

iRho: An Imperative Rewriting-calculus

Luigi Liquori

INRIA, France

Luigi.Liquori@inria.fr

Bernard Paul Serpette

INRIA, France

Bernard.Serpette@inria.fr

Received 26 January 2006

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 big-step *call-by-value* operational semantics of **iRho**. 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-types, which can be assigned to terms like structures (*i.e.*, records).

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 of a pattern-matching imperative language, where a (monomorphic) typed store can be safely manipulated and where fixed-points are built-in into the language itself.

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

1. Introduction

The study of *rewriting-based* languages (*e.g.*, Elan [Pro05], Maude [Mau05], ASF+SDF [vDHK96, Asf05], OBJ* [Gog05]) is a promising line of research unifying the logic paradigm with the functional paradigm.

Although rewriting-based languages are less popular than object-oriented languages such as Java [Sun05], C# [Mic05], etc., for ordinary programming, they can serve as common typed intermediate languages for implementing compilers for rewriting-based, functional, object-oriented, logic, and other high-level modern languages.

Pattern-matching has been widely used in functional and logic programming (ML [MTHM97, Cri05], Haskell [Jon03], Scheme [RK98], Curry [Han97], or Prolog [Kow79]); it is generally considered to be a convenient mechanism for expressing complex require-

ments about the function’s argument, rather than a basis for an *ad hoc* paradigm of computation.

One of the main advantages of rewriting-based languages is *pattern-matching* which allows one to discriminate between alternatives. These languages permit non-determinism in the sense that they can represent a collection of results. That is, pattern matching need not to be exclusive, multiple branches can be “fired” simultaneously. An empty collection of results represents an application failure, a singleton represents a deterministic result, and a collection with more than one element represents a non-deterministic choice between the elements of the collection.

This feature highlights a difference between functional languages featuring pattern matching, such as ML, Haskell, and Scheme. It shares some similarities with backtracking and exhaustive proof search in logic languages like Prolog. It is possible to make a product of two functions having the same pattern; when the product is applied to an argument, both functions will be fired. *Optimistic* and *pessimistic* operational semantics[†], with a fixed strategy, can then be imposed on the language by defining successful results as products that have at least a component (respectively all the components) different from error values. It should then be possible to obtain a logic language on top of this structure by defining an appropriate strategy for backtracking. Useful applications lie in the field of pattern recognition and manipulation of strings and trees.

The *Rewriting-calculus* (Rho) [CKL01b, CLW04] integrates matching, rewriting, and functions in a uniform way; its abstraction mechanism is based on rewrite rule formation: in a term of the form $P \rightarrow A$, one abstracts over the pattern P (instead over a simple variable as in the Lambda-calculus). The Rewriting-calculus is a generalization of the Lambda-calculus since one may choose the pattern P to be a variable. If an abstraction $P \rightarrow A$ is applied to the term B , then the evaluation mechanism is based on (1) bind the free variables present in P to appropriate subterms of B building a substitution θ , and (2) apply θ to A . Indeed, this binding is achieved by matching P against B . In rewriting-based languages, pattern matching can be “customizable” with more sophisticated matching theories, *e.g.* building-in associativity and/or commutativity of equality.

The original Rho calculus is computationally complete, and, thanks to pattern-matching, Lambda-calculus and fixed-points can be encoded and type-checked by using *ad hoc* patterns. In fact, Rho is a direct generalization of the core of a typed (rewriting-based and functional) programming language (of the MLU_Elan family) in which, roughly speaking, an ML-like `let` becomes by default a `let rec`, by abstracting over a suitable pattern P ; 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 structuring and destructuring structures, like lists (we record only the names of the constructor and we discard those of the accessors). Since structures are built into the calculus, it follows that the encoding of constructor/accessors is simpler than the standard encoding in the Lambda-calculus. The Table 1 informally compares the untyped encoding of accessors in the two formalisms.

[†] “Optimistic” means that, if a matching failure occurs, then the computation is not halted; this is in contrast to a “pessimistic” machine, that “kills” the computation once a failure-value is produced.

ops/form	Rho-calculus	Lambda-calculus
cons	(cons $X Y$)	$\lambda X. \lambda Y. \lambda Z. Z X Y$
car	(cons $X Y$) $\rightarrow X$	$\lambda Z. Z (\lambda X. \lambda Y. X)$
cdr	(cons $X Y$) $\rightarrow Y$	$\lambda Z. Z (\lambda X. \lambda Y. Y)$

Table 1. *Accessors and destructors in Rho/Lambda calculi*

Original Contribution. This work presents the first version of the *Imperative Rewriting-calculus* (iRho), an extension of Rho with references, memory allocation, and assignment *à la* ML [FF89]. To our knowledge, no similar study exists in the literature. The iRho-calculus is a powerful calculus, both at the syntactic and at the semantic level. It includes all the features of functional/rewriting-based languages with imperative aspects and pattern-matching facilities.

The controlled use of references, in the style of the ML language [MTHM97] also gives the user the programming ease and expressiveness that might not a priori be expected from such a simple calculus.

