iRho: An Imperative Rewriting Calculus

[Extended Abstract] *

Luigi Liquori
INRIA, France
Luigi.Liquori@inria.fr

Bernard Paul Serpette
INRIA, France
Bernard.Serpette@inria.fr

ABSTRACT

We propose an imperative version of the Rewriting-calculus, a calculus based on pattern-matching, pattern-abstraction, and side-effects, which we call iRho.

We formulate a static and a *call-by-value* dynamic semantics of iRho like that of Gilles Kahn's *Natural Semantics*. The operational semantics is deterministic, and immediately suggests how to build an interpreter for the calculus. The static semantics is given via a first-order type system based on a form of product-type, which can be assigned to iRho-terms like structures (*i.e.* pairs).

The calculus is *à la* Church, *i.e.* pattern-abstractions are decorated with the types of the free variables of the pattern.

iRho is a good candidate for a core or an intermediate language, because it can safely access and modify a (monomorphic) typed store, and because fixed-points can be defined.

Properties like determinism of the interpreter, and subject reduction are completely checked by a machine assisted approach, using the Coq proof assistant. Progress and decidability of type-checking are proved by pen and paper.

Categories and Subject Descriptors

D.3.1 [**Formal Definitions and Theory**]: [Syntax, Semantics]; D.3.2 [**Language Classifications**]: [Applicative (functional) languages, Constraint and logic languages]; F.4.1 [**Mathematical Logic**]: [Lambda calculus and related systems, Logic and constraint programming, Mechanical theorem proving]

General Terms

Languages, Theory.

Keywords

Term Rewriting Systems, Rewriting-calculus, Types, Pattern-Matching, Natural Semantics, Certified Software.

*A full version of this paper is available as `http://www-sop.inria.fr/oasis/Bernard.Serpette/ImpRhoCalculus/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPDP'04, August 24–26, 2004, Verona, Italy.
Copyright 2004 ACM 1-58113-819-9/04/0008 ...$5.00.

1. INTRODUCTION

A promising line of research unifying the logic paradigm with the functional paradigm is that of *rewriting-based* languages (Elan [44], Maude [42], ASF+SDF [35, 49], OBJ* [15], ...). Although these languages are less used than object-oriented languages (Java [34], C# [26], ...), they can also serve as (formal) common typed intermediate languages for implementing compilers for rewriting-based, functional, object-oriented, logic, and other "high-level" modern languages.

One of the main advantages of the rewriting-based languages is *pattern-matching* which allows one to discriminate between alternatives. Once a pattern is recognized, a pattern is associated to 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 make 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.

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 [27, 38], Haskell [40], Scheme [45], or Prolog [39]; 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.

One of the most commonly used models of computation, Lambda-calculus, uses only trivial pattern-matching. This calculus has been extended, initially for programming concerns, either by introducing patterns in Lambda-calculi [30, 50], or by introducing matching and rewrite rules in functional languages.

The *Rewriting-calculus* (Rho) [7,9] 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.

The Rho is computationally complete, since Lambda-calculus and fixed-points can be encoded and type-checked by using *ad hoc* patterns. Thus, Rho comes as a direct generalization of a core of a typed (rewriting-based and functional) programming language (of the ML∪Elan family) in which, roughly speaking, an ML-like let becomes by default a let rec, by abstracting over a suitable pattern P. In fact, through pattern-matching, one can type-check many divergent terms.

One of the main features of the Rewriting-calculus is that it can deal with (de)structuring structures, *e.g.* lists: we record only the names of the constructor and we discard those of the accessors. Since structures are built-in the calculus, it follows that the encoding of constructor/accessors is simpler w.r.t. the standard encoding in the Lambda-calculus. The table below (informally) compares the (untyped) encoding of accessors in both formalisms.

ops/form	Rewriting-calculus	Lambda-calculus
cons	$(\mathsf{cons}\ X\ Y)$	$\lambda XYZ.\ ZXY$
car	$(\mathsf{cons}\ X\ Y) \to X$	$\lambda Z.\ Z(\lambda XY.X)$
cdr	$(\mathsf{cons}\ X\ Y) \to Y$	$\lambda Z.\ Z(\lambda XY.Y)$

This work presents the first version of the *Imperative Rewriting-calculus* (iRho), an extension of Rho with references, memory allocation, and assignment. To our knowledge, no similar study exists in the literature. The iRho-calculus is a "rich" calculus, both at the syntactic and at the semantic level. It features, in a nutshell, all the "idiosyncrasies" of functional/rewriting-based languages with imperative features and modern pattern-matching facilities.

The controlled and conscious use of references, in the style of the ML language [27] also gives the user the programming comfort and expressiveness that would not a priori be expected from such a simple calculus.

The "crucial ingredients" of iRho are the combination of (*i*) modern and safe imperative features, which give full control over the internal data-structure representation, and of (*ii*) the "matching power", which provides the main Lisp-like operations, like cons/car/cdr. For example, iRho make a good theoretical engine for an emerging family of *ad hoc* languages combining rewriting, functions and patterns with semi-structured XML-data, like XDUCE [48], CDUCE [36], or combining object-orientation and patterns with semi-structured data, like HYDROJ [21](*"...object-oriented pattern-matching naturally focuses on the essential information in a message and is insensitive to inessential information..."*), etc. To summarize, even if iRho is a minimalist calculus, its features, like pattern-matching, references, and built-in structures, suggest iRho as a good candidate to be a computational core of a real rewriting-based language.

From Theory to Practice and Vice versa. We design static and dynamic semantics of iRho; the dynamic semantics is given via a natural deduction system *à la* Kahn. The formalization uses *environments* inside "closure-values" to keep the value of free variables in function bodies, and a global *store* to model the imperative traits. We always had in mind the main objectives of a skilled implementor, *i.e.* a sound machine (the interpreter) with a sound type system (the type-checker), respecting the Milner's slogan that "well-typed programs do not go wrong".

Static and dynamic semantics were suitable to be specified with nice mathematics, to be implemented with high-level programming languages, *e.g.* Bigloo [43] (of the Scheme family), and to be certified with a modern and semi-automatic proof assistant, *e.g.* Coq [41].

For this goal, we have *encoded* in Coq the static and dynamic semantics of iRho. All subtle aspects, which are usually "swept under the rug" on the paper, are here highlighted by the rigid discipline imposed by the Logical Framework of Coq. Often, this process has a bearing on the design of the static and dynamic semantics. The continuous cycle between mathematics and manual (*i.e.* pen and paper) *vs.* mechanical proofs, and "toy" implementations using high-level languages such as Scheme (and back) has been fruitful since the very beginning of our project. Although our calculus is rather simple, it is not impossible, in a near future, to scale-up to larger projects, such as the certified implementation of compilers for a "real" programming language of the C family [11].

Therefore, the main contributions of this paper are:

- provide a typed framework that enhances the functional Rho, introduced in [9], with imperative features like referencing (*i.e.* "malloc-like ops", ref term), dereferencing (*i.e.* "goto-memory ops", ! term), and assignments operators (X := term), and enrich the type system with dereferencing-types (*i.e.* pointer-types, int ref), and product-types. The resulting calculus iRho is a good candidate for giving a semantics to a broad family of functional, rewriting, and logic-based languages.

- experiment an interesting "pattern[1]" (in the sense of *"The Gang of Four"* [13]) called DIMPRO, a.k.a. Design-IMplement-PROve, to design safe software, which respects *in toto* its mathematical and functional specifications. 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.

[1] *"A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts..." [31].*

$$\tau \;::=\; \mathsf{b} \mid \tau \to \tau \mid \tau \wedge \tau \mid \tau \;\mathsf{ref}$$

$$\Delta \;::=\; \emptyset \mid \Delta, X{:}\tau \mid \Delta, f{:}\tau$$

$$P \;::=\; X \mid f\,\overline{P} \mid P, P \mid \mathsf{ref}\; P$$

$$A \;::=\; f \mid X \mid P{:}\Delta \to A \mid A\,A \mid A, A \mid \mathsf{ref}\; A \mid\; !A \mid A := A$$

Figure 1: Syntax of iRho.

Road Map. The paper is structured as follows. In Section 2, we present the syntax and the operational semantics of the imperative Rewriting-calculus iRho. Section 3 describes the type system. Section 4 contains various encodings of a quite common decision procedure, *i.e.* computing a negation normal form. All the encodings are type-checked by the type system. For lack of space, Section 5 enumerates the main metatheoretical results. Section 6 contains some hints about our methodology and describes some "views" of the Natural Semantic, conclusions and further work.

