Compilation of Combinatory Reduction Systems

Stefan Kahrs*

University of Edinburgh
Laboratory for Foundations of Computer Science

Abstract. Combinatory Reduction Systems generalise Term Rewriting Systems. They are powerful enough to express β-reduction of λ-calculus as a single rewrite rule. The additional expressive power has its price — CRSs are much harder to implement than ordinary TRSs.
We propose an abstract machine suitable for executing CRSs. We define what it means to execute an instruction, and give a translation from CRS rules into sequences of instructions. Applying a rewrite rule to a term is realised by initialising the machine with this term, and then successively executing the instructions of the compiled rule.

1 Introduction

Combinatory Reduction Systems were introduced by Klop in 1980 [9]. CRSs in their original form generalise *applicative* TRSs [10]. We shall concentrate here on *functional* CRSs, as defined by Kennaway in [8]; they generalise ordinary TRSs. The techniques of this paper can easily be adapted to applicative CRSs.

Functional CRSs extend TRSs in two respects. Firstly, they support a notion of variable binding. Substitution has to respect variable bindings: it is not allowed to capture bound variables. Thus, if a rewrite rule contains a subterm $\lambda x.y$, where y is a free variable, then no instance of the rewrite rule can substitute anything for y which contains x freely. However, this excludes rules such as the S-combinator introduction rule in the translation from λ-calculus into Combinatory Logic:

$$\mathrm{Lambda}([x]\mathrm{App}(y,z)) \;\rightarrow\; \mathrm{App}(\mathrm{App}(\mathrm{S},\mathrm{Lambda}([x]y)),\mathrm{Lambda}([x]z))$$

For the S-combinator introduction rule, it is necessary that the terms substituted for y and z may contain x freely, but this is not possible with ordinary first-order substitutions. To overcome this restriction, CRSs have another extension: they support second-order variables and second-order substitutions. For first-order terms, the application of a substitution to a term and the computation of a matching substitution are rather simple operations. Thus, a rewriting step itself is a rather simple operation. This is not longer true in the second-order world, i.e. if the terms contain second-order variables; see [4] where Huet and Lang present an algorithm to compute all principal matches for the second-order case. For second-order variables, substitution application incorporates β-reduction, and consequently matching may involve β-expansion.

* The research reported here was partially supported by SERC grant GR/J07303.

Combinatory Reduction Systems support second-order rewriting only in a restricted way that uniquely determines the necessary β-expansions during matching. Nevertheless, β-expansion and β-reduction remain rather expensive operations — it would be nice to compile CRSs and to statically perform as much of the expansion/reduction process as possible. This is our goal. We define an abstract machine, suitable for executing CRSs, and we give a translation from rewrite rules into instruction sequences for this machine.

We do not address the problems arising from considering CRSs as a programming language; we look at rewriting in the (very) small, not in the large. Apart from the problem of sheer size, the main reason for concentrating on this issue is the close similarity to corresponding problems in related systems. A CRS operates by performing the same set of elementary actions *when checking the applicability of a rule* as a first-order TRS with some notion of variable binding does. There is only one elementary action that does not exist in the first-order case, the equality check corresponding to a non-initial occurrence of a second-order variable, but even this is not too different.

I wrote the implementation originally in C (for applicative CRSs), but for the sake of presentation I use Standard ML in this paper, see [16, 17]. To shorten the code I shall treat several functions as predefined, i.e. I use — without further explanation — several functions which are not primitive in Standard ML, but are standard in the Functional Programming community. Definitions for all these functions can be found in [17], appendix 2. However, I redefine `select n` to be `hd o (drop n)`, i.e. I index lists starting from 0 (instead of from 1). The ML code for these functions can be obtained by anonymous ftp from `ftp.dcs.ed.ac.uk`, directory `pub/smk/CRS`, file `reade.ml`; the file `compile.ml` contains an extended version of the ML implementation presented in this paper.

2 Preliminaries: Combinatory Reduction Systems

First, we recall the basic definitions for Combinatory Reductions Systems. Instead of giving the usual "mathematical textbook" style definitions, we express these definitions in ML. Another difference from the usual presentation [8] is that we consider CRS terms in "de Bruijn style" [2] with variable names being replaced by natural numbers.

The *alphabet* of a CRS is determined by two types `mvar` and `symbol`, the elements of which we call *metavariables* and *symbols*, respectively. For the purposes of this paper, we take them to be strings of characters, i.e. we fix a (sufficiently large) CRS alphabet.

```
type mvar = string and symbol = string
```

Given a CRS alphabet `A`, we can define the *terms* over it as the values of type `term`:

```
datatype term =
        Var of int | Sym of symbol * term list |
        Meta of mvar * term list | Abst of term;
```

Usually, symbols and metavariables come equipped with an arity function, such that a term of the form `Sym(s,ts)` is only well-formed if the arity of `s` is equal to the length of `ts`. We do not impose this restriction here and allow overloading of symbols and metavariables. CRS theory traditionally [9, 8] distinguishes a subclass of terms not containing the constructor `Meta`, restricting the word "terms" for them and calling terms which may contain metavariables "metaterms". This distinction is not essential and we avoid it here.

