CRSX Howto

Kristoffer Rose
IBM Thomas J. Watson Research Center
http://www.research.ibm.com/people/k/krisrose

Abstract

CRS stands for *Combinatory Reduction Systems* which is a powerful rewriting formalism invented by J. W. Klop [2, 3]. The idea of CRS is to allow rewrite rules that support matching of variable binding constructs and substitution as part of rewrite steps, collectively known as *higher-order rewriting*. CRSX implements CRS with a number of extension to facilitate using CRS for writing compilers and similar transformation systems.

This HOWTO explains how to use CRSX. The latest version of this document[1] can be found at `http://crsx.sf.net` along with some examples of use and all source code, distributed under the terms of the Common Public License version 1.0.

1 Quick start guide

Get the latest *crsxV.zip* file (this guide assumes $V = 7$) from `http://sourceforge.net/projects/crsx`. Unzip it to obtain the *crsx* directory and change into that directory. Run the command

```
java -jar crsx.jar
  rules=samples/derivation/derivation.crs.html
  term="D[&lambda;x.Ln(x+1)]"
```

(on a single line, substituting your proper Java 5+ execution command), and check that the following is output:

```
(&lambda; x . 1 / (x + 1))
```

This was using a CRS for symbolic derivation to prove that the derivative of the function with term λx.Ln(x+1) is λx.1/(x+1), *i.e.*, that the derivative of the function $ln(x + 1)$ (of x) is $\frac{1}{x+1}$. The system is presented in Section 3.2.

[1] This is revision 1.39 of April 14, 2010.

Contents

2 Gentle introduction

Here we'll introduce CRS *terms*, *first order rewriting*, *meta-terms*, and *higher-order rewriting*, including the fundamental rules for composing and running rewrite systems. The reference manual below will give a fuller account of the details.

2.1 Terms

Here are some sample basic terms in CRSX notation:

1. +[2,2]
2. Cons[1, Cons[2, Cons[3, Nil]]]

3. λ[x.@[x,x]]

4. @[λ[foo.foo], S]

The examples illustrate some basic rules:

- Capitalized words (such as Cons and Nil), numbers (such as 2), and most other symbols (such as +, @, and λ) are *constructors* which take an optional ordered (or *positional*) parameter list of subterms in immediately following []s. Special symbols are actually written as HTML entity references, *e.g.*, λ is really written as `&lambda;`.

- Uncapitalized words (such as x and foo) denote *variables*.

- Constructor can take special *binding* parameters which combine some variables followed by a . (dot) and the subterm in which this variable is "bound," *i.e.*, is allowed to occur. The third term should be read as "a λ-construction with a single subterm that binds the variable x (before the .) and only contains a single reference to x."

Variables are really useful when defining binding structures, however, since the notation is completely general it needs to use explicit scoping which is sometimes different. Many languages, for example, include a grammar production like the following,

$$\mathrm{E} ::= \textsf{let } \mathrm{x} := \mathrm{E}_1 \textsf{ in } \mathrm{E}_2$$

where the scope of the variable, x, is the let "body", E_2. Such a construct would most naturally be modeled by CRS terms like $\mathrm{Let}[E_1, x.E_2]$, placing the variable binding such that the scoping is explicit.

2.2 First order rewriting

Rewriting is *first order (term) rewriting* if it does not involve manipulation of any structures that include their own bound variables. Here is a collection of rules that implement addition with unary numbers, so-called Peano arithmetic:

```
Peano[(
  PlusS: +[S[#1],#2] → S[+[#1,#2]] ;
  PlusZ: +[Z,#2] → #2 ;
  TimesS[Copy[#2]]: *[S[#1],#2] → +[*[#1,#2],#2] ;
  TimesZ[Discard[#2]]: *[Z,#2] → Z ;
)]
```

The Peano system illustrates these conventions:

- The rule system itself is written as a construction where the name does not matter but the single argument should be a `()`-enclosed sequence of *rules* each terminated by `;` (semicolon).

- Each rule has the form

  $$\mathit{name}[\mathit{options}] : \mathit{pattern} \rightarrow \mathit{contraction}$$

  where the *name* should be a constructor and the *pattern* and *contraction* should be terms. (The *options* are discussed below, and the → is actually written as the `&rarr;` HTML entity.)

- The pattern of a rule must be a construction term, in this case all the rules rewrite +-constructions.

- Names containing # are special "pattern variables," or *meta-variables*, that are used to match arbitrary subterms at the indicated position.