For obvious lack of space, this paper is an extended abstract. The full version of this paper containing additional definitions, the complete set of typing rules, a wide collection of functional and imperative examples concerning the static and dynamic semantics of iRho, the Bigloo code for iRho and the Coq encoding of the dynamic and static semantics (with their theorems) can be found in: `http://www-sop.inria.fr/oasis/Bernard.Serpette/ImpRhoCalculus/`.

2. IRHO: IMPERATIVE REWRITING

In a nutshell, iRho is a imperative calculus with pattern-matching, and can be seen as the kernel of any (statically typed) programming language based on functions, mutable variables, and (customizable) pattern-matching; term rewriting systems can be encoded too in iRho, following the same lines as [9]. The presentation of iRho is also original because it mimics our current implementation by making use of *closures* instead of (meta) *substitutions*.

Notational Conventions. In what follows, we use the meta-symbols "$\to$" (function- and type-abstraction), and "," (structure operator), and the (hidden) "$\bullet$" (application operator). We assume that the application operator "$\bullet$" associates to the left, while the other operators associate to the right. The priority of "$\bullet$" is higher than that of "$\to$ " which is, in turn, of higher priority than ",".

The symbols $A, B, C, \ldots$ range over the set $\mathcal{T}$ of terms, the symbols $X, Y, Z, \ldots, \mathsf{SELF}, \ldots$ range over the set $\mathcal{X}$ of variables ($\mathcal{X} \subseteq \mathcal{T}$), the symbols $a, b, c, \ldots, f, \ldots$, $\mathsf{car}, \mathsf{cons}, \mathsf{cdr}, \mathsf{true}, \mathsf{false}, \mathsf{not}, \mathsf{and}, \mathsf{or}, \mathsf{dummy}, \ldots$ range over a set $\mathcal{K}$ of term-constants ($\mathcal{K} \subseteq \mathcal{T}$). The symbol P ranges over the set $\mathcal{P}$ of pseudo-patterns, ($\mathcal{X} \subseteq \mathcal{P}$). The symbol τ ranges over the set $\mathcal{T}y$ of types, the symbol b ranges over the set of type-constants, the symbols Γ, Δ range over contexts. The symbols $A_\mathsf{v}, B_\mathsf{v}, C_\mathsf{v}, \ldots$ range over the set $\mathcal{V}al$ of values. We often denote $f\,\overline{A}$ for $(\ldots((f\,A_1)\,A_2)\ldots A_n)$, for $n \geq 0$. The symbol $\equiv$ denotes syntactic equality.

2.1 Syntax

The syntax of iRho (types, contexts, patterns and terms) is presented in Figure 1.

Types and Contexts. The symbol b denotes basic types, the arrow type $\tau_1 \to \tau_2$ is the type of pattern abstractions $P{:}\Delta \to A$, and the product-type $\tau_1 \wedge \tau_2$ is the type of structure terms (A_1, A_2); finally, the type $\tau\ \mathsf{ref}$ is the type of references containing a value of type τ.

Patterns. It is well-known that an unrestricted use of patterns in lambda-abstraction, may lead to loose confluence; this was pointed out by Vincent van Oostrom [50] which introduced the, so called, *Rigid Pattern Condition* (RPC), which forces patterns to be "linear" (*i.e.* no double occurrences of free variables, thus avoiding, the pattern $f(X, X)$), and without "active" variables (thus avoiding the pattern $(X\,P)$).

The solution we adopt in this presentation of the Rewriting-calculus relaxes, safely, the van Oostrom's condition; the main reason to do this is because most real functional programming languages Scheme, relax the linearity restriction (this is not the case of ML). This means that the pattern $f(X, X)$ is allowed in Rho.

This choice induces also a modification in the classical syntactic pattern-matching algorithm, since we "hide" the first binding in favor of the second one. The original syntactic pattern-matching algorithm, due by Gérard Huet [17], forces both occurrences to be matchable with the same value. As a comparison, both solutions are presented in the table below:

patt $\ll$ term	hide	force
$f(X,X) \ll f(3,4)$	$\theta \triangleq \{4/X\}$	fail
$f(X,X) \ll f(4,4)$	$\theta \triangleq \{4/X\}$	$\theta \triangleq \{4/X\}$

Both solutions for matching are sound, in the sense that confluence and type soundness are not lost. Our choice was suggested by the practice found in languages like Bigloo (recall that non linear pattern are rejected in ML); the non-linearity could be easy implemented with some minor modifications in the definition/implementation/proofs). Therefore, our mathematical definition, together with our current implementations (in Bigloo and in Coq) are, in some sense, synchronized; we "hide" all the bindings of the same variable occurring inside a pattern with the binding of last occurrence; this greatly simplifies the implementation. The "force" solution would be worthy to explore since would lead to a redefinition of equality between terms.

Terms. Intuitively, the main intuition behind the term syntax is as follows:

- *(Variable and Constant)* are used as in Lambda-calculus with algebraic constants;
- *(Structure)* allows one to express structures, like lists, sets, objects, etc.
- *(Pattern Abstraction)* allows one to match over patterns, so giving *de facto* a conservative extension of the Lambda-calculus when the pattern is a simple variable; the context Δ in the pattern abstraction records the types of *all* the free variables of P (possibly bound in the body A); as example, the accessor car (in a homogeneous list) can be written in Rho as follows: $\mathsf{car} \triangleq (\mathsf{cons}\,\mathsf{X}\,\mathsf{Y}){:}(\mathsf{X}{:}\tau, \mathsf{Y}{:}\tau) \to \mathsf{X}$;

- *(Application)* allows one to apply a pattern abstraction $P{:}\Delta \to A$ to an argument B, which, of course must match on P; the terms are reduced under a classical call-by-value evaluation strategy; in the evaluation, the body of a pattern abstraction is not evaluated until the function is called on a suitable value (*i.e.* pattern abstraction are values) ; as example, $(\mathsf{car}(\mathsf{car}(\mathsf{cons\ a\ b})\ \mathsf{c}))$ reduces to a;
- *(Ref-Terms)* The term $\mathsf{ref}\ A$ is a "referencing" term (the-location-of); if A is a term of type τ, then $\mathsf{ref}\ A$ is a pointer to A of type $\tau\ \mathsf{ref}$;
- *(Deref-terms)* The term $!A$ is a "dereferencing" term (goto-memory); term A is a pointer in the store;
- *(Assign-terms)* The term $A := B$ is an "assignment" operator, which returns as result the value obtained by evaluating B.

We need not include sequencing since it can easily be defined in iRho as follows (types are omitted):

$$A\,;B \triangleq (X \to B)\,A \quad \text{where} \quad X \notin \mathsf{Fv}(B)$$

When unambiguous, we introduce the following syntactic-sugar for multiple assignments, *i.e.*:

$$(X_1, \ldots X_n) := (A_1, \ldots A_n) \quad \triangleq \quad X_1 := A_1\,; \ldots\,; X_n := A_n$$

As an immediate benefit of the built-in powerful new pattern-matching algorithm, it follows that also the dereferencing term $!A$ can be easily defined as follows (types are omitted):

$$!A \quad \triangleq \quad (\mathsf{ref}\ X \to X)\,A$$

We leave the dereferencing term in the syntax of iRho as "syntactic sugar". Finally, observe that issues related to garbage collection are out of the scope of the paper: new locations created during reduction, via referencing ($\mathsf{ref}\ A$), will remain in the store forever. In principle, classical techniques of Ian Mason and Carolyn Talcott, and Greg Morrisset *et al.* [23, 29] could be applied to iRho. Observe that, w.r.t. "non strategic" implementations of the Rewriting-calculus [2, 6–8], the delayed matching-constraint $[P \ll_\Delta A].B$, becomes now just syntactic sugar for $(P{:}\Delta \to A)\,B$ (hence omitted from the source language but still presents in the set of output values). Moreover, the shape of patterns has been limited to algebraic terms (*i.e.* no function-as-pattern). This restriction is strictly related to the current software development of our interpreter, and of the current mechanical development of the metatheory underneath iRho and not to theoretical problems (see [2]). The choice of call-by-value too was suggested by the practice of current functional languages.