The following notion of substitution supports replacement of more than one variable at a time. A substitution is determined by a function from natural numbers (de Bruijn indices) to terms.

```
fun subst f (Var x) = f x
 |  subst f (Sym(s,ts)) = Sym(s,map (subst f) ts)
 |  subst f (Meta(s,ts)) = Meta(s,map (subst f) ts)
 |  subst f (Abst t) = Abst (subst (fn n =>
        if n=0 then Var 0 else lift (f(n-1))) t)
and lift t = subst (fn x => Var(x+1)) t
```

This definition is slightly unusual and also rather inefficient, but it is more general than the usual definition (see [3], for example) which uses two recursive functions instead of one. The added generality enables us to easily state the substitution lemma of λ-calculus [1]:

Proposition 1. *Let* `t` *be a term and* `f` *and* `g` *functions of type* `int->term`. *Then:*

$$\texttt{subst f (subst g t)} = \texttt{subst ((subst f) o g) t}$$

This proposition does not even require totality of `f` and `g`, in the sense that `a` = `b` holds iff either the expressions `a` and `b` are both undefined or both are defined and denote equal values. Based on substitution, β-reduction and n-fold β-reduction are defined as follows:

```
fun beta arg (Abst b) = subst (fn 0 => arg | k => Var (k-1)) b
 |  beta _ _ = error "no beta-redex"
val betas = foldright beta
```

Using `betas` for multiple argument β-reduction is rather inefficient, as each argument requires its own β-reduction including (repeated) adjustments of de Bruijn indices. We can characterise the effect of n-fold β-reduction more concisely:

Proposition 2. *Let* `t` *be a term and* `ts` *be a list of terms. If* `k=length ts` *then:*

```
betas (repeat Abst k t) ts   =
subst (fn x => if x<k then select x ts else Var(x-k)) t
```

The functional `repeat` is function iteration: `repeat f n` is the function `f` composed with itself `n` times. Proposition 2 is typical for the kind of benefit we can expect from a compilation of CRSs: the expression on the right-hand side of the equation uses much less adjustments of variable indices, e.g. the terms from `ts` are not adjusted at all.

We have a second notion of substitution, the substitution of metavariables. A metavariable substitution is determined by a *valuation*, a function from pairs of metavariables and their arities to terms. Notice that a valuation is defined here on pairs, because we allow overloaded metavariables. A valuation has to satisfy a further condition: each pair `(z,n)` has to be mapped to a term (its *substitute*) of the form `(repeat Abst n t)` for some `t`, i.e. it "starts" with at least `n` abstractions. The reason for this restriction is the definition of `betas`: an application `betas t xs` is only well-defined if `t` starts with at least as many abstractions as the list `xs` is long.

```
fun metasubst v (Var x) = Var x
 |  metasubst v (Sym(s,ts)) = Sym(s,map (metasubst v) ts)
 |  metasubst v (Meta(z,ts)) =
        betas (v (z,length ts)) (map (metasubst v) ts)
 |  metasubst v (Abst t) = Abst (metasubst (lift o v) t)
```

The definitions of metavariable substitution in the literature [9, 8] do not mention that there is a danger of name capture for metavariable substitution — but there is one; in the above version, this is reflected by the presence of `lift` in the last clause.

For the definition of "CRS rule" we need several auxiliary functions, e.g. for extracting the free/meta- variables of a term.

```
val freevars =
    let fun fv n (Var k) = if k<n then [] else [k-n]
          |  fv n (Sym(_,ts)) = foldright (append o fv n) [] ts
          |  fv n (Meta(_,ts)) = foldright (append o fv n) [] ts
          |  fv n (Abst t) = fv (n+1) t
    in fv 0 end;
fun metas (Var _) = []
 |  metas (Sym(_,ts)) = foldright (append o metas) [] ts
 |  metas (Meta(z,ts)) =
        (z,length ts)::foldright (append o metas) [] ts
 |  metas (Abst t) = metas t;
```

Left-hand sides of CRS rules obey a strong syntactic restriction, the so-called *simplicity condition*. Dale Miller coined this name [12] for arbitrary terms in λ^τ, but the corresponding condition for CRS rules was already present in Klop's thesis [9]; Nipkow [14] calls terms satisfying this property "higher-order patterns". The absence of third-order variables from CRSs simplifies the definition of simplicity for CRS terms. Second-order unification of simple terms is decidable and has most general solutions [12] and so has second-order matching of arbitrary terms against simple terms.

```
fun simple (Var _) = true
 |  simple (Sym(_,ts)) = all simple ts
 |  simple (Meta(_,ts)) =
        all (fn Var _ => true | _ => false) ts andalso
        let fun isset [] = true
              |  isset (x::xs) = not(member xs x) andalso isset xs
        in isset (foldright (append o freevars) [] ts) end
 |  simple (Abst t) = simple t;
```

A *CRS rule* is a pair of terms `p` satisfying the predicate `crs_rule`, i.e. such that `crs_rule p` evaluates to `true`.

```
fun crs_rule (left,right) =
    contains (metas right) (metas left) andalso
    (fn Sym _ => true | _ => false) left andalso
    freevars left = [] andalso freevars right = [] andalso
    simple left;
```

The conditions in the first two lines (of `crs_rule`) are typical restrictions for rewrite systems. CRS rules are required to contain no free variables. This restriction is a consequence of the slightly artificial distinction between metavariables and variables. We could indeed identify variables and metavariables in terms, using de Bruijn indices instead of `mvar` for metavariables etc. For the purposes of this paper, it is useful to keep the distinction.

Assume a fixed CRS alphabet and terms over this alphabet. A *Combinatory Reduction System* consists of a set R of CRS rules. The rewrite relation associated with a CRS is (as usual for rewrite systems) obtained from the set of rules, by interpreting the set of rules as a relation (set of pairs) and closing it under certain properties.

Definition 3. A binary relation $\to$ on terms is called a *rewrite relation*, if we have the following (for arbitrary `t`, `z`, `k` etc.):

$$\begin{array}{l} \texttt{metas t} = \texttt{[(z,k)]} \ \wedge\ \texttt{a} \to \texttt{b} \ \wedge \\ \texttt{context} = \texttt{fn x => metasubst (fn \_ => repeat Abst k x) t} \quad \Rightarrow \\ \qquad \texttt{context a} \to \texttt{context b} \end{array}$$

Notice that any rewrite relation is substitutive and compatible (in the usual sense). Notice also that "rewrite relation" only refers to substitution of *variables*, while CRS rules also contain *metavariables*. We therefore need another property:

Definition 4. A binary relation $\to$ on terms is called *meta-substitutive*, if for arbitrary total functions `g` (of the right type) we have:

$$\texttt{a} \to \texttt{b} \ \Rightarrow\ \texttt{metasubst g a} \to \texttt{metasubst g b}$$

The rewrite relation associated with a CRS R is the smallest rewrite relation containing the meta-substitutive closure of R.

