iRho: the Software

[System Description]

Luigi Liquori

INRIA, France

Abstract

This paper describes the first implementation of an interpreter for iRho, an imperative version of the Rewriting-calculus, based on pattern-matching, pattern-abstractions, and side-effects. The implementation contains a parser and a call-by-value evaluator in Natural Semantics; everything is written using the programming language Scheme. The core of this implementation (evaluator) is *certified* using the proof assistant Coq.

Performances are honest compared to the minimal essence of the implementation. This document describes, by means of examples, how to use and to play with iRho. The final objective is to make iRho a, so called, *agile programming language*, in the vein of some useful scripts languages, like, *e.g.* Python and Ruby, where proof search is not only feasible but easy.

1 Introduction to the Rewriting Calculus

One of the main advantages of the *rewriting-based* languages, like Elan [16], Maude [14], ASF+SDF [19, 2], OBJ* [10], Stratego [18] is *pattern-matching*. Pattern-matching allows to discriminate between alternatives: once a pattern is recognized, a pattern is associated with an action. The corresponding pattern is thus rewritten in an appropriate instance of a new one.

Another advantage of rewriting-based languages (in contrast with ML or Haskell) is the ability to handle *non-determinism* in the sense of a collection of results: pattern matching need not to be exclusive, *i.e.* multiple branches can be "fired" simultaneously. An empty collection of results represents an application failure, a singleton represents a deterministic result, and a collection with more than one element represents a non-deterministic choice between the elements of the collection. This feature makes the calculus quite close to logic languages too; this means that it is possible to make a product of two patterns, thus applying "in parallel" both patterns.

Optimistic/pessimistic semantics can then be imposed to the calculus by defining successful results as products that have at least a component (respectively all the components) different from error values. It should be possible to obtain a logic language on top of it by redefining appropriate strategy for backtracking.

[1] Supported by grant Aeolus FP6-2004-IST-FET Proactive, and the French grant ACI Modulogic.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Useful applications lie in the field of pattern recognition, and strings/trees manipulation. Pattern-matching has been widely used in functional and logic programming, as ML [15,7], Haskell [11], Scheme [17], or Prolog [9]; generally, it is considered a convenient mechanism for expressing complex requirements about the function's argument, rather than a basis for an *ad hoc* paradigm of computation.

The *Rewriting-calculus* (Rho) [4, 5] integrates in a uniform way, matching, rewriting, and functions; its abstraction mechanism is based on the rewrite rule formation: in a term of the form $P \to A$, one abstracts over the pattern P. Note that the Rewriting-calculus is a generalization of the Lambda-calculus if the pattern P is a variable. If an abstraction $P \to A$ is applied to the term B, then the evaluation mechanism is based on the binding of the free variables present in P to the appropriate subterms of B applied to A. Indeed, this binding is achieved by matching P against B. One of the advantages of matching is that it is "customizable" with more sophisticated matching theories, *e.g*. the associative-commutative one.

This year, an imperative extension enhancing the (functional) Rho, was presented in [13]; shortly, we introduced imperative features like referencing (*i.e*. "malloc-like ops", ref expr), dereferencing (*i.e*. "goto-memory ops", ! expr), and assignments operators (X := expr). The associated type system was enriched with dereferencing-types (*i.e*. pointer-types, int ref), and product-types (*e.g*. $\mathsf{int} \to \mathsf{int} \wedge \mathsf{nat} \to \mathsf{nat}$). The mathematical content of this extension was validated by the help of the semiautomatic proof assistant Coq. A toy software prototype, mimicking the mathematical behavior of the dynamic semantics was also implemented in Scheme. This paper introduces shortly the first LGPL release of the software; a parser has been implemented and more syntactic sugar has been added to make the interpreted easier to use. The core kernel is conform to the semantics specification of [13]; future releases will also come with a machine assisted "certificate" that the design choices are correct. We may envisage also proof extraction of the main kernel routine, in case of a "port" of the software in Caml or in Haskell [2]. This paper presents the syntax of the iRho language and some examples that can be run directly by cut and paste in the interpreter. The current distribution can be found in: `http://www-sop.inria.fr/mirho/Luigi.Liquori/iRho/`. It contains: two software releases `iRho-1.0.scm`, and `iRho-1.1.scm`, a precompiled binary version for Linux architecture [3], a file `demo.rho` containing many examples, and a copy of the [13] paper (journal version).

