CRSX—Combinatory Reduction Systems with Extensions

Kristoffer H. Rose

IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598, USA
`krisrose@us.ibm.com`

Abstract

Combinatory Reduction Systems with Extensions (CRSX) is a system available from `http://crsx.sourceforge.net` and characterized by the following properties:

- Higher-order rewriting engine based on pure Combinatory Reduction Systems with full strong reduction (but no specified reduction strategy).
- Rule and term syntax based on λ-calculus and term rewriting conventions including Unicode support.
- Strict checking and declaration requirements to avoid idiosyncratic errors in rewrite rules.
- Interpreter is implemented in Java 5 and usable stand-alone as well as from an Eclipse plugin (under development).
- Includes a custom parser generator (front-end to JavaCC parser generator) designed to ease parsing directly into higher-order abstract syntax (as well as permitting the use of custom syntax in rules files).
- Experimental (and evolving) sort system to help rule management.
- Compiler from (well-sorted deterministic subset of) CRSX to stand-alone C code.

1998 ACM Subject Classification F.4.2, F.4.3, F.3.3

Keywords and phrases Higher-Order Rewriting, Compilers

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.81

Category System Description

1 Introduction

CRSX is a project on SourceForge, specifically `http://crsx.sourceforge.net`, currently at version 20 but under intense development. CRSX aims at providing a high quality rewrite engine and development environment for experimenting with higher order rewriting in general and higher order rewrite systems for compilers in particular, and has recently been rather successful at the latter including being the basis for a large internal compiler project in IBM.

CRSX stands for *Combinatory Reduction Systems* (CRS) with "eXtensions," which we detail below. CRS were invented and studied in depth by Jan Willem Klop [2, 3], who in turn credits the idea to Peter Aczel [1]. They are "combinatory" because every meta-variable carries all the (locally bound) variables it depends on, which enables the use of higher order terms and substitution.

In this paper we summarize what the CRSX system includes and is capable of in general terms. We assume some familiarity with rewriting (even higher order rewriting) as well as a general computer science background (such as an understanding of what a compiler and an interpreter are, *etc.*).

Components. CRSX consists of a number of pieces that work together.

Interpreter (CRSX). Executable that is used to load and "execute" rewrite systems. Loads a "script," which is itself a rewrite system using special internal "directive" terms to bootstrap the system. The *default script* is setup to load a file with CRSX rewrite rules and another with an input term, rewrite the term, and output the result in one of several customizable forms. The interpreter is written in Java 5.

Parser Generator (PG). Tool for building parsers from source text directly to CRSX higher-order terms such that input terms and rules can use custom syntax. Includes special support for directly generating higher order abstract syntax (HOAS) [4] and configuring scopes during parsing. Currently written as a JavaCC [8] script.

Eclipse Plugin. A helper to allow editing CRSX rules files with syntax highlighting and rule outlines in Eclipse, and with ambitions for simple debugging and such. Available directly from an Eclipse update site at `http://crsx.sourceforge.net/eclipse-plugin/`.

RulesCompiler. Tool for converting a rule system to a stand-alone C program. Written as a CRSX script including a large rules file for generating C, however, does invoke various experimental low-level directives programmed in Java.

In the following sections we briefly describe some of the special characteristics of these components.

Contributors. Most of the code so far has been written by the author, except the Eclipse plugin, which was written by Takahide Nogayama (IBM Tokyo Research Lab).

Availability. CRSX is available from the SourceForge archive `http://sourceforge.net/projects/crsx/` (click on "Code"). The code is Open Source and all CRSX documentation is being managed using the SourceForge source control system.

2 CRSX Extensions

The notation of CRSX modifies the original base CRS formalism in several ways, described briefly here (for a full description see the evolving system documentation [7]).

Symbols. All symbols can be used as constructor symbols. Unless the symbol has another meaning, it can be used directly, otherwise it can be used in quotes, so although lower case letters and words are normally variables, `'x'` is a constructor, and all symbols containing a `#` are normally meta-variables but `"Expr#"` is a constructor. In addition, CRSX uses Unicode, so many exotic characters are available.

