Braßel, M. Hanus, F. Huch, and G. Vidal. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04)*, pp. 179–190. ACM Press, 2004.

[12] B. Braßel and F. Huch. Translating Curry to Haskell. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 60–65. ACM Press, 2005.

[13] L. Byrd. Understanding the Control Flow of Prolog Programs. In *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.

[14] P.H. Cheong and L. Fribourg. Implementation of Narrowing: The Prolog-Based Approach. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic programming languages: constraints, functions, and objects*, pp. 1–20. MIT Press, 1993.

[15] O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood – A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 176–193. Springer LNCS 2011, 2001.

[16] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electr. Notes Theor. Comput. Sci.*, Vol. 41, No. 1, 2000.

[17] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

[18] M. Hanus. Efficient Translation of Lazy Functional Logic Programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pp. 252–266. Springer LNCS 1048, 1995.

[19] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.

[20] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.

[21] M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.

[22] M. Hanus. Adding Constraint Handling Rules to Curry. In *Proc. 20th Workshop on Logic Programming (WLP 2006)*, pp. 81–90. INFSYS Research Report 1843-06-02 (TU Wien), 2006.

[23] M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pp. 27–38. ACM Press, 2006.

[24] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2006.

[25] M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2005.

[26] M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, Vol. 1999, No. 6, 1999.

[27] M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.

[28] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.

[29] J.A. Jiménez-Martin, J. Marino-Carballo, and J.J. Moreno-Navarro. Efficient Compilation of Lazy Narrowing into Prolog. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'92)*, pp. 253–270. Springer Workshops in Computing Series, 1992.

[30] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 184–200. Springer LNCS 714, 1993.

[31] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.

[32] W. Lux. Implementing Encapsulated Search for a Lazy Functional Logic Language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 100–113. Springer LNCS 1722, 1999.

[33] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.

[34] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 2nd edition, 1994.

[35] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

[36] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.