We conclude with a table showing future releases and evolutions of the present software, like (polymorphic) type inference, powerful matching and unification algorithms, exceptions handlers, strategies, calling external languages, objects, etc.

[2] The extraction mechanism in Coq can currently target Caml or Haskell code.

[3] `ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), stripped.`

2 Playing with iRhoSW

The interpreter iRhoSW greets you as follows:

```
 ----------------------------------------------
|               ----------\                    |
|               | i R h o  >                   |
|               ----------/                    |
| An Imperative Rewriting Calculus Interpreter |
| Kernel Certified by the Proof Assistant Coq  |
|           Powered by Bigloo Scheme           |
|             Copyright Inria 2005             |
|               Version 1.1alpha               |
|             NoEffect Theory Loaded           |
|             $ = Switch Theory                |
|             # = Clean  Namespace             |
|             @ = Exit   iRho                  |
 ----------------------------------------------
```

As usual, the first thing to learn is how to exit from the *read-eval-print* loop: just evaluate "`@;;`" to exit. Evaluating "`$;;`" moves the interpreter to the *empty* (or *syntactic*) matching theory to the *no-stuck* matching theory, introduced firstly in [5]: we will be more precise about this theory in a moment, after presenting the syntax and a sketch of the reduction semantics. Evaluate "`#;;`" allows to clean a *global namespace*, *i.e.* a space where constants, functions, and term rewriting systems can be names and globally reused.

Syntax

The untyped (abstract) syntax of iRho is as follows:

```
key   ::= "(" | ")" | "," | "^" | "!" | ":=" | "->" |
          "<-?" | "?->" | ";" | "=" | "[" | "]" | "|" Keywords
var   ::=  "any sequence of capital alphadigit"          Variables
const ::=  "any sequence of non capital alphadigit"      Constants
patt  ::= var    | const | const patt   |
          patt,patt      | ^ patt                         Patterns
expr  ::= const | var    | patt -> expr | expr expr |
          expr,expr      | ^ expr       | ! expr    |
          expr:=expr     | expr <-? expr ?-> expr   |
          var=expr       | [(var=expr |)*]             Expressions
```

One important point is that *linearity* in pattern is not enforced in the syntax; the solution we adopt in this formalization and implementation of the Rewriting-calculus was influenced by the choice of the implementation language of our operational semantics, namely Scheme and the matching algorithm adopted [12]. As such, the specification of the matching algorithm in iRho accepts non-linear patterns, and compares subparts of the datum (through $\equiv$, implemented via the primitive `equiv?` in Scheme). Confluence is preserved, thanks to the call-by-value strategy of the operational semantics. Examples of legal terms are:

```
iRho  IN > 12;;
iRho OUT > 12
iRho  IN > dummy;;
iRho OUT > dummy
iRho  IN > x;;
iRho OUT > x
iRho  IN > X;;
iRho OUT > (Effect: Unbound Variable X)
iRho  IN > 12;;
iRho OUT > 12
iRho  IN > (12->13 12);;
iRho OUT > 13
iRho  IN > (12->12 14);;
iRho OUT > (Effect: Pattern Mismatch)
iRho  IN > ((a->b,a->d) a);;
iRho OUT > (b , d)
iRho  IN > ((a->b,c->d) a);;
iRho OUT > b
iRho  IN > (f X Y)->X;;
iRho OUT > (Fun ((f X) Y) -> X)
iRho  IN > ((f X X)->X (f 3 4));;
iRho OUT > (Effect: Pattern Mismatch)
iRho  IN > $;;
iRho OUT > Switching_to_empty_theory
iRho  IN > ((a->b,c->d) a);;
iRho OUT > (b , (Effect: Pattern Mismatch))
```

The last example can help to understand that the no-stuck theory absorbs pattern-matching failures, while the empty theory is not. This is perhaps a good point to introduce the reduction semantics.

