Appendix A Detailed Optimisation Example Walk-Through

This appendix elaborates the optimisation example of Section 3.3 in more depth.

We start from the following program:

```
:- effect c/1.
ab.
ab :- c(a), c(b), ab.
query(Lin) :-
  handle ab with
    (c(X) -> Lin1=[X|Lmid], continue(Lmid,Lout1))
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=[]).
```

Step 1 We abstract the goal `handle ab with ...` into a new predicate `ab0/2`. This new predicate takes two arguments: one for every parameter in the handler's `for` clause. The original call is replaced by a call to the new predicate, supplying the actual parameters of the handler as actual arguments.

```
query(Lin) :- ab0(Lin,[]).
```

The predicate `ab0/2` is a copy of `ab/0`'s definition, with the handler wrapped around each clause's body.

```
ab0(Lin,Lout) :-
  handle true with
    (c(X) -> Lin1=[X|Lmid], continue(Lmid,Lout1))
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=Lout).

ab0(Lin,Lout) :-
  handle (c(a), c(b), ab) with
    (c(X) -> Lin1=[X|Lmid], continue(Lmid,Lout1))
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=Lout).
```

Step 2 The optimiser now applies rewrite rules to the two clauses. In the first clause, Rule (O-DROP) can be applied because the effect system provides the information that the goal `true` has no effects. Hence, we drop the operation clause:

```
ab0(Lin,Lout) :-
  handle true with
  finally (Lin1 = Lout1)
  for (Lin1=Lin,Lout1=Lout).
```

Step 3 The handler currently handles no operations[3]. The optimizer proceeds with applying (O-TRIV):

[3] This syntax is only allowed during the compilation process.

```
ab0(Lin,Lout) :-
  true,
  Lin1 = Lout1 ,
  Lin1 = Lin,
  Lout1 = Lout.
```

Step 4 By partially evaluating `true` and the remaining unifications, the first clause is simplified to:

```
ab0(L,L).
```

Step 5 In the second clause the handler's goal starts with the `c/1` operation. The optimiser applies (O-OP) to the handler, producing the following code:

```
ab0(Lin,Lout) :-
  Lin1 = [a|Lmid],
  Lin1 = Lin,
  Lout1 = Lout,
  handle (c(b), ab) with
    (c(X1) -> Lin11=[X1|Lmid1], continue(Lmid1,Lout11))
  finally (Lin11 = Lout11)
  for (Lin11=Lmid,Lout11=Lout1).
```

All the variables in the new handler goal are fresh variables. Observe that the actual arguments in the newly generated `for` clause are taken from the `continue` call of the previous handler. This is to ensure the correct state threading of the handler, and to keep the correct semantics of the program.

Step 6 The optimiser re-applies (O-OP) for `c(b)`, generating the following code:

```
ab0(Lin,Lout) :-
  Lin1 = [a|Lmid],
  Lin1 = Lin,
  Lout1 = Lout,
  Lin11 = [b|Lmid1],
  Lin11 = Lmid,
  Lout11 = Lout1,
  handle (ab) with
    (c(X2) -> Lin12=[X1|Lmid2], continue(Lmid2,Lout12))
  finally (Lin12 = Lout12)
  for (Lin12=Lmid1,Lout12=Lout11).
```

Step 7 The remaining handler goal is now a variant of the original one, which was already abstracted into `ab0/2`. Therefore, we can replace it with `ab0/2`.

```
ab0(Lin,Lout) :-
  Lin1 = [a|Lmid],
```

```
  Lin1 = Lin,
  Lout1 = Lout,
  Lin11 = [b|Lmid1],
  Lin11 = Lmid,
  Lout11 = Lout1,
  ab0(Lmid1,Lout11).
```

Step 8 The clause now consists of several unifications followed by a tail-recursive call. Partially evaluating the unifications leads to the final optimised code:

```
ab0([a,b|Lmid1],Lout) :-
  ab0(Lmid1,Lout).
```

Appendix B State-DCG Handler Example in Detail

This appendix shows the result of optimizing a program that consists of two handlers. We first show the elaboration into delimited control. Then, we show how the original program can be optimised by means of the rewrite rules and partial evaluation.

