Transforming Delimited Control: Achieving Faster Effect Handlers

AMR HANY SALEH
KU Leuven
Department of Computer Science
Celestijnenlaan 200a
3001 Leuven
Belgium
(*e-mail:* ah.saleh@cs.kuleuven.be)

submitted 29 April 2015; accepted 5 June 2015

Abstract

Algebraic effect handlers are a great way for modularizing side effects in Prolog, but they suffer from poor performance due to nested use of delimited control.

Our aim is to propose a systematic program transformation that merges a composition of multiple modular handlers into a single monolithic one. Our transformation combines definition folding/unfolding with rewrite rules that exploit the semantics of delimited control to eliminate their runtime overhead.

This approach enables the programmer to write programs in a modular fashion and at the same time to benefit from the good performance of monolithic code. Our experimental evaluation indicates that merged handlers are twice as fast on average.

1 Introduction

In recent work, Schrijvers et al. (2013) have introduced support for *delimited control* (Felleisen 1988; Danvy and Filinski 1990) in Prolog. Delimited control enables the definition of new high-level language features at the program level (e.g., in libraries) rather than at the meta-level as program transformations. As a consequence, feature extensions based on delimited control are more robust with respect to changes and do not require pervasive changes to existing code bases.

Algebraic effect handlers (Plotkin and Pretnar 2009) are a particularly attractive application of delimited control. They are an elegant way to add many kinds of side-effectful operations (eg. mutable states, reading and writing to files, ...) to a language (far less intrusive than monads (Moggi 1991)) in a compositional fashion. Schrijvers et al. give various examples in Prolog, including handlers for implicit state, DCGs and co-routines.

While the compositionality of effect handlers is one of its main attractions, this modularity comes at the cost of considerably reduced runtime performance. Our experiments in Prolog show programs that are up to 2× or 3× slower due to handler composition. Hence, the efficient implementation of modular effect handlers is very much an active topic of research.

```
get(S):- shift(get(S)).
put(S):- shift(put(S)).

run_state(G,Sin,Sout) :-
  reset(G,Cont,Command),
  ( Cont = 0 ->
        Sin = Sout
  ; Command = get(S) ->
        S = Sin,
        run_state(Cont,Sin,Sout)
  ; Command = put(S) ->
        run_state(Cont,S,Sout)).
```

```
c(X) :- shift(c(X)).

phrase(G,Lin,Lout) :-
   reset(G,Cont,Command),
   ( Cont = 0 ->
          Lin = Lout
   ; Command = c(X) ->
          Lin = [X|NL],
          phrase(Cont,NL,Lout)).
```

Fig. 1. *State* and DCG handlers

2 Background

2.1 Delimited Control

Prolog extended with delimited continuations provides two predicates for delimited control:

- `reset(G,Cont,T)` executes goal `G` untill a `shift/1` call occurs inside `G`.
- `shift(T1)` suspends the execution of the current goal and captures the remainder up to the nearest surrounding `reset/3`. This remainder is called the *continuation*. It unifies the captured continuation with `Cont` and `T` with `T1`. The control is then returned to the call just after the `reset/3`.

The following example shows delimited control in action.

```
main :- reset(p,Cont,Term),      p :- write('a '),         ?- main.
        write('b ').                  write('c ').         a c b
```

Because `p` terminates without shifting, the variables `Cont` and `Term` are unified with `0`. The next example illustrates the interaction between `shift/1` and `reset/3`.

```
main :- reset(p,Cont,Term),      p :- write('a '),         ?- main.
        write(Term),                  shift('hi '),        a hi b c
        write('b '),                  write('c ').
        call(Cont).
```

Executing `?-main.` calls `p` inside the reset, prints `a`, then suspends the execution due to `shift('hi ')`, giving the control back to the `main` clause after the `reset/3` and unifying `Term` with `'hi '` and `Cont` with (`write('c ')`).

2.2 Effect Handlers

Effect handlers (Plotkin and Pretnar 2009) provide a high level interface to delimited control. They are an elegant way to add many kinds of side-effectful operations to a language.