Values and Environments. In order to define a call-by-value operational semantics of the Rewriting-calculus, we need to introduce the set $\mathcal{V}al$ of *values*, and the set of environments $\mathcal{E}nv$ (historically, and by a little abuse of notation, the symbol ρ ranges over environments), and the set $\mathcal{S}tore$ of *stores* (the symbol σ ranges over the set of stores). The symbol ι ranges over the set $\mathcal{L}oc$ of store locations. Values are defined below:

$$A_\mathsf{v} \ ::= \ f\,\overline{A}_\mathsf{v} \mid A_\mathsf{v}, A_\mathsf{v} \mid \langle P{:}\Delta \to A\,.\,\rho\rangle \mid \langle [P \ll_\Delta A_\mathsf{v}].B\,.\,\rho\rangle \mid \iota$$

Environments are partial functions from the set of variables to the set of values, *i.e.* $\rho \in \mathcal{E}nv \simeq [\mathcal{X} \Rightarrow \mathcal{V}al]_\perp$; the extension of an environment is denoted by $\rho[X \mapsto A_\mathsf{v}]$. Stores are partial functions from the set $\mathcal{L}$ of locations to the set of values *i.e.* $\sigma \in \mathcal{S}tore \simeq [\mathcal{L}oc \Rightarrow \mathcal{V}al]_\perp$; we denote the extension of a store by $\sigma[\iota \mapsto A_\mathsf{v}]$. Environment and stores are defined as follows:

$$\rho[X \mapsto A_\mathsf{v}](Y) \quad \triangleq \quad \begin{cases} A_\mathsf{v} & \text{if } X \equiv Y \\ \rho(Y) & \text{otherwise} \end{cases}$$

$$\sigma[\iota \mapsto A_\mathsf{v}](\iota') \quad \triangleq \quad \begin{cases} A_\mathsf{v} & \text{if } \iota \equiv \iota' \\ \sigma(\iota') & \text{otherwise} \end{cases}$$

Remark 1 (On Failure-values and Exceptions). *"Failure-values" $\langle [P \ll A_\mathsf{v}].B\,.\,\rho\rangle$ denote failures occurring when we cannot find a correct substitution θ on the free variables of P such that $\theta(P) \equiv A_\mathsf{v}$; the environment ρ records the value of the free variables of B. Failure-values are obtained during the computation when a matching failure occurs. Unsuccessful matches generate an error value that does not stuck the interpreter. These can, in principle, be discarded, or caught by a suitable exception handler [8] implemented in the interpreter.*

For the sake of simplicity, dealing with pattern-mismatch errors and pattern-exceptions is out of the scope of this paper (but this feature is available with a "flag-option" in our interpreter written in Bigloo*); in all examples presented in Section 4 when a computation terminates with a success (i.e. not a failure-value), all intermediate failure-values are simply discharged from the final output. The interested reader could have a look at [9] showing necessary extensions/enhancements of an operational semantics and a suitable matching theory that would automatically drop failure-values.*

2.2 Imperative Operational Semantics

We define a call-by-value operational semantics via a natural proof deduction system *à la* Kahn [18]. The present interpreter is "optimistic" since it gives a result if at least one computation does not produce a failure-value: of course other choices are possible, *e.g.* a "pessimistic" interpreter which stops if at least one failure-value occurs. The purpose of the deduction system is to map every expression into a value, *i.e.* an irreducible term in weak-head normal form. The semantics is defined via three judgments of the shape:

$$\begin{aligned} \sigma\,.\,\rho \ &\vdash \ A \Downarrow_{\mathsf{val}} A_\mathsf{v}\,.\,\sigma' \\ \sigma \ &\vdash \ \langle A_\mathsf{v}\,.\,B_\mathsf{v}\rangle \Downarrow_{\mathsf{call}} C_\mathsf{v}\,.\,\sigma' \\ \sigma\,.\,\rho \ &\vdash \ \langle A\,.\,A_\mathsf{v}\rangle \Downarrow_{\mathsf{match}} \rho' \end{aligned}$$

All judgments have as premises a global store σ, which can be modified and returned as a result. In the case of $\Downarrow_{\mathsf{val}}$ and $\Downarrow_{\mathsf{call}}$, a store σ is given as input, and a (possibly modified) store σ' is returned as output. In the $\Downarrow_{\mathsf{match}}$ rule, a store σ is needed as input since our matching algorithm allows to match a referencing terms $\mathsf{ref}\ A$ to a pointer-variable, such as in:

$$[\iota_0 \mapsto 3]\,.\,[Y \mapsto \iota_0] \vdash (\mathsf{ref}\ X{:}(X{:}\mathsf{b}) \to X)Y \Downarrow_{\mathsf{val}} 3\,.\,[\iota_0 \mapsto 3]$$

The rules of the dynamic semantics are defined in Figure 2. In a nutshell:

- $(Red{-}\mathsf{v})$ This rule evaluates every constant to itself;
- $(Red{-}Fun)$ This rule evaluates a pattern abstraction to its closure $\langle P{:}\Delta \to A\,.\,\rho\rangle$;

Value Reduction $\Downarrow_{\mathsf{val}}$

$$\frac{}{\sigma \,.\, \rho \vdash P{:}\Delta \to A \Downarrow_{\mathsf{val}} \langle P{:}\Delta \to A \,.\, \rho\rangle \,.\, \sigma}\ (Red{-}Fun)$$

$$\frac{}{\sigma \,.\, \rho \vdash a \Downarrow_{\mathsf{val}} a \,.\, \sigma}\ (Red{-}\mathsf{v})$$

$$\frac{\sigma_0 \,.\, \rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \,.\, \sigma_1 \quad \sigma_1 \,.\, \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \,.\, \sigma_2}{\sigma_0 \,.\, \rho \vdash A, B \Downarrow_{\mathsf{val}} A_{\mathsf{v}}, B_{\mathsf{v}} \,.\, \sigma_2}\ (Red{-}Struct)$$

$$\frac{X \in \mathsf{Dom}(\rho)}{\rho \vdash X \Downarrow_{\mathsf{val}} \rho(X)}\ (Red{-}Var)$$

$$\frac{\iota \notin \mathsf{Dom}(\sigma_1) \quad \sigma_0 \,.\, \rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \,.\, \sigma_1}{\sigma_0 \,.\, \rho \vdash \mathsf{ref}\ A \Downarrow_{\mathsf{val}} \iota \,.\, \sigma_1[\iota \mapsto A_{\mathsf{v}}]}\ (Red{-}Ref)$$

$$\frac{\iota \in \mathsf{Dom}(\sigma_1) \quad \sigma_0 \,.\, \rho \vdash A \Downarrow_{\mathsf{val}} \iota \,.\, \sigma_1}{\sigma_0 \,.\, \rho \vdash\, !A \Downarrow_{\mathsf{val}} \sigma_1(\iota) \,.\, \sigma_1}\ (Red{-}Deref)$$

$$\frac{\sigma_0 \,.\, \rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \,.\, \sigma_1 \quad \sigma_1 \,.\, \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \,.\, \sigma_2 \quad \sigma_2 \vdash \langle A_{\mathsf{v}} \,.\, B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} C_{\mathsf{v}} \,.\, \sigma_3}{\sigma_0 \,.\, \rho \vdash A\,B \Downarrow_{\mathsf{val}} C_{\mathsf{v}} \,.\, \sigma_3}\ (Red{-}\rho_{\mathsf{v}})$$

$$\frac{\iota \in \mathsf{Dom}(\sigma_1) \quad \sigma_0 \,.\, \rho \vdash A \Downarrow_{\mathsf{val}} \iota \,.\, \sigma_1 \quad \sigma_1 \,.\, \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \,.\, \sigma_2}{\sigma_0 \,.\, \rho \vdash A := B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \,.\, \sigma_2[\iota \mapsto B_{\mathsf{v}}]}\ (Red{-}:=)$$

