Extended Abstract: A Functional Derivation of the Warren Abstract Machine

Maciej Piróg and Jeremy Gibbons

Oxford University Computing Laboratory
firstname.lastname@comlab.ox.ac.uk

Abstract. Based on Danvy et al.'s functional correspondence, we give a further example of gradual refinement of an interpreter into a known, low-level abstract machine underlying real-world compilers, by deriving an abstract model of the Warren Abstract Machine from a simple resolution-based Prolog interpreter. We show that other well-known functional programming techniques (namely, explicit laziness and semi-persistent data structures) can help to develop abstract machines without detailed examination of the semantics realised by the interpreter.
Keywords. Abstract machine, defunctionalisation, continuation-passing, semi-persistent data structures.

1 Introduction

The Warren Abstract Machine [4, 11] (WAM) is a detailed, low-level virtual machine for execution of Prolog. It has been widely studied and serves as a foundation for several Prolog implementations. It comes with a great number of variations that boost efficiency or implement additional language features. It consists of a translation scheme from Prolog into a set of instructions, a description of an evaluation environment (a heap, stacks, registers), and a collection of rules defining how each instruction affects the environment.

The most comprehensive study of the WAM and its correctness with respect to a more intuitive semantics is due to Börger and Rosenzweig [6]. Their study successfully isolates the WAM evaluation model from the myriad of low-level technical components, different for different incarnations of the WAM. In this spirit, we are trying to isolate the WAM evaluation model in the form of an abstract machine that operates directly on Prolog terms, rather than on a set of instructions. This way, we maintain a high level of abstraction, leaving a lot of space for further implementation choices. For example, we concentrate on WAM control structure and backtracking, without much concern for how terms are unified and represented in the heap. We start with an evaluator based on SLD-resolution written in Haskell, and apply meaning-preserving program transformations to get an abstract model of the WAM in a few simple steps. Hence, without any heavy formal machinery, we show that the WAM evaluation model is equivalent to SLD-resolution. We do not provide a complete proof of equivalence, but all the transformations we perform on the interpreter are either very intuitive or previously proved correct.

Our main tool is Danvy et al.'s functional correspondence between evaluators and abstract machines [1], which has previously been used for derivation of machines for Prolog [5], but not for any as elaborate as the WAM. In derivations based on functional correspondence, we embed an evaluator in a functional metalanguage, representing various elements of the language and its semantics by abstract objects of the metalanguage; for example, we represent functions in the language by functions in the metalanguage. We convert the evaluator to continuation-passing style (CPS) and defunctionalise the continuations, making it less dependent on the semantics of the metalanguage [1–3].

Our observation is that to liberate the evaluator from the metalanguage, said transformations reify some techniques used to implement the abstract constructs of the metalanguage. Defunctionalisation replaces functional abstractions with explicit closures, while CPS conversion creates an explicit call-stack. We may then extend this repertoire with less elegant, but still useful mechanisms (not necessarily seen as program transformations), such as efficient implementation of semi-persistent data structures [7], which, as we will see, enable us to switch from the persistence-based backtracking present in the evaluator implementing SLD-resolution to the imperative heap unwinding implemented by the WAM.

In this extended abstract we show only the essential steps of our derivation. The full source code with more fine-grained transformations is available at the first author's web page[1].

2 A Prolog interpreter

Program definition. A term is either a variable or an operator with a list of terms as arguments.

$$\begin{aligned}&\textbf{type } \mathit{VarN} = \mathit{String}\\ &\textbf{type } \mathit{OpN} \;= \mathit{String}\\ &\textbf{data } \mathit{Term} = \mathit{Var}\ \mathit{VarN} \mid \mathit{Op}\ \mathit{OpN}\ [\,\mathit{Term}\,]\end{aligned}$$

A substitution is an association list pairing variables and terms. We do not substitute for variables in terms; instead, we keep one global substitution. Thus, when we see a variable, we must always look it up in the substitution to determine whether there is an associated value.

$$\textbf{type } \mathit{Sub} = [(\mathit{VarN}, \mathit{Term})]$$

We model Prolog programs by means of clauses and predicates. In each predicate, the arity and the outermost operator in the head of each clause should coincide. A program is a map from predicate names to predicates. A name in a key should match the name of the associated predicate (that is, the outermost operator in heads of all its clauses). We hold the whole Prolog program as a constant value.