3 Compilation of a CRS

In the following, we restrict our efforts on implementing the meta-substitutive closure of a single rule, i.e. applying a CRS rule at the root of a term. The reasons for this limited effort are:

- A naïve, non-efficient extension to the general case is straightforward (see the `ftp` source). Working with de Bruijn indices makes it unnecessary to treat rewriting within abstractions in any special way; there is no need to "freeze" variables or to adjust indices.
- Eliminating most the inefficiencies of the naïve approach requires techniques known from the implementation of first-order TRSs. However, when I applied such techniques in an implementation of λ-rewriting (see [5]; λ-rewriting systems also require the right-hand sides of rules to be simple), the second-order patterns had no impact on the approach at all. This observation generalises to arbitrary CRSs.
- We are interested in the rewrite relation of a CRS as such, not just in the relation that relates terms to their normal forms. Thinking of CRSs as the kernel of a programming language would be a slightly different undertaking, e.g. it would be worthwhile then to allow conditional CRS rules and to require a constructor discipline.

The implementation of a CRS rule we shall develop is based on the *compilation* of the rule. The CRS rule (the pair of terms) is translated into a different representation, which is more suitable for the computation of CRS matching and metavariable substitution. This representation is an instruction sequence for modifying the state of an abstract machine; the abstract machine is especially designed for the execution of CRS rules.

4 Abstract Machine

The chosen abstract machine for the execution of compiled CRS rules is a stack-based machine, very much in the spirit of abstract machines for term rewriting, or Landin's SECD machine [11] for the λ-calculus.

The most significant difference to the SECD machine is the omission of a *dump*, a stack of states of the machine. The dump is superfluous, because rule application does not depend on other rule applications. Even if we had such a dependency (e.g. conditional CRS rules) and needed a dump, it would not be necessary to store the entire state, as most components are stack-like.

4.1 Machine Components

Instead of carrying the state of the Abstract Machine (the tuple of its components) around, we use the Standard ML state for this purpose. Therefore, all components of the abstract machine are given as references, that is: as updateable pointers to values.

The components of the machine are the following:

– The stack of *current* terms, which contains (subterms of) the term the rule is applied to. We call the top of this stack "the current term".

```
val CURR = ref ([]:term list);
```

– The environment, which represents the matching valuation.

```
val ENV = ref ([]:term list);
```

Notice that the environment is not an association list; the compilation replaces metavariables by relative addresses w.r.t. this environment. Also, entries in the environment are not (necessarily) k-fold abstractions when associated with k-ary metavariables. For the precise correspondence between the environment and the valuation it represents, see section 5.

– The stack which is used to create the instance of the right-hand side of the rule. After successfully executing all instructions corresponding to a rule, the stack contains only one term, the instance of the right-hand side.

```
val STACK = ref ([]:term list);
```

– An offset to adjust de Bruijn indices for free variables.

```
val OFFSET = ref 0;
```

– A stack of numbers to adjust de Bruijn indices for bound variables of the right-hand side of a rule. This is a bit delicate when they occur as arguments of a metavariable.

```
val NSTACK = ref ([]:int list);
```

The rôles of `OFFSET` and in particular of `NSTACK` are rather subtle. The idea behind these components is to allow nested substitutions like $t[u[s/y]/x]$ to be computed outside-in rather than inside-out. This minimises adjustments of variable indices and term traversals. However, there is a problem with implementing outside-in: s may contain a bound variable whose λ-binding is in t and the new index for this variable depends on the occurrences of x and y in t and u. `NSTACK` is used to keep track of such occurrences.

`OFFSET` and `NSTACK` represent a substitution, a certain variable lifting. In any given state of the abstract machine, we can retrieve it as follows:

```
fun offset () =
    let val off = !OFFSET and nst = !NSTACK
    in  subst (fn x=>
            if x < length nst then Var(x+select x nst)
            else Var(x+off))
    end
```

Initially, `OFFSET` is set to 0 and `NSTACK` is empty, making `offset()` the identity on terms.

The abstract machine also has an implicit component: the *control*, the list of instructions that have yet to be executed. The instructions do not modify themselves, thus it is not necessary to include the control as an explicit component of the machine.

The failure of matching is expressed using the exception mechanism of Standard ML. The function `test` is used to check Boolean expressions that have to be true when matching succeeds.

```
exception failure;
fun test p = if p then () else raise failure;
```

4.2 Machine Instructions

The abstract machine has instructions for matching a term against the left-hand side of a rule and instructions for applying the resulting valuation to the right-hand side. The instructions are the following:

```
datatype instruction =
    IS of string*int | NEXT | ISABST | CHECK of int list | SET |
    EQVAR of int | EQI of int*int*((int*int)list) | EQIMM of int |
    PUSHI of int*int*((int*(instruction list)) list) |
    PUSHIMM of int | PUSHVAR of int | CELL of string*int |
    LAMBDA | ADBMAL
```

If `s` is an instruction then `exec s` changes the state of the abstract machine. Because of its size (14 alternatives), the definition of `exec` below is divided into several parts. Execution and compilation are defined independently.