Call Reduction $\Downarrow_{\mathsf{call}}$

$$\frac{\sigma_0 \,.\, \rho \vdash \langle P \,.\, B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho' \quad \sigma_0 \,.\, \rho' \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \,.\, \sigma_1 \quad (Call{-}FunOk)}{\sigma_0 \vdash \langle\langle P{:}\Delta \to A \,.\, \rho\rangle \,.\, B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} A_{\mathsf{v}} \,.\, \sigma_1}$$

$$\frac{\sigma_0 \vdash \langle A_{\mathsf{v}} \,.\, C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} D_{\mathsf{v}} \,.\, \sigma_1 \quad \sigma_1 \vdash \langle B_{\mathsf{v}} \,.\, C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} E_{\mathsf{v}} \,.\, \sigma_2}{\sigma_0 \vdash \langle (A_{\mathsf{v}}, B_{\mathsf{v}}) \,.\, C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} D_{\mathsf{v}}, E_{\mathsf{v}} \,.\, \sigma_2}\ (Call{-}Struct)$$

$$\frac{\nexists \rho'.\ \sigma \,.\, \rho \vdash \langle P \,.\, B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho' \quad A_{\mathsf{v}} \equiv \langle [P \ll_\Delta B_{\mathsf{v}}].A \,.\, \rho\rangle}{\sigma \vdash \langle\langle P{:}\Delta \to A \,.\, \rho\rangle \,.\, B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} A_{\mathsf{v}} \,.\, \sigma}\ (Call{-}FunKo)$$

$$\frac{A_{\mathsf{v}} \equiv \langle [P \ll_\Delta B_{\mathsf{v}}].A \,.\, \rho\rangle}{\sigma \vdash \langle\langle [P \ll_\Delta B_{\mathsf{v}}].A\rangle \,.\, \rho\rangle \,.\, C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} A_{\mathsf{v}} \,.\, \sigma}\ (Call{-}Wrong)$$

$$\frac{}{\sigma \vdash \langle f\,\overline{A}_{\mathsf{v}} \,.\, B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} f\,\overline{A}_{\mathsf{v}}\,B_{\mathsf{v}} \,.\, \sigma}\ (Call{-}Algbr)$$

Matching Reduction $\Downarrow_{\mathsf{match}}$

$$\frac{}{\sigma \,.\, \rho \vdash \langle X \,.\, A_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho[X \mapsto A_{\mathsf{v}}]}\ (Match{-}Var)$$

$$\frac{}{\sigma \,.\, \rho \vdash \langle a \,.\, a\rangle \Downarrow_{\mathsf{match}} \rho}\ (Match{-}Const)$$

$$\frac{\iota \in \mathsf{Dom}(\sigma) \quad \sigma(\iota) \equiv A_{\mathsf{v}} \quad \sigma \,.\, \rho \vdash \langle P \,.\, A_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho'}{\sigma \,.\, \rho \vdash \langle \mathsf{ref}\ P \,.\, \iota\rangle \Downarrow_{\mathsf{match}} \rho'}\ (Match{-}Ref)$$

$$\frac{\sigma \,.\, \rho_0 \vdash \langle A \,.\, A_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho_1 \quad \sigma \,.\, \rho_1 \vdash \langle B \,.\, B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho_2}{\sigma \,.\, \rho_0 \vdash \langle A \star B \,.\, A_{\mathsf{v}} \star B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho_2}\ (Match{-}Pair)$$

Figure 2: Natural Imperative Semantics.

- ($Red{-}Var$) This rule simply fetches the value of X into the environment;
- ($Red{-}Struct$) This rule simply evaluates the elements of the structure;
- ($Red{-}\rho_{\mathsf{v}}$) This rule reduces the term A to a value A_{v}, then evaluates the argument B in B_{v}, and finally applies A_{v} to B_{v} using the $\Downarrow_{\mathsf{call}}$ judgment;
- ($Red{-}Ref$) This rule first reduces A into a value, and then stores it into a "fresh" location ι;
- ($Red{-}Deref$) This rule first reduces A into a memory location ι, and then read the store at ι;
- ($Red{-}:=$) This rule performs assignment: first we reduce the receiver A into an (existent) memory location, then we reduce the expression B (to be assigned) into a value, and finally we give as result the value produced by B, and a new store which performs the modification *in situ*;
- ($Call{-}FunOk$) This rule first matches successfully P against B_{v}, and then evaluates the body of the pattern abstraction A in the new environment calculated by $\Downarrow_{\mathsf{match}}$;
- ($Call{-}FunKo$) This rule applies when the match of P against B_{v} fails: a failure-value is returned;

- $(Call{-}Struct)$ This rule applies every element of the structure-value to the argument C_{v};
- $(Call{-}Algbr)$ This rule builds an algebraic-value under the shape of an application in weak-head normal form;
- $(Call{-}Wrong)$ This rule applies a failure-value to a value; the failure-value is then propagated;
- $(Match{-}Const)$ Matching two equal constants does not modify the resulting environment;
- $(Match{-}Var)$ Matching a variable against a value produces an environment updated with the new binding;
- $(Match{-}Pair)$ Let $\star \in \{\bullet\ ,\}$. Matching either an application (like "$f\,A_{\mathsf{v}}$") or a structure (like "A, B") produces an environment resulting from the composition of two environments;
- $(Match{-}Ref)$ This rule is the only matching rule which needs a store as an input argument ; it first fetch the value A_{v} in the store σ, at the location ι, and then calls the matching of the pattern P against the value A_{v}. An example of imperative pattern-matching is:

  $$[\iota_0 \mapsto 3]\,.\,[X \mapsto 4] \vdash \langle \mathsf{ref}\ X\,.\,\iota_0\rangle \Downarrow_{\mathsf{match}} [X \mapsto 4][X \mapsto 3]$$

  As said before, this kind of imperative pattern-matching gives the dereferencing term $!A$ the status of simple sugar in iRho.

3. THE TYPE SYSTEM

In this section, we present a type system which allows us to give a type to terms of iRho. Our type discipline assigns a semantical meaning to iRho-programs by type-checking and hence, allows to catch some error before run-time. More precisely, the type system is powerful enough to ensure a type consistency, and to give a type to a rich collection of interesting examples, namely decision procedures, meaningful objects, fixed-points, term rewriting systems, etc. This type system is, in principle, suitable to be extended with a *subtyping* relation, or with *bounded-polymorphism*, to capture the behavior of *structures-as-objects*, and object-oriented features.

The main novelty, with respect to previous type systems for the (functional) Rho [2, 7–9] is that term-structures *can have different types*, *i.e.* we introduce the following new rule for structure:

$$\frac{\Gamma \vdash_{\mathsf{A}} A : \tau_1 \quad \Gamma \vdash_{\mathsf{A}} B : \tau_2}{\Gamma \vdash_{\mathsf{A}} A, B : \tau_1 \wedge \tau_2}\,(Term{-}Struct)$$

The new kind of type $\tau_1 \wedge \tau_2$ (reminiscent of product-types discipline) is suitable for heterogeneous (non-commutative) structures, like lists, ordered sets, or objects. This enhancement gives a more flexible type discipline, where the structure-type $\tau_1 \wedge \tau_2$ reflects the implicit non-commutative property of "," in the term "A, B", *i.e.* "A, B" does not behave necessarily as "B, A". This modification greatly improves expressiveness w.r.t. previous typing disciplines on the Rho [9], in the sense that it gives a type to terms that will not be stuck at run-time, but it complicates the metatheory and the mechanical proof development.

The type system $\vdash_{\mathsf{A}}$ we present is *algorithmic*, in the sense that the type rules are *deterministic* and they allow to describe two *decidable procedures* for type-reconstruction and type-checking. More precisely, a set of rules specifies a deterministic typing algorithm if the type rules are *syntax-directed*, and, moreover, if each rule satisfies the *subformula property*, *i.e.* all the formulas appearing in the premise of a rule are subformulas of those appearing in the conclusion.

The main complication in the type system lies in applying a structure to an argument, thus producing a structure-value by dispatching the argument to all the pattern abstractions contained in the structure.

The structure-value will be typed with a structure-type containing all the components of the structure. As a simple example, if we apply a structure (with type $(\mathsf{b}_1 \to \mathsf{b}_2) \wedge (\mathsf{b}_1 \to \mathsf{b}_3)$) to an argument of type b_1, we would obtain as result a structure-value of type $\mathsf{b}_2 \wedge \mathsf{b}_3$. To capture this behavior (which is a direct consequence of dispatching application into structures), we need the partial function arr on types, which transforms a structure-type into a function-type:

$$\mathsf{arr}(\tau_1 \to \tau_2) \triangleq \tau_1 \to \tau_2$$
$$\mathsf{arr}(\tau_1 \wedge \tau_2) \triangleq \tau_3 \to (\tau_4 \wedge \tau_5) \begin{cases} \text{if} & \mathsf{arr}(\tau_1) \equiv \tau_3 \to \tau_4 \\ \text{and} & \mathsf{arr}(\tau_2) \equiv \tau_3 \to \tau_5 \end{cases}$$

Therefore, the type system of iRho derives judgments of the shape:

$$\Gamma \vdash_{\mathsf{T}} ok \qquad \Gamma \vdash_{\tau} \tau : ok \qquad \Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau$$
$$\Gamma \vdash_{\rho} \rho : \Gamma' \qquad \Gamma \vdash_{\sigma} \sigma : \Gamma' \qquad \Gamma \vdash_{\mathsf{P}} P : \tau \qquad \Gamma \vdash_{\mathsf{A}} A : \tau$$

which denote well-typed contexts, types, values, environments, stores, patterns, and terms, respectively. For the lack of space, we present here only the rules for patterns and terms. In the following, we let the symbol α range over $\mathcal{X} \cup \mathcal{K}$. The type system is given by using rule schema presented in Figure 3. In what follows, we give a review of the most intriguing type-checking rules.