$$\begin{aligned}&\textbf{data } \mathit{Cls} \;\; = \mathit{Cls}\ \mathit{Term}\ [\,\mathit{Term}\,]\\ &\textbf{type } \mathit{Pred} = [\,\mathit{Cls}\,]\end{aligned}$$

[1] `http://www.comlab.ox.ac.uk/people/maciej.pirog/wam.tar.gz`

```
type Prog = Data.Map.Map OpN Pred
database :: Prog
database = ...
```

A simple interpreter. The function *predicate* looks up a proper predicate in the program. If no such predicate is defined, the whole evaluation fails. To avoid variable name capture in resolution, the function *freshClause* generates fresh variable names. The function *mgu* computes the most general unifier in the context of some initial global substitution, and, if the unifier exists, returns the global substitution extended with the unifier.

```
predicate :: OpN → Pred
predicate f = case Data.Map.lookup f database of
  Nothing → error ("Predicate " ++ f ++ " not in the database")
  Just p  → p
freshClause :: Integer → Cls → Cls
freshClause n (Cls h tl) = Cls (aux h) (map aux tl) where
  aux (Op f ts) = Op f (map aux ts)
  aux (Var v)   = Var (v ++ "__" ++ show n)
mgu :: Term → Term → Sub → Maybe Sub
mgu t r s = ...
```

The interpreter computes the first solution to a given query, via lazy evaluation. The integer argument indicates the current depth of the resolution tree; it is used solely for generating fresh variable names.

```
type Qry = [Term]
eval :: Sub → Qry → Integer → Maybe Sub
eval s []                 _ = Just s
eval _ ((Var _) : _)      _ = error "Variable as goal"
eval s (g@(Op f _) : gs) n = msum (map try (predicate f)) where
  try c = let Cls h b = freshClause n c in case mgu g h s of
    Nothing → Nothing
    Just s' → eval s' (b ++ gs) (n + 1)
ask :: Qry → Maybe Sub
ask q = eval [] q 0
```

The first step. To prepare for subsequent transformations, we rewrite the program so that it no longer depends on laziness. We cannot look for solutions for all clauses from a predicate at once and then select one; instead, we traverse the list of clauses until we find a solution.

Moreover, we make our first transformation, which is the only one inspired directly by the design of the WAM. Instead of passing the first of the current goals, g, to *try*, we store g in the global substitution under a fresh name, which we pass to *try* as a 'pointer'. Then the third clause of *eval* becomes:

```
eval s (g@(Op s _) : gs) n = let pg = "TOP" ++ show n
                                 s' = (pg, g) : s
                             in  try s' gs n pg (predicate s)
try :: Sub → Qry → Integer → VarN → Pred → Maybe Sub
try _ _ _ _ []       = Nothing
try s gs n pg (c : cs) =
  let Cls h b = freshClause n c
  in  case mgu (Var pg) h s of
        Nothing → try s gs n pg cs
        Just s' → case eval s' (b ++ gs) (n + 1) of
                    Nothing → try s gs n pg cs
                    j       → j
```

3 Semi-persistent substitution

A data structure is *semi-persistent* [7] if we can backtrack to its previous versions, but we never simultaneously keep two different modifications (siblings) of the same ancestor. Then we may define dedicated data structures which make some under-the-hood optimisations. Here, we focus our attention on a regular persistent list used in a semi-persistent way.

Having a list xs, we may create another list by consing an element, obtaining $x : xs$, and yet another by consing a different element, obtaining $y : xs$. Though $x : xs$ and $y : xs$ are different lists, they share a tail, which is a previous version of both. If xs is used in a semi-persistent way, after consing y, there is no need to keep $x : xs$ alive in memory. Consequently, no part of such a list is ever shared with any other list, except for previous versions. We can see the list as a stand-alone object, mutable over time by consing new elements and dropping prefixes. As with any mutable object, we can encode it as a 'global' value, threaded through functions in exactly the same way that stateful computations treat state.

In the case of association lists with no duplicate keys, we may also reify pointers to previous versions as keys of the first elements of appropriate tails. Backtracking then corresponds to dropping the prefix before the right key.

