iRho: An Imperative Rewriting Calculus

(Extended Abstract)

Luigi Liquori and Bernard Paul Serpette

INRIA, France

First.Last@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, subject reduction, and decidability of type-checking 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.

Keywords. Rewriting-calculus with side-effects, Pattern-matching, Call-by-value, Type theory, Theorem proving.

Introduction

A promising line of research unifying the logic paradigm with the functional paradigm is that of *rewriting-based* languages (Elan [46], Maude [44], ASF+SDF [35, 51], OBJ* [15], ...). Although these languages are less used than object-oriented languages (Java [34], C# [26], ...), they can also serve as (formal) common 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.

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 [27,39], Haskell [42], Scheme [47], or Prolog [41]; 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 recently been extended, initially for programming concerns, either by introducing patterns in Lambda-calculi [30, 52], or by introducing matching and rewrite rules in functional languages.

The functional *Rewriting-calculus* (fRho) [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 $P \twoheadrightarrow A$, 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 $P \twoheadrightarrow A$ 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 functional fragment fRho is Turing complete, since Lambda-calculus and fixed-points can be encoded and type-checked by using *ad hoc* patterns. Thus, fRho is the core of a programming language 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	$X \twoheadrightarrow Y \twoheadrightarrow (\mathsf{cons}\ X\ Y)$	$\lambda XYZ.\ ZXY$
car	$(\mathsf{cons}\ X\ Y) \twoheadrightarrow X$	$\lambda Z.\ Z(\lambda XY.X)$
cdr	$(\mathsf{cons}\ X\ Y) \twoheadrightarrow Y$	$\lambda Z.\ Z(\lambda XY.Y)$

This work presents the first version of the imperative *Rewriting-calculus* (iRho), an extension of fRho with references, memory allocation, and assignment. To our knowledge, no similar study exists. 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 Caml language [38] (and other languages of the ML family), also gives the user the programming comfort and expressiveness that would not a priori be expected from such a simple calculus.

The "magic 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 functions, rewriting, and patterns with semi-structured XML-data, like XDUCE [50], 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 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 where suitable to be specified with nice mathematics, to be implemented with high-level programming languages, *e.g.* Bigloo [45] (of the Scheme family), and to be certified with a modern and semi-automatic proof assistant, *e.g.* Coq [43].

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 (see Figure 11). 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:

$$\tau ::= \mathsf{b} \mid \tau \twoheadrightarrow \tau \mid \tau \wedge \tau \quad \text{Types} \qquad P ::= X \mid a\,\overline{P} \mid P, P \quad \text{Patterns}$$

$$\Delta ::= \emptyset \mid \Delta, X{:}\tau \mid \Delta, a{:}\tau \quad \text{Contexts} \qquad A ::= a \mid X \mid P{:}\Delta \twoheadrightarrow A \mid A\,A \mid A, A \quad \text{Terms}$$

Fig. 1. Syntax of fRho.

- provide a typed framework that enhances the functional fRho, 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.

Road Map. The paper is structured as follows. In Section 1 and 2, we present the syntax and the operational semantics of the functional fRho and then 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 imperative encoding are type-checked by our type system. Section 5 presents the formal model and the proof of the metatheory. All lemmas and theorems in this section are encoded and proved in Coq. Section 6 contains some hints about our methodology and describes some "views" of the Natural Semantic, conclusions and further work.

An Appendix gives additional definitions, typing rules, the main meta-theoretical properties proved by pen and paper, and a wide collection of functional and imperative examples concerning the dynamic and the static semantics of fRho and of iRho. A Web-appendix [22] contains both the Bigloo code for iRho and the Coq encoding of the dynamic and static semantics (with their theorems).

1 The Functional Rewriting-calculus

For pedagogical reasons, we start by presenting the fRho calculus. This will allow us to introduce almost all the ingredients and the technicalities needed to scale-up to the full iRho. In a nutshell, fRho is a functional calculus with pattern-matching, and can be seen as the kernel of any (statically typed) programming language based on functions and (customizable) pattern-matching; term rewriting systems can be encoded too in iRho, following the same lines as [9]. The presentation of fRho is original because it mimics our current implementation by making use of closures instead of meta substitutions.

1.1 Functional Syntax

Syntax. The syntax of fRho (types and terms) is presented in Figure 1, where we let the meta-symbols "$\twoheadrightarrow$" (function- and type-abstraction), and "," (structure operator), and the (hidden) "$\bullet$" (application operator). The symbol ";" ranges over the set $\{\bullet\ ,\ \}$. 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 "$\twoheadrightarrow$ " which is, in turn, of higher priority than ",".

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

The symbols $A, B, C, \ldots$ range over the set $\mathcal{T}$ of terms, the symbols $X, Y, Z, \ldots, \mathsf{SELF}, \mathsf{Self}, \ldots$ 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 range 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 sometime denote $\overline{A}$ for $A_1 \cdots A_n$, for $n \geq 0$. The application of a constant function, say f, to a term A will be usually denoted by $f(A)$, following the algebraic "folklore"; this convention can be "curryfied" in order to denote a function taking multiple arguments, *e.g.* $f(A_1 \cdots A_n) \triangleq f\ A_1\ \cdots A_n$. The symbol $\equiv$ denotes syntactic equality.

Types and Contexts. The symbol b denotes basic types, the arrow-type $\tau_1 \twoheadrightarrow \tau_2$ is the type of pattern abstractions $P{:}\Delta \twoheadrightarrow A$, and the product-type $\tau_1 \wedge \tau_2$ is the type of structure terms (A_1, A_2).

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 [52] 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, *e.g.* the pattern $f(X, X)$), and without "active" variables (thus avoiding *e.g.* 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 (like Scheme and ML) relax the linearity restriction. This means that the pattern $f(X, X)$ is allowed.

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 Caml or Bigloo. 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, and some research directions are sketched in the conclusions.

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 we be written in fRho as follows: $\mathsf{car} \triangleq \mathsf{cons}(\mathsf{X}, \mathsf{Y}){:}(\mathsf{X}{:}\tau_1, \mathsf{Y}{:}\tau_2) \twoheadrightarrow \mathsf{X}$;
- *(Application)* allows one to apply a pattern abstraction $P{:}\Delta \twoheadrightarrow A$ to a "value" B_v, which, of course must match on P; the terms are reduced under a call-by-value evaluation strategy; lazy evaluation can be simulated, because the body of a pattern abstraction is not evaluated until the function is called; as example, $\mathsf{car}(\mathsf{cons}(\mathsf{a}, \mathsf{b}))$ reduces to a;

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 $(P{:}\Delta \twoheadrightarrow A)\, B$ (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). They are defined below:

$$A_{\mathsf{v}} ::= a\,\overline{A_{\mathsf{v}}} \mid A_{\mathsf{v}}, A_{\mathsf{v}} \mid \langle P{:}\Delta \twoheadrightarrow A \cdot \rho\rangle \mid \langle [P \ll_{\Delta} A_{\mathsf{v}}].B \cdot \rho\rangle \qquad \text{Functional Values}$$

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 $\rho[X \mapsto A_{\mathsf{v}}]$ with the following meaning:

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

Remark 1 (On Failure-values and Exceptions). "Failure-values" $\langle [P \ll A_{\mathsf{v}}].B \cdot \rho\rangle$ 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. 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 and in Appendix A.3, 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.

1.2 The Rhosetta Stone.

The fRho calculus is well-known as a conservative extension of the Lambda-calculus, with built-in pattern-matching facilities. Nevertheless, historically it is often presented using an infix notation, using as binder the meta-symbol "arrow" ($\twoheadrightarrow$), instead of the *infix* notation using as a binder the meta-symbol "lambda" (λ) in conjunction with the meta-symbol "point" (.). Moreover, since an abstraction can bind in its body more than one variable, the decoration of the pattern is given by a "context" (Δ) instead of a simple type. The rationale is:

$$\lambda X{:}\tau_1.A \;\simeq\; \lambda X{:}\underbrace{X{:}\tau_1}_{\Delta}.A \;\simeq\; X{:}\underbrace{X{:}\tau_1}_{\Delta} \twoheadrightarrow A \qquad \text{for trivial patterns, a.k.a. variables}$$

$$\lambda f(X\,Y){:}\underbrace{X{:}\tau_1, Y{:}\tau_2}_{\Delta}.A \;\simeq\; f(X\,Y){:}\underbrace{X{:}\tau_1, Y{:}\tau_2}_{\Delta} \twoheadrightarrow A \qquad \text{for algebraic pattern}$$

Since the context Δ declares the types of all the free variables of P, we have:

$$\mathsf{Fv}(P{:}\Delta \twoheadrightarrow A) \triangleq \mathsf{Fv}(A) \setminus \mathsf{Fv}(P)$$

Let-like and conditionals. As usual, let-like constructs can be generalized with pattern and becomes simple syntactic sugar for applications (types are omitted), *i.e.*

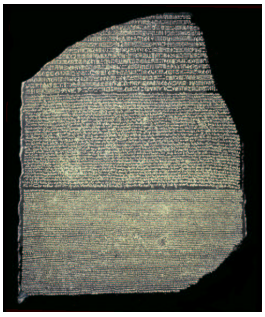
$$\mathsf{let}\ P \ll A\ \mathsf{in}\ B \triangleq (P \twoheadrightarrow B)\,A$$

Conditional too can be easily encoded in fRho, using pair and applications (true, false are constants), *i.e.*

$$\mathsf{if}\ A\ \mathsf{then}\ B\ \mathsf{else}\ C \triangleq (\mathsf{true} \twoheadrightarrow B, \mathsf{false} \twoheadrightarrow C)\,A \qquad \mathsf{neg} \triangleq (\mathsf{true} \twoheadrightarrow \mathsf{false}, \mathsf{false} \twoheadrightarrow \mathsf{true})$$

Pair encoding. It is also well-known that structures can be easily encoded in the Lambda-calculus, using the standard pair-encoding.