- Rules should generally contain each meta-variable exactly once in the pattern and once in the contraction. If a meta-variable occurs twice in the contraction, then what it matches is effectively *copied*, which should be indicated by giving the rule name a special Copy option as shown for the TimesS rule. If a meta-variable is used in the pattern but not in the contraction, then what it matches is effectively *discarded*, which must be indicated, as shown for the TimesZ rule.[2]

The Peano system can rewrite

$$\begin{aligned}
&\underline{+[S[S[Z]],S[S[Z]]]}\\
&\rightarrow_{\text{Peano-PlusS}}\ S[\underline{+[S[Z],S[S[Z]]]}]\\
&\rightarrow_{\text{Peano-PlusS}}\ S[S[\underline{+[Z,S[S[Z]]]}]]\\
&\rightarrow_{\text{Peano-PlusZ}}\ S[S[S[S[Z]]]]
\end{aligned}$$

(where each term has the relevant redex underlined and each rewrite arrow is annotated with the full name of the used rule), corresponding to the fact that $2+2=4$. The final term cannot be further rewritten and is called a *normal form*; the default for CRSX is to *normalize* by rewriting until a normal form is reached, *i.e.*, no further possibilities for rewriting exist.

Notice that the order of these rewrites is not deterministic: each step is only guaranteed to locate *some* rewritable subterm (or "redex") and contract with the appropriate rule. So if provided

[2] Requiring such pedantry may seem like overkill but catches an amazing number of errors in rules.

with term ∗[+[S[Z],S[Z]],+[S[Z],S[Z]]], either of the +-subterms may be subject to the first rewrite. If the first is chosen, the rewrite with the PlusS rule gives +[∗[+[Z,S[Z]],+[S[Z],S[Z]]],S[Z]], which permits rewriting of either of the three +-subterms, *etc.*

For some rule systems there are terms where the choice of which rule to apply can influence the normal form, or even whether a normal form exists at all: we detail later the actual rules used and how to deal with such non-confluent systems.

Finally, functional programmers should notice that rules are applied freely under binders. So the above system can also rewrite

Let[Z, v.+[S[S[Z]],S[S[v]]]]
→Peano-PlusS Let[Z, v.S[+[S[Z],S[S[v]]]]]
→Peano-PlusS Let[Z, v.S[S[+[Z,S[S[v]]]]]]
→Peano-PlusZ Let[Z, v.S[S[S[S[v]]]]]

where the final term is the Peano normal form so far since the Peano system as presented here has no rules for substituting the value Z for the variable v.

2.3 Higher order patterns

The CRS formalism adds some special terms to first-order terms, called *meta-applications*, which generalize meta-variables explained above by adding parameters; the generalized forms are called meta-terms when the distinction needs to be made. Here is a meta-term with two meta-applications, a simple meta-variable, $\#_1$, as above, and a proper *meta-application pattern*, $\#_2$[x]:

Let[$\#_1$, x.$\#_2$[x]]

This is actually a pattern that will match Let-terms discussed above, where

- the constructor must be Let,
- there must be exactly two subterms,
- the first subterm cannot have any bindings and will be matched by $\#_1$,
- the second subterm must have a single variable binder which will be matched by x, and
- the second subterm under the binder can be anything wherein the variable matched by x may occur, and is matched by $\#_2$ where we keep track of all the actual occurrences of the bound variable.

Meta-terms with variable bindings are used to form higher-order rewrite rules. Think of $\#_2$[x] as a way to express a matched subterm where we keep track of all references to occurrences of the bound variable x.

Specifically, we say that the pattern Let[$\#_1$, x.$\#_2$[x]] matches the *redex* Let[Z, v.S[S[S[S[v]]]]] by mapping the individual components as follows:

- the meta-variable $\#_1$ in the pattern maps to Z in the redex;
- the bound variable x in the pattern maps to v in the redex;
- the meta-variable $\#_2$ in the pattern maps to a so-called *substitute* function generated from the redex, which we can write as [v]↦S[S[S[S[v]]]], or simply $\#_2$[v]↦S[S[S[S[v]]]].

In CRS such a collection of components for a succesful match is called a *valuation*. The general rule is that in higher order patterns all meta-applications must apply a meta-variable to the list of all the distinct bound variables. (There are exceptions discussed later.)

2.4 Higher order rewriting

CRSX higher-order rewriting is rewriting with rules that involve binders and *substitution*, which here merely means applying substitute components obtained as discussed above.

One such rule would be the one for evaluating a "Let" expression from above:

Let[Copy[$\#_1$]] : Let[$\#_1$, x.$\#_2$[x]] → $\#_2$[$\#_1$] ;

The Let rule is composed as follows:

- The name is Let, and we have indicated that what matches $\#_1$ may be copied by rewriting (see below).
- The pattern is the part that follows the :, which matches $\#_1$ as well as $\#_2$ where we furthermore keep track of all occurrences of the actual variable that matched x, as discussed above.
- The contraction follows the → and constructs a new term by exploiting the way we kept track of occurrences of the bound variable matched by x to substitute all occurrences of the variable: $\#_2$[$\#_1$] means "a subterm like what matched $\#_2$ except we have inserted a copy of $\#_1$ instead

of the variable that occurs in all the places we tracked."