Note that *mgu* and *eval* may only extend the global substitution, and if *eval* in *try* returns *Nothing*, we use the ancestor of the substitution from before the unification. So, we may encode the substitution as a semi-persistent data structure, with explicit backtracking provided by the function *unwind*:

```
unwind :: Sub → VarN → Sub
unwind []            _ = error "Wrong trail address"
unwind h@((v, _) : t) s | v ≡ s      = h
                        | otherwise = unwind t s
```

Since we want the substitution to be a state now, it must always be returned by each function of the interpreter, successful or not. Thus, we change the return type of the functions from *Maybe Sub* to (Res, Sub), where

```
data Res = Flr | Scs
```

Apart from appropriate refactoring of return values, this impacts mostly on the second clause for *try*:

```
try s gs n pg (c : cs) =
  let Cls h b          = freshClause n c
      (trailPtr, _) : _ = s
  in  case mgu (Var pg) h s of
        (Flr, s') → try (unwind s' trailPtr) gs n pg cs
        (Scs, s') → case eval s' (b ++ gs) (n + 1) of
                      (Flr, s'') → try (unwind s'' trailPtr) gs n pg cs
                      j          → j
```

To assure that the matching of *trailPtr* never fails, we start with a substitution which holds a dummy variable that stands for the bottom of the substitution:

```
ask :: Qry → (Res, Sub)
ask q = eval [("BOTTOM", ⊥)] q 0
```

4 CPS transformation and defunctionalisation

Now we can CPS-transform the functions with return type (Res, Sub), so that they take a continuation as an additional argument. Instead of returning a value directly, the functions either tail-call themselves or apply the continuation to the result. Since *Res* is an enumeration of two elements, the type $(Res, Sub)\rightarrow k$ is isomorphic to $(Sub\rightarrow k, Sub\rightarrow k)$—*success* and *failure* continuations [10]. Apart from the obvious changes to *mgu*:

```
mgu :: Term → Term → Sub → (Sub→k) → (Sub→k) → k
mgu t r s ks kf = case ... of
  (Scs, s) → ks s
  (Flr, s) → kf s
```

the most significant changes are to the *try* function:

```
try :: Sub → Qry → Integer → VarN → Pred → (Sub→k) → (Sub→k) → k
try s _  _ _  []       _  kf = kf s
try s gs n pg (c : cs) ks kf =
  let Cls h b          = freshClause n c
      (trailPtr, _) : _ = s
  in  mgu (Var pg) h s
        (λs' → eval s' (b ++ gs) (n + 1)
                 (λs'' → ks s'')
                 (λs'' → try (unwind s'' trailPtr) gs n pg cs ks kf))
        (λs' →           try (unwind s' trailPtr) gs n pg cs ks kf)
```

Inlining the success continuation. In the previous interpreter, *mgu* is used in one place only, so we know exactly the continuations it is applied to. Also, the failure continuation for *eval* in *try* is the same as the failure continuation

for *mgu*. So, instead of giving *mgu* its complicated success continuation, we can rewrite it so that in case of success it calls *eval* directly with *try*'s success continuation and *mgu*'s failure continuation. As we may notice, the original *mgu*'s success continuation is the only non-identity success continuation in the whole interpreter. Since we managed to inline it into *mgu*, we may get rid of the success continuations altogether, and replace them with direct returns.

```
mgu :: Sub → Qry → Integer → VarN → Term → (Sub→Sub) → Sub
mgu s gs n pg t kf = case ... of
  (Scs, s') → eval s' gs (n + 1) kf
  (Flr, s') → kf s'

try :: Sub → Qry → Integer → VarN → Pred → (Sub→Sub) → Sub
try s _ _ _ []       kf = kf s
try s gs n pg (c : cs) kf =
  let Cls h b          = freshClause n c
      (trailPtr, _) : _ = s
  in  mgu s (b ++ gs) n pg h
        (λs' → try (unwind s' trailPtr) gs n pg cs kf)
```