We use the following program, which was used to generate the results of the first benchmarks in Table 2. As described in Section 4, there are two handlers: one handles the implicit state operations and the other handles the DCG operations.

```
abinc.
abinc :- c(a), c(b), get_state(St), St1 is St+1, put_state(St1), abinc.

state_phrase_handler(Sin,Sout,Lin,Lout) :-
  handle
      (handle abinc
        with
          ( get_state(Q)  -> Q = Sin1, continue(Sin1,Sout1)
          ; put_state(NS) -> continue(NS,Sout1)
          )
        finally
          Sout1 = Sin1
        for
          (Sin1 = Sin, Sout1 = Sout)
      )
    with
      (c(X) -> Lin1 = [X|Lmid], continue(Lmid,Lout1))
    finally
      Lin1 = Lout1
    for
      (Lin1=Lin, Lout1=Lout).
```

The inner handler's goal is `abinc`, which consumes two elements, `a` and `b`, by using the operation `c/1` and then increments the state using the operations `get_state/1` and `put_state/1`.

```
?- state_phase_handler(0,Sout,[a,b,a,b,a,b],Lout).
Sout = 0
Lout = [a,b,a,b,a,b];
Sout = 1
Lout = [a,b,a,b];
Sout = 2
Lout = [a,b];
Sout = 3
Lout = [].
```

The immediate elaboration into delimited control yields:

```
state_phrase_handler(A, B, C, D) :-
  handler_0(handler_1(abinc,A,B), C, D).
handler_1(A, B, C) :-
  reset(A, D, E),
  ( D == 0 ->
      C = B
  ; E = get_state(F) ->
      F = B,
      handler_1(D, B, C)
  ; E = put_state(G) ->
      handler_1(D, G, C)
  ;   shift(E),
      handler_1(D, B, C)
  ).
handler_0(A, B, C) :-
  reset(A, D, E),
  ( D == 0 ->
      B = C
  ; E = c(F) ->
      B = [F|G],
      handler_0(D, G, C)
  ;   shift(E),
      handler_0(D, B, C)
  ).
```

The predicates `handler_0/3` and `handler_1/3` correspond to the elaborated DCG and state handlers respectively. They follow the semantics described in Section 2.3.

Using the rewrite rules first, yields the following elaborated program instead:

```
state_phrase_handler(A, B, C, D) :-
  handler_2(abinc, A, B, C, D).
handler_2(A, B, C, D, E) :-
```

```
  reset(A, F, G),
  ( F == 0 ->
      C = B,
      D = E
  ; G = get_state(H) ->
      H = B,
      handler_2(F, B, C, D, E)
  ; G = put_state(I) ->
      handler_2(F, I, C, D, E)
  ; G = c(J) ->
      D = [J|K],
      handler_2(F, B, C, K, E)
  ;   shift(G),
      handler_2(F, B, C, D, E)
  ).
```

The two handlers have been merged into one, with the corresponding performance improvement.

When partial evaluation is enabled as well, the optimisation goes one step further and yields the following final program:

```
state_phrase_handler(A, B, C, D) :-
  abinc0(A, B, C, D).
abinc0(A, A, B, B).
abinc0(A, B, [a,b|C], D) :-
  E is A+1,
  abinc0(E, B, C, D).
```

Partial evluation has pushed the handlers into the definition of `abcinc/0` where the rewrite rules have been able to replace the operations by the corresponding handler clauses. As a consequence, the handlers are eliminated and no delimited control primitives are generated.

Appendix C Soundness of Rule (O-Disj)

This appendix proves the soundness of the (O-Disj) rewrite rule. Our proof relies on the elaboration of the handler syntax into delimited control and the corresponding semantics for delimited control given by Schrijvers et al. (2013). This semantics is expressed in terms of a Prolog meta-interpreter that we show in Figure C 1.