Matching Instructions The instructions in the first two lines of the definition of type `instruction` are used for computing the matching valuation. Matching manipulates two objects: the stack of current terms and the environment (valuation).

```
fun exec (IS(s,n)) =
        (case !CURR of Sym(s',bs)::ps =>
            (test (s=s' andalso length bs = n); CURR := bs @ ps)
         | _ => raise failure)
 |  exec ISABST =
        (case !CURR of Abst t::ps => CURR:=t::ps
         | _ => raise failure)
 |  exec NEXT = CURR := tl (!CURR)
```

`IS(s,n)` checks that the current term is an application of the symbol `s` to `n` arguments and replaces it by its argument list. `ISABST` is the corresponding instruction for abstractions. Matching fails if the current term does not have the required form. `NEXT` is used to select the next argument. In Higher-Order Rewrite

Systems [13, 6], an ISABST instruction would not have to check anything, as the type system already guarantees that the current term is an abstraction. In a certain sense, ISABST checks for the presence of the symbol Λ introduced by the translation from CRSs into HRSs, see [6, 15].

```
| exec (CHECK vs) =
  let fun check n (Var k) = k<n orelse not(member vs (k-n))
        | check n (Sym(_,bs)) = all (check n) bs
        | check n (Abst t) = check (n+1) t
        | check n (Meta(_,bs)) = all (check n) bs
  in test (check 0 (hd(!CURR))) end
```

The instruction CHECK vs checks that the current term does not contain any of the variable indices in vs. These checks are necessary because of the presence of bound variables, *not* because of the presence of higher-order variables.

```
| exec SET = ENV := hd(!CURR) :: !ENV
```

SET stores the current term on top of the environment. Since the current term is always a subterm of the original current term, the environment does not directly contain the substitutes of k-ary metavariables, but rather the substitutes without the k abstractions and modulo some change of bound variables.

```
| exec (EQVAR n) = (case hd(!CURR) of Var m => test (m=n)
                                     | _ => raise failure)
```

The instruction (EQVAR n) corresponds to a “positive” occurrence of a bound variable with index n in the left-hand side of a rule. Only bound variables outside metavariable applications correspond to EQVAR instructions.

```
| exec (EQI(n,d,vvs)) =
  let fun equal k (Var x, Var y)=
              if x<k then x=y
              else assoc vvs (fn a=>a+d) (x-k) = y-k
        | equal k (Sym(s,bs), Sym(t,at)) =
              s=t andalso eqlist k bs at
        | equal k (Abst a, Abst b) = equal (k+1) (a,b)
        | equal k (Meta(z,bs), Meta(y,at)) =
              z=y andalso eqlist k bs at
        | equal _ _ = false
      and eqlist k xs ys = length xs=length ys andalso
              all(equal k)(pairlists xs ys)
  in test (equal 0 (select n (!ENV),hd(!CURR))) end
| exec (EQIMM n) = test (hd(!CURR) = select n (!ENV))
```

EQI instructions are used for non-left-linear CRS rules to match non-initial occurrences of metavariables. EQIMM n is a cheap version that compares the current term with the term stored earlier at the n-th place in the environment.

EQI(n,d,xs) does the same, but in a more difficult setting when the variables cannot be compared one-to-one; xs is an association list for translating bound variables, and d has to be added to the indices corresponding to free variables.

The instruction EQIMM is redundant in the sense that all that it does can be done with EQI as well. But it cannot be done quite so well, e.g. if terms are uniquely represented (using a cache; see [7]) then the execution of the instruction EQIMM takes constant time, while EQI is linear in the size of the term to which the rule is applied.

Instantiating the Right-Hand Side The instructions in the last three lines of the definition of type instruction are used for generating the instance of the right-hand side of the rule. In particular, they manipulate the STACK which will finally contain this instance.

```
 |   exec (CELL (s,n)) =
         let val (args,rest) = split n (!STACK)
         in STACK := Sym(s,rev args)::rest end
```

CELL(s,n) pops n elements $t_1, \cdots, t_n$ from the stack (n may be 0) and replaces them by the term $\mathtt{s}(t_1, \cdots, t_n)$.

```
 |   exec LAMBDA = NSTACK := 0 :: !NSTACK
 |   exec ADBMAL = let val t::ts = !STACK in
         STACK := Abst t :: ts; NSTACK := tl(!NSTACK) end
```

The instructions LAMBDA and ADBMAL are used to create abstractions. The changes to NSTACK are very often insignificant, but they do matter if the code executed between a LAMBDA and an ADBMAL involves PUSHVAR instructions.

```
 |   exec (PUSHI(n,d,acs)) = push (select n (!ENV)) d acs
 |   exec (PUSHIMM n) = STACK := select n (!ENV) :: !STACK
```

PUSHI(n,d,acs) pushes the term stored at the n-th place of the environment onto the stack, or more precisely: a substitution instance of this term. The substitution is the one corresponding to the k-fold β-reduction that occurs when a k-ary metavariable is meta-substituted; the function push, which creates the substitution instance is defined below. For each occurrence of a free variable, d has to be added to its index. Each occurrence of a (non-local) bound variable invokes an instruction sequence from acs producing a term, i.e. acs associates bound variables to instruction sequences. PUSHIMM is a cheap version of PUSHI, analogous to EQIMM for EQI.

```
 |   exec (PUSHVAR x) =
         STACK := Var(x+ select x (!NSTACK)) :: !STACK
```

PUSHVAR(x) pushes the variable with index x onto the stack. We have to add the x-th component of !NSTACK to that index to adapt it to the context in which it occurs. This will always be 0 if no metavariable intervenes between binding and using occurrence of that variable. For the same reason, the corresponding matching instruction `EQVAR` does not have to bother about `NSTACK` — `EQVAR` is only generated for variables outside metavariable applications.