The crucial ingredients of iRho are the combination of (i) modern and safe imperative features, which give full control over the internal data-structure representation, and of (ii) “matching power”, which provides the main Lisp-like operations, like cons/car/cdr. The language iRho provides a good theoretical foundation for an emerging family of languages combining rewriting, functions, and patterns with semi-structured XML-data (e.g., XDUCE [Xdu05], CDUCE [Fri05]) or combining object-orientation and patterns with semi-structured data (e.g., HYDROJ [LLC03][‡]).

From Theory to Practice and Vice versa. We present static and dynamic semantics of iRho. The dynamic semantics is given via a natural deduction system (big-step) [Mar84, Mil87, Plo81, Tof87, Kah87, Plo04]. 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. In this design phase we try to not forget the needs and the objectives of a future implementor of the language, *i.e.*, a to build 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”, and a bit of care on performances.

The static and dynamic semantics are suitable to be specified mathematically, to be implemented with high-level programming languages, *e.g.* Bigloo [Ser05] (of the Scheme family), and to be certified with a modern and semi-automatic proof assistant, *e.g.* Coq [Log05].

With this goal in mind 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. This process has often influence the design of the semantics. The continuous interplay between mathematics and manual (*i.e.*, pen and paper) *vs.* mechanical proofs, and prototype

[‡] ...object-oriented pattern-matching naturally focuses on the essential information in a message and is insensitive to inessential information...”

implementations using high-level languages such as Scheme (and back) has been fruitful since the very beginning of our project. Although our calculus is rather simple, we expect in the near future to scale-up to larger projects, such as the certified implementation of compilers for a programming language of the C family [CFCL⁺03, Ler05].

Therefore, the main contributions of this paper are as follows.

- We provide a typed framework that enhances the functional language Rho, with imperative features like referencing, dereferencing, and assignment operators, and we enrich the type system with dereferencing-types 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.
- We experiment an interesting “pattern”[§] (in the sense of “*The Gang of Four*” [GHJT94]) called DIMPRO, a.k.a. Design-IMplement-PROve, to design safe software, which respects *in toto* its mathematical and functional specifications. Essentially, we started from a clean and elegant mathematical design, we continued with an implementation of a prototype satisfying the design, 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.

Refinement of this process was 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 [Abr96]). All three phases have the same status, and each influences the other.

Road Map. The paper is structured as follows. In Sections 2 and 3, we present respectively the syntax and the operational semantics of the functional Rho and of the imperative Rewriting-calculus iRho. Section 4 describes the type system. Section 5 presents the metatheory for iRho. Section 6 contains various examples of terms, reductions, and type-checking. Section 7 presents the formalization of iRho in Coq. Section 8 contains some remarks about our methodology and describes some “views” of the Natural Semantics, conclusions and further works.

The Coq encoding of the dynamic and static semantics (with their theorems) and the prototype implementation of an interpreter in Bigloo can be found at: <http://www-sop.inria.fr/mirho/Luigi.Liquori/iRho/>.

2. The Functional Rho

For pedagogical reasons we start by presenting the functional Rho. This will allow us to introduce almost all the ingredients and the technicalities needed to scale-up to the full imperative iRho. In a nutshell, Rho is a functional calculus with pattern-matching,

[§] “A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts...” [RZ96].

$\tau ::= \mathbf{b} \mid \tau \rightarrow \tau \mid \tau \wedge \tau$	Types
$\Delta ::= \emptyset \mid \Delta, X:\tau \mid \Delta, f:\tau$	Contexts
$P ::= X \mid f \bar{P} \mid P, P$	Patterns
$A ::= a \mid X \mid P \rightarrow_{\Delta} A \mid A A \mid A, A$	Terms

Fig. 1. Syntax of Rho

and can be seen as the kernel of any (statically typed) programming language based on functions, term rewriting and pattern-matching; many term rewriting systems can be encoded as well in *iRho*, (see [CLW04] for the exact class of term rewriting systems that can be encoded). Since the presentation of *Rho* mimics our current implementation, we make use of closures instead of meta substitutions.

2.1. Functional Syntax

Notational Conventions. We use the meta-symbols \rightarrow (function- and type-abstraction), and “,” (structure operator), and the implicit \bullet (application operator). We assume that the application operator \bullet associates to the left, while the other operators associate to the right. The priority of \bullet is higher than that of \rightarrow which is, in turn, of higher priority than “,”.

The symbols A, B, C, \dots range over the set \mathcal{T}_A of terms, the symbols X, Y, Z, \dots , **SELF**, \dots ranges over the set \mathcal{X} of variables ($\mathcal{X} \subseteq \mathcal{T}_A$), the symbols a, b, c, \dots, f, \dots , **cons**, **true**, **false**, **not**, **and**, **or**, **dummy**, \dots range over a set \mathcal{K} of term-constants ($\mathcal{K} \subseteq \mathcal{T}_A$). 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}_{τ} of types, the symbol \mathbf{b} ranges over the set of type-constants, the symbols Γ, Δ ranges over contexts. The symbols A_v, B_v, C_v, \dots range over the set \mathcal{Val} of values. We sometimes write \bar{A} for $A_1 \cdots A_n$, for $n \geq 0$. The symbol \equiv denotes syntactic equality. The syntax of *Rho* is presented in Figure 1.