Figure 1 shows two handlers, for mutable states and for parsing. The *State* handler on the left provides two operations: `get/1` for reading an implicit state and `put/1` for writing it. The *State* handler executes the goal `G` in the scope of the `reset/3` and afterwards discriminates between the different possible outcomes. If the goal finishes without shifting, then the handler unifies the input state `Sin` with the output state `Sout`. If the goal shifts the term `get(S)`, the handler unifies `S` with `Sin` and recurses over the continuation. If the term `put(S)` was shifted, it recurses over the continuation with `S` as the new state.

The predicate `inc/0` uses the two operations to increment the implicit state.

```
inc :- get(S), S1 is S + 1, put(S1).
```

The query `?- run_state( (inc,inc), 0, Sout)` uses the *State* handler to increment the state twice, unifying `Sout` with `2`.

The right part of Figure 1 defines a handler for Definite Clause Grammars (DCG).[1] This effect handler introduces one operation `c(E)` to consume the head `E` of the input list `Lin`. For instance, the `ab/0` predicate defines the $(ab)^*$ grammar.

```
ab.
ab :- c(a), c(b), ab.
```

The query `?- phrase(ab,[a,b,a,b],[]).` checks whether the string *abab* matches the grammar. We refer to Schrijvers et al. (2013) for more examples of effect handlers in Prolog.

2.2.1 Combining Effect Handlers

Effects become more interesting when they are combined. For example, `ab_inc` combines the *State* and DCG effects. It counts the number occurrences of *ab*.

```
ab_inc.
ab_inc:- c(a), c(b), inc, ab_inc.
```

How can we handle these combined effects? We see two possible ways:

Modular Handlers. We can handle multiple effects by composing *modularized* versions of the handlers. A modular handler is one that propagates unknown operations to the next handler in line. We modularize a handler by adding a default case that takes care of such propagation. For instance, in the *State* handler, we add the following disjunct.

```
; shift(Command),
  run_state(Cont,Sin,Sout)
```

This disjunct shifts unknown `Command`s upwards to the next handler and then handles the continuation recursively. We modify the DCG handler in the same fashion.

Now it is easy to combine both handlers: The query `?-run_state(phrase(ab_inc,[a,b,a,b,a,b],[]),0,Sout).` unifies `Sout` with `3`.

[1] DCGs are a well-known Prolog extension to sequentially access the elements of an implicit list.

```
state_phrase(G,Lin,Lout,Sin,Sout) :-
  reset(G,Cont,Command),
  (  Cont = 0 ->
     Sin = Sout,
     Lin = Lout
  ; Command = get(S) ->
     S = Sin,
     state_phrase(Cont,Lin,Lout,Sin,Sout)
  ; Command = put(S) ->
     state_phrase(Cont,Lin,Lout,S,Sout)
  ; Command = c(X) ->
     Lin = [X|NL],
     state_phrase(Cont,NL,Lout,Sin,Sout)).
```

Fig. 2. state_phrase handler

Monolithic Handlers. Another way of combining multiple effects is to write a single monolithic handler that handles all effects.

The *state_phrase* handler in Figure 2 tackles the *state* and *DCG* effects together using only one `reset/3`. The query `?-state_phrase(ab_inc,[a,b,a,b,a,b],[],0,Sout).` unifies `Sout` with 3.

3 Research Goal and Current Status

3.1 Research Goal

Both ways to handle multiple effects have their strengths and weaknesses.

Modular handlers nicely isolate separate effects in components that can be reused independently in arbitrary combinations. In contrast, monolithic handlers are highly inflexible; they only serve one combination of effects. However, monolithic handlers can be much more efficient. This is due to the overhead generated by going through many reset layers in the case of modular handlers, which is mostly eliminated in the case of monolithic handlers. Therefore, the main aim of this research is to have the flexibility of modular handlers without sacrificing efficiency.

3.2 Current Status

Currently, our approach consists of systematically deriving the monolithic definition of handlers from the modular ones. This way the programmer can write his programs in terms of the modular handlers, but the Prolog system can actually run the corresponding monolithic handler. Hence we get both modularity and efficiency.