- $(Patt{-}Start), (Term{-}Start)$ Those rules fetch from the context the correct type of variables and constants, respectively;
- $(Patt{-}Struct), (Term{-}Struct)$ Those rules assign a product-type to a structure which records the type of both elements;
- $(Patt{-}Algbr), (Term{-}Appl)$ Those rules deal with application. We discuss the application term-term, the pattern-pattern being similar. The application rule is the usual one can expect for an algorithmic version of a type system; note that, before applying terms, we need to transform the type τ_1 of A into an arrow type, since it could happen that A is a structure containing more branches of the same domain type;
- $(Term{-}Abs)$ In this rule we note that the context Δ is charged in the premises, using the decidable function $\mathsf{Fv}(P)$; the context Γ gives types only for algebraic constants;
- $(Term{-}Assign)$ This rule deals with assignment: the only possible choice is to assign to an expression A, of type τ ref , an object B of type τ;

Pattern Rules

$$\frac{\Gamma_1, \alpha{:}\tau, \Gamma_2 \vdash_\Gamma ok}{\Gamma_1, \alpha{:}\tau, \Gamma_2 \vdash_\mathsf{P} \alpha : \tau}\ (Patt{-}Start)$$

$$\frac{\Gamma \vdash_\mathsf{P} P_1 : \tau_1 \quad \Gamma \vdash_\mathsf{P} P_2 : \tau_2}{\Gamma \vdash_\mathsf{P} P_1, P_2 : \tau_1 \wedge \tau_2}\ (Patt{-}Struct)$$

Term Rules

$$\frac{\Gamma_1, \alpha{:}\tau, \Gamma_2 \vdash_\Gamma ok}{\Gamma_1, \alpha{:}\tau, \Gamma_2 \vdash_\mathsf{A} \alpha : \tau}\ (Term{-}Start)$$

$$\frac{\mathsf{arr}(\tau_1) \equiv \tau_2 \to \tau_3 \quad \Gamma \vdash_\mathsf{P} f\,\overline{P} : \tau_1 \qquad \Gamma \vdash_\mathsf{P} P : \tau_2}{\Gamma \vdash_\mathsf{P} f\,\overline{P}\,P : \tau_3}\ (Patt{-}Algbr)$$

$$\frac{\mathsf{Dom}(\Delta) = \mathsf{Fv}(P) \quad \Gamma, \Delta \vdash_\mathsf{P} P : \tau_1 \qquad \Gamma, \Delta \vdash_\mathsf{A} A : \tau_2}{\Gamma \vdash_\mathsf{A} P{:}\Delta \to A : \tau_1 \to \tau_2}\ (Term{-}Abs)$$

$$\frac{\mathsf{arr}(\tau_1) \equiv \tau_2 \to \tau_3 \quad \Gamma \vdash_\mathsf{A} A : \tau_1 \qquad \Gamma \vdash_\mathsf{A} B : \tau_2}{\Gamma \vdash_\mathsf{A} A\,B : \tau_3}\ (Term{-}Appl)$$

$$\frac{\Gamma \vdash_\mathsf{A} A : \tau_1 \quad \Gamma \vdash_\mathsf{A} B : \tau_2}{\Gamma \vdash_\mathsf{A} A, B : \tau_1 \wedge \tau_2}\ (Term{-}Struct)$$

$$\frac{\Gamma \vdash_\mathsf{A} A : \tau\ \mathsf{ref} \qquad \Gamma \vdash_\mathsf{A} B : \tau}{\Gamma \vdash_\mathsf{A} A := B : \tau}\ (Term{-}Assign)$$

$$\frac{\Gamma \vdash_\mathsf{A} A : \tau}{\Gamma \vdash_\mathsf{A} \mathsf{ref}\,A : \tau\ \mathsf{ref}}\ (Term{-}Ref)$$

$$\frac{\Gamma \vdash_\mathsf{A} A : \tau\ \mathsf{ref}}{\Gamma \vdash_\mathsf{A} !A : \tau}\ (Term{-}Deref)$$

Figure 3: Well Formed Pattern and Terms

- $(Term{-}Ref)$ This rule says that, if an object A has type τ, then a pointer to this object, denoted by $\mathsf{ref}\,A$, has type $\tau\ \mathsf{ref}$;
- $(Term{-}Deref)$ This rule says that, if A is a pointer to an object of type τ, then its access in memory, denoted by $!A$, has type τ.

4. EXAMPLES

This section presents two encodings of a quite common decision procedure, computing a negation normal form; both encodings are type checked by our type system. In what follows, following the algebraic "folklore", we denote the application $(A\,B)$ by $A(B)$.

Example 1 (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 two imperative encodings: in the first, the function is shared via a pointer and recursion is achieved via dereferencing. In the second, formulas are shared too with back-pointers to shared-subtrees. The variable "*SELF*" plays the role of the metavariable "*self*" (or "*this*") common in object-orientation. Then we type-check the encodings. For the sake of readability, all type decorations inside terms are omitted.*

(Imperative, I) *this encoding uses a variable* SELF *which contains a pointer to the recursive code: here the recursion is achieved directly via pointer dereferencing, assignment and classical imperative fixed-point in order to implement recursion. Given the constant* dummy*, the function* nnf1 *is defined as in Figure 4 and the imperative encoding is:*

$$\mathsf{let\ SELF} \ll \mathsf{ref\ dummy\ in\ let\ NNF} = \mathsf{nnf1\ in}\ \mathsf{SELF} := \mathsf{NNF}; \mathsf{NNF}(\phi)$$

(Imperative-with-Sharing, IS) *this encoding uses a variable* SELF *which contains a pointer to the recursive code and a flag-pointer to a boolean value associated to each node: all flag-pointers are initially set to* false*; each time we scan a (possibly) shared-formulas we set the corresponding flag-pointer to* true*. The grammar of shared-formulas is as follows:*

$$\mathsf{bool} ::= \mathsf{true} \mid \mathsf{false} \quad \mathsf{flag} ::= \mathsf{bool\ ref} \quad \psi ::= \mathsf{ref}\,\phi$$

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

The function nnf2 *is defined as in Figure 4 and the imperative encoding is:*

$$\mathsf{let\ SELF} \ll \mathsf{ref\ dummy\ in\ let\ NNF} = \mathsf{nnf2\ in}\ \mathsf{SELF} := \mathsf{NNF}; \mathsf{NNF}(\psi)$$

(Typing The Imperative Encodings) *Fixed-points and* let rec *definitions are introduced using the well-known result of Nax Paul Mendler [24, 25]; in fact, when introducing recursive definitions in the typed Lambda-calculus, the strong normalization is no longer enforced by typing, if the type constructors do not satisfy a* "positiveness condition"*.*