The replacement of a variable in one part of the contraction with copies of another matched subterm is what we call substitution, and when a term can be used for substitution we must permit that it is copied with a Copy[] annotation on the rule, as shown.

If we add the Let rule to the Peano system then it can rewrite

Let[Z, v.S[S[S[S[v]]]]] $\rightarrow_{\text{Peano-Let}}$ S[S[S[S[Z]]]]

This works by first constructing the valuation for the match above, and then generating the contraction $\#_2[\#_1]$ which amounts to substitute (a copy of) what matched $\#_1$, namely Z, in every position marked by v in S[S[S[S[v]]]]. (Or, formally, we have applied the meta-level function [v]$\mapsto$S[S[S[S[v]]]] to the argument Z.)

In order for contraction to be well defined, meta-applications in the contraction should have the same arity as the corresponding meta-application with the same meta-variable in the pattern.

In general a CRSX rewrite system takes the form

name`[(` *rule...rule* `)]`

where the *name* is some constructor naming the system, and each *rule* in turn has the form

name`[`*options*`]` `:` *pattern* `&rarr;` *contraction* `;`

where the individual components have been discussed above.

2.5 Syntactic convenience

The basic term notation turns out to be rather klunky, especially for large terms, so the CRSX system permits using syntactic sugar.

Specifically, the system has the following built in abbreviations:

- Parentheses can be freely used to make grouping explicit.
- Simple concatenation, as in "1 2 3", is short for "applicative" left-recursive use of the special constructor @, *i.e.*, "(1 2 3)" is the same as "@[@[1,2],3]".
- Infix use of ";" (semicolon) is really short for right-recursive use of the special symbol $Cons with the additional rule that any empty segment is represented as $Nil. So, for example, "(1;2;;3;)" becomes "$Cons[1,$Cons[2,$Cons[$Nil,$Cons[3,$Nil]]]]". (This also explains how entire rewrite systems are actually terms subject to rewriting.)
- Finally, concatenating a constructor with some binders has special meaning: "C x y z.*t*" is short for "C[x.C[y.C[z.*t*]]]", *i.e.*, the constructor is inserted for every nested binder. So to get a constructor over a single argument with three binders one must write "C[x y z.*t*]".

The parser also permits some use of HTML markup, notably,

- most tags are ignored, specifically all upper case tags;
- a few elements, notably blockquote and headings, are skipped in their entirety so useful for comments;
- groups of entities, such as "`&lambda;&beta;`", are permitted constructors;
- subscripts such as "`<sub>1</sub>`", are permitted at the end of constructors, variables, and meta-variables;
- the "` `" and "` `" entities are considered white space;
- as a special case symbols written on the form `<var>x</var>` and `<em>X</em>` are considered variables and meta-variables, respectively, and the single letter tags like `<b>C</b>` make constructors;

with these conventions we can express the canonical higher order rewrite system – the β reduction of the λ calculus – with the following text:

```
<u>&Lambda;&beta;</u>[(
  &beta;[Copy[<em>M</em><sub><var>x</var></sub>]]:
   ((&lambda;<var>x</var>.<em>M</em>[<var>x</var>])
    <em>N</em>)
   &rarr; <em>M</em>[<em>N</em>]
;)]
```

which, in a browser, will display nicely as

$\underline{\Lambda\beta}$[(β[Copy[N]]: $((\lambda x.M[x])\ N) \rightarrow M[N]$;)]

(We return to the λ calculus in some detail in the examples section below.)

2.6 Running CRSX

The basic command line interface for CRSX is the class net.sf.crsx.run.Crsx, which is run in the default configuration by the Java invocation command

crsx `rules=`*crs-file* `term=`*term*

where *crsx* is the command – typically `java -jar crsx.jar`, the *crs-file* should be a file containing a CRSX rewrite system as the ones above. Several other parameters are possible; in each case the entire *key=value* must be passed as a single command line argument.

- `input=`*term-file* – load the term to rewrite from a file.
- `verbose=`*number* – set the verbosity to the number, with higher numbers showing more output.
- `lax` – change some errors into warnings.

(Many more commands are explained later.)

3 Examples

Some complete examples.