Our main technique for the systematic derivation is the folding/unfolding framework of Pettorossi and Proietti (1994; 1999), a well-established static program transformation technique. We complement the basic folding/unfolding with a number of transformation rules that capture the semantics of delimited control and enable us to eliminate its runtime overhead.

The main job of basic folding and unfolding of predicate definitions is to expose the delimited control built-ins, but the actual job of simplifying their uses is performed by a number of additional tranformation rules.

3.2.1 Simplification of Delimited Control

$$\frac{\texttt{pure(G)}}{\texttt{reset(G,C,T)} \equiv \texttt{G, C=0, T=0}} \quad (\textsc{ResetPure})$$

$$\texttt{reset((shift(S),G),C,T)} \equiv \texttt{C=G, T=S} \quad (\textsc{ResetShift})$$

$$\frac{\texttt{pure}(G_1)}{\texttt{reset}((G_1,G_2),\texttt{Cont,Term}) \equiv G_1,\texttt{reset}(G_2,\texttt{Cont,Term})} \quad (\textsc{ResetConj})$$

$$\frac{\texttt{pure(C)}}{\begin{array}{c}\texttt{reset((C->}G_1\texttt{;}G_2\texttt{),Cont,Term)} \equiv \\ \texttt{(C->reset(}G_1\texttt{,Cont,Term);reset(}G_2\texttt{,Cont,Term))}\end{array}} \quad (\textsc{ResetCond})$$

$$\texttt{((C->}G_1\texttt{;}G_2\texttt{),}G_3\texttt{)} \equiv \texttt{(C->}G_1\texttt{,}G_3\texttt{;}G_2\texttt{,}G_3\texttt{)} \quad (\textsc{Distributivity})$$

Fig. 3. Simplification Rules of Delimited Control

Figure 3 formulates these as inference rules[2] of the judgement $G \equiv G'$. The interpretation of this judgement is that G and G' are denotationally equivalent. Even though the judgement is in principle symmetric, we have oriented the sides in all inference rules in such a fashion that the more complex form is in the left and the simpler on the right. This way the inference rules can be easily used as left-to-right rewrite rules. We also proved the correctness of the rules in operational semantics settings.

Four of the rules allow us to narrow the scope of a `reset/3` or make it disappear altogether. Rule (ResetPure) states that a `reset/3` around a *pure* goal `G` can be dropped. A pure goal is one that does not call `shift/1`. Formally, we can define this property in terms of the meta-interpreter as follows:

$$\texttt{pure(G)} \Leftrightarrow \nexists \texttt{T},\texttt{C} : \texttt{eval(G,shift(T,C))}$$

Examples of pure goals are unifications, calls to `reset/3` and user-defined predicates that are exclusively defined in terms of pure goals.

Rule (ResetShift) captures the interaction between reset and shift in the simple case where the continuation is a conjunct of the `shift/1` call.

Rule (ResetConj) expresses that a `reset/3` can be pushed into the second goal of a conjunction if the first goal is pure. Similarly, Rule (ResetCond) says that a `reset/3` can be be pushed into the branches of a conditional if the condition is pure.

Finally, the fifth rule is not strictly speaking related to delimited control; nevertheless, it is important for our transformation. This rule expresses the distributivity of conjunction with respect to conditionals.

[2] Inference rules provide a vertical layout for Horn clauses, with a consequence below the bar and optional antecedents above the bar. Variables are implicitly quantified like in Prolog.

3.3 Transformation Example

The aim of the transformation is to eliminate the nested use of `reset/3` and the delegation with `shift/1` of unknown commands from the first to the second handler. Because the handlers are recursive, we follow the usual transformation strategy for recursive predicates:

1. *Unfolding:* We unfold the nested handlers to expose opportunities for simplification.
2. *Local transformation:* We improve one level of the recursion using the transformation rules, constant propagation and more unfolding.
3. *Folding:* We massage the recursive calls into variants of the toplevel call to tie the knot and distribute the improvement over all levels of the recursion.