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

Patterns. An unrestricted use of patterns in lambda-abstraction may lead to a failure of confluence in small-step semantics (see [Klo80]). To retain confluence, Vincent van Oostrom [vO90] introduced a suitable syntactical condition on the formation of patterns, called, the *Rigid Pattern Condition* (RPC), which (i) forces patterns to be “linear” (*i.e.*, no double occurrences of free variables, thus avoiding, the pattern $(f X X)$), and (ii) forbids “active” variables (thus avoiding the pattern $(X P)$).

Not all functional languages with pattern-matching apply the same restrictions concerning linearity in patterns. For example *Scheme* accepts non-linear patterns, permitting comparison of subparts of the datum (through **eq?**), while *ML* enforces linearity in pat-

terns (but **when** guards can be used to test for equality between two parts of a data structure). **Haskell**, because of its lazy evaluation strategy, accepts only patterns that are linear. The solution we adopt in this formalization and implementation of the Rewriting-calculus was influenced by the choice of the implementation language of our operational semantics, namely **Scheme**. The specification of the matching algorithm in **iRho** accepts non-linear patterns, and compares subparts of the datum (through \equiv , implemented via the primitive `equiv?` in **Scheme**). Confluence is preserved thanks to the call-by-value strategy of the operational semantics.

The shape of patterns has been limited to algebraic terms and structures (*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 [BCKL03]).

Terms. The main intuitions behind the term syntax are as follows.

- (*Variable and Constant*) are exactly as in the Lambda-calculus with constants;
- (*Structure*) allows one to express lists, sets, objects, etc.;
- (*Pattern Abstraction*) allows one to match over patterns. This gives a conservative extension of the simply-typed Lambda-calculus when the pattern is a simple variable, *i.e.* $\lambda X:\tau.A \simeq X \rightarrow_{X:\tau} A$; the context Δ in the pattern abstraction records the types of *all* the free variables of P (possibly bound in the body A). For example, the accessors `car` (in a homogeneous list) can be written in **Rho** as follows:

$$\text{car} \triangleq (\text{cons } X Y) \rightarrow_{\Delta} X \text{ with } \Delta \equiv X:\tau, Y:\tau'$$

- (*Application*) allows one to apply a pattern abstraction $P \rightarrow_{\Delta} A$ to an argument B , which of course must match on P . The terms are reduced under a classical call-by-value evaluation strategy; in the evaluation, the body of a pattern abstraction is not evaluated until the function is called on a suitable value (*i.e.*, pattern abstraction are values). For example, `(car (car (cons a b) c))` will reduce to `a`;

Observe that compared with “non-strategic” implementations of the Rewriting-calculus [CKL01a, CKL01b, CKL02, BCKL03], the delayed matching-constraint $[P \ll_{\Delta} A].B$, becomes now just syntactic sugar for $(P \rightarrow_{\Delta} A) B$ (omitted from the source language but still present in the set of output values).

Values and Environments. The set \mathcal{Val} of *values*, and the set of environments \mathcal{Env} are defined below:

$$A_v ::= a \bar{A}_v \mid A_v, A_v \mid \langle P \rightarrow_{\Delta} A \cdot \rho \rangle \mid \langle [P \ll_{\Delta} A_v].B \cdot \rho \rangle \quad \text{Functional Values}$$

Environments (denoted by ρ) are partial functions from the set of variables to the set of values, *i.e.*, $\rho \in \mathcal{Env} \simeq [\mathcal{X} \Rightarrow \mathcal{Val}]_{\perp}$. The extension of an environment is denoted by $\rho[X \mapsto A_v]$ and it is defined by:

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

Remark 1 (On Failure-values and Exceptions). “Failure-values” $\langle [P \ll_{\Delta} A_v].B.\rho \rangle$ denote failures occurring when we cannot find a correct substitution θ on the free variables of P such that $\theta(P) \equiv A_v$; the environment ρ records the value of the free variables of B . Failure-values are obtained during the computation when a matching failure occurs. They can in principle be discarded, or caught by an exception handler (see [CKL02]) that can be implemented in the interpreter (see the discussion of future work below).

In this paper, for the sake of simplicity, we will not deal with pattern-mismatch errors and pattern-exceptions (but this feature is available as an option in our interpreter). In the examples of Section 6, when a computation terminates with success (*i.e.*, not a failure-value), all intermediate failure-values are simply discharged from the final output. The interested reader can have a look at [CLW04] for necessary extensions/enhancements of an operational semantics and for a suitable matching theory that automatically drops failure-values.

2.2. The Rhosetta Stone

The Rewriting-calculus is known to be a conservative extension of the Lambda-calculus [CKL01a, CK01]. Nevertheless, it is typically presented using an infix notation, using as binder the meta-symbol “arrow” (\rightarrow), instead of the prefix notation using as a binder the meta-symbol “lambda” (λ) in conjunction with the meta-symbol “point” (\cdot). Moreover, since an abstraction can bind more than one variable, the type decoration of a pattern is given by a “context” (Δ) instead of a simple type. The rationale is:

$$\begin{aligned} \lambda X: \underbrace{X:\tau_1}_{\Delta}.A &\simeq X \rightarrow_{\Delta} A && \text{variables as patterns} \\ \lambda(f X Y): \underbrace{X:\tau_1, Y:\tau_2}_{\Delta}.A &\simeq (f X Y) \rightarrow_{\Delta} A && \text{algebraic patterns} \end{aligned}$$

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

$$\text{Fv}(P \rightarrow_{\Delta} A) \triangleq \text{Fv}(A) \setminus \text{Fv}(P)$$

The other cases of the Fv definition are in Definition 1.