We start from the left-hand side of the rewrite rule and turn it into the right-hand side by means of a number of equivalence preserving transformations.

$$\begin{array}{c}\texttt{handle (G1;G2) with} \\ \overline{op \to G;} \\ \texttt{finally } G_f \\ \texttt{for } G_s.\end{array} \tag{C1}$$

```
eval(G) :- eval(G,Signal),
          ( Signal = shift(Term,Cont) ->
            fail
          ; true).

eval(shift(Term),Signal) :- !,Signal = shift(Term,true).

eval(reset(G,Cont,Term),Signal) :- !, eval(G,Signal1),
                                   ( Signal1 = ok -> Cont = 0, Term = 0
                                   ; Signal1 = shift(Term,Cont)),
                                   Signal = ok.

eval((G1,G2),Signal) :- !, eval(G1,Signal1),
                       ( Signal1 = ok -> eval(G2,Signal)
                       ; Signal1 = shift(Term,Cont),
                         Signal = shift(Term,(Cont,G2))).

eval((G1;G2),Signal) :- !, ( eval(G1,Signal)
                           ; eval(G2,Signal)).

eval((C->G1;G2),Signal) :- !, ( eval(C,Signal1) ->
                                ( Signal1 = ok -> eval(G1,Signal)
                                ; fail
                                )
                              ; eval(G2,Signal)).

eval(Goal,Signal) :- built_in_predicate(Goal), !, call(Goal), Signal = ok.

eval(Goal,Signal) :- clause(Goal,Body), eval(Body,Signal).
```

Fig. C 1: Delimited Control Meta-Interpreter

The elaboration of this handler goal into delimited control yields the following auxiliary predicate:

```
h(Goal,P1,..,Pn) :-
  reset(Goal,Cont,Term),
  ( Term == 0  -> Gf
  ; Term =  op → G
  ; shift(Signal), h(Cont,P1,...,Pn)
  ).
```

Here the variables P_i are the formal parameters of G_s. The goal itself is then by definition equivalent to

$$\texttt{h((G1;G2),}A_1,\ldots,A_n\texttt{)} \tag{C2}$$

where the A_i are the actual parameters of G_s.

This is equivalent to evaluation the goal in the meta-interpreter:

$$\texttt{eval(h((G1;G2),}A_1,\ldots,A_n\texttt{))} \tag{C3}$$

We can now unfold the `eval/1` call and subsequently unfold the resulting call to the auxiliary predicate `eval/2` which selects the last clause. After also evaluating the call to `clause/2` to unfold $\texttt{h}/n+1$ we get:

```
eval( ( reset((G1;G2),Cont,Term),
        ( Term == 0  -> G_f
        ; Term =  op  ->  G
        ; shift(Signal), h(Cont,P_1,...,P_n)
        )                                                    (C4)
      )
    , Signal
    ),
(Signal = shift(Term,Cont) -> fail ; true)
```

For the sake of space, we refer to the if-then-else block after the `reset/3` call as `<Switches>`. We can thus abbreviate the above as:

```
eval( (reset((G1;G2),Cont,Term), <Switches>)
    , Signal                                                 (C5)
    ),
(Signal = shift(Term,Cont) -> fail ; true)
```

Unfolding `eval/2` using the appropriate clause for conjunction, yields:

```
eval(reset((G1;G2),Cont,Term), Signal1),
( Signal1 = ok -> eval(<Switches>, Signal)
; Signal1 = shift(Term,Cont) -> Signal = shift(Term,(Cont,<Switches>))   (C6)
),
(Signal = shift(Term,Cont) -> fail ; true)
```

Now we unfold the first call to `eval/2` using the clause for `reset/3`:

```
eval((G1;G2), Signal2),
( Signal2 = ok -> Cont = 0, Term = 0
; Signal2 = shift(Term,Cont)
),
Signal1 = ok,                                                (C7)
( Signal1 = ok -> eval(<Switches>, Signal)
; Signal1 = shift(Term,Cont) -> Signal = shift(Term,(Cont,<Switches>))
),
(Signal = shift(Term,Cont) -> fail ; true)
```