We now explain the transformation in detail in terms of our running example.

3.3.1 Unfolding

We start with the toplevel query that uses the modular handlers:

```
?- run_state( phrase( G, Lin, Lout), Sin, Sout).
```

Step 1. We abstract over the query with a new predicate `query/5`.[3]

```
query(G,Lin,Lout,Sin,Sout) :- run_state( phrase( G, Lin, Lout), Sin,  Sout) .
```

Then the original query can be rewritten as:

```
?- query(G,Lin,Lout,Sin,Sout).
```

Step 2. Now we unfold the *State* and DCG handlers in the `query/5` predicate to expose opportunities for fusing the handlers:

```
query(G,Lin,Lout,Sin,Sout) :-
   reset(reset(G,Contin,Commandin),
         ( Contin = 0 -> Lin = Lout
         ; Commandin = c(E) -> Lin = [E|Lmid], phrase(Contin,Lmid,Lout)
         ; shift(Commandin),phrase(Contin,Lin,Lout)
         )
         , Cont, Command),
   ( Cont = 0 -> Sin = Sout
   ; Command = get(S) -> S = Sin, run_state(Cont,Sin,Sout)
   ; Command = put(S) -> run_state(Cont,S,Sout)
   ; shift(Command), run_state(Cont,Sin,Sout)).
```

3.3.2 Local Transformation

Now we simplify the unfolded handler code. This comprises a series of steps that simplify the goal arguments of the `reset/3` calls.

[3] We highlight each time in gray the code that changes in the next step.

Step 3. Because `reset/3` is pure, Rule (ResetConj) can flatten the nested `reset/3`.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  reset(( Contin = 0 -> Lin = Lout
  ; Commandin = c(E) -> Lin = [E|Lmid], phrase(Contin,Lmid,Lout)
  ; shift(Commandin),phrase(Contin,Lin,Lout)), Cont, Command) ,
  ( Cont = 0 -> Sin = Sout
  ; Command = get(S) -> S = Sin, run_state(Cont,Sin,Sout)
  ; Command = put(S) -> run_state(Cont,S,Sout)
  ; shift(Command), run_state(Cont,Sin,Sout)).
```

Step 4. Rule (ResetCond) simplifies the second `reset/3` as the unification conditions are pure.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  ( Contin = 0 -> reset(( Lin = Lout) ,Cont,Command)
  ; Commandin=c(E)-> reset(( Lin=[E|Lmid], phrase(Contin,Lmid,Lout)),Cont,Command)
  ; reset((shift(Commandin),phrase(Contin,Lin,Lout)), Cont, Command)
  ),
  ( Cont =0 -> Sin = Sout ; Command = get(S) -> S = Sin, run_state(Cont,Sin,Sout)
  ; Command = put(S) -> run_state(Cont,S,Sout)
  ; shift(Command), run_state(Cont,Sin,Sout)).
```

Step 5. Now Rule (ResetPure) applies to the second `reset/3`.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  ( Contin = 0 -> Lin=Lout, Cont = 0 , Command = 0
  ; Commandin=c(E)-> reset(( Lin = [E|Lmid] ,phrase(Contin,Lmid,Lout)),Cont,Command)
  ; reset((shift(Commandin),phrase(Contin,Lin,Lout)), Cont, Command) ),
  ( Cont = 0 -> Sin = Sout
  ; Command = get(S) -> S = Sin, run_state(Cont,Sin,Sout)
  ; Command = put(S) -> run_state(Cont,S,Sout)
  ; shift(Command), run_state(Cont,Sin,Sout)).
```

Step 6. Rule (ResetConj) simplifies the second `reset/3`.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  ( Contin = 0 -> Lin=Lout, Cont = 0 , Command = 0
  ; Commandin=c(E)-> Lin = [E|Lmid], reset(phrase(Contin,Lmid,Lout),Cont,Command)
  ; reset((shift(Commandin),phrase(Contin,Lin,Lout)), Cont, Command)) ,
  ( Cont = 0 -> Sin = Sout
  ; Command = get(S) -> S = Sin, run_state(Cont,Sin,Sout)
  ; Command = put(S) -> run_state(Cont,S,Sout)
  ; shift(Command), run_state(Cont,Sin,Sout)).
```