Let-like and conditionals. Let-like constructs can be generalized to include pattern by viewing them as syntactic sugar for applications, *i.e.*

$$\text{let } P \ll A \text{ in } B \triangleq (P \rightarrow B) A$$

Conditionals can also be easily encoded in Rho, using pairing and application (true , and false are constants), *i.e.*

$$\text{neg} \triangleq (\text{true} \rightarrow \text{false}, \text{false} \rightarrow \text{true})$$

$$\text{if } A \text{ then } B \text{ else } C \triangleq (\text{true} \rightarrow ((X \rightarrow B) \text{dummy}), \text{false} \rightarrow ((X \rightarrow C) \text{dummy})) A$$

Observe that the then and the else branches are wrapped in a dummy abstraction, and X is fresh in B and C , *i.e.*, $X \notin \text{Fv}(B) \cup \text{Fv}(C)$.

Context Syntax

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

Rewriting-like Syntax

$A ::= P \rightarrow_{\Delta} A \mid A, B \mid \overbrace{((X_1, X_2) \rightarrow_{\Delta} X_{1,2}) (A, B)}^{\text{proj}} \mid \dots$

Lambda-like Syntax

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

with $\Delta \equiv X_1:\tau_1, X_1:\tau_2$ and $\tau_3 \equiv \tau_1 \rightarrow \tau_2 \rightarrow \tau_{1,2}$ and $\tau_4 \equiv \tau_3 \rightarrow \tau_{1,2}$

Fig. 2. The Rhosetta (Functional) Stone

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 Rewriting-like one, with a particular focus on the pair/projection encoding.

2.3. Functional Operational Semantics

We define a big-step call-by-value operational semantics via a natural proof deduction system. The purpose of the deduction system is to map every expression into a value, which is an irreducible term in weak head-normal form. The semantics is defined via three judgments, of the shapes:

$$\rho \vdash A \Downarrow_{\text{val}} A_v \quad \text{and} \quad \vdash \langle A_v \cdot B_v \rangle \Downarrow_{\text{call}} C_v \quad \text{and} \quad \rho \vdash \langle A \cdot A_v \rangle \Downarrow_{\text{match}} \rho'$$

The first judgment evaluates a term in **Rho**, while 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, where the symbol \star stands either for \bullet or “ \cdot ”. In a nutshell:

- (Red·Val) This rule evaluates every constant to itself;
- (Red·Var) This rule simply fetches the value of X into the environment;
- (Call·Struct) This rule applies every element of the structure-value to the argument C_v ;
- (Red·Fun) This rule evaluates a pattern abstraction to a closure;
- (Red·Appl_v) This rule first 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_{call} judgment;
- (Call·Algr) This rule builds an algebraic-value under the shape of an application in weak head-normal form;
- (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 $\Downarrow_{\text{match}}$;
- (Call·FunKo) This rule applies when the match of P against B_v fails: a failure-value is returned;

Value Reduction \Downarrow_{val}

$$\begin{array}{c}
\frac{}{\rho \vdash a \Downarrow_{\text{val}} a} \text{(Red.Val)} \\
\\
\frac{X \in \text{Dom}(\rho)}{\rho \vdash X \Downarrow_{\text{val}} \rho(X)} \text{(Red.Var)} \\
\\
\frac{\rho \vdash A \Downarrow_{\text{val}} A_v \quad \rho \vdash B \Downarrow_{\text{val}} B_v}{\rho \vdash A, B \Downarrow_{\text{val}} A_v, B_v} \text{(Red.Struct)} \\
\\
\frac{}{\rho \vdash P \rightarrow_{\Delta} A \Downarrow_{\text{val}} \langle P \rightarrow_{\Delta} A \cdot \rho \rangle} \text{(Red.Fun)} \\
\\
\frac{\rho \vdash A \Downarrow_{\text{val}} A_v \quad \rho \vdash B \Downarrow_{\text{val}} B_v \quad \vdash \langle A_v \cdot B_v \rangle \Downarrow_{\text{call}} C_v}{\rho \vdash AB \Downarrow_{\text{val}} C_v} \text{(Red.Applv)}
\end{array}$$