Defunctionalisation of continuations. We replace each construction of a failure continuation with an explicit closure, which stores all free variables of the continuation, and each application of a continuation with a call to *apply*.

```
data BStack = Btm | Frm Qry Integer VarN Pred VarN BStack

apply :: BStack → (Sub→Sub)
apply Btm                          _ = error "No"
apply (Frm gs n pg cs trailPtr kf) s' =
  try (unwind s' trailPtr) gs n pg cs kf

try :: Sub → Qry → Integer → VarN → Pred → BStack → Sub
try s _ _ _ []       kf = apply kf s
try s gs n pg (c : cs) kf =
  let Cls h b          = freshClause n c
      (trailPtr, _) : _ = s
  in  mgu s (b ++ gs) n pg h (Frm gs n pg cs trailPtr kf)

ask :: Qry → Sub
ask q = eval [("BOTTOM", ⊥)] q 0 Btm
```

5 Comparison with a realistic WAM

In this section we compare our abstract machine with the exemplary low-level WAM implementation described by Aït-Kaci [4].

Queries. The code compiled from the Prolog query and bodies of clauses (the `put...` instructions) is mirrored by the *eval* function. The basic idea is to put the terms in the heap (represented here by the global substitution) and then the clauses from the suitable predicates will try to unify them with their heads.

Predicates. The code for a predicate (the `try...` and `trust...` instructions) is mirrored by the *try* function. Each clause first pushes a backtrack frame on the stack, which will allow the system to go back to the current state to try another clause, and then enters the code for the head of the clause.

Clauses. The code for the head of a clause tries to unify the head with an appropriate term in the heap (the `get...` and `unify...` instructions). If it succeeds, we continue with the body of the clause or the next goal from the stack. Otherwise, we backtrack. This is reflected by the *mgu* function.

Backtracking. If the unification fails at some point, the WAM calls its `backtrack` subroutine, which restores a previous state using data stored in the top backtrack frame on the stack, and jumps to code which tries to unify an another potentially matching clause. This is reflected by the *apply* function.

The heap and the trail. Our global substitution models the WAM heap as an association list, enabling backtracking based on its structure. We could use a more elaborate data structure, for example a map implemented using binary-search trees. To use it as semi-persistent state, we must implement some backtracking mechanism, for example keeping a stack of variables for which we substitute, a *trail*. To backtrack to a previous state, we need to revert the appropriate front of the trail. Our model shows that the exact heap implementation is not important for the control structure of the WAM. We just need to write and unify terms, and take a small piece of information to store on a stack and then use to backtrack to a previous version of the heap.

The backtrack stack. In the WAM, if the query or body of a clause consists of more than one goal, we put a pointer to the code for other goals on the stack before entering the code for the first goal. After entering a pointer from the stack, we cannot always pop it, since it may be needed again after backtracking. That is why we put backtrack frames on the same stack, and do not pop pointers to goals if there is a backtrack frame above them. Thus, there may be goals on the stack that are already satisfied, and in real implementations we keep both goals and backtrack frames as linked lists to bypass the satisfied goals when looking for a new goal, or all the goals in between when looking for another backtrack frame. The careful analysis of this fragile ecosystem of the WAM stack may seem difficult at first. Our model reveals that this complicated structure is just an implementation of a stack ($BStack$) of stacks (Qry). Each backtrack frame holds the complete list of goals to be satisfied after backtracking, but goals may be shared between different frames. In the WAM, the sharing is implemented by linked lists.

The goal stack. Our goal stack seems crude in comparison to WAM goal management. The problem is that when trying a clause, our model puts its whole body on the stack, even before we check whether the head of the clause unifies with the current goal. The WAM delays this kind of stack operation as much

as possible, which means that we push a single pointer to the next goal just before dealing with the previous one. For instance, if the body of the clause is $[g_1, g_2, g_3]$, before evaluation of g_1, we push a single pointer to the code for g_2 (and not two pointers to g_2 and g_3), and then it is the job of g_2 to push a pointer to g_3 before evaluation. Of course this is all done after the unification of the head of the clause.

It is easy to move the goal management after the unification. We just need to give *mgu* the whole clause as an argument (not only its head) and push the body onto the stack after unification of the head. It would not violate the correspondence with the WAM, since both *eval* and *mgu* reflect code created for the whole clause. It seems much more difficult to push goals one by one, and not all the goals from the body at once. The solution comes as yet another technique used by functional programmers: explicit laziness.