Step 7. Rule (ResetShift) eliminates the last `reset/3` call.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  ( Contin = 0 -> Lin=Lout, Cont = 0 , Command = 0
  ; Commandin=c(E)-> Lin = [E|Lmid], reset(phrase(Contin,Lmid,Lout),Cont,Command)
  ; Cont = phrase(Contin,Lin,Lout), Commandin = Command ),
   ( Cont = 0 -> Sin = Sout
   ; Command = get(S) -> S = Sin, run_state(Cont,Sin,Sout)
   ; Command = put(S) -> run_state(Cont,S,Sout)
   ; shift(Command), run_state(Cont,Sin,Sout)) .
```

Step 8. We now use Rule (DISTRIBUTIVITY) to move the second conditional into the branches of the first one. For the sake of brevity, we refer to the second conditional as $\langle StateConditional \rangle$.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  (  Contin = 0 -> Lin=Lout, Cont = 0 , Command = 0,⟨StateConditional⟩)
  ; Commandin=c(E)->
     Lin = [E|Lmid],reset(phrase(Contin,Lmid,Lout),Cont,Command),⟨StateConditional⟩
  ;  Cont = phrase(Contin,Lin,Lout), Commandin = Command,⟨StateConditional⟩) .
```

Step 9. With constant propagation we propagate `Cont = 0` in the first branch. Then we simplify the $\langle StateConditional \rangle$ conditional with the statically known condition. In the same fashion, we simplify the last branch using constant propagation on `Cont = phrase(Contin,Lin,Lout)` and `Commandin = Command`.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  ( Contin = 0 -> Lin=Lout, Cont = 0 , Command = 0, Sin = Sout),
  ; Commandin=c(E)->
     Lin = [E|Lmid], reset(phrase(Contin,Lmid,Lout),Cont,Command),⟨StateConditional⟩
  ; Commandin = get(S) -> S = Sin, run_state(phrase(Contin,Lin,Lout),Sin,Sout)
  ; Commandin = put(S) -> run_state(phrase(Contin,Lin,Lout),S,Sout)
  ; shift(Commandin), run_state(phrase(Contin,Lin,Lout),Sin,Sout)
  ).
```

3.3.3 Folding Phase

Step 10. In the second branch, we can fold the *state* handler.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  ( Contin = 0         -> Lin = Lout, Sin = Sout),
  ; Commandin=c(E)   -> Lin = [E|Lmid], run_state(phrase(Contin,Lmid,Lout),Sin,Sout)
  ; Commandin =get(S) -> S = Sin,  run_state(phrase(Contin,Lin,Lout),Sin,Sout)
  ; Commandin =put(S) ->  run_state(phrase(Contin,Lin,Lout),S,Sout)
  ; shift(Commandin), run_state(phrase(Contin,Lin,Lout),Sin,Sout)
  ).
```

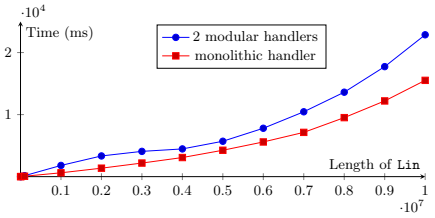

Fig. 4. 2 modular handlers *vs.* monolithic handler

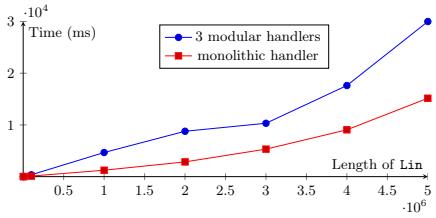

Fig. 5. 3 modular handlers *vs.* monolithic handler

Step 11. Finally, we fold the four occurrences of the composite handlers `run_state(phrase(_,_,_),_,_)` to obtain a tight and tidy definition of `query/5`.

```
query(G,Lin,Lout,Sin,Sout) :-
  reset(G,Contin,Commandin),
  ( Contin = 0          -> Lin = Lout, Sin = Sout),
  ; Commandin = c(E)    -> Lin = [E|Lmid], query(Contin,Lmid,Lout,Sin,Sout)
  ; Commandin = get(S) -> S = Sin, query(Contin,Lin,Lout,Sin,Sout)
  ; Commandin = put(S) -> query(Contin,Lin,Lout,S,Sout)
  ; shift(Commandin), query(Contin,Lin,Lout,Sin,Sout)
  ).
```