Structure brackets. Basic term formation uses square brackets for subterms, and every symbol not otherwise specified is a constructor symbol. So simple arithmetic terms can be written, *e.g.*, as `+[*[1,2],3]`, where all of the symbols `+`, `*`, `1`, `2`, and `3`, are constructors; notice how we have omitted `[]` after the nullary constructors.

Subterm binder dot. Bound variables are specified with a "dot" prefix and are restricted to construction subterms. A standard "**let** $a = 1 \times 2$ **in** $a + 3$" binder construct can for example correspond to the term `Let[*[1,2], a.+[a,3]]` where the term makes the binder's scope explicit.[1]

λ-calculus conventions. Two special conventions facilitate λ-calculus notation: prefixing a constructor to some binders corresponds to nesting, *e.g.*, `λ x y z . x` really means

[1] The main effect of the subterm restriction for binders is that a fragment like `x.x` is not in itself a term.

λ`[x.`λ`[y.`λ`[z.x]]]`, and juxtaposition and normal parenthesis correspond to left-associative use of a special application operator, *e.g.*, `x z (y z)` really means `@[@[x,z],@[y,z]]`.[2]

Meta-application #tags. Words with `#` in them are meta-variables used to form meta-applications with brackets, *e.g.*, `Expr#1[x,Sub#Expr]`, where we can also omit `[]` for ground meta-applications.

Free variables. Unlike standard CRS, CRSX permits *free variables*, *i.e.*, `x` is a proper term. (This turns out to be crucial when defining recursive compilation and analysis schemes as these invariably have a case for free variables.)

Property lists. CRSX includes special syntax for constant properties and for properties of variables, written as *prefixes* of the form `{` *`Constant`* `:` *`Term`* `;...;` `v:` *`Term`* `;...; }`, where a *`Constant`* is a constructor symbol, `v` a variable, and *`Term`* arbitrary terms. We explain below how rules can match against the existence and value of properties.

Rules. Directives of the form *`Name`* `[`*`Options`*`]:` *`Pattern`* $\to$ *`Contraction`* are rules describing how subterms that match the *`Pattern`* can be replaced by subterms built according to the *`Contraction`*.[3] As usual in CRS, the *`Pattern`* is restricted to constructions where each contained meta-application must be applied to distinct bound variables, and the arity of each meta-variable must be consistent throughout the rule. We shall discuss rules in more details below, including the role and form of the *`Options`*.

Sorts. Directives of the form *`Sort`* `::=(`*`Form`*`;...)` and *`Form`* `::` *`Sort`* declare algebraic sorts of the included *`Form`*s. Each *`Sort`* is just a constant name,[4] and each *`Form`*, in turn, defines the shape of one construction by *`Constructor`* `[`B_1*`Sort`*$_1$`,...,`B_n*`Sort`*$_n$`]`, which specifies several constraints. First the *`Constructor`* can only occur where instances of the defined *`Sort`* are permitted. Second, the i'th subterm of the constructor must have the *`Sort`*$_i$ with precisely the binders described by B_i, which may be empty or have the form $\vec{\mathtt{x}}_i$`.{`$\mathtt{x}_{i1}$`:`*`Sort`*$_{ik}$`;...;`$\mathtt{x}_{i1}$`:`*`Sort`*$_{i1}$`}`, which combines the constraints that

- there must be precisely k binders on the i'th subterm, and
- each bound variable corresponding to binder $\mathtt{x}_{ij}$ must occur inside the subterm scope with $Sort_{ij}$.

Sort declarations with `::=` are called *data sort declarations* in that they enumerate forms of a sort that can only be included as non-outermost constructions in patterns, whereas `::`-declarations are called *function sort declarations* that (separately) enumerate forms of a sort that can only be included in patterns as the outermost construction. (Constructors that have not been included as part of a sort do not have any such restrictions.)