Reduction Semantics

The semantics behaves as follows (see [13] for a detailed presentation):

```
 (patt -> expr exprnf) => sigma(expr)  where sigma=patt<<exprnf
((expr1,expr2) exprnf) => ((expr1 exprnf),(expr2 exprnf))
```

The first rule fires an application if the argument is in normal-form (call-by-value semantics) and if it *matches* with the pattern, while the second rule distributes the application to all elements of a structure. That's all you need to do if you want to play just with the functional fragment of the Rho. In a nutshell, the functional fragment is "just" a Lambda-calculus with patterns, records, and non exclusive pattern-matching (*i.e.* multiple branches can be fired simultaneously). The possibility to fire, in parallel, multiple matching branches is one of the biggest peculiarity of the Rewriting-calculus w.r.t. other languages featuring (exclusive and sequential) pattern-matching.

Adding imperative features causes to introduce a store, *i.e.* an global partial mapping `s` from locations to expressions in normal forms (*i.e.* values), and to add the following reduction rules:

```
  ^ exprnf  /s =>    loc/(s,loc=exprnf)  where loc not in Dom(s)
   !loc     /s => s(loc)/ s              where loc     in Dom(s)
loc:=exprnf /s => exprnf/(s,loc=exprnf)  where loc     in Dom(s)
exprs1;expr2/s => ((X->expr2) expr1)/s   where  X   fresh in  expr2
```

In a nutshell: the first rule allocates a new fresh location `loc` in the store and binds it to the value `exprnf`; the second rule reads the content of the location `loc`; the third rule writes in the location `loc` the value `exprnf`. The last rule (sequence) is just a macro for a dummy function application; the call-by-value strategy ensures that `expr1` will be evaluated (possibly with a store modification) before `expr2`. Examples of legal terms are:

```
iRho  IN > ^ 1,2;;
iRho OUT > ((Ref 1) , 2)
iRho  IN > ^ (1,2);;
iRho OUT > (Ref (1 , 2))
iRho  IN > (!^(X->X)  4);;
iRho OUT > 4
iRho  IN > ((X,Y)->(X,Y) (^3,^4));;
iRho OUT > ((Ref 3) , (Ref 4))
iRho  IN > ((X,Y)->(Y:=!X;(!X,!Y)) (^3,^4));;
iRho OUT > (3 , 3)
iRho  IN > ((X,Y)->(Y:=!X;(X,!X,Y,!Y)) (^3 , ^4));;
iRho OUT > ((Ref 3) , (3 , ((Ref 3) , 3)))
iRho  IN > ((f X Y)->(Z->(X:=!Z) X) (f ^3 ^4));;
iRho OUT > 3
iRho  IN > (X->((^ Y->Y) X) ^ 4);;
iRho OUT > 4
iRho  IN > ((XREF->((X->XREF:=X) 5)) ^dummy);;
iRho OUT > 5
```

The macro "="