The evaluation of `push t d acs` pushes a term `subst f t` to the stack, where the substitution function `f` is determined by `d`, `acs` and the state of the abstract machine. The state of the abstract machine is important, because `push` may execute instructions (`push` and `exec` are mutually recursive) from `acs`, including accesses to the environment. This is the above mentioned outside-in strategy of computing nested substitutions.

```
and push t d acs = let
    fun push' n (t as Var x) =
           if x<n then STACK := t::(!STACK)
           else let val bs=(assoc acs (fn k =>[])(x-n));
                in if bs=[] then
                        STACK:= Var(x+d+ !OFFSET)::(!STACK)
                   else (NSTACK := map (fn y=>y+n) (!NSTACK);
                         OFFSET := !OFFSET + n;
                         map exec bs;
                         OFFSET:= !OFFSET - n;
                         NSTACK := map (fn y=>y-n) (!NSTACK))
                end
     |  push' n (Abst t) = (push'(n+1) t;
              let val b::rs= !STACK in STACK:=(Abst b)::rs end)
     |  push' n (Sym(s,bs)) =
              (map (push' n) bs; exec(CELL(s,length bs)))
     |  push' n (Meta(z,bs)) = (map (push' n) bs;
               let val (args,rest) = split (length bs) (!STACK)
               in STACK:= Meta(z,rev args) :: rest end)
    in push' 0 t end;
```

The most interesting part of `push` is the treatment of variables. Local bound variables (`x<n`) are pushed unchanged. Free variables (not found in the association list) are pushed with a slight change, their variable index is increased by `d`. This is the statically known difference in number of surrounding abstractions of the first and the current occurrence of the metavariable corresponding to this call of `push`. Other bound variables are associated with an instruction sequence `bs` from `acs`. For the execution of such an instruction sequence, all entries of `NSTACK` have to be increased by the number of abstractions that surround the occurrence of this bound variable in `t`.

A CRS rule corresponds to a sequence of instructions. To apply a CRS rule to a term we push the term onto `CURR` and execute the instructions:

```
fun run t cs = (CURR := t :: !CURR; map exec cs;
                case !STACK of r::rs => (STACK:=rs; [r]))
    handle failure => [];
```

The function `run` returns a singleton list containing the rewrite result if rule application succeeds and the empty list otherwise — the exception `failure` indicates the non-existence of a matching substitution.

It is not always meaningful to apply `run t` to an instruction sequence, because several instructions (e.g. `CELL(s,n)`) assume the abstract machine to be in a certain state. However, execution and compilation should fit together in the sense that they implement the meta-substitutive closure of a CRS rule. For the compilation function `compile` we are going to define, the following proposition should hold: Let $\texttt{l} \to \texttt{r}$ be a CRS rule, `t` and `u` be CRS terms, then

$$\exists\texttt{f}.\,\texttt{metasubst f l} = \texttt{t} \wedge \texttt{metasubst f r} = \texttt{u} \quad\Rightarrow$$
$$\texttt{run t (compile(l,r))} = \texttt{[u]}.$$

This is the completeness of compilation, i.e. the ability to rewrite each redex. We also want a soundness property, the non-ability to rewrite any non-redex:

$$\neg\exists\texttt{f}.\,\texttt{metasubst f l} = \texttt{t} \quad\Rightarrow\quad \texttt{run t (compile(l,r))} = \texttt{[]}$$

Complete code is almost by default sound; the only likely sources of a soundness violation are overly weak `CHECK` instructions or a wrong treatment of non-left-linear rules, and indeed for certain CRS rules completeness implies soundness.

5 Symbol Table

The compilation function presented later uses a symbol table for the metavariables of a rule. It has a similar purpose as the symbol table (for identifiers) used in an implementation of a programming language. The symbol table is a list of entries, each entry having the following form:

```
type entry = { MV: mvar*int, BV: int,
               LOC: int ref, ARG: int list };
fun lookup z (e::es : entry list) =
        if #MV(e)=z then e else lookup z es
 |  lookup z [] = error "internal error"
```

Type `entry` is an SML record type; each entry corresponds to a metavariable in a CRS rule. `MV` is its name plus arity, `BV` is the number of abstractions at its first occurrence, which is essential because of the representation of variables as de Bruijn indices. `LOC` is the location of the metavariable in the environment. `ARG` represents the list of arguments of the first occurrence — it can be given as an integer list, because the restrictions for CRS rules ensure that it is a list of bound variables, hence a list of de Bruijn indices. For each element `e` of type entry we assume that the arity and the length of the argument list are equal, i.e. `#2(#MV e)` = `length(#ARG e)`.

```
val unvar = map (fn (Var x) => x | _ => error "lhs not simple");
```

```
fun update n t (Var x) = t:entry list
 |  update n t (Sym(s,ts)) = foldleft (update n) t ts
 |  update n t (Abst m) = update (n+1) t m
 |  update n t (Meta(z,ts)) =
        let val mv = (z,length ts)
            fun enter [] =
                    [ {LOC=ref 0, MV=mv, BV=n, ARG=unvar ts} ]
              |  enter (tab as (e as {MV=y, ...})::es) =
                    if mv=y then tab else e::enter es
        in foldleft (update n) (enter t) ts end;
```

The function `update` traverses a simple term t and updates the symbol table, storing all initial occurrences of metavariables in t that are not already in the table. If the local function `enter` is applied to an empty symbol table, then we have an initial occurrence of a metavariable and we create a new entry. By the restrictions on CRS rules it is guaranteed that the argument list `ts` of an initial occurrence is always a list of variables, i.e. the error in `unvar` does not arise.

```
fun create_table l =
    let fun locations rn ({LOC=rl, ...}:entry) =
                    (rl:= !rn; rn:= !rn+1);
        val tab = update 0 [] l
        val no = ref 0
    in map (locations no) (rev tab); (tab,no) end
```

The function `create_table` generates the symbol table for a CRS rule $\mathtt{l} \to \mathtt{r}$. It traverses the left-hand side of the rule, and then assigns a relative address to each metavariable. A symbol table for a CRS rule $\mathtt{l} \to \mathtt{r}$ contains all metavariables occurring in the rule, because they all have to occur in the left-hand side `l`; each metavariable in a symbol table of length n is associated with a unique location between 0 and $n-1$. This location is the relative address (in `!ENV`) of the substitute of the metavariable after matching has succeeded.