The *"Rhosetta"* stone (presented in Figure 2) gives an intuitive comparison between the lambda-like notation and the arrow-like one, with a particular focus on the pair/projection encoding.



Contexts Syntax

$$\Delta ::= \emptyset \mid \Delta, X{:}A \mid \Delta, a{:}A$$

Arrow-like Syntax

$$A ::= P{:}\Delta \twoheadrightarrow A \mid \quad A, B \quad \mid \overbrace{((X_1, X_2){:}\underbrace{X_1{:}\tau_1, X_1{:}\tau_2}_{\Delta} \twoheadrightarrow X_{1,2})\,(A, B)}^{\mathrm{proj}} \mid \ldots$$

Lambda-like Syntax

$$A ::= \lambda P{:}\Delta.A \quad \mid \underbrace{\lambda X{:}\tau_3.XA\,B}_{\mathrm{pair}} \mid \overbrace{(\lambda X{:}\tau_4.\lambda Y{:}\tau_3.Y\,X)\,\mathrm{pair}\,\underbrace{\lambda X_1{:}\tau_1.\lambda X_2{:}\tau_2.X_{1,2}}_{\mathrm{bool}}}^{\mathrm{proj}} \mid \ldots$$

with $\tau_3 \equiv \tau_1 \twoheadrightarrow \tau_2 \twoheadrightarrow \tau_{1,2}$ and $\tau_4 \equiv \tau_3 \twoheadrightarrow \tau_{1,2}$

Fig. 2. The Rhosetta (Functional) Stone (196 b.c., courtesy of The British Museum).

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

$$\frac{}{\rho \vdash a \Downarrow_{\mathsf{val}} a}\,(Red{-}\mathsf{v}) \qquad \frac{}{\rho \vdash P{:}\Delta \twoheadrightarrow A \Downarrow_{\mathsf{val}} \langle P{:}\Delta \twoheadrightarrow A \cdot \rho\rangle}\,(Red{-}Fun) \qquad \frac{X \in \mathsf{Dom}(\rho)}{\rho \vdash X \Downarrow_{\mathsf{val}} \rho(X)}\,(Red{-}Var)$$

$$\frac{\rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \quad \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \quad \vdash \langle A_{\mathsf{v}} \cdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} C_{\mathsf{v}}}{\rho \vdash A\,B \Downarrow_{\mathsf{val}} C_{\mathsf{v}}}\,(Red{-}\rho_{\mathsf{v}}) \qquad \frac{\rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \quad \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}}}{\rho \vdash A, B \Downarrow_{\mathsf{val}} A_{\mathsf{v}}, B_{\mathsf{v}}}\,(Red{-}Struct)$$

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

$$\frac{\rho \vdash \langle P \cdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho' \quad \rho' \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}}}{\vdash \langle\langle P{:}\Delta \twoheadrightarrow A \cdot \rho\rangle \cdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} A_{\mathsf{v}}}\,(Call{-}FunOk) \qquad \frac{\vdash \langle A_{\mathsf{v}} \cdot C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} D_{\mathsf{v}} \quad \vdash \langle B_{\mathsf{v}} \cdot C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} E_{\mathsf{v}}}{\vdash \langle (A_{\mathsf{v}}, B_{\mathsf{v}}) \cdot C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} D_{\mathsf{v}}, E_{\mathsf{v}}}\,(Call{-}Struct)$$

$$\frac{\nexists \rho'.\ \rho \vdash \langle P \cdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho'}{\vdash \langle\langle P{:}\Delta \twoheadrightarrow A \cdot \rho\rangle \cdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} \langle [P \ll_\Delta B_{\mathsf{v}}].A \cdot \rho\rangle}\,(Call{-}FunKo) \qquad \frac{}{\vdash \langle a\,\overline{A}_{\mathsf{v}} \cdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} a\,\overline{A}_{\mathsf{v}}\,B_{\mathsf{v}}}\,(Call{-}Algbr)$$

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

Fig. 3. Natural Functional Semantics.

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

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

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

Fig. 4. Natural Matching Semantics.

1.3 Functional 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.

τ	$::=$	... as in fRho ... $\mid \tau$ ref	Types	P	$::=$	... as in fRho ... $\mid$ ref P	Patterns
Δ	$::=$	... as in fRho ...	Contexts	A	$::=$	... as in fRho ... $\mid$ ref $A \mid !A \mid A := A$	Terms

Fig. 5. Syntax of iRho.

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:

$$\rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \quad \text{and} \quad \vdash \langle A_{\mathsf{v}} \cdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{call}} C_{\mathsf{v}} \quad \text{and} \quad \rho \vdash \langle A \cdot A_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho' .$$

The first judgment evaluates a term in fRho, the second applies a value to another, producing a result value, and the last updates a correct environment obtained by matching a term against a value. All the rules are presented in Figure 3 and 4. In a nutshell:

- $(Red{-}\mathsf{v})$ This rule evaluates every constant to itself;
- $(Red{-}Fun)$ This rule evaluates a pattern abstraction to a closure;
- $(Red{-}Var)$ 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;
- $(Call{-}FunOk)$ 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;
- $(Match{-}Pair)$ Matching either an application or a structure (recall that $; \in \{\bullet\ , \}$) produces an environment resulting from the composition of two environments.

The natural operational semantics is deterministic, and immediately suggests how to build an interpreter for the calculus.

2 The Imperative Rewriting-calculus

2.1 Imperative Syntax

We introduce imperative traits in our Rewriting-calculus, to yield the full iRho. Hence, we extended the syntax of terms by adding (de)referencing and assignment operators, by extending the set of values and contexts including references, by adding new reference-types, and by recasting our natural semantics with store locations and environments.

Syntax. The syntax of iRho (types, contexts, patterns and terms) is presented in Figure 5. Intuitively, iRho deals with references *à la* Caml *i.e.*:

- **(Ref-types)** The type τ ref is the type of references containing a value of type τ;
- **(Ref-Terms)** The term ref A is a "referencing" term (the-location-of); if A is a term of type τ, then ref A is a pointer to A of type τ 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):

$$A\,;B \triangleq (X \multimap B)\,A \qquad X \notin \mathsf{Fv}(B)$$

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

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

Moreover, thanks again to the built-in powerful pattern-matching, it follows that also the dereferencing term $!A$ can be easily defined as follows (types are omitted):

$$!A \;\triangleq\; (\mathsf{ref}\, X \multimap X)\,A$$

We leave the dereferencing term in the syntax of iRho as "syntactic sugar". Finally, observe that issues related to garbage collection as 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.

Values and Stores. The new set of values is enriched by locations. The symbol ι ranges over the set $\mathcal{L}oc$ of store locations, and the symbol σ ranges over the set of global stores $\mathcal{S}tore$.

$$A_{\mathsf{v}} ::= \ldots \text{ as in } \mathsf{fRho} \ldots \mid \iota \qquad \text{Imperative Values}$$

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 $\sigma[\iota \mapsto A_{\mathsf{v}}]$ with the following meaning:

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

2.2 Imperative Operational Semantics

As in the functional case, we define an "optimistic" operational semantics via a natural proof deduction system *à la* Kahn [18]. Again, the chosen strategy is *call-by-value*. The semantics is defined via three judgments of the shape:

$$\sigma \cdot \rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \cdot \sigma' \quad \text{and} \quad \sigma \vdash \langle A_{\mathsf{v}} \cdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{call}} C_{\mathsf{v}} \cdot \sigma' \quad \text{and} \quad \sigma \cdot \rho \vdash \langle A \cdot A_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho'.$$

The main difference w.r.t. the functional calculus fRho is that 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:

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

The rules of the dynamic semantics are defined in Figure 6. In a nutshell:

– $(Red{-}\{\mathsf{v}, \rho_{\mathsf{v}}, Var, Fun, Struct\})$ All those rules essentially behave as in the functional case, with the exception that the store parameter is propagated over the judgments in the premises and in the conclusion;
– $(Red{-}Ref)$ 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*;
– $(Call{-}\{FunOk, Struct, FunKo, Algbr, Wrong\})$ Those rules present no surprise w.r.t. the corresponding rules in fRho; the only difference lies in the store propagation from the input of the conclusion (through the premises) to the output of the conclusion ($\mathsf{value} \cdot \mathsf{store}$);

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

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

Plus the update of $(Red{-}\mathsf{v}), (Red{-}Fun), (Red{-}Var)$ with the extra (unused) store parameter.

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

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

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

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

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

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

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

Plus the update of $(Call{-}FunKo), (Call{-}Algbr), (Call{-}Wrong)$, with the extra (unused) store parameter.

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

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

Plus the update of $(Match{-}Const), (Match{-}Var), (Match{-}Pair)$ with the extra (unused) store parameter.

Fig. 6. Natural Imperative Semantics.

- $(Match{-}\{Const, Var, Pair\})$ All the matching judgments of Figure 4 are still valid by adding an (unused) extra parameter σ;
- $(Match{-}Ref)$ This rule is intriguing; first of all, it is the only matching rule which needs a store as an input argument ; it first read the value A_{v} in the store σ, and then calls recursively the matching of the pattern P against the value A_{v}. An example of imperative pattern-matching is:

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

  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-flow" 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 aspects.