This condition forces an algebraic constructor to be typed without negative occurrences of "recursive", (potentially infinite) entities; This condition is also enforced in the Calculus of Inductive Constructions *(see [14]), which is the basis of the* Coq *proof assistant; the condition avoids inconsistencies in the system itself, such as proving the* Russell Paradox*; termination issues are essentials in Curry-Howard based proof assistants. The same problem also appears*

$$\mathsf{nnf1} \triangleq \left(\begin{array}{lcl} \mathsf{p} & \rightarrow & \mathsf{p}, \\ \mathsf{not(not(X))} & \rightarrow & \mathsf{!SELF(X)}, \\ \mathsf{not(or(X,Y))} & \rightarrow & \mathsf{and(!SELF(not(X)),!SELF(not(Y)))}, \\ \mathsf{not(and(X,Y))} & \rightarrow & \mathsf{or(!SELF(not(X)),!SELF(not(Y)))}, \\ \mathsf{and(X,Y)} & \rightarrow & \mathsf{and(!SELF(X),!SELF(Y))}, \\ \mathsf{or(X,Y)} & \rightarrow & \mathsf{or(!SELF(X),!SELF(Y))} \end{array}\right)$$

$$\mathsf{nnf2} \triangleq \left(\begin{array}{lcl} \mathsf{p} & \rightarrow & \mathsf{p}, \\ \mathsf{not}(B_1, \mathsf{ref\ not}(B_2, \mathsf{X})) & \rightarrow & \mathsf{!SELF(!X)}, \\ \mathsf{not}(B_1, \mathsf{ref\ or}(B_2, \mathsf{X}, \mathsf{Y})) & \rightarrow & \mathsf{and(ref\ false, !SELF(ref\ not(ref\ false, X)), !SELF(ref\ not(ref\ false, Y)))}, \\ \mathsf{not}(B_1, \mathsf{ref\ and}(B_2, \mathsf{X}, \mathsf{Y})) & \rightarrow & \mathsf{or(ref\ false, !SELF(ref\ not(ref\ false, X)), !SELF(ref\ not(ref\ false, Y)))}, \\ \mathsf{and}(B, \mathsf{X}, \mathsf{Y}) & \rightarrow & \mathsf{if\ (neg\ ref}\ B) \ \mathsf{then}\ (B, \mathsf{X}, \mathsf{Y}) := (\mathsf{true}, \mathsf{!SELF(!X)}, \mathsf{!SELF(!Y)}) \\ & & \mathsf{else\ and}(B, \mathsf{X}, \mathsf{Y}), \\ \mathsf{or}(B, \mathsf{X}, \mathsf{Y}) & \rightarrow & \mathsf{if\ (neg\ ref}\ B) \ \mathsf{then}\ (B, \mathsf{X}, \mathsf{Y}) := (\mathsf{true}, \mathsf{!SELF(!X)}, \mathsf{!SELF(!Y)}) \\ & & \mathsf{else\ or}(B, \mathsf{X}, \mathsf{Y}) \end{array}\right)$$

Figure 4: Imperative Encoding with(out) Sharing

in programming languages: for instance, in Caml*, one can define a recursive function without using the keyword* let rec*.*

There are many techniques to efficiently and effectively implement recursive definitions in call-by-value functional languages: among them, it is worth noticing the "in-place update tricks" outlined by Guy Cousineau et al. [10], and the more recent techniques due by Gérard Boudol and Pascal Zimmer [3,4], and by Tom Hirschowitz et al. [16], or the Peter Landin's classical trick [20].

If b *is the type of formulas* ϕ*, and* b ref *is the type of the shared-formulas* ψ*, and* $\overset{n}{\wedge}\tau \triangleq \underbrace{\tau \wedge \cdots \wedge \tau}_{n}$*, and* $\tau_1 \triangleq \mathsf{b} \rightarrow \mathsf{b}$*, and* $\tau_2 \triangleq \mathsf{b\ ref} \rightarrow \mathsf{b\ ref}$*, then the reader can verify that the following judgments are derivable (let* $\Gamma_1 \triangleq \mathsf{dummy}: \overset{n}{\wedge}\tau_1, \mathsf{SELF}: \overset{n}{\wedge}\tau_1\ \mathsf{ref}$*, and* $\Gamma_2 \triangleq \mathsf{dummy}: \overset{n}{\wedge}\tau_2, \mathsf{SELF}: \overset{n}{\wedge}\tau_2\ \mathsf{ref}$*)*

$$(\mathsf{I}) \qquad \Gamma_1, X: \overset{n}{\wedge}\tau_1, \mathsf{NNF}: \overset{n}{\wedge}\tau_1 \ \vdash\ \mathsf{NNF}(\phi) : \overset{n}{\wedge}\mathsf{b}$$

$$(\mathsf{IS}) \qquad \Gamma_2, X: \overset{n}{\wedge}\tau_2, \mathsf{NNF}: \overset{n}{\wedge}\tau_2 \ \vdash\ \mathsf{NNF}(\psi) : \overset{n}{\wedge}\mathsf{b\ ref}$$

Example 2 (Simple First-order Fixed-point [9]). *The type systems of* iRho *relax the classical property that "well-typed programs normalize". More precisely, non-termination can be encoded in our calculus thanks to ad hoc patterns. We present here a term inspired by the classical* Ω *term of the untyped Lambda-calculus. Let* $\Gamma \equiv \mathsf{fix}{:}(\mathsf{b} \rightarrow \mathsf{b}) \rightarrow \mathsf{b}$*, and* $\Delta \equiv X{:}\mathsf{b} \rightarrow \mathsf{b}$*. A derivation for* $\Omega \triangleq \mathsf{fix}(X){:}\Delta \rightarrow X\,\mathsf{fix}(X)$ *is shown in Figure 5. The reader can verify that our interpreter diverges.*

5. MECHANICAL METATHEORY

In the previous sections, we have given a mathematical presentation of iRho better suited to an encoding in Coq. The formalization of iRho in the specification language of the proof assistant is nevertheless a complex task, since we have to face many subtle details which are left implicit on paper. Due to lack of space, here we will briefly discuss the most interesting aspects of this development.

The encoding of iRho in Coq rephrases naturally the previous sections. Adequacy of the Coq encoding w.r.t. the mathematical presentation is proved by pen and paper.

A well-known problem we have to deal with is the encoding of the $\rightarrow$-binder. Binders are known to be difficult to encode in proof assistants; our encoding was essentially based on *closures*, *i.e.* pairs <pattern abstraction · environment>. Environments are partial functions from variables to values. Substitution is replaced by a simple look-up in the environment; variable scoping, and all name-related matters are simply ignored. This technique is widely used in efficient implementations of functional languages, and greatly simplifies mechanical metatheory.

The signature of the encoding of iRho is therefore presented in Figure 6. The natural semantics is given by means of two mutually recursive functions, namely, `eval` and `call`, and a third function `match` devoted to calculate matching; they are sketched in Figure 7. The encoding of the type system is rather intuitive, again by a positive consequence of our DIMPRO pattern. The most intriguing rules are sketched in Figure 8. For obvious lack of space we omit any discussion of the mechanical formalization of iRho in the Coq proof assistant: the interested reader can refers to [22]. Again, for obvious lack of space, we just enumerate the main metatheoretical results. We label with a "$\surd$" all theorems proved by the proof assistant Coq.

(Determinism)$^{\surd}$ If $\sigma \,.\, \rho \vdash A \Downarrow_{\mathsf{val}} A'_{\mathsf{v}} \,.\, \sigma'$, and $\sigma \,.\, \rho \vdash A \Downarrow_{\mathsf{val}} A''_{\mathsf{v}} \,.\, \sigma''$, then $A'_{\mathsf{v}} \equiv A''_{\mathsf{v}}$, and $\sigma' \equiv \sigma''$;

(Unique Type)$^{\surd}$ If $\Gamma \vdash_{\mathsf{A}} A : \tau$, then τ is unique;

(Coherence)$^{\surd}$ $\sigma \,.\, \rho \vdash_{\mathsf{coh}} \Gamma$ if there exist two sub-contexts Γ_1, and Γ_2, such that $\Gamma_1, \Gamma_2 \equiv \Gamma$, and $\Gamma \vdash_{\sigma} \sigma : \Gamma_1$, and $\Gamma \vdash_{\rho} \rho : \Gamma_2$;

(Subject-reduction)$^{\surd}$ If $\emptyset \vdash_{\mathsf{A}} A : \tau$, and $\emptyset.\emptyset \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}}.\sigma$, then there exists Γ' which extend Γ, such that $\Gamma' \vdash_{\sigma} \sigma : ok$, and $\Gamma' \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau$.