The combination of a symbol table, which uses locations between 0 and $k-1$, and an environment `!ENV` of length at least k represents a valuation defined on the entries of the symbol table.

```
fun getval tab = fn z =>
        let val e = lookup z tab
            val k = length(#ARG e)
            val su = assoc
                    (pairlists (#ARG e)(map Var (0 upto (k-1))))
                    (fn m => Var(m + k - #BV e))
        in  repeat Abst k (subst su (select (!(#LOC e)) (!ENV)))
        end;
```

The environment always carries subterms of the initial current term. This is simpler than to replace them by the proper substitutes of metavariables. But this means that we always make an assumption about the connection between symbol table and environment:

Definition 5. Let `tab` be a symbol table. An environment `ENV` *covers* a pair `(mv,n)` of a metavariable `mv` with its arity `n` iff

1. `(lookup (mv,n) tab)` is defined (call it `e`);
2. `(select (!(#LOC e))(!ENV))` is defined (call it `a`) and
3. the free variables of `a` are all either greater or equal than `#BV e` or occur in `#ARG e`.

We say that an environment *covers a term* if it covers all metavariables occurring in it.

Similarly as in proposition 2, we can characterise the n-fold β-reduction that corresponds to substitution of metavariables:

Lemma 6. *Let* `tab` *be a symbol table,* `mv` *be a metavariable, and* `ts` *be a list of terms, such that* `(mv,length ts)` *is covered by* `ENV`*. For arbitrary natural numbers* `n` *we have then:*

```
metasubst (repeat lift n o getval tab) (Meta(mv,ts))   =
subst (assoc(pairlists(#ARG e)ts')
        (fn m=>Var(m + n - #BV e))) (select(!(#LOC e))(!ENV))
```

where `ts'` *abbreviates* `map (metasubst(repeat lift n o getval tab)) ts`.

Proof.
```
metasubst (repeat lift n o getval tab) (Meta(mv,ts))
= betas (repeat lift n (repeat Abst k
                (subst su (select(!(#LOC e))(!ENV)))) ts'
= { fun lift' n k = subst (fn x=>if x<k then Var x else Var(x+n)) }
betas (repeat Abst k (lift' n k
                (subst su (select(!(#LOC e))(!ENV)))) ts'
= subst (fn x=>if x<k then select x ts' else Var(x-k))
          (lift' n k (subst su (select (!(#LOC e))(!ENV))))
= subst(fn x=>if x<k then select x ts' else Var(x+n-k))
          (subst su (select (!(#LOC e))(!ENV)))
= { fun f' x = if x<k then select x ts' else Var(x+n-k) }
subst ((subst f') o su) (select(!(#LOC e))(!ENV))
```
We now simplify the substitution function:
```
subst f' o su
= assoc(pairlists(#ARG e)(map (subst f')(map Var(0 upto(k-1)))))
          (subst f' o (fn m=>Var(m+k-#BV e)))
= assoc(pairlists(#ARG e)(map f' (0 upto (k-1)))))
          (fn m=>f'(m+k-#BV e))
= assoc(pairlists(#ARG e)ts')(fn m=>f'(m+k-#BV e))
```
We can restrict any substitution function to the free variables of the term it is applied to. By the "cover" assumption about the environment, we know that `m` is greater or equal than `#BV e` whenever `assoc` uses its default function. This allows us to simplify `f'(m+k-#BV e)` to `Var(m+n-#BV e)`. □

The natural number `n` in `repeat lift n` corresponds to the number of abstractions that surround a metavariable occurrence. This number is statically known: the component `d` in an instruction `PUSHI(k,d,acs)` is the difference `(n-#BV e)`. This static information is useful to detect the case in which the substitution the lemma describes is the identity substitution; see the section on optimisation.

The components `OFFSET` and `NSTACK` of the abstract machine are used for a different variable lifting, the offsets of which are not statically known. The idea is not to compute the list `ts'` in advance and then to lift its variable indices whenever the substitution of the lemma is applied to an abstraction, but rather to collect all lifting information before instantiating `ts`.

6 Generating Code

Code generation for left-hand and right-hand sides of a rule are fairly independent — the only dependency between the two is the symbol table, since it gives relative addresses of metavariable substitutes in the environment.

6.1 Generating Match Code

The generated match code is very similar to how an interpretative implementation of CRS rewriting would operate.

```
fun bvset n ts = filter (not o member (unvar ts)) (0 upto (n-1));
fun lhs tab n k =
    let fun lhs' (Sym(s,ts)) = IS(s,length ts) ::
                foldleft(fn cs=>fn t => cs @ lhs' t) [] ts
          | lhs' (Var x) = [EQVAR x, NEXT]
          | lhs' (Abst t) = ISABST::lhs tab (n+1) k t
          | lhs' (Meta(z,ts)) =
                let val {LOC=ref l, ARG=ns, BV=no, ... } =
                    lookup (z,length ts) tab
                in if !k>l then
                       (k:= !k-1; [CHECK(bvset n ts),SET,NEXT])
                   else [EQI(l - !k,n-no,
                       pairlists ns (unvar ts)),NEXT]
                end
    in lhs' end;
```

The code for the initial occurrence of a metavariable checks for illegal name capture (`CHECK`) and stores the current term in the environment (`SET`); the `CHECK` establishes the precondition for environment components used in lemma 6. At a non-initial occurrence, the earlier stored term has to be compared with the current one. Which occurrence is considered to be the initial one depends on the term traversal — we traverse the term from left to right (for no good reason). Executing (successfully) the code of `lhs tab n k` means to remove one term from the stack of current terms.