Again, we unfold the first call to `eval/2` using the clause for disjunction:

```
( eval(G1, Signal2) ; eval(G2, Signal2) ),
( Signal2 = ok -> Cont = 0, Term = 0
; Signal2 = shift(Term,Cont)
),
Signal1 = ok,                                                (C8)
( Signal1 = ok -> eval(<Switches>, Signal)
; Signal1 = shift(Term,Cont) -> Signal = shift(Term,(Cont,<Switches>))
),
(Signal = shift(Term,Cont) -> fail ; true)
```

We now distribute what comes after the first disjunction into both branches.

```
(
  eval(G1, Signal2),
  ( Signal2 = ok -> Cont = 0, Term = 0
  ; Signal2 = shift(Term,Cont)
  ),
  Signal1 = ok,
  ( Signal1 = ok -> eval(<Switches>, Signal)
  ; Signal1 = shift(Term,Cont) -> Signal = shift(Term,(Cont,<Switches>))
  ),
  (Signal = shift(Term,Cont) -> fail ; true)
;                                                                   (C9)
  eval(G2, Signal2),
  ( Signal2 = ok -> Cont = 0, Term = 0
  ; Signal2 = shift(Term,Cont)
  ),
  Signal1 = ok,
  ( Signal1 = ok -> eval(<Switches>, Signal)
  ; Signal1 = shift(Term,Cont) -> Signal = shift(Term,(Cont,<Switches>))
  ),
  (Signal = shift(Term,Cont) -> fail ; true)
)
```

At this point we change gear and start folding again. First we fold the `reset/2` clause of `eval/2` twice, once in each branch.

```
(
  eval(reset(G1,Term,Cont),Signal1),
  ( Signal1 = ok -> eval(<Switches>, Signal)
  ; Signal1 = shift(Term,Cont) -> Signal = shift(Term,(Cont,<Switches>))
  ),
  (Signal = shift(Term,Cont) -> fail ; true)
;                                                                   (C10)
  eval(reset(G2,Term,Cont),Signal1),
  ( Signal1 = ok -> eval(<Switches>, Signal)
  ; Signal1 = shift(Term,Cont) -> Signal = shift(Term,(Cont,<Switches>))
  ),
  (Signal = shift(Term,Cont) -> fail ; true)
)
```

Then we fold the conjunction clause of `eval/2` in each branch.

```
(
  eval((reset(G1,Term,Cont),<Switches>),Signal),
  (Signal = shift(Term,Cont) -> fail ; true)
;
  eval((reset(G2,Term,Cont),<Switches>),Signal),
  (Signal = shift(Term,Cont) -> fail ; true)
)
```

(C11)

Subsequently, we fold `eval/1` twice.

```
(
  eval((reset(G1,Term,Cont),<Switches>))
;
  eval((reset(G2,Term,Cont),<Switches>))
)
```

(C12)

Now we can drop the meta-interpretation layer again.

```
(
  (reset(G1,Term,Cont),<Switches>)
;
  (reset(G2,Term,Cont),<Switches>)
)
```

(C13)

Then we fold $\texttt{h}/n+1$ twice.

$$\texttt{(h(G1,}\mathtt{A}_1\texttt{,...,}\mathtt{A}_n\texttt{); h(G2,}\mathtt{A}_1\texttt{,...,}\mathtt{A}_n\texttt{))} \tag{C14}$$

Finally, we invert the elaboration to obtain the right-hand side of the rewrite rule.

$$\begin{array}{c} \texttt{handle}\ G_1\ \texttt{with} \\ \overline{op \to G;} \\ \texttt{finally}(G_f) \\ \texttt{for}(G_s) \end{array} \; ; \; \begin{array}{c} \texttt{handle}\ G_2\ \texttt{with} \\ \overline{op \to G;} \\ \texttt{finally}(G_f) \\ \texttt{for}(G_s) \end{array} \tag{C15}$$