3.1 λ Calculus

A more complete version of the type-free λ calculus is provided by the *samples/lambda/lambda.crs.html* file, which contains the text

```
<HTML>
&Lambda;&beta;&eta;[(
<UL>
<LI>
  &beta;[Copy[#<sub>2</sub>]] :
  ((&lambda;x.#<sub>1</sub>[x]) #<sub>2</sub>)
  &rarr;
  #<sub>1</sub>[#<sub>2</sub>] ;
<LI>
  &eta;[Weak[#]] :
  (&lambda;x.#[] x)
  &rarr;
  #[] ;
</UL>
)]
</HTML>
```

which appears in browsers as

Λβη[(

- β[Copy[#₂]] : ((λx.#₁[x]) #₂) → #₁[#₂] ;
- η[Weak[#]] : (λx.#[] x) → #[] ;

)]

First we remark that all the HTML tags in upper case are ignored, allowing for the formatting of the rules.

Second, the system is named Λβη and contains two rules, β and η.

The β rule is the classical application evaluation that rewrites an application (with the implicit constructor @) of a λ subterm to an argument subterm with the corresponding substitution. Let us work through how it rewrites the term ((λa.a)(λb.b)) (which is really @[λ[a.a],λ[b.b]]): Matching constructs the valuation with #₁[a]↦a and #₂↦λb.b. The contraction of #₁[#₂] is then constructed by applying the "function" [a]↦a to the "argument" λb.b, which gives the result λb.b.

The η rule is the classical extensionality reduction that rewrites a λ term that contains an trivial application of the following special form:

- the function part matches #[], which is restricted to match subterms where the bound variable (matched by x) does *not occur* – this is so because if it *could* occur then we would have written #[x], and
- the argument part is a simple occurrence of the bound variable x.

Since the bound variable matched by x does not occur in the function part of the application matched by the η rule, there is no need to substitute anything for it in the contraction, as no occurrences can "escape" in this way. So the underlined part of

λ[y.[λ[z.@[@[y,y],z]]]]

is an η-redex because it matches the pattern of the η rule (# can match @[y,y] where there are no zs, the variable that corresponds to x), and the term rewrites to λ[y.[@[y,y]]]. In contrast, the term λ[y.λ[z.@[@[y,z],y]]] does not match the rule.

3.2 Symbolic Differentiation

A classic example of a higher-order rewrite system is *symbolic differentiation* based on the rules of Knuth [4, p.337]. In CRS notation we write Knuth's $D_x(e)$, where e can contain free occurrences of x, as D[x.e,x] (notice that the two occurrences to x do not refer to the same variable: the first is bound in e, the second free). The rules are in Figure 1 following these conventions:

KnuthDerivation [(

Rules for symbolic differentiation based on a system from The Art of Computer Programming.

Top-level derivor

Function: D[λx.E[x]] → (λx.D[y.E[y], x]) ;

Standard function derivatives

Ln: D[Ln] → (λx.1/x) ; **Exp**: D[Exp] → Exp ; **Sin**: D[Sin] → Cos ; **Cos**: D[Cos] → (λx.0 − (Sin x)) ;

Arithmetic derivatives

Plus[Copy[X]]: D[x.E_1[x] + E_2[x], X] → (D[x_1.E_1[x_1], X] + D[x_2.E_2[x_2], X]) ;
Minus[Copy[X]]: D[x.E_1[x] − E_2[x], X] → (D[x_1.E_1[x_1], X] − D[x_2.E_2[x_2], X]) ;
Times[Copy[E_1,E_2,X]]: D[x.E_1[x] × E_2[x], X] → (((D[x_1.E_1[x_1], X] × E_2[X]) + (E_1[X] × D[x_2.E_2[x_2], X])) ;
Divide[Copy[E_1,E_2,X]]: D[x.E_1[x] / E_2[x], X] → (((D[x.E_1[x], X] × E_2[X]) − (E_2[X] × D[x.E_1[x], X])) / (E_2[X] × E_2[X])) ;
Power[Copy[E_1,E_2,X]]: D[x.E_1[x] ^ E_2[x], X] → ((E_2[X] × (E_1[X] ^ (E_2[X] − 1))) × D[x.E_1[x], X]) ;

Composite function derivatives

Constant[Weak[E],Discard[E,X]]: D[x.E[], X] → 0 ; **Identity**[Discard[X]]: D[x.x, X] → 1 ;
Chain[Weak[E_1],Copy[E_1,E_2,X]]: D[x.E_1[] E_2[x], X] → ((D[E_1] E_2[X]) × D[x.E_2[x], X]) ;

Function simplification

β[Copy[E_1]]: ((λx.E_2[x]) E_1) → E_2[E_1] ; η[Weak[E]]: (λx.E[] x) → E[] ;

Arithmetic simplification

Plus-0-left: (0 + E) → E ; **Plus-0-right**: (E + 0) → E ;
Times-0-left[Discard[E]]: (0 × E) → 0 ; **Times-0-right**[Discard[E]]: (E × 0) → 0 ;
Times-1-left: (1 × E) → E ; **Times-1-right**: (E × 1) → E ;
Frac-1-under: (E/1) → E ; **Frac-0-over**[Discard[E]]: (0/E) → 0 ;
Frac-1-over-times: ((1/E_1) × (1/E_2)) → (1/(E_1 × E_2)) ;
Double[Comparable[E]]: (E + E) → (2 × E) ; **Square**[Comparable[E]]: (E × E) → (E ^ 2) ;)]

Figure 1: Symbolic differentiation CRS.