Lemma 7. *Let* `t` *be a simple and closed term.*
Let `(tab,rn)=create_table t`*. Let* `f` *be a valuation defined on the metavariables in* `t`*. Then* `f` *is pointwise the same as*

```
(CURR:=metasubst f t :: !CURR;
map exec (lhs tab 0 rn t);
getval tab)
```

and the environment `ENV` *covers* `t` *w.r.t.* `tab` *after the evaluation of this expression.*

Proof. By induction on the structure of `t`. Sketch: we have to prove a more general lemma, because most of the required properties are not closed under taking subterms. In particular: `t` is simple and does not contain free variables greater or equal than `n`; `tab` is a symbol table containing all metavariables of `t`; `rn` refers to a natural number, such that for all entries `e` in `tab` of metavariables occurring in `t` the following holds: either `!(#LOC e)<!rn`, or the environment covers `#MV e`. To get the right valuation using `getval`, we have to compose `getval tab` with `repeat lift n`, and we also have to place `!rn` arbitrary terms on top of the environment after executing the code from `(lhs tab n rn t)`; after evaluating this expression to an instruction sequence containing k `SET` instructions, `!rn` is reduced by k. □

Lemma 7 states the completeness of our matching procedure. We can also claim a soundness property (with a very similar proof), i.e. that execution raises the exception `failure`, if the current term is not a valuation instance of `t`.

6.2 Generating Code for the Right-hand Side

Code generation for the right-hand side is slightly simpler, because we do not have to distinguish between initial and non-initial occurrences of metavariables. The complications concerning possible nesting of metavariable applications arise at run-time, not at compile-time.

```
fun rhs tab n =
    let fun rhs' (Sym(s,ts)) =
                foldright append [CELL(s,length ts)] (map rhs' ts)
          |  rhs' (Var x) = [PUSHVAR x]
          |  rhs' (Abst t) = [LAMBDA] @ rhs tab (n+1) t @ [ADBMAL]
          |  rhs' (Meta(z,ts)) =
                let val {LOC = ref l, ARG=ns, BV=m, ...}
                      = lookup (z,length ts) tab
                in  [PUSHI(l,n-m,pairlists ns (map rhs' ts))]
                end
    in rhs' end
```

For code generation with `rhs`, we can make the following claim:

Lemma 8. *Let* `r` *be a term and* `tab` *be a symbol table such that the environment covers* `r`*. Let* `n` *be the length of* `!NSTACK`*. If all free variables in* `r` *are smaller than* `n`*, then the evaluation of* `map exec (rhs tab n r)` *puts a term* `r'` *on top of* `STACK` *and leaves all other components of the abstract machine unchanged, where* `r'` *is equal to*

```
offset()(metasubst (repeat shift n o getval tab) r).
```

Proof. By induction over the term structure of `r`. Sketch: Symbol applications and variables are trivial; for abstractions notice that `offset()(Abst t)` is equivalent to `Abst(#2(exec LAMBDA, lift(offset() t), exec ABDMAL))`, see the definition of `subst`. For metavariables we can use lemma 6 and proposition 1 to characterise the `metasubst` application and its composition with `offset()` as a single substitution. It then remains to show that the evaluation of the expression `(push t d (pairlists ns (map (rhs tab n) ts)))` stores the term we obtain from `(subst(assoc(pairlists ns us)(fn m=>Var(m+d+!OFFSET)))t)` on top of `STACK`; in this expression, `us` is shorthand for pointwise applying `offset() o (metasubst ...)` to the list `ts`. This requires again an inductive proof on the term structure of `t`; notice here that the evaluation of

```
(OFFSET:=!OFFSET+k; NSTACK:=map(fn x=>x+k)(!NSTACK);
map exec (rhs tab n u);
OFFSET:=!OFFSET-k; NSTACK:=map(fn x=>x-k)(!NSTACK))
```

is equivalent to

```
(map exec (rhs tab n u);
STACK:=repeat lift k(hd(!STACK)) :: tl(!STACK))
```

Notice further that `rhs` never returns an empty list; therefore, using `(fn x=>[])` as default function for `assoc` (see the definition of `push`) is a proper way to distinguish free variables. □

The proof requires two nested inductions, because I have chosen a lazy compilation scheme for outside-in computation of nested substitutions. An eager compilation scheme for computing them inside-out would generate the instances of the argument list `ts` of a metavariable on the stack.

6.3 Full Compilation

Compiling a rule simply involves of compiling its parts and concatenating the code afterwards.

```
fun compile (l,r) =
    let val (tab,rn) = create_table l
    in  lhs tab 0 rn l @ rhs tab 0 r
    end;
```

Completeness of compilation is now easily established:

Theorem 9. *Let* `(l,r)` *be a CRS rule and* `f` *be a valuation defined on the metavariables in* `l`. *Let* `!NSTACK=[]` *and* `!OFFSET=0`. *Then:*

$$\texttt{run(metasubst f l)(compile(l,r))} = \texttt{[metasubst f r]}.$$

Proof. Immediate by distributivity of `map` over `@` and lemmas 7 and 8, taking $\mathtt{n} = 0$. □

Similarly, soundness of compilation follows from the soundness of matching.

6.4 An Example

To get a feeling for the code the compilation produces, let us look at an example. Here is the chain rule of symbolic derivation, for the sake of legibility represented as a nameful CRS rule, i.e. with variable names instead of de Bruijn indices.

$$D([x]App(f, g(x))) \ \rightarrow \ M(D([x]g(x)), B(D(f), [x]g(x)))$$

Explanation: x is the only variable, f and g are metavariables, and the symbols D, B, App, and M can be interpreted as follows: D is the derivation operator, B is function composition, App is function application, and M is multiplication of functions (pointwise). The substitute for f cannot contain x freely; this is the usual side-condition of the chain rule, which is here implicit, as f is a nullary metavariable. Applying the function `compile` to the above rule (after converting it to a pair of CRS terms, of course) we obtain the code in figure 1.