Evaluators. Rules can contain special evaluation terms of the form `$[`*`Constant`*`,...]`, where the *`Constant`* indicates what evaluation to perform in the pattern or contraction in question. The evaluator `$[Plus,#arg,1]`, for example has these properties:

- if used in a contraction it first contracts `#arg`, and if the result is a constant that can be interpreted as a number, then the contraction of the entire evaluator is the constant representing the number obtained by adding one;
- if used when matching a pattern then it first matches `#arg` against the redex subterm and, if that succeeds for some *`Term`*, then it records that the valuation of `#arg` should effectively be `$[Minus,#arg,1]` (with the obvious meaning).

The available evaluators are summarized in the documentation, including the functional restrictions to prevent the need for backtracking.

[2] We further follow the common convention that application binds tighter than other constructs.
[3] The "$\to$" in the directive is a the Unicode character U2192.
[4] We *are* working on parametrically polymorphic sorts...

Meta-rules. Special directives of the form `$Meta[( Rule ;... )]` permit rules that rewrite the rewrite rules themselves.

Notice that CRSX does not specify a reduction strategy and has no special evaluation strategy mechanism. Evaluation is only specified to always make progress and thus be weakly normalizing, and indeed the interpreter and C implementations use quite different strategies that we furthermore change over time. (Like all weakly normalizing strategies over higher order terms both reduction strategies reduce terms under binders, of course.) To enforce a particular reduction order it is necessary to insert control symbols that block unwanted reductions.[5]

We present a simple example of how to enter and run a rewrite system here; a larger example with properties and custom syntax will follow below.

▶ **Example 1** (λ-calculus)**.** Let us illustrate the use of the CRSX interpreter from the command line on a simple example. Create a file *beta-eta.crs* with the following content:[6]

```
BetaEta[(
  β[Copy[#X]] : ((λx.#[x]) #X) → #[#X] ;  η[Weak[#]] : (λx. # x) → # ;
  S → (λ x y z.x z (y z)) ;  K → (λ x y.x) ;  I → (λ x.x) ;
)]
```

Now copy the *crsx.jar* file from the CRSX release [7] and execute the following command (using your proper Java 5 `java` command):

```
$ java -jar crsx.jar rules=beta-eta.crs term="S K I"
(λ z . z)
```

The rules use the λ-calculus conventions discussed above, and further illustrate several points:

- A named group of rules (here BetaEta) is specified as a construction with a parenthesized sequence of `;`-terminated rules.
- Each rule is named (here β and η).
- After the rule name some options are needed whenever a rule does something beyond straight linear term rewriting: here we have to declare that, in β, the argument to the application may be copied (by the substitution), and, in η, the second argument under the abstraction omits a free variable from the substitution parameter list (which corresponds to the η-rule not permitting occurrences of that variable).
- The name part of a rule can be omitted (then the rule will be named for the pattern constructor).

To see the purpose of the options, removing them and running the same command would give these errors:

```
Error: BetaEta-β rule contractum uses non-shared/duplicatable meta-variable in place that may be copied (#X).
Error: BetaEta-η rule pattern meta-application # omits bound variables yet is not declared Weak.
Errors prevent normalization.
```

▶ **Example 2** (simply typed λ-calculus)**.** To illustrate the use of environments, we include a rule system for rewriting a simply typed λ-calculus to its type in Figure 1. The rule system defines a `T` scheme and some helper schemes that together rewrite a simply typed λ-term to its type, which can be used as follows:

[5] We are experimenting with generating force/delay operators automatically from the sorts but this is not available yet.
[6] Use the Unicode coding for the Greek letters and encode with UTF-8.

```
BetaTypes[(
 TYPE ::=( BASE; TYPE→TYPE; ERROR; );
 TERM ::=( x; λ[TYPE, x.{x : TERM}TERM]; TERM TERM; );

 T[TERM] :: TYPE ;
 -[Free[x]] : {x : #α} T[x] → #α ;
 -[Free[x]] : {¬x} T[x] → BASE ;
 -[Fresh[v],Copy[#α]] : {#Γ} T[λ[#α, x.#[x]]] → (#α → {#Γ; v : #α} T[#[v]]) ;
 {#Γ} T[#1 #2] → {#Γ} TA[{#Γ} T[#1], #2] ;

 TA[TYPE, TERM] :: TYPE ;
 -[Discard[#]] : TA[BASE, #] → ERROR ;
 {#Γ} TA[#α → #β, #] → M[#α, {#Γ} T[#], #β] ;

 M[TYPE, TYPE, TYPE] :: TYPE ;
 M[BASE, BASE, #γ] → #γ ;
 M[#α→#β, #α2→#β2, #γ] → M[#α2, #α, M[#β, #β2, #γ]] ;
 -[Discard[#α2,#β2]] : M[BASE, #α2→#β2, #γ] → #γ ;
 -[Discard[#γ]] : M[BASE, ERROR, #γ] → ERROR ;
 -[Discard[#α,#β,#γ]] : M[#α→#β, BASE, #γ] → ERROR ;
 -[Discard[#α,#β,#γ]] : M[#α→#β, ERROR, #γ] → ERROR ;
)]
```

Figure 1 *samples/lambda/beta-type.crs*: type analysis of simply typed λ-calculus.

```
$ java -jar crsx.jar check-sorts rules=beta-type.crs term="T[λ[BASE,x.x]]"
(BASE → BASE)
```

(the precise options used can be found in the system manual).

The system includes two sorts `TYPE` and `TERM` with terms describing simple types and simply typed terms, respectively. Notice that `TERM` permits free variables as well as a typed binder under λ; in addition to the λ-calculus conventions it makes use of the built-in parsing of infix use of →.

The second block defines the sort and rules for the `T` scheme. The first two rules give cases for `T` on free variables: one rule for variables that are defined in the type environment and another for globally free variables. Both rules include a `Free[x]` option, which is required to allow the free variable `x`, and both have a property constraint on the properties of the root `T` (the first requiring it to be `#α` and the second requiring it to be absent).

The third rule constructs the function type of an abstraction, which involves replacing the bound variable `x` with a fresh variable `v`, therefore declared `Fresh[v]` (in addition to `Copy[#α]` for the copied substructure). It illustrates how the property list in the contraction extends the matched properties with an additional binding.

The fourth and final `T` rule constructs the type of an application with a helper `TA` scheme that in turn uses the `M` scheme for type matching of the formal and actual argument types; the last two blocks define these. Notice the use of the `Discard` option, which is required for rules to not contract specific matched subterms: this is the single most effective error catching mechanism in the system.

```
// Simple XQuery-like language.
grammar net.sf.crsx.samples.x.X : <P>, <E>, <S>, <Q>

meta[<E>] ::= "#<PRODUCTION_NAME>" i?, "[", "]" .    // Meta-applications over AST.

skip ::= " " | "\r" | "\n" | "\t" .                 // White space.

<P> ::= {program} <E> .                             // Program.

<E> ::= <S>:#S ("," <E>:#E ⟦"comma"[#S,#E]⟧ | ⟦#S⟧) . // Expression.

<S> ::= "(" (<E> | {empty}) ")"                     // Simple expression.
     |  "element"_{} <N> "{" <E> "}"
     |  {query} <Q>
     |  "if"_{} <S> "then" <S> "else" <S>
     |  {call} <N> "(" (<E> | {empty}) ")"
     |  v_?
     |  {literal} <L>
     .

<Q> ::= "for"_{} v_x "in" <S> <Q>[x]                // Query.
     |  "let"_{} v_x ":=" <S> <Q>[x]
     |  "where"_{} <S> <Q>
     |  "return"_{} <S>
     .

<N> ::= n_{} .                                      // Names.
<L> ::= l_{} .                                      // Literals.

token l ::= i | "’" (¬[\’] | "’’")* "’" .
token i ::= [0-9]+ .
token n ::= [A-Za-z_] [A-Za-z0-9_-]* .
token v ::= "$" n .
```

Figure 2 *samples/x/x.pg*: sample nested loop language parser.