Call Reduction \Downarrow_{call}

$$\begin{array}{c}
\frac{}{\vdash \langle f \bar{A}_v \cdot B_v \rangle \Downarrow_{\text{call}} f \bar{A}_v B_v} \text{(Call.Algbr)} \\
\\
\frac{\vdash \langle A_v \cdot C_v \rangle \Downarrow_{\text{call}} D_v \quad \vdash \langle B_v \cdot C_v \rangle \Downarrow_{\text{call}} E_v}{\vdash \langle (A_v, B_v) \cdot C_v \rangle \Downarrow_{\text{call}} D_v, E_v} \text{(Call.Struct)} \\
\\
\frac{\rho \vdash \langle P \cdot B_v \rangle \Downarrow_{\text{match}} \rho' \quad \rho' \vdash A \Downarrow_{\text{val}} A_v}{\vdash \langle \langle P \rightarrow_{\Delta} A \cdot \rho \rangle \cdot B_v \rangle \Downarrow_{\text{call}} A_v} \text{(Call.FunOk)} \\
\\
\frac{\nexists \rho'. \rho \vdash \langle P \cdot B_v \rangle \Downarrow_{\text{match}} \rho'}{\vdash \langle \langle P \rightarrow_{\Delta} A \cdot \rho \rangle \cdot B_v \rangle \Downarrow_{\text{call}} \langle [P \ll_{\Delta} B_v].A \cdot \rho \rangle} \text{(Call.FunKo)} \\
\\
\frac{}{\vdash \langle [P \ll_{\Delta} B_v].A \cdot \rho \rangle \cdot C_v \rangle \Downarrow_{\text{call}} \langle [P \ll_{\Delta} B_v].A \cdot \rho \rangle} \text{(Call.Wrong)}
\end{array}$$

Fig. 3. Natural Functional Semantics

- (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-VarNew)(Match-VarEq) Matching a variable against a value produces an environment updated with the new binding, or the same environment if the variable is already bound with exactly the same value;

$$\begin{array}{c}
\textbf{Matching Reduction } \Downarrow_{\text{match}} \\
\frac{}{\rho \vdash \langle a \cdot a \rangle \Downarrow_{\text{match}} \rho} \text{(Match·Const)} \\
\\
\frac{\rho(X) = \perp}{\rho \vdash \langle X \cdot A_v \rangle \Downarrow_{\text{match}} \rho[X \mapsto A_v]} \text{(Match·VarNew)} \\
\\
\frac{\rho(X) = A_v}{\rho \vdash \langle X \cdot A_v \rangle \Downarrow_{\text{match}} \rho} \text{(Match·VarEq)} \\
\\
\frac{\rho_0 \vdash \langle A \cdot A_v \rangle \Downarrow_{\text{match}} \rho_1 \quad \rho_1 \vdash \langle B \cdot B_v \rangle \Downarrow_{\text{match}} \rho_2}{\rho_0 \vdash \langle A \star B \cdot A_v \star B_v \rangle \Downarrow_{\text{match}} \rho_2} \text{(Match·Pair)}
\end{array}$$

Fig. 4. Natural Matching Semantics

— (Match·Pair) Matching either an application or a structure (recall that $\star \in \{\cdot, \}$) produces an environment resulting from the composition of two environments.

The natural operational semantics is deterministic and immediately suggests how to build an interpreter. The standard first-order matching algorithm of [Hue76] is implemented by a judgment $\rho \vdash \langle A \cdot A_v \rangle \Downarrow_{\text{match}} \rho'$ that given an environment, a term and a value, either enriches the environment or fails. However, the algorithm has been enhanced in order to take into account the imperative features of the calculus.

Remark 2 (Optimistic vs. Pessimistic vs. Clean Machines). Our natural semantics is “optimistic”, in the sense that, if a matching failure occurs, then the computation is not halted and we explicitly list, in the final result, such a failure. Of course, other choices are possible, such as the one of “killing” the computation once a failure-value is produced, or “cleaning” all failure-values from the final result. The distinction is clearly visible when dealing with structures, as shown in the following trivial example (we let $\Downarrow_{\text{val}}^{\text{opt}}$ – resp. $\Downarrow_{\text{val}}^{\text{pex}}$ – resp. $\Downarrow_{\text{val}}^{\text{clean}}$, denote the optimistic – resp. pessimistic – resp. clean machine)

$$\begin{array}{c}
\vdots \\
\frac{}{\emptyset \vdash (3 \rightarrow 3, 4 \rightarrow 4) 4 \Downarrow_{\text{val}}^{\text{opt}} \langle [3 \ll 4].3 \cdot \emptyset \rangle, 4} \\
\vdots \\
\frac{}{\emptyset \vdash (3 \rightarrow 3, 4 \rightarrow 4) 4 \Downarrow_{\text{val}}^{\text{pex}} \langle [3 \ll 4].3 \cdot \emptyset \rangle} \\
\vdots \\
\frac{}{\emptyset \vdash (3 \rightarrow 3, 4 \rightarrow 4) 4 \Downarrow_{\text{val}}^{\text{clean}} 4}
\end{array}$$

We conclude with a simple example of a two functional evaluations.