- Functions are written "curried", *i.e.*, using the built-in application operator.
- Primitive operators are written in traditional infix notation, which in CRSX in reality corresponds to applications.
- Numbers are written as themselves.
- The **Constant** rule exploits the check for free variables to verify that the dependant variable cannot occur in the first subterm of D which implies that the subterm is constant.
- The **Double** and **Square** rules introduce a new option that we have not seen before: Comparable[E] means that the meta-variable E can be used more than once in a pattern, and then the pattern only matches when the matched subterms are identical.

The command from the quick start guide (now with HTML rendering but still on one line)

```
crsx rules=samples/derivation/derivation.crs.html
  term="D[λx.Ln(x+1)]"
```

reveals the file name and prints the result

```
(λ x . 1 / (x + 1))
```

Similarly,

```
java -jar crsx.jar term="D[λx.Sin(x × x)]"
rules=samples/derivation/derivation.crs.html
```

prints the result

```
(λ x . Cos (x ^ 2) × (2 × x))
```

which is a slightly different way to write the usual derivative $2x\cos(x^2)$.

One-pass Call-by-Value CPS transform

One-pass Call-by-Value CPS transformation from Danvy & Rose (RTA '98).

CbvCps[(

Top-level rule.

Top : CBV[#1] → ↑[λ̲ k.CBV[↓[#1], λ m.@̲[k,m]]] ;

Derivor.

@̲ : CBV[@̲[#1, #2], λ k.#3[k]] → CBV[#1, λm.CBV[#2, λn.@̲[@̲[m,n],λ̲a.#3[a]]]] ;
λ̲[Copy[#1]] : CBV[λ̲x.#1[x], λ k.#2[k]] → #2[λ̲x k.CBV[#1[x], λ m.@̲[k,m]]] ;
Var[Free[v]] : CBV[v, λ k.#[k]] → #[v] ;

Drop

↓-App : ↓[@[#1,#2]] → @̲[↓[#1],↓[#2]] ; ↓-Abs : ↓[λx.#[x]] → (λ̲ x.↓[#[x]]) ; ↓-Abs1 : ↓[λ x.#[x]] → (λ̲ x.↓[#[x]]) ;
↓-Var[Free[x]] : ↓[x] → x ;

Lift

↑-App : ↑[@̲[#1,#2]] → @[↑[#1],↑[#2]] ; ↑-Abs : ↑[λ̲x.#[x]] → (λx.↑[#[x]]) ; ↑-Abs1 : ↑[λ̲ x.#[x]] → (λx.↑[#[x]]) ;
↑-Var[Free[x]] : ↑[x] → x ;
)]

Figure 2: One-pass call-by-value CPS transform CRS.

3.3 CPS Transform

The one-pass "call-by-value" CPS transform [1] is shown as a CRS in Figure 2. The command

```
java -jar crsx.jar term="CBV[λx.x]"
rules=samples/cps/cbv.crs.html
```

will print

```
(λ k . k (λ x k-1 . k-1 x))
```

The system demonstrates the following.

- The special option Free[x] used, for example, in the Var rule, permits a pattern to contain a *free variable* that will match any variable in the redex that is not matched by a bound variable in the pattern (this enforces what is known as Barendregt's "variable convention").
- In the λ̲ rule the substitution inserts a copy of the whole of λ̲x k.CBV[#1[x], λ m.@̲[k,m]] and since #1 is contained inside we must declare it copyable.

Also note that we sometimes use the explicit application form @[...] instead of the implicit one.

4 Manual

This section gives reference information for the CRSX higher-order rewriting engine in the conventional bottom up fashion.[3]

4.1 Lexical conventions

The standard CRSX parser generates tokens from the input character stream as follows.

White space. White space characters, most HTML tags (specifically everything between < and > not otherwise mentioned below), and the following complete HTML "comment" structures:

```
<!--...-->
<h1>...</h1>, <h2>...</h2>, ...
<head>...</head>
<blockquote>...</blockquote>
<address>...</address>
```

Case and spaces matter, so tags like `<H1>` or `<h1 class=foo>` are just skipped and not recognized as the above cases; indeed in each case

[3] The manual is still quite sketchy, and needs serious expansion.

the skipping is a simple search for the literal end tag, so nesting does not work.