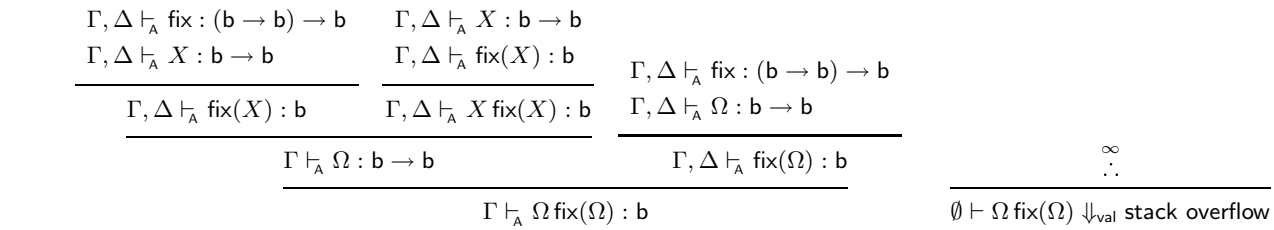

Figure 5: One Fixed-point.

```
Variable basic    :  Set.  Variable eqbasic : basic -> basic -> bool.  Variable var  : Set.              (* Bricks *)
Variable boperator:  Set.  Variable eqvar   : var -> var -> bool.      Variable sbrk : store -> loc.
Definition env    := (PartialFunction var value).  Definition envt    := (PartialFunction var type).
Definition store  := (PartialFunction loc value).  Definition storet  := (PartialFunction loc type).
Definition loc    := nat.                          Definition values  := (list value).
Inductive  type   : Set := Basic    : basic -> type                                                    (* Types *)
                         | FunType  : type  -> type -> type
                         | ProdType : type  -> type -> type
                         | RefType  : type  -> type.
Definition   operator   := boperator * type.
Inductive pattern : Set := POpe     : operator  -> (list pattern) -> pattern                        (* Patterns *)
                         | PVar     : var       -> type           -> pattern
                         | PCons    : pattern   -> pattern        -> pattern
                         | PRef     : pattern   -> pattern.
Definition patterns     := (list pattern).
Inductive  expr   : Set := Ope      : operator  -> expr                                             (* Expressions *)
                         | Var      : var       -> expr
                         | Abs      : pattern   -> expr -> expr
                         | App      : expr      -> expr -> expr
                         | Cons     : expr      -> expr -> expr
                         | Assign   : expr      -> expr -> expr
                         | Ref      : expr      -> expr
                         | Deref    : expr      -> expr.
Inductive value   : Set := VOpe     : operator  -> (list value) -> value                            (* Values *)
                         | Loc      : loc       -> value
                         | Pair     : value     -> value -> value
                         | Closure  : pattern   -> expr  -> env  -> value
                         | Wrong    : pattern   -> value -> expr -> env -> value.
```

Figure 6: Semantics Domains in Coq.

(Type-soundness) If $\emptyset \vdash_{\mathsf{A}} A : \tau$, then $\emptyset \,.\, \emptyset \vdash A \Downarrow_{\mathsf{call}} A_{\mathsf{v}}$;

(Type-reconstruction) It is decidable if, for a given τ, is it true that $\emptyset \vdash_{\mathsf{A}} A : \tau$;

(Type-checking) It is decidable if there a type τ such that $\emptyset \vdash_{\mathsf{A}} A : \tau$.

6. CONCLUSIONS, RELATED, FUTURE

In this paper, we have presented a formal development of the theory of iRho, a typed rewriting-based calculus featuring term-rewriting, pattern-matching on imperative terms, structures, functions, and side-effects. We mix rewriting (for rule-based languages), with functions (for functional languages), structures (for logic-like languages) and safe imperative structures, all "glued" by a "imperative-tolerant" pattern-matching algorithm. To our knowledge, no similar study can be found in the literature.

We presented a clean and compact formalization of iRho in the proof assistant Coq. The Subject Reduction theorem, which is particularly tricky on the paper, was proved in Coq with relatively little effort. The full proof development amounts approximately to `43Kbyte` and the size of the `.vo` file is approximately `1Mbyte`, working with CoqV7.2.

As we said in the introduction, the continuous cycle between mathematics, manual (*i.e.* pen and paper) *vs.* mechanical proofs, "toy" implementations using high-level languages (and back), has been very fruitful since the very beginning of the design of iRho.

We sometime had the feeling that the design using mathematics was driven both by the machine assisted certification and by the software implementation, and that the feedback between those three (usually considered distinct) phases was the crucial point in order to make "safe software."

The lesson learned with iRho, beside from the originality of adding imperative features to a typed calculus featuring functions, pattern-matching, and rewriting, was that the hand of the math's designer must be in strict contact with the hand of the software's designer, which, in turn, must be in strict contact with the hand of the proof's certifier. Our recipe probably suggests a new schema, or "pattern", in the sense of *"The Gang of Four"* [13], for design-implement-certify safe software. This could be subject of future work. A small software interpreted for our

```
Mutual Inductive eval : expr -> env -> store -> value -> store -> Prop :=                    (* Eval *)
...
| evalApp :       (F:expr)(e:env)(s:store)(f:value)(s1:store)
                  (eval F e s f s1)  -> (A:expr)(a:value)(s2:store)
                  (eval A e s1 a s2) -> (v:value)(s3:store)
                  (call f a s2 v s3) ->
                  (eval (App F A) e s v s3)
| evalRef :       (A:expr)(e:env)(s:store)(a:value)(s1:store)
                  (eval A e s a s1)  -> (i:loc)
                  (i=(sbrk s1))      ->
                  (eval (Ref A) e s (Loc i) (extend_store s1 i a))
| evalDeref :     (A:expr)(e:env)(s:store)(i:loc)(s1:store)
                  (eval A e s (Loc i) s1)   -> (v:value)
                  ((s1 i)=(Some value v))   ->
                  (eval (Deref A) e s v s1)
| evalAssign :    (A:expr)(e:env)(s:store)(i:loc)(v:value)(s1:store)
                  (eval A e s (Loc i) s1)   -> (B:expr)(s2:store)
                  (eval B e s1 v s2)        -> (old:value)
                  ((s1 i)=(Some value old)) ->
                  (eval (Assign A B) e s v (extend_store s2 i v))
with call : value -> value -> store -> value -> store -> Prop :=Eval                       (* Call *)
...
| callClosureOK : (P:pattern)(v:value)(s:store)(e,e':env)
                  (match P v s e e') -> (B:expr)(r:value)(s1:store)
                  (eval B e' s r s1) ->
                  (call (Closure P B e) v s r s1).

Inductive match : pattern -> value -> store -> env -> env -> Prop :=                       (* Match *)
...
| matchCons:      (left:pattern)(car:value)(s:store)(e,e':env)
                  (match left car s e e')  -> (right:pattern)(cdr:value)(r:env)
                  (match right cdr s e' r) ->
                  (match (PCons left right) (Pair car cdr) s e r)
| matchRef:       (i:loc)(s:store)(v:value)
                  ((s i)=(Some value v))   -> (x:pattern)(e,r:env)
                  (match x v s e r)        ->
                  (match (PRef x) (Loc i) s e r).
```

Figure 7: Sketch of Natural Imperative Semantics in Coq.

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* [37] (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.

Related. Some implementations of the untyped Rewriting-calculus (uRho) can be found in the literature: among them we recall:

- RhoStratego [46] is an implementation of an early version of the uRho [5], written in the strategic language Stratego [47]. The implementation tests strategic programming with higher-order functional programming;
- Rogue [32] is another implementation of a dialect of the uRho [5]: this implementation is very interesting since some imperative features are added to the language, *e.g.* reading and writing "attributes" of expressions and a fixed strategy. Rogue has an interesting application, namely, it is the implementation language for building a new Validity Checker based on the CVC [33] infrastructure;
- JRho [12] is a Java implementation of uRho [5], using the TOM pattern-matching compiler [28].