3 Parser Generator

The system includes the “PG” parser generator for generating custom parsers for higher order terms. Figure 2 contains the input file to PG for a small “nested loop” query language like XQuery [6]. The PG notation is designed to express common higher order term constructions.

- The top “grammar” declaration declares the name of the grammar as well as the externally available syntactic categories, with the first, `<P>`, being the default.
- The `meta` declaration sets up the format for meta-variables in the language, in this case using the same `#-[-]` notation as in CRSX itself.
- The `skip` declaration just specifies the format for white space.
- The `<P>` production states that a program will be represented as a *program* construction (indicated by the `{}`s) with the contained `<E>` as a subterm.
- The `<E>` production says that an `<E>` is an `<S>` followed either by a comma and a sub-`<E>` or nothing. The `:#S` allows us to refer to the `<S>` as `#S`, and in the first case `#E` can be

used to reference the sub-`<E>`; in both cases the result term is specified in ⟦...⟧s.[7]

- The `<S>` production has several choices; for each the generated term is specified either by giving the constructor prefix in `{}`s, or by indicating with a `_{}` marker that an existing token should itself be used as a constructor prefix, or (for the single variable token case) with the marker `_?` that the token should be the name of an in-scope variable.
- The `<Q>` production introduces binding constructs with the `_x` markers that indicate that the `v` token should be interpreted as a variable and, in each case, the correponding `[x]` marker indicates the (single) subterm that is the scope of the variable.
- The `<N>` and `<L>` productions just use the tokens as constructors.
- Finally, the four token kinds are defined using regular expressions.

In Java, the parser is then generated by first invoking PG to generate *X.jj* and then JavaCC [8] to generate the appropriate Java classes which then permits parsing such as the following, where we explicitly request the grammar name and syntax category to use:

```
$ java -jar crsx.jar "grammar=('net.sf.crsx.samples.x.X';)" category=P \
>   term="for $x in child(doc()) for $y in child(doc()) where eq($x,$y) return plus($x,$y)"
"program"[
 "query"[
  "for"["call"["child", "call"["doc", "empty"]],
   v"$x" . "for"[
     "call"["child", "call"["doc", "empty"]],
     v"$y" . "where"["call"["eq", "comma"[v"$x", v"$y"]],
                    "return"["call"["plus", "comma"[v"$x", v"$y"]]]]]]]]
```

Finally, the `meta`-declaration makes it possible to use rules files with embedded syntax, *e.g.*, the *samples/x/N.crs* rules file contains a rule

```
-[Free[id],Fresh[f]] :
 NQ[%Q⟦ for $v in #S #Q[$v] ⟧, id, t.#op[t]]
 → NQ[#Q[f], id, id3.MapConcat[
        Dep[id2.Map[Dep[id1.Tuple[ACons[f id1, ANil]]], N[#S, id2]]],
        #op[id3]]] ;
```

where the special `%Q⟦...⟧` notation invokes the parser described above to parse the `for`-construct allowing for binders and embedded meta-variables of the appropriate sorts.