Reserved characters. `{}[]().,:;` (braces, brackets, parenthesis, period,comma, colon, and semicolon) are always separate tokens, unless included in the white space forms above.

Tokens. The basic rules for forming tokens are that they consist of *parts* that are joined with the connector character - (dash). The first part determines the kind of token. Parts are either identifiers (including `$_` as letters), numbers, strings (with XML conventions), concatenations of the symbols `@^*+-'|/\%!?$=:`, or HTML markup <*m*>...</*m*> where *m* is one of `i`, `b`, `u`, `tt`, and `q`. In addition, parts can be "embellished" with any number of trailing _{...} and ^{...}.

Variables. If the first part is an identifier starting with a lower case letter, or it has the form <var>...</var>, then the token is a variable.

Metavariables. If the first part is an identifier that includes the special symbol #, or it has the form ..., then the token is a metavariable.

Constructors. In all other cases the token is a constructor.

Embedded terms. Embedded terms are lexically one unit (passed to an external parser). They take one of the following forms:

```
{{...}}, [[...]],
<code>...</code>, <pre>...</pre>
```

If the involved parser supports more than one nonterminal then one can indicate that with a special wrapper like

```
%NonTerminal[<code>...</code>]
```

(The precise details are in the source file *ClassicParser.jj*.)

4.2 Term Syntax

The syntax for CRS terms and meta-terms is outlined in Figure 3. The notation is usual extended BNF (more precisely that of JJCRS described in the companion "JJCRS HOWTO" [5]) with the following remarks:

```
Term ::= ⟨var⟩
 | (Props)? ⟨con⟩ ("[" (BList)? "]" | ⟨var⟩+ "." Term)
 | (Props)? ⟨mvar⟩ "[" (List)? "]"
 | "(" Sequence ")" | ⟨embedded⟩ .

Props ::= "{" (⟨mvar⟩ ";")? (Prop ("," Prop)*)? "}" .
Prop ::= ⟨con⟩ ":" Term | ⟨con⟩ | "&not;" ⟨con⟩
 | ⟨var⟩ ":" Term | ⟨var⟩ | "&not;" ⟨var⟩ .

BList ::= BTerm ("," BTerm)* .
BTerm ::= (⟨var⟩+ ".")? Term .

List ::= Term ("," Term)* .

Sequence ::= (Application)? (";" (Application)? )* .
Application ::= (Term)+ .
```

Figure 3: CRSX term syntax.

- The precise lexical form of variables, metavariables, and constructors, were given with the lexical conventions above.
- Parentheses are for convenience: all terms can be written without ()s, ;, and concatenation, instead using the @, $Cons, and $Nil constructors using the equivalences

$$
\begin{aligned}
t_0\ t_1 \cdots t_n &\equiv @[\ldots @[t_0, t_1], \ldots, t_n] \\
c\ v_1\ v_2 \cdots v_n\ .\ t &\equiv c[v_1.c[v_2 \ldots c[v_n.t] \ldots]] \\
(t) &\equiv t \\
t_1; \ldots; t_n; t &\equiv \$\mathrm{Cons}[t_1, \ldots \$\mathrm{Cons}[t_n, t] \ldots] \\
() &\equiv \$\mathrm{Nil} \\
(t;) &\equiv t; () \\
(;t) &\equiv (); t
\end{aligned}
$$

  In addition, two special application forms are translated as follows (used below):

$$
\begin{aligned}
t_0\ :\ t_1\ \texttt{\→}\ t_2 &\equiv\ \$\mathrm{Rule}[t_0, t_1, t_2] \\
t_1\ \texttt{\→}\ t_2 &\equiv \$\mathrm{Rule}[t_1, t_2]
\end{aligned}
$$

- A variable occurrence v is *bound* if it occurs inside the body b of a binder term (BTerm) like $v.b$, otherwise it is *free*.
- The special Props properties prefix is an extension to allow easy manipulation of annotations. In proper terms only the $c = t$ form is allowed and only on constructions; in patterns we furthermore allow the leading meta-variable for "all the properties" as well as the special form $\neg c$

which means "there is no c annotation" (with ¬ being `&not;` in HTML).

- An embedded term is a term in a foreign notation where a special parser has been setup; such are typically generated with the JJCRS parser generator [5] accompanying CRSX.

4.3 Normalization

When the CRSX interpreter is asked to normalize a term, it follows the following strategy, starting with the root term.

1. Reduce the current term as much as possible.
2. Reduce each of the children in turn (recursively) as much as possible.
3. If any child was actually modified then start over (otherwise reduction is done).

This strategy is implemented in the *normalize* method of the GenericCRS class.

4.4 Expressions

Several terms have special meaning when "evaluated" by directives and rewriting, described below. Notice that these are not needed for most rewrite systems, indeed one can argue that using these changes your rewrite system into a program in a more traditional sense.