$$\begin{array}{c}
\frac{\frac{\frac{\emptyset \vdash \langle f \cdot f \rangle \Downarrow_{\text{match}} \emptyset}{\emptyset \vdash \langle X \cdot 3 \rangle \Downarrow_{\text{match}} \rho_0}}{\emptyset \vdash \langle (f X) \cdot (f 3) \rangle \Downarrow_{\text{match}} \rho_0} \quad \frac{\frac{\frac{\rho_0 \vdash \langle 3 \cdot 3 \rangle \Downarrow_{\text{match}} \rho_0 \quad \rho_0 \vdash 3 \Downarrow_{\text{val}} 3}{\vdash \langle \langle (3 \rightarrow 3) \cdot \rho_0 \rangle \cdot 3 \rangle \Downarrow_{\text{call}} 3}}{\rho_0 \vdash X \Downarrow_{\text{val}} 3}}{\rho_0 \vdash (3 \rightarrow 3) \Downarrow_{\text{val}} \langle (3 \rightarrow 3) \cdot \rho_0 \rangle}}{\rho_0 \vdash (3 \rightarrow 3) X \Downarrow_{\text{val}} 3}} \\
\circledast \quad \Delta \quad \frac{\vdash \langle \langle (f X) \rightarrow (3 \rightarrow 3) X \rangle \cdot \emptyset \rangle \cdot (f 3) \rangle \Downarrow_{\text{call}} 3}{\emptyset \vdash ((f X) \rightarrow (3 \rightarrow 3) X) (f 3) \Downarrow_{\text{val}} 3}
\end{array}$$

and

$$\begin{array}{c}
\frac{\frac{\frac{\emptyset \vdash \langle f \cdot f \rangle \Downarrow_{\text{match}} \emptyset}{\emptyset \vdash \langle X \cdot 4 \rangle \Downarrow_{\text{match}} \rho_1}}{\emptyset \vdash \langle (f X) \cdot (f 4) \rangle \Downarrow_{\text{match}} \rho_1} \quad \frac{\frac{\frac{\not\exists \rho_2. \rho_1 \vdash \langle 3 \cdot 4 \rangle \Downarrow_{\text{match}} \rho_2}{\vdash \langle \langle (3 \rightarrow 3) \cdot \rho \rangle \cdot 4 \rangle \Downarrow_{\text{call}} A_v}}{\rho_1 \vdash X \Downarrow_{\text{val}} 4}}{\rho_1 \vdash (3 \rightarrow 3) \Downarrow_{\text{val}} \langle (3 \rightarrow 3) \cdot \rho_1 \rangle}}{\rho_1 \vdash (3 \rightarrow 3) X \Downarrow_{\text{val}} A_v}} \\
\circledast \quad \square \quad \frac{\vdash \langle \langle (f X) \rightarrow (3 \rightarrow 3) X \rangle \cdot \emptyset \rangle \cdot (f 4) \rangle \Downarrow_{\text{call}} A_v}{\emptyset \vdash ((f X) \rightarrow (3 \rightarrow 3) X) (f 4) \Downarrow_{\text{val}} A_v}
\end{array}$$

Fig. 5. Natural Deduction of $((f X) \rightarrow (3 \rightarrow 3) X) (f 3)$ and $((f X) \rightarrow (3 \rightarrow 3) X) (f 4)$

Example 1 (Two Functional Evaluations). Consider the following term in Rho:

$$((f X) \rightarrow (3 \rightarrow 3) X) (f 3) \quad \text{and} \quad ((f X) \rightarrow (3 \rightarrow 3) X) (f 4)$$

and let

$$\begin{aligned}
\circledast &\equiv \emptyset \vdash \langle \langle (f X) \rightarrow (3 \rightarrow 3) X \rangle \cdot \emptyset \rangle \Downarrow_{\text{val}} \langle \langle (f X) \rightarrow (3 \rightarrow 3) X \rangle \cdot \emptyset \rangle \\
\Delta &\equiv \emptyset \vdash (f 3) \Downarrow_{\text{val}} (f 3) \\
\square &\equiv \emptyset \vdash (f 4) \Downarrow_{\text{val}} (f 4) \\
A_v &\equiv \langle [3 \ll 4] \cdot 3 \cdot \rho \rangle \quad \rho_0 \triangleq [X \mapsto 3] \quad \rho_1 \triangleq [X \mapsto 4]
\end{aligned}$$

The deduction trees are shown in Figure 5.

3. The Imperative Rewriting-calculus

3.1. Imperative Syntax

We introduce imperative features in our Rewriting-calculus, to yield the full iRho. We extend 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 [Tof87, FF89].

$\tau ::= \dots$ as in Rho...	τ ref	Types
$\Delta ::= \dots$ as in Rho...		Contexts
$P ::= \dots$ as in Rho...	ref P	Patterns
$A ::= \dots$ as in Rho...	ref A $A := A$	Terms

Fig. 6. Syntax of iRho

Syntax. The syntax of iRho (types, contexts, patterns and terms) is presented in Figure 3. Intuitively, iRho deals with references *à la* ML *i.e.*:

- (*Ref-Terms*) The term $\text{ref } A$ is a referencing term (the-location-of); if A is a term of type τ , then $\text{ref } A$ is a pointer to A of type τ ref;
- (*Assignment-terms*) The term $A := B$ is an assignment operator, which returns as result the value obtained by evaluating B .

As an immediate benefit of the built-in powerful new pattern-matching algorithm, the classical dereferencing term (goto-memory), denoted by $!A$, where A is a pointer in the store can be easily defined as follows (types are omitted):

$$!A \triangleq (\text{ref } X \rightarrow X) A$$

This is a nice feature with respect to functional core calculi, mixing imperative and functional features, like, *e.g.* Caml.