This simple macro allows to modify a global namespace; it is also useful to define quickly constants values, functions, and term rewriting systems with built-in fixpoints.

```
iRho  IN > ID = (X->X);;
iRho OUT > (Fun X -> X)
iRho  IN > IDID = (X->X X->X);;
iRho OUT > (Fun X -> X)
iRho  IN > ID;;
iRho OUT > (Fun X -> X)
iRho  IN > (ID 4);;
iRho OUT > 4
iRho  IN > IDID;;
iRho OUT > (Fun X -> X)
iRho  IN > (IDID 4);;
iRho OUT > 4
iRho  IN > MATCHPAIR = ((f(X,Y))->X);(MATCHPAIR (f(2,3)));;
iRho OUT > 2
iRho  IN > MATCHCURRY = (f X Y)->X;(MATCHCURRY (f 2 3));;
iRho  IN > SWAP=((X,Y)->((AUX->(AUX:=!X;X:=!Y;Y:=!AUX;
                               (!X,!Y,!AUX)))(^0)));;
iRho OUT > (Fun (X , Y) -> ((Fun AUX ->
           ((Fun FRESH1005 -> ...
           ((Bang X) , ((Bang Y) , (Bang AUX))))
           (Ass Y (Bang AUX)))) (Ass X (Bang Y))))
           (Ass AUX (Bang X)))) (Ref 0)))     Swapping two variables
iRho  IN > (SWAP(^4,^5));;
iRho OUT > (5 , (4 , 4))
iRho  IN > FIXV = FUN->VAL->(FUN (FIXV FUN) VAL);;
iRho OUT > (Fun FUN -> (Fun VAL ->
           ((FUN (FIXV FUN)) VAL)))          A call-by-value  fix point
iRho  IN > (FIXV ID 3);;
Segmentation fault                          Sorry, reload everything ...
iRho  IN > LETRECPLUS = ((PLUS ->
           (PLUS ((succ (succ 0)),(succ (succ 0)))))
           (FIXV (PLUS -> VAL ->
           (((0,N)       -> N  ,
            ((succ M),N) -> (succ (PLUS (M,N)))) VAL))));;
                              letrec PLUS = ''Peano's plus'' in (PLUS (2,2))
```

If-then-else

Control structures can be easily be defined as follows:

```
iRho  IN > NEG = (true -> false, false -> true);;
iRho OUT > ((Fun true -> false) , (Fun false -> true))
iRho  IN > (NEG true);;
iRho OUT > false
iRho  IN > AND = ((true, true)  -> true,
                  (true, false) -> false,
                  (false,true)  -> false,
                  (false,false) -> false);;
iRho OUT > ((Fun (true  , true)  -> true) ,
           ((Fun (true  , false) -> false) ,
           ((Fun (false , true)  -> false) ,
            (Fun (false , false) -> false))))
iRho  IN > OR = ((true, true)  -> true,
                 (true, false) -> true,
                 (false,true)  -> true,
                 (false,false) -> false);;
iRho OUT > ((Fun (true  , true)  -> true) ,
           ((Fun (true  , false) -> true) ,
           ((Fun (false , true)  -> true) ,
           (Fun (false  , false) -> false))))
iRho  IN > OMG = (X->(X X));;
iRho OUT > (Fun X -> (X X))
iRho  IN > ((OMG OMG) <-? (AND (true,true)) ?-> 4);; Happy syntax
iRho OUT > 4                                  Don't try with false :-)
```

Defining Term Rewriting Systems

One may wonder a simpler way to define a term rewriting system and a fix-point operator allowing to use a term rewriting system; the iRhoSW offers two ways to do it in a simpler and efficient way. The first is by using the macros "=" while the latter is by using the macros "[...]". The main difference between those two alternatives is in efficiency (the former being faster the the latter). We first introduce some macros for Peano's numbers

```
iRho  IN > ZERO     = 0;;
           ONE      = (succ 0);;
           TWO      = (succ ONE);;
           THREE    = (succ TWO);;
           ...
```

Then we simply define our PLUS term rewriting system as follows:

```
iRho  IN > PLUS = ((0,N)        -> N,
                   ((succ N),M) -> (succ (PLUS (N,M))));;
iRho OUT > ((Fun        (0 , N) -> N) ,
            (Fun ((succ N) , M) -> (succ (PLUS (N , M)))))
iRho  IN > (PLUS (THREE,THREE));;
iRho OUT > (succ (succ (succ (succ (succ (succ 0))))))
```

or as follows:

```
iRho  IN > [PLUS = ((0,N)       -> N,
                   ((succ N),M) -> (succ (PLUS (N,M))))];;
iRho OUT > Term Rewriting System Definition
iRho  IN > [PLUS = ((0,N)       -> N,
                   ((succ N),M) -> (succ (PLUS (N,M))))];
           (PLUS (THREE,THREE));;
iRho OUT > (succ (succ (succ (succ (succ (succ 0))))))
```