The main novelty, with respect to previous type systems of the Rewriting-calculus [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 functional Rewriting-calculus [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-checking and type reconstruction. 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 $(\mathsf{b}_1 \twoheadrightarrow \mathsf{b}_2) \wedge (\mathsf{b}_1 \twoheadrightarrow \mathsf{b}_3)$) 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:

$$\mathsf{arr}(\tau_1 \twoheadrightarrow \tau_2) \triangleq \tau_1 \twoheadrightarrow \tau_2$$

$$\mathsf{arr}(\tau_1 \wedge \tau_2) \triangleq \tau_3 \twoheadrightarrow (\tau_4 \wedge \tau_5) \text{ if } \mathsf{arr}(\tau_1) \equiv \tau_3 \twoheadrightarrow \tau_4 \text{ and } \mathsf{arr}(\tau_2) \equiv \tau_3 \twoheadrightarrow \tau_5$$

Therefore, the type system of iRho derives (among others see Appendix A.1) judgments of the shape:

$$\Gamma \vdash_{\mathsf{P}} P : \tau, \quad \text{and} \quad \Gamma \vdash_{\mathsf{A}} A : \tau$$

which denote well-typed patterns and terms, respectively. In the following, we let the symbol χ range over $\mathcal{X} \cup \mathcal{K}$. The type system is completed by using rule schema presented in Figure 7. In what follows, we give a review the type-checking rules and we focus on the most intriguing ones.

- $(Patt{-}Start), (Term{-}Start)$ 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;
- $(Term{-}Ref)$ This rule says that, if an object A has type τ, then a pointer to this object, denoted by $\mathsf{ref}\ A$, has type τ 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.

Pattern Rules

$$\frac{\Gamma \equiv \Gamma_1, \chi{:}\tau, \Gamma_2 \quad \chi \notin \mathsf{Dom}(\Gamma_1)}{\Gamma \vdash_{\mathsf{P}} \chi : \tau}\,(Patt{-}Start) \qquad \frac{\Gamma \vdash_{\mathsf{P}} P_1 : \tau_1 \quad \Gamma \vdash_{\mathsf{P}} P_2 : \tau_2}{\Gamma \vdash_{\mathsf{P}} P_1, P_2 : \tau_1 \wedge \tau_2}\,(Patt{-}Struct)$$

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

Term Rules

$$\frac{\Gamma \equiv \Gamma_1, \chi{:}\tau, \Gamma_2 \quad \chi \notin \mathsf{Dom}(\Gamma_1)}{\Gamma \vdash_{\mathsf{A}} \chi : \tau}\,(Term{-}Start) \qquad \frac{\Gamma \vdash_{\mathsf{A}} A : \tau_1 \quad \Gamma \vdash_{\mathsf{A}} B : \tau_2}{\Gamma \vdash_{\mathsf{A}} A, B : \tau_1 \wedge \tau_2}\,(Term{-}Struct)$$

$$\frac{\mathsf{Dom}(\Delta) = \mathsf{Fv}(P) \quad \Gamma, \Delta \vdash_{\mathsf{P}} P : \tau_1 \quad \Gamma, \Delta \vdash_{\mathsf{A}} A : \tau_2}{\Gamma \vdash_{\mathsf{A}} P{:}\Delta \rightarrowtriangle A : \tau_1 \rightarrowtriangle \tau_2}\,(Term{-}Abs) \qquad \frac{\mathsf{arr}(\tau_1) \equiv \tau_2 \rightarrowtriangle \tau_3 \quad \Gamma \vdash_{\mathsf{A}} A : \tau_1 \quad \Gamma \vdash_{\mathsf{A}} B : \tau_2}{\Gamma \vdash_{\mathsf{A}} A\,B : \tau_3}\,(Term{-}Appl)$$

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

Fig. 7. Well Formed Pattern and Terms

4 Examples

This section presents various encodings of a quite common decision procedure, computing a negation normal form. All imperative encodings are type checked by our type system. More examples can be found in Appendix A.3.

Example 1 (Computing a Negation Normal Form). This function is used in implementing *decision procedures*, present in almost all model checkers. The processed input is an implication-free languages 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 is purely functional, since it just makes use of recursion via self-application. The second is imperative: the function is shared via a pointer and recursion is achieved via dereferencing. The third encoding is still imperative; it manipulates formulas with sharing (back-pointers to shared-subtrees). Then we type-check all the imperative encodings. For the sake of readability, all type decorations inside terms are omitted.

(Functional, F**)** this encoding uses the Samuel Kamin's *self application* mechanism [19] to implement the recursion via the *object itself*. This technique is well-understood in object-oriented languages, where message passing is formalized via the "dot notation". The variable "Self" plays the role of the metavariable "self" (or "this") common in object-orientation. More precisely, we use the following notation as syntactic sugar in iRho:

$$\mathsf{obj.meth} \triangleq \mathsf{obj}(\mathsf{meth}(\mathsf{obj}))$$

The function nnf is defined as follows, and called as in, *e.g.* $(\mathsf{nnf.fix})\,\phi$.

$$\mathsf{nnf} \triangleq \mathsf{fix}(\mathsf{Self}) \rightarrowtriangle \begin{pmatrix} \mathsf{p} & \rightarrowtriangle \mathsf{p}, \\ \mathsf{not}(\mathsf{not}(\mathsf{X})) & \rightarrowtriangle \mathsf{Self.fix}(\mathsf{X}), \\ \mathsf{not}(\mathsf{or}(\mathsf{X},\mathsf{Y})) & \rightarrowtriangle \mathsf{and}(\mathsf{Self.fix}(\mathsf{not}(\mathsf{X})), \mathsf{Self.fix}(\mathsf{not}(\mathsf{Y}))), \\ \mathsf{not}(\mathsf{and}(\mathsf{X},\mathsf{Y})) & \rightarrowtriangle \mathsf{or}(\mathsf{Self.fix}(\mathsf{not}(\mathsf{X})), \mathsf{Self.fix}(\mathsf{not}(\mathsf{Y}))), \\ \mathsf{and}(\mathsf{X},\mathsf{Y}) & \rightarrowtriangle \mathsf{and}(\mathsf{Self.fix}(\mathsf{X}), \mathsf{Self.fix}(\mathsf{Y})), \\ \mathsf{or}(\mathsf{X},\mathsf{Y}) & \rightarrowtriangle \mathsf{or}(\mathsf{Self.fix}(\mathsf{X}), \mathsf{Self.fix}(\mathsf{Y})) \end{pmatrix}$$

(Imperative, I) this imperative 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 nnf is defined as follows:

$$\mathsf{nnf} \triangleq \mathsf{SELF} \rightarrow \begin{pmatrix} \mathsf{p} & \rightarrow \mathsf{p}, \\ \mathsf{not}(\mathsf{not}(\mathsf{X})) & \rightarrow \mathsf{SELF}(\mathsf{X}), \\ \mathsf{not}(\mathsf{or}(\mathsf{X},\mathsf{Y})) & \rightarrow \mathsf{and}(\mathsf{SELF}(\mathsf{not}(\mathsf{X})), \mathsf{SELF}(\mathsf{not}(\mathsf{Y}))), \\ \mathsf{not}(\mathsf{and}(\mathsf{X},\mathsf{Y})) & \rightarrow \mathsf{or}(\mathsf{SELF}(\mathsf{not}(\mathsf{X})), \mathsf{SELF}(\mathsf{not}(\mathsf{Y}))), \\ \mathsf{and}(\mathsf{X},\mathsf{Y}) & \rightarrow \mathsf{and}(\mathsf{SELF}(\mathsf{X}), \mathsf{SELF}(\mathsf{Y})), \\ \mathsf{or}(\mathsf{X},\mathsf{Y}) & \rightarrow \mathsf{or}(\mathsf{SELF}(\mathsf{X}), \mathsf{SELF}(\mathsf{Y})) \end{pmatrix}$$

and the imperative encoding is:

$$\mathsf{let\,Self} \ll \mathsf{ref\ \ dummy\ \ in\ \ Self} := (\mathsf{nnf\ \ !Self}); \mathsf{let\ \ Nnf} \ll \mathsf{!Self\ \ in\ \ Nnf}(\phi)$$

(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:

$$\mathsf{bool} ::= \mathsf{true} \mid \mathsf{false} \qquad \psi ::= \mathsf{ref}\ \phi$$

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

Given the constant dummy, the function nnf is defined as follows:

$$\mathsf{nnf} \triangleq \mathsf{SELF} \rightarrow$$

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

and the imperative encoding is:

$$\mathsf{let\,Self} \ll \mathsf{ref\ \ dummy\ \ in\ \ Self} := (\mathsf{nnf\ \ !Self}); \mathsf{let\ \ Nnf} \ll \mathsf{!Self\ \ in\ \ Nnf}(\psi)$$

(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; in our case, the algebraic constructor fix does not satisfy the above condition, since it is applied to a recursive object represented by the Self variable. 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 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].

```
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.
```

Fig. 8. Semantics Domains in Coq.