Sequencing can be also defined in iRho as follows (types are omitted):

$$A; B \triangleq (X \rightarrow B) A \quad X \notin \text{Fv}(B)$$

Issues related to garbage collection are out of the scope of this paper: new locations created during reduction, via referencing ($\text{ref } A$), will remain in the store forever. In principle, classical techniques of Ian Mason and Carolyn Talcott, and Greg Morrisset *et al.* [MT92, MFH95] could be applied to iRho.

Values and Stores. The set of values is enriched by locations. The symbol ι ranges over the set \mathcal{Loc} of store locations, and the symbol σ ranges over the set of global stores Store .

$$A_v ::= \dots \text{ as in Rho} \dots \mid \iota \quad \text{Imperative Values}$$

Stores are partial functions from the set \mathcal{L} of locations to the set of values *i.e.*, $\sigma \in \text{Store} \simeq [\mathcal{Loc} \Rightarrow \mathcal{Val}]_{\perp}$; we denote the extension of a store by $\sigma[\iota \mapsto A_v]$ with the following meaning:

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

Value Reduction \Downarrow_{val}

$$\frac{\sigma_0 \cdot \rho \vdash A \Downarrow_{\text{val}} A_v \cdot \sigma_1 \quad \sigma_1 \cdot \rho \vdash B \Downarrow_{\text{val}} B_v \cdot \sigma_2}{\sigma_0 \cdot \rho \vdash A, B \Downarrow_{\text{val}} A_v, B_v \cdot \sigma_2} \text{(Red·Struct)}$$

$$\frac{\iota \notin \text{Dom}(\sigma_1) \quad \sigma_0 \cdot \rho \vdash A \Downarrow_{\text{val}} A_v \cdot \sigma_1}{\sigma_0 \cdot \rho \vdash \text{ref } A \Downarrow_{\text{val}} \iota \cdot \sigma_1[\iota \mapsto A_v]} \text{(Red·Ref)}$$

$$\frac{\sigma_0 \cdot \rho \vdash A \Downarrow_{\text{val}} A_v \cdot \sigma_1 \quad \sigma_1 \cdot \rho \vdash B \Downarrow_{\text{val}} B_v \cdot \sigma_2 \quad \sigma_2 \vdash \langle A_v \cdot B_v \rangle \Downarrow_{\text{call}} C_v \cdot \sigma_3}{\sigma_0 \cdot \rho \vdash AB \Downarrow_{\text{val}} C_v \cdot \sigma_3} \text{(Red·Appl}_v\text{)}$$

$$\frac{\sigma_0 \cdot \rho \vdash A \Downarrow_{\text{val}} \iota \cdot \sigma_1 \quad \iota \in \text{Dom}(\sigma_1) \quad \sigma_1 \cdot \rho \vdash B \Downarrow_{\text{val}} B_v \cdot \sigma_2}{\sigma_0 \cdot \rho \vdash A := B \Downarrow_{\text{val}} B_v \cdot \sigma_2[\iota \mapsto B_v]} \text{(Red·Ass)}$$

Update of (Red·Val), (Red·Fun), (Red·Var) with the unused store parameter.

Call Reduction \Downarrow_{call}

$$\frac{\sigma_0 \cdot \rho \vdash \langle P \cdot B_v \rangle \Downarrow_{\text{match}} \rho' \quad \sigma_0 \cdot \rho' \vdash A \Downarrow_{\text{val}} A_v \cdot \sigma_1}{\sigma_0 \vdash \langle \langle P \rightarrow_{\Delta} A \cdot \rho \rangle \cdot B_v \rangle \Downarrow_{\text{call}} A_v \cdot \sigma_1} \text{(Call·FunOk)}$$

$$\frac{\sigma_0 \vdash \langle A_v \cdot C_v \rangle \Downarrow_{\text{call}} D_v \cdot \sigma_1 \quad \sigma_1 \vdash \langle B_v \cdot C_v \rangle \Downarrow_{\text{call}} E_v \cdot \sigma_2}{\sigma_0 \vdash \langle \langle A_v, B_v \rangle \cdot C_v \rangle \Downarrow_{\text{call}} D_v, E_v \cdot \sigma_2} \text{(Call·Struct)}$$

Update of (Call·FunKo), (Call·Algr), (Call·Wrong) with the unused store parameter.

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

$$\frac{\iota \in \text{Dom}(\sigma) \quad \sigma(\iota) \equiv A_v \quad \sigma \cdot \rho \vdash \langle P \cdot A_v \rangle \Downarrow_{\text{match}} \rho'}{\sigma \cdot \rho \vdash \langle \text{ref } P \cdot \iota \rangle \Downarrow_{\text{match}} \rho'} \text{(Match·Ref)}$$

Update of (Match·Const), (Match·Var), (Match·Pair) with the unused store parameter.