Note that in the two encodings (using "=" or "[...]") one term rewriting system can "call" another term rewriting system as follows (using sequencing):

```
iRho  IN > PLUS = ((0,N)        -> N ,
                   ((succ N),M) -> (succ (PLUS (N,M))));
           FIB  = (0            -> (succ 0) ,
                   (succ 0)     -> (succ 0) ,
                (succ (succ X)) -> (PLUS ((FIB (succ X)),
                                          (FIB X))));
          (FIB FOUR);;                              First encoding
```

```
iRho  IN > [PLUS = ((0,N)       -> N ,
                   ((succ N),M) -> (succ (PLUS (N,M))))
            |
             FIB = (0           -> (succ 0) ,
                    (succ 0)    -> (succ 0) ,
                (succ (succ X)) -> (PLUS ((FIB (succ X)),
                                          (FIB X))))];
           (FIB FOUR);;                           Second encoding
iRho OUT > (succ (succ (succ (succ (succ 0)))))
iRho  IN > [PLUS = ((0,N)        -> N ,
                    ((succ N),M) -> (succ (PLUS (N,M))))
            |
            MULT = ((0,M)        -> 0,
                    ((succ N),M) -> (PLUS (M,(MULT (N,M)))))
            |
            POW  = ((N,0)        -> (succ 0),
                    (N,(succ M)) -> (MULT (N,(POW (N,M)))))];;
           (POW (TWO,TEN));;                                Power
iRho  IN > [ACK  =
            ((0,N)              ->(succ N),
             ((succ M),0)       ->(ACK(M,(succ 0))),
             ((succ M),(succ N))->(ACK(M,(ACK((succ M),N)))))];
           (ACK  (THREE,FOUR));;                        Ackermann
iRho  IN > LIST = (10,11,12,13,15,16,nil);;
iRho  In > [FINDN = ((0,nil)         -> fail,
                    ((succ N),nil)   -> fail,
                    ((succ 0),(X,Y)) -> X,
                    ((succ N),(X,Y)) -> (FINDN (N,Y)))];
           (FINDN (THREE,LIST));;          Find an element in a list
iRho  In > [KILLM = ((m,(n,nil)) -> (n,nil),
                     (m,(m,X))   -> X,
                     (m,(n,X))   -> (n,(KILLM (m,X))))];
           (KILLM (13,LIST));;             Kill an element in a list
```

A More Tricky Example: Negation Normal Form

This function is used in implementing *decision procedures*, present in almost all model checkers. The processed input is an implication-free language of formulas with generating grammar:

$$\phi ::= \mathsf{p} \mid \mathsf{and}(\phi,\phi) \mid \mathsf{or}(\phi,\phi) \mid \mathsf{not}(\phi)$$

We present three encodings, the first uses the "=" macro, the second uses the "[...]" macro and the last is just the macro-expansion of the second one (some outputs are omitted).

```
iRho  IN > PHI = (and ((not (and (p,q))),(not (and (p,q)))));;
iRho  IN > NNF = (  p ->  p,
                    q ->  q,
    (not (not  X))    -> (NNF X),
    (not (or  (X,Y))) -> (and ((not (NNF X)),(not (NNF Y)))),
    (not (and (X,Y))) -> (or  ((not (NNF X)),(not (NNF Y)))),
    (and (X,Y))       -> (and ((NNF X),(NNF Y))),
    (or  (X,Y))       -> (or  ((NNF X),(NNF Y))));
           (NNF PHI);;                                  First encoding
iRho  IN > [NNF = (  p ->  p,
                     q ->  q,
     (not (not  X))    -> (NNF X),
     (not (or  (X,Y))) -> (and ((not (NNF X)),(not (NNF Y)))),
     (not (and (X,Y))) -> (or  ((not (NNF X)),(not (NNF Y)))),
     (and (X,Y))       -> (and ((NNF X),(NNF Y))),
     (or  (X,Y))       -> (or  ((NNF X),(NNF Y))))];
           (NNF PHI);;                                 Second encoding
iRho OUT > (and((or((not p),(not q))),(or((not p),(not q)))))
```