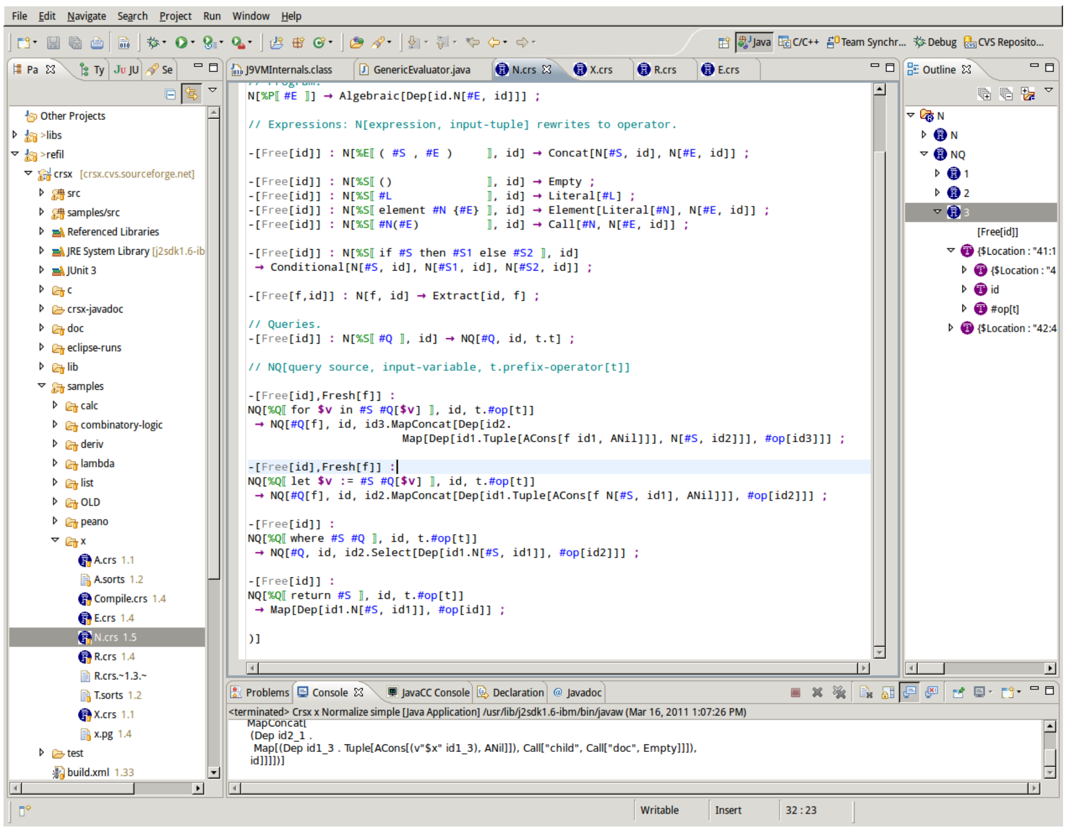
4 Eclipse Plugin

We include an Eclipse plugin for editing CRSX files (and, to some extent, PG grammar files). Figure 3 shows a screen shot of the plugin running in Eclipse. The plugin follows the usual Eclipse style of having a main editing pane for the rules and support panes with additional navigation and status information. The views include:

- Syntax highlighting in the editor view, including flagging of syntax errors.
- A structured outline view for quick navigation to rules and rule components.
- Simple stepping through an execution of a rewrite system with highlighting of the current rule and a way to access the current state of the term.
- A simple "observe rule" mechanism to allow very primitive break points; we hope to extend this capability to a proper debugging facility.

At the time of writing, the plugin is still under heavy development thus should not be expected to always work fully.