In strict languages, the cost of concatenation of two lists, say *xs* and *ys*, is linear in the length of *xs*, because we need to rebuild it in order to connect its last element with *ys*. The trick to avoid this is to traverse *xs* only when it is really needed. Thus, *ys* is represented as a list of thunks containing lists to be traversed, via **type** $LList\ a = [[a]]$; then $+\!\!+$, *head* and *tail* can all be implemented in constant time. From the point of view of the datatype interface, we still just pop goals from the stack and concatenate bodies of clauses. Hence, we need no conceptual changes in the interpreter, just some syntactic adjustments. But inlining the exact definitions of operations on *LList*s reveals that concatenation of a body $[g_1, g_2, g_3]$ to the goal stack is in reality pushing a single element $[g_1, g_2, g_3]$, and when we want to pop the soon-to-be-satisfied goal g_1, the *tail* function removes this element and pushes $[g_2, g_3]$ instead. That is exactly the WAM behaviour; we just need to think of elements of the goal stack as pointers to tails of the bodies of the clauses, not individual goals.

Argument registers. The outermost operator in the head of each clause of a predicate is the same as the name of the predicate in the program, so there is no reason to compare it to the outermost operator in the current goal during unification: it always matches. We can optimise the evaluation by writing only the arguments of the current goal in the substitution, and give them to *try* using some set of registers, instead of the single register *pg*. This is an optimisation that originates on the level of the interpreter and does not require any understanding of the WAM. Moreover, it can be almost forced by a more precise program representation, representing clauses with **data** $Cls = Cls\ [Term]\ [Term]$, so that we hold the head of a clause as a list of its arguments; we no longer need the outermost operator, since it is equivalent to the name of the predicate.

6 Conclusion and future work

We present a derivation of a model of the WAM from a simple Prolog interpreter. We could obtain a more precise model with the same method simply by tweaking the interpreter and the definition of Prolog programs. For a further example, we

may avoid pushing a backtrack frame for the last clause of a predicate by using a more elaborate datatype for clauses, with an explicit constructor for the last element. Many such optimisations can be expressed on the level of interpreter, including tail-calls, the optimised treatment of lists, or switching on atoms.

It would be interesting to build an instruction set for our model and compare it to that of the WAM. We know all the possible goals in *eval* (up to fresh variable names) and all the possible clauses for *try*, as they are always parts of the original program, so they may be easily encoded as instructions. It would also be possible to formalise our derivation in a proof system like Coq. This may lead to a novel construction of a certified Prolog compiler—the existing developments are just formalisations of the proof by Börger and Rosenzweig [8, 9].

Acknowledgements. We thank Dariusz Biernacki for helpful comments and pointers to related work. This work was partially supported by UK EPSRC grant *Reusability and Dependent Types*.

References

1. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Miller, D. (ed.) Principles and Practice of Declarative Programming. pp. 8–19 (2003)
2. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between call-by-need evaluators and lazy abstract machines. Inf. Process. Lett. 90(5), 223–232 (2004)
3. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Theor. Comput. Sci. 342(1), 149–172 (2005)
4. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991)
5. Biernacki, D., Danvy, O.: From interpreter to logic engine by defunctionalization. In: Bruynooghe, M. (ed.) Logic Based Program Synthesis and Transformation. LNCS, vol. 3018, pp. 143–159 (2003)
6. Börger, E., Rosenzweig, D.: The WAM—definition and compiler correctness. In: Logic Programming: Formal Methods and Practical Applications, pp. 20–90. Elsevier (1995)
7. Conchon, S., Filliâtre, J.C.: Semi-persistent data structures. In: Programming Languages and Systems. LNCS, vol. 4960, pp. 322–336 (2008)
8. Pusch, C.: Verification of compiler correctness for the WAM. In: Theorem Proving in Higher Order Logics. pp. 347–362. Springer-Verlag (1996)
9. Schellhorn, G., Ahrendt, W.: The WAM case study: Verifying compiler correctness for Prolog with KIV. In: Bibel, W., Schmitt, P.H. (eds.) Automated Deduction — A Basis for Applications, pp. 165–194. Kluwer, Dordrecht (1998)
10. Wand, M., Vaillancourt, D.: Relating models of backtracking. In: International Conference on Functional Programming. pp. 54–65 (2004)
11. Warren, D.H.D.: An abstract Prolog instruction set. Tech. rep., SRI International, Menlo Park (1983)

Appendix: Haskell notation

We have used the notation of the Haskell programming language throughout. For the benefit of reviewers who may not be familiar with Haskell, this appendix presents a brief summary of the important points—assuming familiarity with functional programming in general, but not Haskell in particular.