Certification: the DIMPRO *pattern*

In [13] we experimented with an interesting "pattern (in the sense of *"The Gang of Four"* [8]) called DIMPRO, a.k.a. Design-IMplement-PROve, to design safe software, which respects *in toto* its mathematical and functional specifications. The iRhoSW is a direct derivative of such a methodology.

Intuitively, we started from a clean and elegant mathematical design, from which we continued with an implementation of a prototype satisfying the design (using a functional language), and finally we completed it with a mechanical certification of the mathematical properties of the design, by looking for the simplest "adequacy" property of the related software implementation. These three phases are strictly coupled and, very often, one particular choice in one phase induced a corresponding choice in another phase, very often forcing backtracking.

The process refinement is done by iterating cycles until all the global properties wanted are reached (the process is reminiscent of a fixed-point computation, or of a B-refinement [1]). All three phases have the same status, and each can influence the other.

Our recipe probably suggests a new schema, or "pattern", in the sense of *"The Gang of Four"* [8], for design-implement-certify safe software. This could be subject of future work. A small software interpreter for our core-calculus is surely a good test of the "methodology". More generally, this methodology could be applied in the setting of raising quality software to the highest levels of the *Common Criteria, CC* [6] (from EAL5 to EAL7), or level five of the *Capability Maturity Model, CMM*. We schedule in our agenda our novel DIMPRO, in the folklore of "design pattern", hoping that it would be useful to the community developing safe software for crucial applications.

Agenda

Our iRhoSW is really young: the table below sketch some possible improvements planned in the next two future releases.

MAJOR IMPROVEMENTS/RELEASE	2.0	3.0
exceptions on pattern-matching failure	✓	
first-order type inference	✓	
more control structures and strategies	✓	
simple objects and object-based inheritance	✓	
type-inference *à la* Damas-Milner		✓
unification and AC matching theory		✓
rewriting-rule as patterns [3]		✓
calling externals Scheme/Java/C		✓
I/O (files)		✓
certification using Coq	✓	??

References

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*.

[2] Asf+Sdf Team. The Asf+Sdf Meta-Environment Home Page, 2005.

[3] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems. In *Proc. of POPL*. 2003.

[4] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.

[5] H. Cirstea, L. Liquori, and B. Wack. Rho-calculus with Fixpoint: First-order system. In *Proc. of TYPES*. Springer-Verlag, 2004.

[6] Common Criteria Consortium. The Common Criteria Home Page, 2005.

[7] Cristal Team. OCaml Home page, 2005.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides (The Gang of Four). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] GNU Prolog Team. The Prolog Home Page, 2005.

[10] J. Goguen. The OBJ Family Home Page, 2005.

[11] Haskell Team. The Haskell Home Page, 2005.

[12] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Ph.d. thesis, Université de Paris 7 (France), 1976.

[13] Liquori, L. and Serpette, B. iRho, An Imperative Rewriting Calculus. In *Proc. of PPDP*, pages 167–178. The ACM Press, 2004.

[14] Maude Team. The Maude Home Page, 2005.

[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[16] Protheo Team. The Elan Home Page, 2005.

[17] Scheme Team. The Scheme Language, 2005.

[18] Stratego Team. The Stratego Home Page, 2005.

[19] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping*, 1996.