Future. The iRho calculus is suitable for extension with more powerful pattern-matching algorithms, and more sophisticated type systems capturing all modern object-oriented features, both class-based and prototype-based ones. Among the possible developments, the next questions on our agenda are:

- add to our type system a subtyping relation; this would allow one to type-check considerably more programs in iRho, by enhancing the type system with bounded polymorphism and object-types, together with the design of a type inference algorithm;
- enhance the calculus with garbage collection: today, new locations created during reduction remain in the store forever; extending the calculus with suitable modern exception mechanisms would be also worth studying;
- analyze, perhaps using abstract interpretation or static analysis techniques, the possibility to statically catch some pattern-matching failures;
- enhance our work, using the DIMPRO pattern, building an abstract machine for iRho;

```
Inductive TypeCheckPattern : envt -> pattern -> envt -> type -> Prop :=                (* Type-check for patterns *)
...
 | tcPOpCons : (E,E1:envt)(op:operator)(lp:patterns)(t:type)
               (TypeCheckPattern E (POp op lp) E1 t)  -> (t1,t2:type)
               (NormalizeFunType t (FunType t1 t2))   -> (P:pattern)(E2:envt)
               (TypeCheckPattern E1 P E2 t1)          ->
               (TypeCheckPattern E (POp op (cons P lp)) E2 t2).
Inductive TypeCheckExpr : envt -> expr -> type -> Prop :=                             (* Type-check for expressions *)
...
 | tcApp :     (E:envt)(F:expr)(t:type)
               (TypeCheckExpr E F t)  -> (t1,t2:type)
               (NormalizeFunType t (FunType t1 t2)) -> (A:expr)
               (TypeCheckExpr E A t1) ->
               (TypeCheckExpr E (App F A) t2)
 | tcRef :     (E:envt)(A:expr)(t:type)
               (TypeCheckExpr E A t)  ->
               (TypeCheckExpr E (Ref A) (RefType t))
 | tcDeref :   (E:envt)(A:expr)(t:type)
               (TypeCheckExpr E A (RefType t)) ->
               (TypeCheckExpr E (Deref A) t)
 | tcAssign :  (E:envt)(A:expr)(t1:type)
               (TypeCheckExpr E A (RefType t1)) -> (B:expr)(t2:type)
               (TypeCheckExpr E B t1) ->
               (TypeCheckExpr E (Assign A B) t1).
Mutual Inductive TypeOf : storet -> value -> type -> Prop :=                          (* Type-check for values *)
...
 | toClosure : (S:storet)(e:env)(E:envt)
               (AbstractEnv S e E) -> (P:pattern)(B:expr)(t1,t2:type)
               (TypeCheckExpr E (Abs P B) (FunType t1 t2)) ->
               (TypeOf S (Closure P B e) (FunType t1 t2))
with AbstractEnv : storet -> env -> envt -> Prop :=       (* Coherence environment-type via coherence store-type *)
...
 | aeExtend :  (S:storet)(e:env)(E:envt)
               (AbstractEnv S e E) -> (v:value)(t:type)
               (TypeOf S v t) -> (x:var)
               (AbstractEnv S (extend_env e x v) (extend_envt E x t)).
Definition AbstractStore: storet -> store -> storet -> Prop :=                        (* Coherence store-type *)
 [S1:storet][s:store][S2:storet]
    (((i:loc)(v:value) (s i)=(Some value v)  ->
           (EX t:type | ((S2 i)=(Some type (RefType t)) /\ (TypeOf S1 v t) )))
/\  ((i:loc) (s i)=(None value)              -> (S2 i)=(None type))).
Definition FixAbstract: env -> store -> envt -> storet -> Prop :=            (* Coherence environment-type-store *)
 [e:env][s:store][E:envt][S:storet] ((AbstractEnv S e E) /\ (AbstractStore S s S)).
```

Figure 8: Sketch of Type-checking Rules in Coq.

- add some *ad hoc* XML primitives to iRho;
- enhance our proof development, in order to reach software extraction via Coq; this would be particularly appealing, since it would eliminate one cycle in our DIMPRO pattern;
- conceive, following the "design pattern" jargon, the pattern DIMPRO;
- apply DIMPRO to the design of a simple compiler from iRho toward an abstract machine, like JVM, or .NET, or to a variant of a Landin's machine [4];

Acknowledgment. The authors would like to thank all the members of the Protheo Team in Nancy for their comments and interactions on Rewriting Calculus. Luigi was visiting the University of Sussex, Brighton; he would like to thank his hosts Matthew Hennessy and Vladimiro Sassone, and the whole Department of Informatics for the ideal working conditions they provided, and Matt Wall for the careful reading of the paper. Finally, the authors are sincerely grateful to all anonymous referees for their extremely useful comments, in particular for pointing out some subtleties in the interaction of patterns, structures and logic programming.

7. REFERENCES

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems. In *Proc. of POPL*, pages 250–261. The ACM Press, 2003.

[3] G. Boudol. The Recursive Record Semantics of Objects Revisited. *Journal of Functional Programming*, 200X.

[4] G. Boudol and P. Zimmer. Recursion in the Call-by-Value Lambda-Calculus. In *Proc. of FICS*, Note Series NS-02-2. BRICS, 2002.

[5] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, 2001.

[6] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

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

[8] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In *Proc. of WRLA*, ENTCS, 2002.

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

[10] G. Cousineau, P.-L. Curien, and M. Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[11] Cristal, Foc-CNAM, Lemme, Mimosa, Miró, and Oasis. Concert: Compilateurs Certifiés, 2004. ARC INRIA 2003-2004, `http://www-sop.inria.fr/lemme/concert`.

[12] G. Faure and P. Moreau. Jrho: a Java Implementation of the Rho Calculus, 2002. `http://elan.loria.fr/Soft/jrho-0.1.tar.gz`.

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

[14] E. Gimenez. Structural Recursive Definitions in Type Theory. In *Proc. of ICALP*, pages 397–408, 1998.

[15] J. Goguen. The OBJ Family Home Page, 2004. `http://www.cs.ucsd.edu/users/goguen/sys/obj.html`.

[16] T. Hirschowitz, X. Leroy, and J. B. Wells. Compilation of Extended Recursion in Call-by-Value Functional Languages. In *Proc. of PPDP*. The ACM Press, 2003.

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

[18] G. Kahn. Natural Semantics. In *Proc. of STACS*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[19] S. N. Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In *Proc. of POPL*, pages 80–87. The ACM Press, 1988.

[20] P. J. Landin. The Mechanical Evaluation of Expression. *The Computer Journal*, 6:308–320, 1964.

[21] K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-Oriented Pattern Matching for Evolvable Distributed Systems. In *Proc. of OOPSLA*. The ACM Press, 2003.

[22] L. Liquori and B. Serpette. The Full Version of this paper, 2004. `http://www-sop.inria.fr/oasis/Bernard.Serpette/ImpRhoCalculus/`.

[23] I. A. Mason and C. L. Talcott. References, Local Variables and Operational Reasoning. In *Proc. of LICS*, pages 66–77, 1992.

[24] N. P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, USA, 1987.

[25] N. P. Mendler, P. Panangaden, and R. L. Constable. Infinite Objects in Type Theory. In *Proc. of LICS*, pages 249–255, 1986.

[26] Microsoft. The C# Home Page, 2004. `http://msdn.microsoft.com/vcsharp/`.

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

[28] P. Moreau, C. Ringeissen, and M. Vittek. The Tom Home Page, 2004. `http://tom.loria.fr/`.

[29] J. G. Morrisett, M. Felleisen, and R. Harper. Abstract Models of Memory Management. In *Proc. of FPCA*, pages 66–77. The ACM Press, 1995.

[30] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[31] D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.

[32] A. Stump. The Rogue Home Page, 2004. `http://www.cse.wustl.edu/~stump/rogue.html`.

[33] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In *CAV*, 2002. System Description.

[34] Sun. Java Technology, 2004. `http://java.sun.com/`.

[35] The Asf+Sdf Team. The Asf+Sdf Meta-Environment Home Page, 2004. `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment`.

[36] The Cduce Team. The Cduce Home Page, 2004. `http://www.cduce.org`.

[37] The Common Criteria Consortium. The Common Criteria Home Page, 2004. `http://www.commoncriteria.org`.

[38] The Cristal Team. The Objective Caml Home Page, 2004. `http://www.ocaml.org/`.

[39] The GNU Prolog Team. The GNU Prolog Home Page, 2004. `http://pauillac.inria.fr/~diaz/gnu-prolog/`.

[40] The Haskell Team. The Haskell Home Page, 2004. `http://www.haskell.org/`.

[41] The Logical Team. The Coq Home Page, 2004. `http://coq.inria.fr`.

[42] The Maude Team. The Maude Home Page, 2004. `http://maude.cs.uiuc.edu/`.

[43] The Mimosa Team. The Schme Bigloo Home Page, 2004. `http://www.sop.inria.fr/mimosa/fp/bigloo/`.

[44] The Protheo Team. The Elan Home Page, 2004. `http://elan.loria.fr`.

[45] The Scheme Team. The Scheme Language, 2004. `http://www.swiss.ai.mit.edu/projects/scheme/`.

[46] The Stratego Team. The Rho Stratego Home Page, 2004. `http://www.stratego-language.org/twiki/bin/view/Stratego/RhoStratego`.

[47] The Stratego Team. The Stratego Home Page, 2004. `http://www.stratego-language.org`.

[48] The Xduce Team. The Xduce Home Page, 2004. `http://xduce.sourceforge.net`.

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

[50] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.