3.3.4 Preliminary Results

Figure 4 shows that the monolithic handler is about 1.4 times faster than the composition of two modular handlers, and Figure 5 shows that a monolithic handler is about 2 times faster than the composition of three modular handlers. Moreover, there are more cases that show that the performance gain can reach up to 4× times faster when using the transformed monolithic handler in comparison with the composite handler.

4 Open Issues

The main open issue of this project is to develop a correct and terminating algorithm to automate the transformation using the rules we developed. We are considering either an adhoc heuristic-based approach or a more systematic embedding in a partial evaluation framework (Lloyd and Shepherdson 1991).

We are currently leaning towards the first option. Starting by developing a higher abstract syntax to define handlers in order to restrict the programmer to define handlers in a transformable fashion. We are aiming to do program analysis to know the positions of the shifts and nested resets to ease the process of the automation.

In some handlers, there are multiple shifts within the same branch of a handler. This can occur due to a recursive predicate call within the branch of the handler. Capturing the recursive pattern of these predicates and transforming them to eliminate the delimited control code within them is still another open issue. However, *conjunctive partial deduction* (CPD) approach of De Schreye et al. (1999) seems to be a a promising solution. It has recently been extended by De Schreye and Nys (2014) to cope with linear recursion patterns.

One other solution that we are currently investigating for capturing recursive patterns is a technique developed by Pettorossi and Proietti (2002). It adds a list to the inputs arguments of the recursive predicate. Then it puts the calls needed to be executed after the recursion is finished.

This technique is close to explicit continuation-passing style (CPS), similar to BinProlog's binarization (Tarau 2012). With the program in CPS form the delimited control primitives can be expressed in terms of plain Prolog and optimized with partial evaluation. The downside is that CPS is rather indiscriminate and introduces lots of meta-calls.

In the future, we are aiming to eliminate all delimited control code from a program by using these transformation techniques.

References

Danvy, O. and Filinski, A. 1990. Abstracting control. Lisp and Functional Programming '90. 151–160.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. 1999. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming 41,* 2, 231–277.

De Schreye, D., Nys, V., and Nicholson, C. 2014. Analysing and compiling coroutines with abstract conjunctive partial deduction. In *Proceedings of the International Symposium on Logic-Based Program Synthesis and Transformation*, M. Proietti and H. Seki, Eds.

Felleisen, M. 1988. The theory and practice of first-class prompts. Principles of Programming Languages '88. 180–190.

Lloyd, J. W. and Shepherdson, J. C. 1991. Partial evaluation in logic programming. *The Journal of Logic Programming 11,* 3, 217–242.

Moggi, E. 1991. Notions of computation and monads. *Information and Computation 93,* 1.

Pettorossi, A. and Proietti, M. 1994. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming 19/20*, 261–320.

Pettorossi, A. and Proietti, M. 1999. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming 41,* 2-3, 197–230.

Pettorossi, A. and Proietti, M. 2002. The list introduction strategy for the derivation of logic programs. *Formal aspects of computing 13,* 3-5, 233–251.

Plotkin, G. and Pretnar, M. 2009. Handlers of algebraic effects. In *Programming Languages and Systems*. Springer, 80–94.

Schrijvers, T., Demoen, B., Desouter, B., and Wielemaker, J. 2013. Delimited continuations for Prolog. *Theory and Practice of Logic Programming 13,* 4-5, 533–546.

Tarau, P. 2012. The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *Theory and Practice of Logic Programming 12,* 1-2, 97–126.