- Haskell's semantics is *lazy*, or call-by-need: arguments to functions are not evaluated at call time, but only when their value is needed to make further progress, and moreover, multiple copies of an argument in a function body are shared, so that no argument is evaluated more than once. In particular, list cons is lazy; one may define a (perhaps infinitely) long list, but look at only the first few elements, and then the remaining elements are never evaluated.
- As a lexical rule, identifiers with an initial lowercase letter are used for type- and value variables, and identifiers with an initial uppercase letter for type- and constructor names.
- The type expression $[a]$ denotes the type of (finite or infinite) lists, with elements of type a; the constructors are $[]$ ('nil') and $:$ ('cons'); there is a shorthand $[1, 2, 3]$ for enumerating the elements of a list, and the operator $+\!\!+$ appends two lists.
- The keyword **type** introduces a new synonym for an existing type; **data** declares a new algebraic datatype—typically with multiple variants, each variant having a constructor and zero or more arguments. For example, the standard prelude provides a datatype **data** $Maybe\ a = Just\ a \mid Nothing$ of 'optional values'; we also use the library function $msum :: [Maybe\ a] \to Maybe\ a$, which takes a list of optional values and returns *Just* the first one present, or *Nothing* if none are.
- There is a module *Data.Map* in the standard library, which provides a datatype *Map* of finite mappings, and (among many other functions) an operation *lookup* of type $k \to Map\ k\ a \to Maybe\ a$ (for which the key type k has to provide an ordering).
- In 'patterns' on the left-hand side of function definitions, $\_$ is a wildcard, and @ allows matches against the same value; so matching $xs@(y : ys)$ against the list $[1, 2, 3]$ binds $xs = [1, 2, 3]$ but also binds $y = 1$ and $ys = [2, 3]$.
- The binary comparison operator $\equiv$ yields a boolean result (in contrast to $=$, which is used only in definitions), and $\bot$ denotes the undefined value.

Appendix: full versions of programs

For reasons of space, we have been constrained to presenting in the body of the paper only the programs fragments with significant changes for each of the transformation steps. For ease of reading, this appendix presents fuller definitions.

The first step

The transformation on page 3 eliminates the dependency on lazy evaluation, yielding the following definitions.

```
eval :: Sub → Qry → Integer → Maybe Sub
eval s []                    _ = Just s
eval _ ((Var _) : _)         _ = error "Variable as goal"
eval s (g@(Op s _) : gs) n = let pg = "TOP" ++ show n
                                 s' = (pg, g) : s
                             in  try s' gs n pg (predicate s)
try :: Sub → Qry → Integer → VarN → Pred → Maybe Sub
try _ _  _ _  []       = Nothing
try s gs n pg (c : cs) =
  let Cls h b = freshClause n c
  in  case mgu (Var pg) h s of
        Nothing → try s gs n pg cs
        Just s' → case eval s' (b ++ gs) (n + 1) of
                    Nothing → try s gs n pg cs
                    j       → j
```

Semi-persistent substitutions

By explicitly managing the backtracking, we can make the substitution data structure semi-persistent, as described in Section 3. That leads to the following definitions.

```
mgu :: Term → Term → Sub → (Res, Sub)
mgu t r s = ...
eval :: Sub → Qry → Integer → (Res, Sub)
eval s []                  _ = (Scs, s)
eval _ ((Var _) : _)       _ = error "Variable as goal"
eval s (g@(Op s _) : gs) n = let pg = "TOP" ++ show n
                                 s' = (pg, g) : s
                             in  try s' gs n pg (predicate s)
try :: Sub → Qry → Integer → VarN → Pred → (Res, Sub)
try s _ _ _ []       = (Flr, s)
try s gs n pg (c : cs) =
  let Cls h b          = freshClause n c
      (trailPtr, _) : _ = s
  in  case mgu (Var pg) h s of
        (Flr, s') → try (unwind s' trailPtr) gs n pg cs
        (Scs, s') → case eval s' (b ++ gs) (n + 1) of
                      (Flr, s'') → try (unwind s'' trailPtr) gs n pg cs
                      j          → j
```