If b is the type of formulas ϕ, and $\mathsf{b\ ref}$ is the type of the shared-formulas ψ, and $\tau^{\wedge^n} \triangleq \overbrace{\tau \wedge \cdots \wedge \tau}^{n}$, and $\tau_1 \triangleq \mathsf{b} \rightarrowtriangle \mathsf{b}$, and $\tau_2 \triangleq \mathsf{b\ ref} \rightarrowtriangle \mathsf{b\ ref}$, then the reader can verify that the following judgments are derivable (recall $A\,;B \triangleq (X \rightarrowtriangle B)\,A$, if $X \notin \mathsf{Fv}(B)$).

$$(\mathsf{I})\quad \mathsf{dummy}{:}\tau_1^{\wedge^6}, \mathsf{Self/SELF}{:}\tau_1^{\wedge^6}\,\mathsf{ref} \vdash \mathsf{nnf} : \tau_1^{\wedge^6} \rightarrowtriangle \tau_1^{\wedge^6}$$
$$\mathsf{dummy}{:}\tau_1^{\wedge^6}, \mathsf{Self/SELF}{:}\tau_1^{\wedge^6}\,\mathsf{ref}, X{:}\tau_1^{\wedge^6}, \mathsf{Nnf}{:}\tau_1^{\wedge^6} \vdash \mathsf{Nnf}(\phi) : \mathsf{b}^{\wedge^6}$$

$$(\mathsf{IS})\quad \mathsf{dummy}{:}\tau_2^{\wedge^6}, \mathsf{Self/SELF}{:}\tau_2^{\wedge^6}\,\mathsf{ref} \vdash \mathsf{nnf} : \tau_2^{\wedge^6} \rightarrowtriangle \tau_2^{\wedge^6}$$
$$\mathsf{dummy}{:}\tau_2^{\wedge^6}, \mathsf{Self/SELF}{:}\tau_2^{\wedge^6}\,\mathsf{ref}, X{:}\tau_2^{\wedge^6}, \mathsf{Nnf}{:}\tau_2^{\wedge^6} \vdash \mathsf{Nnf}(\psi) : \mathsf{b\ ref}^{\wedge^6}$$

5 Formalization in Coq

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 $\rightarrowtriangle$-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.

5.1 Syntactic and Semantics Structures

The signature of the encoding of iRho is therefore presented in Figure 8. We comment the most interesting choices:

```
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).
```

Fig. 9. Sketch of Natural Imperative Semantics in Coq.

- An *ad hoc* type `var` is introduced for variables;
- Another *ad hoc* type `boperator` is introduced for algebraic constants; algebraic constants come with their types, so giving the category `operator` as a pair `boperator*type`;
- Locations `loc` are faithfully represented by natural numbers;
- (Un)typed environments (`env` and `envt`) and (un)typed stores (`store` and `storet`) are partial functions from `var`/`loc` to the sets `value`/`type`;
- A special variable `sbrk` denotes a function that, for any store, gives the topmost unused location: the `sbrk` variable is essential when we are looking to extend the store with fresh locations during new allocations (via the operator ref);
- Types `type` needs no special comments: they are implemented with an inductive datatype; patterns `pattern`, expressions `expr`, and values `value` are implemented too with an inductive datatype.

5.2 Natural Semantics

As we said in the previous section, 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 9. The web-appendix [22] contains all the encoding of the natural semantics. No rule presents surprises compared to rules in Natural Semantics (*i.e.* we get adequacy almost directly): this is a positive consequence of our DIMPRO pattern. We comment the most interesting choices:

- (`evalApp`) This rule is an "ASCII-clone" of the $(Red{-}\rho_{\mathsf{v}})$ natural semantic rule;
- (`evalRef`) This rule encodes the semantic rule $(Red{-}Ref)$: observe the use of the `sbrk` function, which extends a given store (partial function), via the (here omitted) auxiliary function `extend_store`;
- (`evalDeref`) This rule first verifies that the required location belongs to the store-domain (recall that stores are partial functions), and then directly accesses the store leaving the store itself unmodified;

```
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)).
```

Fig. 10. Sketch of Type-checking Rules in Coq.

- (`evalAssign`) This rule first evaluates the lvalue and the rvalue, then verifies that the location corresponding to the lvalue defined in the store, and finally modifies the store *in situ*.
- (`callclosureOK`), (`matchCons`) Again another two "ASCII-clones" of the $(Call{-}FunOK)$ (resp. $(Match{-}Pair)$ natural semantic rules;
- (`matchRef`) This rule first verifies that the given location `i` has some meaning in the store `s`, and then matches the `x` pattern in `(PRef x)` against `(Loc i)`.

5.3 Type System

The encoding of the type system is rather intuitive, again by a positive consequence of our DIMPRO pattern. The encoding is composed by three inductive functions, namely `TypeCheckPattern`, `TypeCheckExpr`, and `TypeOf`, to type-check patterns, terms and values, respectively. The latter function needs two important auxiliary functions, namely `AbstractEnv`, and `AbstractStore`, to keep consistency between types environments (Γ) and typed stores (σ). We discuss the most intriguing rules presented in Figure 10.

- (`tcPOpCons`), (`tcApp`) Those rules encode the type-checking rule for patterns $(Patt{-}Algbr)$, and terms $(Term{-}Appl)$, respectively: they make use of the function `NormalizeFunType` which behaves as a coercion to a functional type;
- (`tcRef`), (`tcDeref`), (`tcAssign`) Those rules encode the type rules $(Type{-}Ref)$, $(Type{-}Deref)$, and $(Type{-}Assign)$: they just mimic the corresponding rules in natural semantics;

– (aeExtend) This rule is the counterpart of the judgment $\vdash_\rho$ (see Appendix A.1) that assigns a type (*i.e.* a context) to an untyped environment; to ease the proof development, the encoding makes use of two different partial functions, namely envt and storet, to give a type to untyped environments and store; this "nuance" disappears in the typing rule, where a context Γ binds variables and/or locations to types; a coherence theorem (see Appendix A.2) bridges the gap from mathematical presentation to the encoding;
– (toClosure) This rule faithfully encodes rule $(Value{-}Clos)$ (see Appendix A.1);
– (FixAbstract) This definition is crucial to establish a coherence relation between (un)typed environments and (un)typed stores.

5.4 Some Metatheory in Coq

The following theorems collect some results we proved in Coq on the dynamic and on the static Semantics: we refer to Appendix A.2 for the the full metatheory, and to [22], for its complete mechanical counterparts.

Theorem 1 (Run-Time Galleria).

1. `Lemma LowerWhenSbrk : (s:store)(i:loc)` **(* writable store for sbrk *)**
   `i=(sbrk s) -> (a:value)`
   `(Lower s (extend_store s i a)).`
2. `Lemma NoGarbageCollection : (E:expr)(e:env)(s:store)(v:value)(s1:store)` **(* the store grows *)**
   `(eval E e s v s1) -> (LowerDomain s s1).`
3. `Lemma match_deterministic : (P:pattern)(v:value)(s:store)(e:env)(e1:env)` **(* algo pattern-matching *)**
   `(match P v s e e1) -> (e2:env)`
   `(match P v s e e2) -> (e1=e2).`
4. `Theorem eval_deterministic : (A:expr)(e:env)(s:store)(v1:value)(s1:store)` **(* algo eval *)**
   `(eval A e s v1 s1) -> (v2:value)(s2:store)`
   `(eval A e s v2 s2) -> ((v1=v2) /\ (s1=s2)).`
5. `Theorem call_deterministic: (v1,v2:value)(s:store)(r1:value)(s1:store)` **(* algo call *)**
   `(call v1 v2 s r1 s1) -> (r2:value)(s2:store)`
   `(call v1 v2 s r2 s2) -> ((r1=r2) /\ (s1=s2)).`

Theorem 2 (Compile-Time Galleria).

1. `Lemma NormalizeFunType_deterministic: (t,t1:type)` **(* arr is deterministic *)**
   `(NormalizeFunType t t1) -> (t2:type)`
   `(NormalizeFunType t t2) -> (t1=t2).`
2. `Lemma TypeCheckPattern_deterministic: (E:envt)(P:pattern)(E1:envt)(t1:type)` **(* algo type-check pattern *)**
   `(TypeCheckPattern E P E1 t1) -> (E2:envt)(t2:type)`
   `(TypeCheckPattern E P E2 t2) -> ((E1=E2) /\ (t1=t2)).`
3. `Lemma TypeCheckExpr_deterministic: (E:envt)(A:expr)(t1:type)` **(* algo type-check expr *)**
   `(TypeCheckExpr E A t1) -> (t2:type)`
   `(TypeCheckExpr E A t2) -> (t1=t2).`
4. `Lemma open_subject_reduction: (A:expr)(e:env)(s:store)(v:value)(s2:store)` **(* SR for open expr *)**
   `(eval A e s v s2) -> (E:envt)(S:storet)`
   `(FixAbstract e s E S) -> (t:type)`
   `(TypeCheckExpr E A t) -> (EX S2:storet | ((Coherent s S s2 S2) /\ (TypeOf S2 v t))).`
5. `Theorem subject_reduction: (A:expr)(v:value)(s2:store)` **(* SR for closed expr *)**
   `(eval A env_init store_init v s2) -> (t:type)`
   `(TypeCheckExpr envt_init A t) ->`
   `(EX S2:storet | ((Coherent store_init storet_init s2 S2) /\ (TypeOf S2 v t))).`

Since data-structures for stores, environments, terms, values, types, are isomorphic in mathematics and in Coq, the adequacy result comes directly as a matter of fact.

6 Conclusions, Lesson Learned and Further Work

In this paper, we have presented a formal development of the theory of iRho, a typed calculus featuring functions, pattern-matching, and side-effects. 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.