1 `IS(D,1)`	10 `LAMBDA`
2 `ISABST`	11 `PUSHI(0,0,[(0,[PUSHVAR 0])])`
3 `IS(App,2)`	12 `ADBMAL`
4 `CHECK [0]`	13 `CELL(D,1)`
5 `SET`	14 `PUSHI(1,-1,[])`
6 `NEXT`	15 `CELL(D,1)`
7 `CHECK []`	16 `LAMBDA`
8 `SET`	17 `PUSHI(0,0,[(0,[PUSHVAR 0])])`
9 `NEXT`	18 `ADBMAL`
	19 `CELL(B,2)`
	20 `CELL(M,2)`

Fig. 1. Code of a compiled rule

The first 9 instructions (left column) were generated by `lhs`, i.e. their purpose is to match the left-hand side of the rule; the remaining 11 instructions are for the creation of an instance of the right-hand side. The code mimics exactly what an interpreted rule application would do, only the term structure of the left-hand and right-hand sides of the rule has been flattened into a list of instructions and the names of metavariables have been replaced by relative addresses.

7 Optimisation

The code generated by the naïve compilation can be improved in many ways. Most of these improvements are minor, they allow to compactify the code, sometimes requiring extensions to the instruction set to make better use of the resources of the abstract machine. We do not have the space here to elaborate on that; several optimisations can be found in the `ftp` source.

One kind of optimisation has a major effect: it would be nice to "compile away" as many second-order substitutions as possible. An important observation here is that the (named) term $(\lambda x.t)x$ can be β-reduced to t, but that it is a waste of resources to perform this β-reduction in the standard way. This waste of resources is the difference between the complexities $\mathcal{O}(n)$ and $\mathcal{O}(1)$ of executing a `PUSHI` and `PUSHIMM` instruction, respectively.

Such wasteful β-reductions typically occur in λ-rewriting systems, i.e. in CRSs which only move bound variables around rather than replacing them, e.g. symbolic derivation, translation of λ-calculus into Combinatory Logic, etc. Similarly to the way in which any HRS can be translated into a CRS plus β-reduction (see [15]), any CRS can be translated into a λ-rewriting system plus β-reduction; however, β-reduction is expressible within CRSs but not within λ-rewriting systems.

```
fun single (c as EQI(n,0,xs)) =
        if all (op =) xs then EQIMM n else c
 |  single (c as PUSHI(n,0,acs)) =
        if all (fn (m,xs)=> xs=[PUSHVAR m]) acs
        then PUSHIMM n else c
 |  single c = c
```

The function `single` maps an instruction to an equivalent instruction, being exactly the mentioned detection of trivial β-redexes. Notice that the `d` component, the difference in surrounding abstractions between the current and initial occurrence of the metavariable these instructions correspond to, is required to be 0 — otherwise we need a term traversal to adjust indices of free variables. But even with `d` $= 0$ we also need `OFFSET` to be set to 0 for the same reason, and `NSTACK` to contain only zeros for a similar one. We can say that `exec c` and `exec(single c)` have the same effect on the state of the abstract machine, provided `!OFFSET=0` and `all(fn x=>x=0)(!NSTACK)`. Initially, `OFFSET` and `NSTACK` satisfy these conditions; moreover they are only ever locally violated within the function `push`. Thus, we can apply `single` to all instructions on the outermost level, i.e. to those that do not occur within a `PUSHI` instruction.

This optimisation is applicable to the example in figure 1; instructions 11 and 17 can both be replaced by `PUSHIMM 0`. These instructions correspond to right-hand side occurrences of the second-order metavariable g. The corresponding optimisation for f (instruction 14) is not possible, for we have to decrease de Bruijn indices of free variables in its substitute by 1.

8 Conclusion

We have defined an abstract machine for executing Combinatory Reduction Systems and a compiler for translating CRS rules into instructions of that machine. The correctness of this translation has been established. Nearly all important actions of the system are performed on the low level of instructions — the only exception being the function `push` which provides an interaction between an interpretative term traversal and the execution of code. The code can be seen as a linearisation of the actions one would perform in a similar way when *interpreting* the rewrite rule.

References

1. Hendrik P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics.* North-Holland, 1984.
2. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.
3. T. Hardin. How to get confluence for explicit substitutions. In M.R. Sleep, M.J. Plasmeijer, and M.C.J.D. van Eekelen, editors, *Term Graph Rewriting*, chapter 3, pages 31–45. John Wiley & Sons, 1993.
4. Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
5. Stefan Kahrs. *λ-rewriting.* PhD thesis, Universität Bremen, 1991. (in German).
6. Stefan Kahrs. Context rewriting. CTRS'92, pages 21–35. LNCS 656.
7. Stefan Kahrs. Unlimp – uniqueness as a leitmotiv for implementation. PLILP'92, pages 115–129. LNCS 631.
8. J.R. Kennaway. Sequential evaluation strategies for parallel-or and related reduction systems. *Annals of Pure and Applied Logic*, 43:31–56, 1989.
9. Jan Willem Klop. *Combinatory Reduction Systems.* PhD thesis, Centrum voor Wiskunde en Informatica, 1980.
10. Jan Willem Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbai, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, pages 1–116. Oxford University Press, 1992.
11. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
12. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Extensions of Logic Programming*, pages 253–281, 1991. LNCS 475.
13. Tobias Nipkow. Higher order critical pairs. LICS'91, pages 342–349.
14. Tobias Nipkow. Functional unification of higher-order patterns. LICS'93.
15. Vincent van Oostrom and Femke van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. HOA'93. (This volume).
16. Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.
17. Chris Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

This article was processed using the LaTeX macro package with LLNCS style