[7] The double brackets are Unicode characters U27E6 and 7.



```
N[%P⟦ #E ⟧] → Algebraic[Dep[id.N[#E, id]]] ;

// Expressions: N[expression, input-tuple] rewrites to operator.

-[Free[id]] : N[%E⟦ ( #S , #E )      ⟧, id] → Concat[N[#S, id], N[#E, id]] ;

-[Free[id]] : N[%S⟦ ()               ⟧, id] → Empty ;
-[Free[id]] : N[%S⟦ #L               ⟧, id] → Literal[#L] ;
-[Free[id]] : N[%S⟦ element #N {#E} ⟧, id] → Element[Literal[#N], N[#E, id]] ;
-[Free[id]] : N[%S⟦ #N(#E)           ⟧, id] → Call[#N, N[#E, id]] ;

-[Free[id]] : N[%S⟦ if #S then #S1 else #S2 ⟧, id]
 → Conditional[N[#S, id], N[#S1, id], N[#S2, id]] ;

-[Free[f,id]] : N[f, id] → Extract[id, f] ;

// Queries.
-[Free[id]] : N[%S⟦ #Q ⟧, id] → NQ[#Q, id, t.t] ;

// NQ[query source, input-variable, t.prefix-operator[t]]

-[Free[id],Fresh[f]] :
NQ[%Q⟦ for $v in #S #Q[$v] ⟧, id, t.#op[t]]
 → NQ[#Q[f], id, id3.MapConcat[Dep[id2.
                     Map[Dep[id1.Tuple[ACons[f id1, ANil]]], N[#S, id2]]], #op[id3]]] ;

-[Free[id],Fresh[f]] :
NQ[%Q⟦ let $v := #S #Q[$v] ⟧, id, t.#op[t]]
 → NQ[#Q[f], id, id2.MapConcat[Dep[id1.Tuple[ACons[f N[#S, id1], ANil]]], #op[id2]]] ;

-[Free[id]] :
NQ[%Q⟦ where #S #Q ⟧, id, t.#op[t]]
 → NQ[#Q, id, id2.Select[Dep[id1.N[#S, id1]], #op[id2]]] ;

-[Free[id]] :
NQ[%Q⟦ return #S ⟧, id, t.#op[t]]
 → Map[Dep[id1.N[#S, id1]], #op[id]] ;

)]
```

```
<terminated> Crsx x Normalize simple [Java Application] /usr/lib/j2sdk1.6-ibm/bin/javaw (Mar 16, 2011 1:07:26 PM)
   MapConcat[
    (Dep id2_1 .
     Map[(Dep id1_3 . Tuple[ACons[(v"$x" id1_3), ANil]]), Call["child", Call["doc", Empty]]]),
    id]]]]))]
```

Figure 3 Screen shot of Eclipse plugin.

5 RulesCompiler

In order for CRSX to be used as a compiler generator, it needs to generate code. We have chosen to do this by translating the rules of the rewrite systems itself directly to C in the following stages:

1. First make sure that the rewrite system is an orthogonal sorted constructor system (this should be automated with a completion procedure but we do not have that yet): any actual term should only match a single rule and all symbols should be sorted (which implies that they are categorized cleanly as function or data symbols).
2. Assign sorts to all constructors of the rewrite system to make sure that it is precisely understood what the possible constructors at every subterm are. Sort assignment is currently a standard monomorphic algebraic type analysis (we hope to extend this to permit polymorphic data constructors as these add convenience without complication).
3. The rewrite system is "dispatchified" by splitting all rules with nested patterns into a separate rule per choice point (similarly to the way normal functional language pattern matching is compiled).
4. For each constructor a C structure is created with the specifics of that construction, notably the number of subterms and rank (binder count) for each subterm.
5. Code is generated for every rule where
   a. pattern matching is translated to a switch on the root constructor of the investigated subterm that generates the specific rule function symbol for that pattern case; the switch has a case for "free variable" precisely when the sort of the subterm permits it;

```
// RULE: β : ((λ x . #[x]) #X) → #[#X]
int stepFunction_M__40(Sink sink_1, Term term_1)
{
  DEBUGT(sink_1->context, "\n==========\nMATCH β \n==========\n", term_1);
  Term sub_1 = FORCE(sink_1->context, SUB(term_1,0));
  if (!IS_DATA(sub_1)) return 0;
  Term m_M__23 = SUB(sub_1,0);
  Variable mbind0_M__23 = BINDER(sub_1,0,0);
  Term m_M__23X = SUB(term_1,1);
  int mcount_M__23X = 0;
  DEBUGF(sink_1->context, "%s","\n==========\nCONTRACT β\n==========\n");
  PROPERTIES_RESET(sink_1);
  {
    Variable vars_1[1] = {mbind0_M__23};
    Term subs_1[1];
    {
      Sink buf_1 = MAKE_BUFFER(sink_1->context);
      COPY(buf_1, m_M__23X, mcount_M__23X++);
      subs_1[0] = BUFFER_TERM(buf_1);
      FREE_BUFFER(sink_1->context, buf_1);
    }
    struct _SubstitutionFrame substitution_1 = {NULL, 1, vars_1, subs_1};
    SUBSTITUTE(sink_1, m_M__23, 0, &substitution_1);
  }
  DEBUGF(sink_1->context, "%s","\n==========\nEND β\n==========\n");
  return 1;
}
```

Figure 4 Fragment of generated C "step" code.

   b. rules drive evaluation of arguments that need to be pattern matched;
   c. every component of each pattern is extracted, including binders;
   d. the result term is constructed from left to right by combining new constructors and binders with copies of existing terms, possibly subject to substitutions.