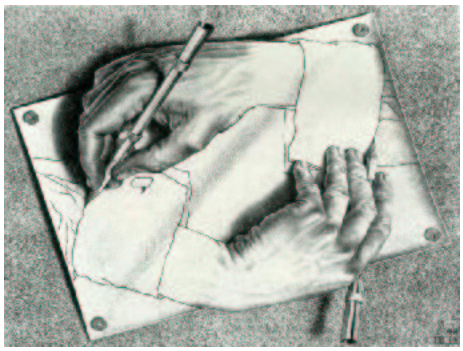


Fig. 11. From Theory to Practice and vice-versa (Escher, Drawing Hands, 1948).

(LaTeX)

$$\frac{\sigma_0 \cdot \rho \vdash F \Downarrow_{\mathsf{val}} f \cdot \sigma_1 \qquad \sigma_1 \cdot \rho \vdash A \Downarrow_{\mathsf{val}} a \cdot \sigma_2 \qquad \sigma_2 \vdash \langle f \cdot a \rangle \Downarrow_{\mathsf{call}} v \cdot \sigma_3}{\sigma_0 \cdot \rho \vdash F\, A \Downarrow_{\mathsf{val}} v \cdot \sigma_3}\,(Red{-}\rho_{\mathsf{v}})$$

(Bigloo)

```
(define-method (Eval::value t::App env) ;()
   (with-access::App t (F A)
      (let ( (f (Eval F env)) )
         (let ( (a (Eval A env)) )
            (call f a) ))))
```

(Coq)

```
Mutual Inductive eval : expr->env->store->value->store->Prop :=
.....
 | 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)
.....
```

Fig. 12. Natural Semantic: LaTeX *vs.* Bigloo *vs.* Coq views.

6.1 From LaTeX to Bigloo via Coq (and back)

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, as pictorially shown in Figure 11[2].

Our recipe probably suggests a new schema, or "pattern", in the sense of *"The Gang of Four"* [13], for design-implement-certify safe software (see Figure 12). This could be subject of future work. A small software interpreted 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* [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.

[2] Copyright demanded to the Escher's Foundation [40].

6.2 Related and Future Work

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

- RhoStratego [48] is an implementation of an early version of the uRho [5], written in the strategic language Stratego [49]. The implementation tests strategic programming with higher-order functional programming;
- Rogue [33] 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 [32] 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;
- analyze, in the pattern-matching algorithm, the impact of the "hide" choice *vs.* the "force" choice ; the "force" solution could be of interest since leads to a redefinition of equality between terms;
- 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];

Acknowledgement. The authors would like to thank all the members of the Protheo Team in Nancy for their comments and interactions on Rewriting Calculus, and Matt Wall for the careful reading of the paper.

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*. 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 *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, 2003. 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. V. (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 *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 T. A. press, editor, *Proc. of POPL*, pages 80–87, 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 Web Appendix of this paper, 2003. http://www-sop.inria.fr/oasis/Bernard.Serpette/webapp.html.
23. I. A. Mason and C. L. Talcott. References, Local Variables and Operational Reasoning. In *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, 2003. http://tom.loria.fr/.
29. J. G. Morrisett, M. Felleisen, and R. Harper. Abstract Models of Memory Management. In *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. *In Theory and Practice of Object Systems*, 2(1):3–13, 1996.
32. A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In *CAV*, 2002. System Description.
33. A. Stump and C. Schürmann. The Rogue Home Page, 2003. http://www.cse.wustl.edu/\%7Estump/rogue.html.
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, 2003. http://www.cduce.org.
37. The Common Criteria Consortium. The Common Criteria Home Page, 2003. http://www.commoncriteria.org.
38. The Cristal Team. The Caml Home Page, 2003. http://caml.inria.fr.
39. The Cristal Team. The Objective Caml Home Page, 2003. http://www.ocaml.org/.
40. The Escher Foundation. M.C. Escher. The Official Web Site, 2004. http://www.mcescher.com/.
41. The GNU Prolog team. The GNU Prolog Home Page, 2003. http://pauillac.inria.fr/~diaz/gnu-prolog/.
42. The Haskell Team. The Haskell Home Page, 2004. http://www.haskell.org/.
43. The Logical Team. The Coq Home Page, 2003. http://coq.inria.fr.
44. The Maude Team. The Maude Home Page, 2003. http://maude.cs.uiuc.edu/.
45. The Mimosa Team. The Schme Bigloo Home Page, 2003. http://www.sop.inria.fr/mimosa/fp/bigloo/.
46. The Protheo Team. The Elan Home Page, 2003. http://elan.loria.fr.
47. The Scheme Team. The Scheme Language, 2004. http://www.swiss.ai.mit.edu/projects/scheme/.
48. The Stratego Team. The Rho Stratego Home Page, 2003. http://www.stratego-language.org/twiki/bin/view/Stratego/RhoStratego.
49. The Stratego Team. The Stratego Home Page, 2003. http://www.stratego-language.org.

50. The Xduce Team. The Xduce Home Page, 2003. `http://xduce.sourceforge.net`.
51. A. van Deursen, J. Heering, and P. Klint. *Language Prototyping*. World Scientific, 1996.
52. V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

A Appendix: for Referees

See also `http://www-sop.inria.fr/oasis/Bernard.Serpette/webapp.html`

A.1 Extra Typing Rules

The presentation of the type system is completed by five complementary judgments of the shape:

$$\Gamma \vdash_{\Gamma} ok, \quad \text{and} \quad \Gamma \vdash_{\tau} \tau : ok, \quad \text{and} \quad \Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau \quad \text{and} \quad \Gamma \vdash_{\rho} \rho : \Gamma' \quad \text{and} \quad \Gamma \vdash_{\sigma} \sigma : \Gamma',$$

denoting well-formed contexts, types, values, environments, and stores. Those judgments are necessary when we encode iRho in the Logical Framework of Coq. The type rules of those five new judgments are really much more intuitive and they do not need any particular comment. It is worth noting that also rule $(Value{-}Algbr)$ needs a transformation step for structure-types into arrow-types. Also interesting are the $(Value{-}Clos)$ and the $(Value{-}Fail)$ rules, since the inferred type for the environment ρ is "charged" into the derivation for the pattern abstraction. Finally, observe that environment-typing allows "type-overriding" over free variables (recall that our interpreter is rather economic, since does not use Henk Barendregt's hygiene condition nor work under "expensive" α-conversion steps), while store-typing still force locations to have a monomorphic type. All extra rules are presented in Figure 14.

A.2 Properties of the Imperative Calculus

The main objectives of this section is to prove the main properties of iRho, namely that:

1. Natural Semantics for $\Downarrow_{\mathsf{val}}$ is deterministic;
2. Type-checking with $\vdash_{\mathsf{A}}$ is unique;
3. Subject Reduction holds (*i.e*. types are preserved under reduction);
4. Type Soundness holds (*i.e*. the type system preserves the evaluator from "stuck" states);
5. Both Type-checking and Type Reconstruction are decidable.

We start with a natural definition of free variables.

Definition 1 (Free variables Fv).

$$\begin{array}{llll}
\mathsf{Fv}(a) & \triangleq \emptyset & \mathsf{Fv}(P{:}\Delta \rightarrowtail A) & \triangleq \mathsf{Fv}(A) \setminus \mathsf{Fv}(P) \\
\mathsf{Fv}(X) & \triangleq \{X\} & \mathsf{Fv}(A\,B) & \triangleq \mathsf{Fv}(A) \cup \mathsf{Fv}(B) \\
\mathsf{Fv}(!A) & \triangleq \mathsf{Fv}(A) & \mathsf{Fv}(A,B) & \triangleq \mathsf{Fv}(A) \cup \mathsf{Fv}(B) \\
\mathsf{Fv}(\mathsf{ref}\ A) & \triangleq \mathsf{Fv}(A) & \mathsf{Fv}(A := B) & \triangleq \mathsf{Fv}(A) \cup \mathsf{Fv}(B)
\end{array}$$

Then, we prove that our natural semantics is deterministic, and immediately suggest how to build an interpreter. A software prototype for the untyped iRho can be found in our web appendix [22];

Theorem 3 (Deterministic Semantics).

1. *If* $\sigma \cdot \rho \vdash A \Downarrow_{\mathsf{val}} A'_{\mathsf{v}} \cdot \sigma'$, *and* $\sigma \cdot \rho \vdash A \Downarrow_{\mathsf{val}} A''_{\mathsf{v}} \cdot \sigma''$, *then* $A'_{\mathsf{v}} \equiv A''_{\mathsf{v}}$, *and* $\sigma' \equiv \sigma''$;
2. *If* $\sigma \vdash \langle A_{\mathsf{v}} \cdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{cal}} C_{\mathsf{v}} \cdot \sigma'$, *and* $\sigma \vdash \langle A_{\mathsf{v}} \cdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{cal}} D_{\mathsf{v}} \cdot \sigma''$, *then* $C_{\mathsf{v}} \equiv D_{\mathsf{v}}$, *and* $\sigma' \equiv \sigma''$;
3. *If* $\sigma \cdot \rho \vdash \langle P \cdot A_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho'$, *and* $\sigma \cdot \rho \vdash \langle P \cdot A_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho''$, *then* $\rho' \equiv \rho''$.

The type system is algorithmic, hence it enjoys the nice property of uniqueness of typing;

Theorem 4 (Uniqueness of Typing). *If* $\Gamma \vdash_{\mathsf{A}} A : \tau$, *then* τ *is unique.*

This definition will be useful to state subject reduction.