Arithmetic

The following expressions are evaluated by

1. evaluating the arguments, and
2. if provided with integer constant arguments, replace with the constant corresponding to the result of the arithmethic expression.

`$[Plus,` i_1`,` i_2`, ...]`
`$[Minus,` i_1`,` i_2`]`
`$[Times,` i_1`,` i_2`, ...]`
`$[Div,` i_1`,` i_2`]`
`$[Mod,` i_1`,` i_2`]`

Comparison

The following expressions are evaluated by

1. evaluating the arguments, and
2. if provided with constant arguments, compare them appropriately and replace with either True or False.

`$[Equals,` c_1`,` c_2`]`
`$[NotEquals,` c_1`,` c_2`]`
`$[LessThan,` c_1`,` c_2`]`
`$[LessThanOrEquals,` c_1`,` c_2`]`
`$[GreaterThan,` c_1`,` c_2`]`
`$[GreaterThanOrEquals,` c_1`,` c_2`]`

Symbol manipulation

The following expressions operate on the constructor symbol of constants.

`$[:,` c_1`,` c_2`, ...]`
`$[Length,` c_1`]`
`$[Index,` c_1`,` c_2`]`
`$[Substring,` c_1`,` c_2`,` c_3`]`
`$[Replace,` c_1`,` c_2`,` c_3`]`

Term construction

The following expressions are evaluated by

1. evaluating the arguments, and
2. if provided with constant *name* argument, generate the appropriate construction, meta-application, or variable occurrence, term.

`$[C,` *name*`,` t`...]`
`$[M,` *name*`,` t`...]`
`$[V,` *name*`]`

Matching

```
$[Match, #p, #t]
$[NotMatch, #p, #t]
$[MatchRegex, #r, #t]
$[IsInteger, #p]
```

Property matching

```
$[NamedProperty, #name, #value, #t]
$[NotNamedProperty;, #name, #t]
$[VariableProperty, #name, #value, #t]
$[NotVariableProperty;, #name, #t]
$[CollectsProperties, #name, #t]
```

Parsing

```
$[Parse, #filename]
$[ParseURL, #parser, #category, #url]
$[ParseResource, #parser, #category, #name]
$[ParseText, #parser, #category, #text]
```

Evaluation

```
$[Script, #t]
$[If, #test, #true, #false]
$[Get, name-or-index[, fallback]]
$[Print, #term [, #result]]
$[Dump, #prefix, #term]
$[Error, #message [, #name]]
$[Trace, #message, #value]
```

4.5 Directives

A directive is a term of one of the following special forms. Notice that except for the first two, nesting and rules, these are not needed for most rewrite systems, indeed one can argue that using these changes your rewrite system into a program in a more traditional sense.

Nesting

```
( directive ;...; directive ;)
constructor[( directive ;...; directive ;)]
```

Process the nested directives. If the constructor form is used then the name of all embedded rules will have their name prefixed with the *constructor* symbol. The outermost such naming is also used as the name of the CRSX.

Rule

```
$Rule[pattern, contraction]
$Rule[name[options], pattern, contraction]
```

Add rule to the CRS being built.

- *name*: name to assign to the rule.
- *options*: properties of rules: list of rule exceptions:
  - Free[v,...] –
  - Fresh[v,...] –
  - Meta[$\#$,...] –
  - Leaf –
  - Share[$\#$,...] –
  - Copy[$\#$,...] –
  - Discard[$\#$,...] –
  - Weak[$\#$,...] –
  - Comparable[$\#$,...] –
- *pattern*: must be a construction where all contained meta-applications have only different bound variables as subterms (allowing C[x.y.#[x,y]] but not C[x.#[x,x]]).
- *contraction* of the corresponding pattern: all free variables in the contraction must occur free in the pattern and every meta-variable in the contraction must occur with the same arity in the pattern.

Term

```
$Term[ term ]
```

Send term to the context sink. The directive is first evaluated.

Normalization

```
$Normalize[ term ]
```

Normalize the term with the constructed CRS and send the result to the context sink. The directive is first evaluated.

Environment

```
$Set[ name ]
$Set[ name, directive ]
```

Process directive with the context sink set to store the result term in the name context variable. The directive is first evaluated.

Message

```
$Message[ term ]
$Message[ term, write-file ]
```

Append printable form of term to write-file (defaults to '' corresponding to System.out). The directive is first evaluated.

Verbosity

```
$Verbose
$Verbose[ verbose ]
$Verbose[ verbose, write-file ]
```

Set verbosity to the integer level verbose (defaults to 1), sending the verbose messages to the write-file (defaults to '' corresponding to System.out). The directive is first evaluated.

Meta

```
$Meta[ rules ]
```

The subsequent rules will be rewritten with the *rules* before being loaded.