Explicit success and failure continuations

The CPS transformation described in Section 4 leads to versions of the interpreter that explicitly manage success and failure continuations.

```
mgu :: Term → Term → Sub → (Sub→k) → (Sub→k) → k
mgu t r s ks kf = case ... of
  (Scs, s) → ks s
  (Flr, s) → kf s

eval :: Sub → Qry → Integer → (Sub→k) → (Sub→k) → k
eval s []                  _ ks kf = ks s
eval _ ((Var _) : _)       _ ks kf = error "Variable as goal"
eval s (g@(Op s _) : gs) n ks kf = let pg = "TOP" ++ show n
                                       s' = (pg, g) : s
                                   in try s' gs n pg (predicate s) ks kf

try :: Sub → Qry → Integer → VarN → Pred → (Sub→k) → (Sub→k) → k
try s _ _ _ []       _ kf = kf s
try s gs n pg (c : cs) ks kf =
  let Cls h b          = freshClause n c
      (trailPtr, _) : _ = s
  in mgu (Var pg) h s
       (λs' → eval s' (b ++ gs) (n + 1)
                (λs'' → ks s'')
                (λs'' → try (unwind s'' trailPtr) gs n pg cs ks kf))
       (λs' →          try (unwind s' trailPtr) gs n pg cs ks kf)

ask :: Qry → Sub
ask q = eval [("BOTTOM", ⊥)] q 0 id (const (error "No"))
```

Inlining the success continuation

On page 6 we showed how to streamline the continuation-passing interpreter, eliminating the success continuations.

```
mgu :: Sub → Qry → Integer → VarN → Term → (Sub→Sub) → Sub
mgu s gs n pg t kf = case ... of
  (Scs, s') → eval s' gs (n + 1) kf
  (Flr, s') → kf s'

eval :: Sub → Qry → Integer → (Sub→Sub) → Sub
eval s []                    _ kf = s
eval _ ((Var _) : _)         _ kf = error "Variable as goal"
eval s (g@(Op s _) : gs) n kf = let pg = "TOP" ++ show n
                                    s' = (pg, g) : s
                                in try s' gs n pg (predicate s) kf

try :: Sub → Qry → Integer → VarN → Pred → (Sub→Sub) → Sub
try s _ _ _ []       kf = kf s
try s gs n pg (c : cs) kf =
  let Cls h b          = freshClause n c
      (trailPtr, _) : _ = s
  in  mgu s (b ++ gs) n pg h
        (λs' → try (unwind s' trailPtr) gs n pg cs kf)

ask :: Qry → Sub
ask q = eval [("BOTTOM", ⊥)] q 0 (const (error "No"))
```

Defunctionalisation

Finally, on page 6 we defunctionalised the failure continuations using the datatype *BStack*.

```
data BStack = Btm
            | Frm Qry Integer VarN Pred VarN BStack
apply :: BStack → (Sub→Sub)
apply Btm                          _ = error "No"
apply (Frm gs n pg cs trailPtr kf) s' =
  try (unwind s' trailPtr) gs n pg cs kf
mgu :: Sub → Qry → Integer → VarN → Term → BStack → Sub
mgu s gs n pg t kf = case ... of
  (Scs, s') → eval s' gs (n + 1) kf
  (Flr, s') → apply kf s'
eval :: Sub → Qry → Integer → BStack → Sub
eval s []                  _ kf = s
eval _ ((Var _) : _)       _ kf = error "Variable as goal"
eval s (g@(Op s _) : gs) n kf = let pg = "TOP" ++ show n
                                    s' = (pg, g) : s
                                in  try s' gs n pg (predicate s) kf
try :: Sub → Qry → Integer → VarN → Pred → BStack → Sub
try s _ _ _ []       kf = apply kf s
try s gs n pg (c : cs) kf =
  let Cls h b          = freshClause n c
      (trailPtr, _) : _ = s
  in  mgu s (b ++ gs) n pg h (Frm gs n pg cs trailPtr kf)
ask :: Qry → Sub
ask q = eval [("BOTTOM", ⊥)] q 0 Btm
```