Definition 2 (Coherence). *The context* Γ *is coherent with a store* σ, *and an environment* ρ, *denoted by*

$$\sigma \cdot \rho \vdash_{\mathsf{coh}} \Gamma$$

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* $\Gamma \vdash_{\rho} \rho : \Gamma_2$.

Context Rules

$$\frac{}{\emptyset \vdash_\Gamma ok}\,(Ctx\text{-}Axiom) \qquad \frac{\chi \notin \mathsf{Dom}(\Gamma) \quad \Gamma \vdash_\tau \tau : ok \qquad \Gamma \vdash_\Gamma ok}{\Gamma, \chi{:}\tau \vdash_\Gamma ok}\,(Ctx\text{-}Var/Const) \qquad \frac{\Gamma \vdash_\Gamma ok \quad \mathsf{b} \notin \mathsf{Dom}(\Gamma)}{\Gamma, \mathsf{b}{:}ok \vdash_\Gamma ok}\,(Ctx\text{-}Type)$$

Type Rules

$$\frac{\Gamma_1, \mathsf{b}{:}ok, \Gamma_2 \vdash_\Gamma ok}{\Gamma_1, \mathsf{b}{:}ok, \Gamma_2 \vdash_\tau \mathsf{b} : ok}\,(Type\text{-}Start) \qquad \frac{\Gamma \vdash_\tau \tau : ok}{\Gamma \vdash_\tau \tau\ \mathsf{ref}\ : ok}\,(Type\text{-}Ref)$$

$$\frac{\Gamma \vdash_\tau \tau_1 : ok \quad \Gamma \vdash_\tau \tau_2 : ok}{\Gamma \vdash_\tau \tau_1 \rightarrowtriangle \tau_2 : ok}\,(Type\text{-}Arrow) \qquad \frac{\Gamma \vdash_\tau \tau_1 : ok \quad \Gamma \vdash_\tau \tau_2 : ok}{\Gamma \vdash_\tau \tau_1 \wedge \tau_2 : ok}\,(Type\text{-}Struct)$$

Value Typing

$$\frac{\Gamma_1, a{:}\tau, \Gamma_2 \vdash_\Gamma ok}{\Gamma_1, a{:}\tau, \Gamma_2 \vdash_\mathsf{v} a : \tau}\,(Value\text{-}Start_1) \qquad \frac{\Gamma_1, \iota{:}\tau\ \mathsf{ref}\ , \Gamma_2 \vdash_\Gamma ok}{\Gamma_1, \iota{:}\tau\ \mathsf{ref}\ , \Gamma_2 \vdash_\mathsf{v} \iota : \tau\ \mathsf{ref}}\,(Value\text{-}Start_2)$$

$$\frac{\Gamma \vdash_\mathsf{v} A_\mathsf{v} : \tau_1 \quad \Gamma \vdash_\mathsf{v} B_\mathsf{v} : \tau_2}{\Gamma \vdash_\mathsf{v} A_\mathsf{v}, B_\mathsf{v} : \tau_1 \wedge \tau_2}\,(Value\text{-}Struct) \qquad \frac{\mathsf{arr}(\tau_1) \equiv \tau_2 \rightarrowtriangle \tau_3 \quad \Gamma \vdash_\mathsf{v} a\ \overline{A}_\mathsf{v} : \tau_1 \quad \Gamma \vdash_\mathsf{v} B_\mathsf{v} : \tau_2}{\Gamma \vdash_\mathsf{v} a\ \overline{A}_\mathsf{v}\ B_\mathsf{v} : \tau_3}\,(Value\text{-}Algbr)$$

$$\frac{\Gamma \vdash_\rho \rho : \Gamma' \quad \Gamma' \vdash_\mathsf{p} P{:}\Delta \rightarrowtriangle A : \tau_1 \rightarrowtriangle \tau_2}{\Gamma \vdash_\mathsf{v} \langle P{:}\Delta \rightarrowtriangle A \cdot \rho \rangle : \tau_1 \rightarrowtriangle \tau_2}\,(Value\text{-}Clos) \qquad \frac{\Gamma \vdash_\mathsf{v} A_\mathsf{v} : \tau_1 \quad \Gamma \vdash_\rho \rho : \Gamma' \quad \Gamma' \vdash_\mathsf{p} P{:}\Delta \rightarrowtriangle B : \tau_1 \rightarrowtriangle \tau_2}{\Gamma \vdash_\mathsf{v} \langle [P \ll_\Delta A_\mathsf{v}].B \cdot \rho \rangle : \tau_2}\,(Value\text{-}Fail)$$

Store Typing

$$\Gamma[\iota{:}\tau] \equiv \begin{cases} \Gamma, \iota{:}\tau & \text{if } \iota \notin \mathsf{Dom}(\Gamma) \\ \Gamma_1, \iota{:}\tau, \Gamma_2 & \text{if } \Gamma \equiv \Gamma_1, \iota{:}\tau, \Gamma_2 \end{cases}$$

$$\frac{\Gamma \vdash_\Gamma ok}{\Gamma \vdash_\sigma \emptyset : \Gamma}\,(Store\text{-}Axiom)$$

$$\frac{\Gamma \vdash_\sigma \sigma : \Gamma' \quad \Gamma'[\iota{:}\tau] \vdash_\Gamma ok \quad \Gamma \vdash_\mathsf{v} \iota : \tau\ \mathsf{ref} \quad \Gamma \vdash_\mathsf{v} A_\mathsf{v} : \tau}{\Gamma \vdash_\sigma \sigma[\iota \mapsto A_\mathsf{v}] : \Gamma'[\iota{:}\tau]}\,(Store\text{-}Loc)$$

Environment Typing

$$\Gamma[X{:}\tau] \equiv \begin{cases} \Gamma, X{:}\tau & \text{if } X \notin \mathsf{Dom}(\Gamma) \\ \Gamma_1, X{:}\tau, \Gamma_2, & \text{if } \Gamma \equiv \Gamma_1, X{:}\tau', \Gamma_2 \end{cases}$$

$$\frac{\Gamma \vdash_\Gamma ok}{\Gamma \vdash_\rho \emptyset : \Gamma}\,(Env\text{-}Axiom)$$

$$\frac{\Gamma \vdash_\rho \rho : \Gamma' \quad \Gamma'[X{:}\tau] \vdash_\Gamma ok \quad \Gamma \vdash_\mathsf{A} X : \tau \quad \Gamma \vdash_\mathsf{v} A_\mathsf{v} : \tau}{\Gamma \vdash_\rho \rho[X \mapsto A_\mathsf{v}] : \Gamma'[X{:}\tau]}\,(Env\text{-}Var)$$

Fig. 13. Extra Type Rules

Subject reduction for open terms is preliminary for subject reduction for closed terms.

Theorem 5 (Subject Reduction for Open Terms).

1. *If $\sigma_1 \cdot \rho_1 \vdash A \Downarrow_{\mathsf{val}} A_\mathsf{v} \cdot \sigma_2$, and $\sigma_1 \cdot \rho_1 \vdash_{\mathsf{coh}} \Gamma$, and $\Gamma \vdash_\mathsf{A} A : \tau$, then there exists Γ', which extend Γ, such that $\sigma_2 \cdot \rho_1 \vdash_{\mathsf{coh}} \Gamma'$, and $\Gamma' \vdash_\mathsf{v} A_\mathsf{v} : \tau$.*
2. *If $\sigma_1 \vdash \langle A_\mathsf{v} \cdot B_\mathsf{v} \rangle \Downarrow_{\mathsf{call}} C_\mathsf{v} \cdot \sigma_2$, and $\sigma_1 \cdot \rho_1 \vdash_{\mathsf{coh}} \Gamma$, and $\Gamma \vdash_\mathsf{v} A_\mathsf{v} : \tau_1$, and $\Gamma \vdash_\mathsf{v} B_\mathsf{v} : \tau_2$, and $\mathsf{arr}(\tau_1) \equiv \tau_2 \rightarrowtriangle \tau_3$, then there exists Γ', which extend Γ, such that $\sigma_2 \cdot \rho_1 \vdash_{\mathsf{coh}} \Gamma'$, and $\Gamma' \vdash_\mathsf{v} C_\mathsf{v} : \tau_3$;*
3. *If $\sigma_1 \cdot \rho_1 \vdash \langle P \cdot A_\mathsf{v} \rangle \Downarrow_{\mathsf{match}} \rho_2$, and $\sigma_1 \cdot \rho_1 \vdash_{\mathsf{coh}} \Gamma$, and $\Gamma \vdash_\mathsf{A} P : \tau$, and $\Gamma \vdash_\mathsf{v} A_\mathsf{v} : \tau$, then there exists Γ', which extend Γ, such that $\sigma_1 \cdot \rho_2 \vdash_{\mathsf{coh}} \Gamma'$.*

This result is crucial to state type soundness.

Theorem 6 (Subject Reduction for Closed Terms). *If* $\emptyset \vdash_{\mathsf{A}} A : \tau$, *and* $\emptyset \cdot \emptyset \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \cdot \sigma$, *then there exists* Γ' *which extend* Γ, *such that* $\Gamma' \vdash_{\sigma} \sigma : ok$, *and* $\Gamma' \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau$.