6. A top level normalization algorithm is applied that repeatedly attempts to reduce the outermost functional subterm until there are none or reduction is stuck.

Figure 4 contains step function code generated for the β rule of the λ-calculus.

The `sub_1` term is set to the value of the first argument after it has been "forced" to be a value. If the value is then not a construction (so a free variable) then the step fails because some rule in the context must perform the substitution. In pure λ calculus we then now know that it must be a λ construction and thus we can merely extract the binder and subterm of the λ construction. The contraction can then start: we create a substitution valuation for replacing instances of the bound variable with copies of the application argument and process the substitution sending the result to the buffer sink that the rewrite step is configured to produce.

For general β reduction this is very naive, however, in practice compilers do few operations as general as this: most rewrite steps are subject to optimizations. The most important one is to eliminate copies and substitutions where possible. Specifically, many substitutions replace a bound variable with another variable either bound or free. In most cases this can be replaced with permuting and reusing the existing variables such that the substitution itself is a copy. Similarly, most rules are linear in that they contract to use just a single copy of the matched subterms. The rules will reuse a pointer to the existing copy for the first use.

6 Conclusions

CRSX is the result of more than three years of development [5], and is beginning to mature. It is definitely there and can be played with, even if the documentation is not scheduled to be properly available until "later this year." The author encourages anyone interested in compilers and (especially) higher order rewriting to download and try CRSX; if you furthermore wish to participate I will be thrilled!

Current work is rather focused on getting a proper understanding of the formal underpinnings of the extension that have seemed necessary to get CRSX to do what it does, and also the cleanest way to allow rewrite systems themselves direct access to the "formal toolbox" that is effectively embedded in the system.

In addition, CRSX is in full production use as a compiler generator framework, which the author plans to keep pushing as far as it can because it is only getting more interesting with time.

Acknowledgements. I am grateful to our "Compilers by Higher Order Rewriting" team at IBM Research: Scott Boag, Takahide Nogoyama, Naoto Sato, Lionel Villard, and Bob Schloss, for participating in the great adventure it is to build a compiler mostly from scratch in this way (as well as serve as first line of defense against interesting if sometimes unintended features), to Morris Matsa for believing that such compilers can eventually be effective, to the anonymous referees for numerous helpful suggestions, and finally, thanks to SourceForge for enabling the disseminiation of projects such as this.

References

1 Peter Aczel. A general Church-Rosser theorem. `http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf`, July 1978. Corrections at `http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGRT_corrections.pdf`.

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 Frank Pfenning and Conal Elliot. Higher-order abstract syntax. *SIGPLAN Notices*, 23(7):199–208, 1988.

5 Kristoffer Rose. CRSX – an open source platform for experimenting with higher order rewriting. Presented in absentia at HOR 2007—`http://kristoffer.rose.name/papers`, June 2007.

6 Kristoffer Rose. Higher-order rewriting for executable compiler specifications. In Eduardo Bonelli, editor, *Proceedings 5th International Workshop on Higher-Order Rewriting*, volume 49 of *EPTCS*, pages 31–45, Edinburgh, Scotland, July 2010.

7 Kristoffer Rose. Combinatory reduction systems with extensions. `http://crsx.sourceforge.net`, March 2011.

8 Sreeni Viswanadha, Sriram Sankar, et al. *Java Compiler Compiler (JavaCC) - The Java Parser Generator.* Sun, 4.0 edition, January 2006.