Dispatchify

```
$Dispatchify
```

Transform the rules into constructor system form, necessary for compiling into plain code.

4.6 On CRSX's notation

When the rewrite and normalize commands are used then the current CRS rules are actually used to change the current term. One rule is applied at the time, anywhere in the term, and the affected subterm (redex) is replaced with the result of the rewrite.

At present there is no way to control the strategy used by the rewrite engine: the rewrite and normalize commands will pick seemingly random redices in an unspecified order.

Here are the differences between CRSX rewriting and the original CRS work [2, 3]. The extensions described here are mostly from [6].

- Meta-variables are written with # instead of reserving the letter Z.
- CRSX uses square brackets [] instead of round parentheses for all basic CRS constructs (reserving the round ones for applicative notations).
- Variable binders are written in λ-calculus style with a . (dot) instead of square brackets; furthermore, variable binders are restricted in CRSX to only occur on construction parameters. A subtle consequence of this is that variable binding is not "curried" because the pattern C[x.#[x]] will not match, for example, the term C[x y.x].
- Free variables are allowed in patterns where they will only match other variables. Such "free variable patterns" should not be used as arguments to meta-application patterns, however.
- We have extended CRS terms with *properties* and more importantly *property patterns*. They work as follows:

  First, a construction term with a "property prefix" has the form

  $$\{C_1.v_1, \ldots, C_n.v_n\}C[b_1, \ldots, b_m]$$

  where the C_i are property name constructors, the v_i are the corresponding property *value* terms (and the b_i are the usual construction subterms with optional binders).

  A construction with properties can be used as a pattern, where it matches when the pattern and term construction have *at least* the same property names where the corresponding property values match.

  Second, a *property pattern* involves prefixes of the form $\{\#; \neg C_1\}p$, where the special $\neg C_1$ means that matching terms can not have the property C_1, and in addition we make # a reference to all the properties the matched term actually had. p can be either a construction pattern or a meta-variable.

  In the rule contraction we can then add stuff like $\{\#; C_1.t_1, \neg C_2\}t$ which means "on the result of contracting t add all the properties remembered as # but then replace the value for C_1 with the contraction of t_1 and remove the C_2 property, if any".
- Finally, we have extended CRS with *constant matches* and *computing contractions*, all through the special term form $[*operator*, ...].

The extensions should all be manageable as rule schemas, *i.e.*, infinite enumerations of rules in a systematic way, but a formal treatment has not yet been undertaken.

4.7 Command line

This summarizes all the possible command line arguments. The first four are built in:

- `script=`*script-name* – use *script-name*, which can be a predefined name or a URL, as master script instead of default.
- `factory=`*class* – instantiate class as Factory, which it must implement. (Defaults to GenericFactory unless the grammar option is used.)
- `grammar=`*grammar-configuration* – use the indicated ANTLR grammar configuration; see

GenericFactoryTreeAdaptor for details (which is also set as the default Factory).

- `output=`*file* – send output from processing to *file*.

The remaining command line arguments are implemented by the *default* script and thus only supported in this way if the script option above is not used:

- `rules=`*system* – use CRS rules from the *system*, which may either be a resource name or a URL, to normalize the input term.
- `input=`*url* – get the input term from the *url*. (Cannot be combined with `term=...`).
- `term=`*term* – use input *term*. (Cannot be combined with `input=...`).
- `category=`*category* – use the non-terminal category for the root term. As a special case, category can be the value ?xml, which parses the special XML format of SAXSink.
- `verbose=`*number* – set the verbosity of normalization to *number*. (Default is 0.)
- `verbose-compiler=`*number* – set the verbosity of rule compilation to *number*. (Default is 0.)
- `embedded-parser=`*class* – use the *class* to parse embedded terms.

4.8 Missing

Some things are still missing:

- A better explanation of how rewriting works.
- Examples of the extensions.
- Examples of combination of several CRSX systems.
- Example of how to use the inference rule helper system to expand inference rules to CRSX systems.
- A description of the XML format and customizations.

References

[1] Olivier Danvy and Kristoffer H. Rose. Higher-order rewriting and partial evaluation. In Tobias Nipkow, editor, *RTA '98—Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 124–140, Tsukuba, Japan, March 1998. Springer. Extended version available as the technical report BRICS-RS-97-46 (`http://www.brics.dk/RS/97/46/`).

[2] Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980. Also available as Mathematical Centre Tracts 127.

[3] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[4] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[5] Kristoffer Rose. JJCRS howto. `http://crsx.sourceforge.net`, April 2010.

[6] Kristoffer Høgsbro Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, February 1996. DIKU report 96/1, `http://diku.dk/publikationer/tekniske.rapporter/rapporter/96-01.pdf`.