The reduction rules for the operational semantics given in Table 6 readily suggest how an interpreter for iRho can be defined. Run-time errors for this interpreter correspond to stuck-states when using the rules to evaluate a closed expression. An inspection of the three judgments shows that there are only few ways in which evaluation may "get-stuck":

($\Downarrow_{\mathsf{val}}$) This judgment gets stuck when we access a variable not defined in the environment, when the evaluation of A in (ref A, or $A := B$) gives a fresh location (*i.e.* not in the current used store), or if one premise in some judgment gets stuck;

($\Downarrow_{\mathsf{call}}$) This judgment gets stuck when the we try to apply a location-value to a value (*e.g.* $\langle \iota \cdot A_{\mathsf{v}} \rangle$), or a failure-value to a value, (*e.g.* $\langle \langle [P \ll_{\Delta} B_{\mathsf{v}}].C \cdot \rho \rangle \cdot A_{\mathsf{v}} \rangle$), or if one premise in some judgment gets stuck;

($\Downarrow_{\mathsf{match}}$) This judgment gets stuck when we try to match a pattern against a value with a different (unmatchable) shape, *e.g.* $\langle P\ P \cdot A, B \rangle$, etc.

The following theorem proves the absence of such errors in the evaluation of a well-typed closed expression: type soundness follows from this result. Those last two theorems are proved "on paper"; the reader is invited to try and formalize those two theorems, which follow quite easily from our previous results proved in Coq.

Theorem 7 (Type Soundness/Absence of Stuck States).

($\Downarrow_{\mathsf{val}}$) *Let* A_0 *be a closed expression such that* $\emptyset \vdash_{\mathsf{A}} A_0 : \tau$ *is derivable.*

- *if* $A_0 \equiv \mathsf{C}[X]$, *then* $\sigma \cdot \rho \vdash X \Downarrow_{\mathsf{val}} \rho(X)$, *and* $\rho(X)$ *is defined (for some* σ, *and* ρ*);*
- *if* $A_0 \equiv$ ref A, *then* $\emptyset \cdot \emptyset \vdash A \Downarrow_{\mathsf{val}} \iota \cdot \sigma_1$, *and* $\iota \in \mathsf{Dom}(\sigma_1)$ *(for some* σ_1*);*
- *if* $A_0 \equiv (A := B)$, *and* $\emptyset \cdot \emptyset \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \cdot \sigma_1$, *and* $\sigma_1 \cdot \emptyset \vdash A \Downarrow_{\mathsf{val}} \iota \cdot \sigma_2$, *and* $\iota \in \mathsf{Dom}(\sigma_2)$ *(for some* σ_1, *and* σ_2*);*

($\Downarrow_{\mathsf{call}}$) *Let* A, *and* B *be closed term such that* $\emptyset \cdot \emptyset \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \cdot \sigma_0$, *and* $\sigma_0 \cdot \emptyset \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \cdot \sigma_1$.

- *if* $\sigma_1 \vdash \langle A_{\mathsf{v}} \cdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{call}} C_{\mathsf{v}}$, *then* $A_{\mathsf{v}} \not\equiv \iota$, *or* $A_{\mathsf{v}} \not\equiv [P \ll_{\Delta} B_{\mathsf{v}}].C$*;*

($\Downarrow_{\mathsf{match}}$) *Let* A, *and* B *be closed term such that* $\emptyset \cdot \emptyset \vdash A \Downarrow_{\mathsf{val}} \langle P{:}\Delta \rightarrowtriangle C \cdot \rho \rangle \cdot \sigma_0$, *and* $\sigma_0 \cdot \emptyset \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \cdot \sigma_1$.

- *if* $\sigma_1 \cdot \rho \vdash \langle P \cdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho'$, *then* P *will successfully match with* B_{v}.

We conclude with the decidability results.

Theorem 8. *Given a closed expression* A, *the following propositions are decidable:*

1. **(Type-checking)** *Given a type* τ, *the judgment* $\emptyset \vdash_{\mathsf{A}} A : \tau$ *is derivable;*
2. **(Type-reconstruction)** *There exists a type* τ, *such that* $\emptyset \vdash_{\mathsf{A}} A : \tau$.

A.3 More-examples

The first example presents a functional evaluation, while the second shows an imperative evaluation, dealing with referencing and dereferencing of operators. To simplify the derivations, types are omitted. The third example gives a type to the imperative term of the second example. The fourth example plays with a simple typed derivation using term-structures, and the last example shows static and dynamic description of a simple functional fixed-point.

Example 2 (Two Functional Evaluations). Consider the following term in fRho:

$$(f(X) \rightarrowtriangle (3 \rightarrowtriangle 3)X)\ f(3)$$

and let $\odot \equiv \emptyset \vdash \langle (f(X) \rightarrowtriangle (3 \rightarrowtriangle 3)\ X) \cdot \emptyset \rangle \Downarrow_{\mathsf{val}} \langle (f(X) \rightarrowtriangle (3 \rightarrowtriangle 3)\ X) \cdot \emptyset \rangle$, and $\triangle \equiv \emptyset \vdash f(3) \Downarrow_{\mathsf{val}} f(3)$, and $\rho_0 \triangleq [X \mapsto 3]$. The deduction tree is shown in Figure 14. The reader is invited to try and find a derivation for $(f(X) \rightarrowtriangle (3 \rightarrowtriangle 3)X)\ f(4)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \langle f \cdot f \rangle \Downarrow_{\mathsf{match}} \emptyset} \quad \dfrac{}{\emptyset \vdash \langle X \cdot 3 \rangle \Downarrow_{\mathsf{match}} \rho_0}}{\emptyset \vdash \langle f(X) \cdot f(3) \rangle \Downarrow_{\mathsf{match}} \rho_0} \quad \dfrac{}{\rho_0 \vdash X \Downarrow_{\mathsf{val}} 3} \quad \dfrac{}{\rho_0 \vdash (3 \rightarrow 3) \Downarrow_{\mathsf{val}} \langle (3 \rightarrow 3) \cdot \rho_0 \rangle}}{\rho_0 \vdash (3 \rightarrow 3)\, X \Downarrow_{\mathsf{val}} 3} \quad \dfrac{\dfrac{}{\rho_0 \vdash \langle 3 \cdot 3 \rangle \Downarrow_{\mathsf{match}} \rho_0} \quad \dfrac{}{\rho_0 \vdash 3 \Downarrow_{\mathsf{val}} 3}}{\vdash \langle \langle (3 \rightarrow 3) \cdot \rho_0 \rangle \cdot 3 \rangle \Downarrow_{\mathsf{call}} 3}}{\vdash \langle \langle (f(X) \rightarrow (3 \rightarrow 3)\, X) \cdot \emptyset \rangle \cdot f(3) \rangle \Downarrow_{\mathsf{call}} 3} \; \odot \quad \triangle}{\emptyset \vdash (f(X) \rightarrow (3 \rightarrow 3)\, X)\, f(3) \Downarrow_{\mathsf{val}} 3}$$

Fig. 14. Natural Deduction of $(f(X) \rightarrow (3 \rightarrow 3)\, X)\, f(3)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\iota_0 \in \mathsf{Dom}(\sigma_0)} \quad \dfrac{}{\sigma_0 \cdot \rho_0 \vdash X \Downarrow_{\mathsf{val}} \iota_0 \cdot \sigma_0} \quad \dfrac{\dfrac{}{\sigma_0 \cdot \rho_0 \vdash Y \Downarrow_{\mathsf{val}} \iota_1 \cdot \sigma_0}}{\sigma_0 \cdot \rho_0 \vdash !Y \Downarrow_{\mathsf{val}} 4 \cdot \sigma_0}}{\sigma_0 \cdot \rho_0 \vdash X := !Y \Downarrow_{\mathsf{val}} 4 \cdot \sigma_1} \quad \dfrac{}{\sigma_0 \cdot \rho_0 \vdash \langle 3 \cdot 3 \rangle \Downarrow_{\mathsf{match}} \rho_0}}{\sigma_0 \vdash \langle \langle 3 \rightarrow X := !Y \cdot \rho_0 \rangle \cdot 3 \rangle \Downarrow_{\mathsf{call}} 4 \cdot \sigma_1} \quad \dfrac{}{\sigma_0 \cdot \rho_0 \vdash !X \Downarrow_{\mathsf{val}} 3 \cdot \sigma_0} \quad \dfrac{}{\sigma_0 \cdot \rho_0 \vdash 3 \rightarrow X := !Y \Downarrow_{\mathsf{val}} \langle 3 \rightarrow X := !Y \cdot \rho_0 \rangle \cdot \sigma_0}}{\sigma_0, \rho_0 \vdash (3 \rightarrow X := !Y)\, !X \Downarrow_{\mathsf{call}} 4 \cdot \sigma_1} \quad \dfrac{}{\sigma_0 \cdot \emptyset \vdash \langle f(X,Y) \cdot f(\iota_0, \iota_1) \rangle \Downarrow_{\mathsf{match}} \rho_0}}{\sigma_0 \cdot \emptyset \vdash \langle \langle (f(X,Y) \rightarrow (3 \rightarrow X := !Y)\, !X) \cdot \emptyset \rangle \cdot f(\iota_0, \iota_1) \rangle \Downarrow_{\mathsf{call}} 4 \cdot \sigma_1} \quad \dfrac{}{\emptyset \cdot \emptyset \vdash f(\mathsf{ref}\ 3, \mathsf{ref}\ 4) \Downarrow_{\mathsf{val}} f(\iota_0, \iota_1) \cdot \sigma_0} \quad \dfrac{}{\emptyset \cdot \emptyset \vdash (f(X,Y) \rightarrow (3 \rightarrow X := !Y)\, !X) \Downarrow_{\mathsf{val}} \langle (f(X,Y) \rightarrow (3 \rightarrow X := !Y)\, !X) \cdot \emptyset \rangle \cdot \emptyset}}{\emptyset \cdot \emptyset \vdash (f(X,Y) \rightarrow (3 \rightarrow X := !Y)\, !X)\, f(\mathsf{ref}\ 3, \mathsf{ref}\ 4) \Downarrow_{\mathsf{val}} 4 \cdot \sigma_1}$$

Fig. 15. Natural Deduction of $(f(X,Y) \rightarrow (3 \rightarrow X := !Y)\, !X)\, f(\mathsf{ref}\ 3, \mathsf{ref}\ 4)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \Delta \vdash_{\mathsf{A}} f : (\mathsf{b\ ref}\ \wedge \mathsf{b\ ref}) \rightarrow \mathsf{b}} \quad \dfrac{}{\Gamma, \Delta \vdash_{\mathsf{A}} X, Y : \mathsf{b\ ref}\ \wedge \mathsf{b\ ref}}}{\Gamma, \Delta \vdash_{\mathsf{A}} f(X,Y) : \mathsf{b}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma, \Delta \vdash_{\mathsf{A}} Y : \mathsf{b\ ref}}}{\Gamma, \Delta \vdash_{\mathsf{A}} !Y : \mathsf{b}} \quad \dfrac{}{\Gamma, \Delta \vdash_{\mathsf{A}} X : \mathsf{b\ ref}}}{\Gamma, \Delta \vdash_{\mathsf{A}} X := !Y : \mathsf{b}}}{\Gamma \vdash_{\mathsf{A}} (f(X,Y) : \Delta \rightarrow (3 : \emptyset \rightarrow X := !Y)\, !X) : \mathsf{b} \rightarrow \mathsf{b}} \quad \dfrac{\dfrac{}{\Gamma, \Delta \vdash_{\mathsf{A}} f : (\mathsf{b\ ref}\ \wedge \mathsf{b\ ref}) \rightarrow \mathsf{b}} \quad \dfrac{}{\Gamma, \Delta \vdash_{\mathsf{A}} \mathsf{ref}\ 3, \mathsf{ref}\ 4 : \mathsf{b\ ref}\ \wedge \mathsf{b\ ref}}}{\Gamma \vdash_{\mathsf{A}} f(\mathsf{ref}\ 3, \mathsf{ref}\ 4) : \mathsf{b}}}{\Gamma \vdash_{\mathsf{A}} (f(X,Y) : \Delta \rightarrow (3 : \emptyset \rightarrow X := !Y)\, !X)\, f(\mathsf{ref}\ 3, \mathsf{ref}\ 4) : \mathsf{b}}$$

Fig. 16. Type-checking of Term of Figure 15.

Example 3 (An Imperative Evaluation). Take the imperative term:

$$(f(X,Y) \rightarrow (3 \rightarrow X := !Y)\, !X)\, f(\mathsf{ref}\ 3, \mathsf{ref}\ 4)$$

with $\sigma_0 \triangleq [\iota_0 \mapsto 3][\iota_1 \mapsto 4]$, and $\sigma_1 \triangleq \sigma_0[\iota_0 \mapsto 4]$, and $\rho_0 \triangleq [X \mapsto \iota_0][Y \mapsto \iota_1]$. The deduction tree is shown in Figure 15.

$$\dfrac{\dfrac{\dfrac{\Gamma,\Delta \vdash_{\mathsf{A}} X : \mathsf{b\ ref} \quad \Gamma,\Delta \vdash_{\mathsf{A}} Y : \mathsf{b\ ref}}{\Gamma,\Delta \vdash_{\mathsf{A}} X,!Y : \mathsf{b\ ref} \wedge \mathsf{b\ ref}} \quad \dfrac{\Gamma,\Delta \vdash_{\mathsf{A}} X : \mathsf{b\ ref} \quad \Gamma,\Delta \vdash_{\mathsf{A}} !Y : \mathsf{b}}{\Gamma,\Delta \vdash_{\mathsf{A}} X := !Y : \mathsf{b}}}{\Gamma \vdash_{\mathsf{A}} (X,Y){:}\Delta \twoheadrightarrow X := !Y : (\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \twoheadrightarrow \mathsf{b}} \quad \dfrac{\dfrac{\Gamma,\Delta \vdash_{\mathsf{A}} X : \mathsf{b\ ref} \quad \Gamma,\Delta \vdash_{\mathsf{A}} Y : \mathsf{b\ ref}}{\Gamma,\Delta \vdash_{\mathsf{A}} X,Y : \mathsf{b\ ref} \wedge \mathsf{b\ ref}} \quad \Gamma,\Delta \vdash_{\mathsf{A}} Y : \mathsf{b\ ref}}{\Gamma \vdash_{\mathsf{A}} (X,Y){:}\Delta \twoheadrightarrow Y : (\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \twoheadrightarrow \mathsf{b}}}{\Gamma \vdash_{\mathsf{A}} ((X,Y){:}\Delta \twoheadrightarrow X := !Y, (X,Y){:}\Delta \twoheadrightarrow !Y) : (\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \twoheadrightarrow \mathsf{b} \wedge (\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \twoheadrightarrow \mathsf{b}} \quad \begin{array}{l}\mathsf{arr}((\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \twoheadrightarrow \mathsf{b} \wedge (\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \twoheadrightarrow \mathsf{b}) \equiv (\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \twoheadrightarrow (\mathsf{b} \wedge \mathsf{b}) \quad \Gamma \vdash_{\mathsf{A}} (\mathsf{ref}\, a, \mathsf{ref}\, b) : \mathsf{b\ ref} \wedge \mathsf{b\ ref}\end{array}}{\Gamma \vdash_{\mathsf{A}} ((X,Y){:}\Delta \twoheadrightarrow X := !Y, (X,Y){:}\Delta \twoheadrightarrow !Y)\,(\mathsf{ref}\, a, \mathsf{ref}\, b) : \mathsf{b} \wedge \mathsf{b}}$$

Fig. 17. Type-checking of Terms.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma,\Delta \vdash_{\mathsf{A}} f : (\mathsf{b} \twoheadrightarrow \mathsf{b}) \twoheadrightarrow \mathsf{b} \quad \Gamma,\Delta \vdash_{\mathsf{A}} X : \mathsf{b} \twoheadrightarrow \mathsf{b}}{\Gamma,\Delta \vdash_{\mathsf{A}} f(X) : \mathsf{b}} \quad \dfrac{\Gamma,\Delta \vdash_{\mathsf{A}} X : \mathsf{b} \twoheadrightarrow \mathsf{b} \quad \Gamma,\Delta \vdash_{\mathsf{A}} f(X) : \mathsf{b}}{\Gamma,\Delta \vdash_{\mathsf{A}} X\, f(X) : \mathsf{b}}}{\Gamma \vdash_{\mathsf{A}} \Omega : \mathsf{b} \twoheadrightarrow \mathsf{b}} \quad \dfrac{\Gamma,\Delta \vdash_{\mathsf{A}} f : (\mathsf{b} \twoheadrightarrow \mathsf{b}) \twoheadrightarrow \mathsf{b} \quad \Gamma,\Delta \vdash_{\mathsf{A}} \Omega : \mathsf{b} \twoheadrightarrow \mathsf{b}}{\Gamma,\Delta \vdash_{\mathsf{A}} f(\Omega) : \mathsf{b}}}{\Gamma \vdash_{\mathsf{A}} \Omega\, f(\Omega) : \mathsf{b}} \quad \dfrac{\vdots^{\infty}}{\emptyset \vdash \Omega\, f(\Omega) \Downarrow_{\mathsf{val}} \mathsf{stack\ overflow}}$$

Fig. 18. One Fixed-point.

Example 4 (Typed Derivation). We decorate the term presented in Example 3: let $\Gamma \equiv 3{:}\mathsf{b}, 4{:}\mathsf{b}, f{:}(\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \twoheadrightarrow \mathsf{b}$, and $\Delta \equiv X{:}\mathsf{b\ ref}, Y{:}\mathsf{b\ ref}$. A derivation for $(f(X,Y){:}\Delta \twoheadrightarrow (3{:}\emptyset \twoheadrightarrow X := !Y)\,!X)\,f(\mathsf{ref}\,3, \mathsf{ref}\,4)$ is shown in Figure 16 (all trivial subtyping derivations are omitted):

Example 5 (Structured-terms and Normalized-types). Let $\Gamma \equiv \mathsf{b}{:}ok, a{:}\mathsf{b}, b{:}\mathsf{b}$ and $\Delta \equiv X{:}\mathsf{b\ ref}, Y{:}\mathsf{b\ ref}$. A derivation for $((X,Y){:}\Delta \twoheadrightarrow X := !Y, (X,Y){:}\Delta \twoheadrightarrow Y)\,(a,b)$ is shown in Figure 17.

Example 6 (Simple Functional Fixed-point [9]). The type systems of fRho (and 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 $\Gamma \equiv f{:}(\mathsf{b} \twoheadrightarrow \mathsf{b}) \twoheadrightarrow \mathsf{b}$, and $\Delta \equiv X{:}\mathsf{b} \twoheadrightarrow \mathsf{b}$. A derivation for $\Omega \triangleq f(X){:}\Delta \twoheadrightarrow X\, f(X)$ is shown in Figure 18. The reader can verify that our interpreter diverges.