Fig. 7. Natural Imperative Semantics

3.2. Imperative Operational Semantics

As in the functional case, we define an “optimistic” big-step operational semantics. Again, the chosen strategy is *call-by-value*, and the semantics is defined via three judgments of the shape:

$$\sigma \cdot \rho \vdash A \Downarrow_{\text{val}} A_v \cdot \sigma' \quad \text{and} \quad \sigma \vdash \langle A_v \cdot B_v \rangle \Downarrow_{\text{call}} C_v \cdot \sigma' \quad \text{and} \quad \sigma \cdot \rho \vdash \langle A \cdot A_v \rangle \Downarrow_{\text{match}} \rho'$$

The main difference with respect to the functional calculus Rho is that all judgments have as premise a global store σ , which can be modified and returned as a result. In the case of \Downarrow_{val} and \Downarrow_{call} , a store σ is given as input, and a (possibly modified) store σ' is returned as output. In the $\Downarrow_{\text{match}}$ rule, a store σ is needed as input since our matching algorithm allows to match a referencing terms $\text{ref } A$ to a pointer-variable, such as in:

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

The rules of the dynamic semantics are defined in Figure 7. In a nutshell:

- (Red·{v, Appl_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·Ass) This rule performs assignment: first we reduce the receiver A into an (existing) memory location, then we reduce the expression B (to be assigned) to 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 with respect to the corresponding rules in Rho ; the only difference lies in the store propagation from the input of the conclusion (through the premises) to the output of the conclusion (**value · store**);
- (Match·{Const, VarNew, VarEq, Pair}) All the matching judgments of Figure 4 are still valid by adding an (unused) extra parameter σ ;
- (Match·Ref) This rule is the only matching rule which needs a store as an input argument ; it first fetches the value A_v in the store σ , at the location ι , and then calls the matching of the pattern P against the value A_v . An example of imperative pattern-matching is:

$$[\iota_0 \mapsto 3] \cdot \emptyset \vdash (\text{ref } X \cdot \iota_0) \Downarrow_{\text{match}} [X \mapsto 3]$$

This pattern-matching rule allows to consider the dereferencing term $!A$ the status of simple sugar in iRho .

Observe that the following rule that first reduces A into a location-value ι , and then return the value stored at ι is admissible:

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

We conclude with a simple example of an imperative evaluation.

Example 2 (An Imperative Evaluation). Take the imperative term:

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

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 8.

$$\begin{array}{c}
\begin{array}{c}
\iota_0 \in \text{Dom}(\sigma_0) \qquad \sigma_0 \cdot \rho_0 \vdash Y \Downarrow_{\text{val}} \iota_1 \cdot \sigma_0 \\
\sigma_0 \cdot \rho_0 \vdash X \Downarrow_{\text{val}} \iota_0 \cdot \sigma_0 \qquad \sigma_0 \cdot \rho_0 \vdash !Y \Downarrow_{\text{val}} 4 \cdot \sigma_0
\end{array} \\
\hline
\begin{array}{c}
\sigma_0 \cdot \rho_0 \vdash X := !Y \Downarrow_{\text{val}} 4 \cdot \sigma_1 \\
\sigma_0 \cdot \rho_0 \vdash \langle 3 \cdot 3 \rangle \Downarrow_{\text{match}} \rho_0
\end{array} \\
\hline
\begin{array}{c}
\sigma_0 \vdash \langle \langle 3 \rightarrow X := !Y \cdot \rho_0 \rangle \cdot 3 \rangle \Downarrow_{\text{call}} 4 \cdot \sigma_1 \\
\sigma_0 \cdot \rho_0 \vdash !X \Downarrow_{\text{val}} 3 \cdot \sigma_0 \\
\sigma_0 \cdot \rho_0 \vdash 3 \rightarrow X := !Y \Downarrow_{\text{val}} \langle 3 \rightarrow X := !Y \cdot \rho_0 \rangle \cdot \sigma_0
\end{array} \\
\hline
\begin{array}{c}
\sigma_0, \rho_0 \vdash (3 \rightarrow X := !Y) !X \Downarrow_{\text{call}} 4 \cdot \sigma_1 \\
\sigma_0 \cdot \emptyset \vdash \langle (f(X, Y)) \cdot (f(\iota_0, \iota_1)) \rangle \Downarrow_{\text{match}} \rho_0
\end{array} \\
\hline
\begin{array}{c}
\sigma_0 \cdot \emptyset \vdash \langle \langle (f(X, Y)) \rightarrow (3 \rightarrow X := !Y) !X \rangle \cdot \emptyset \rangle \cdot (f(\iota_0, \iota_1)) \Downarrow_{\text{call}} 4 \cdot \sigma_1 \\
\emptyset \cdot \emptyset \vdash (f(\text{ref } 3, \text{ref } 4)) \Downarrow_{\text{val}} (f(\iota_0, \iota_1)) \cdot \sigma_0 \\
\emptyset \cdot \emptyset \vdash ((f(X, Y)) \rightarrow (3 \rightarrow X := !Y) !X) \Downarrow_{\text{val}} \langle (f(X, Y)) \rightarrow (3 \rightarrow X := !Y) !X \rangle \cdot \emptyset \cdot \emptyset
\end{array} \\
\hline
\begin{array}{c}
\emptyset \cdot \emptyset \vdash ((f(X, Y)) \rightarrow (3 \rightarrow X := !Y) !X) (f(\text{ref } 3, \text{ref } 4)) \Downarrow_{\text{val}} 4 \cdot \sigma_1
\end{array}
\end{array}$$

Fig. 8. Natural Deduction of $((f(X, Y)) \rightarrow (3 \rightarrow X := !Y) !X) (f(\text{ref } 3, \text{ref } 4))$

4. The Type System

In this section, we present a type system for iRho. We use, as usual, type checking to catch some errors before run-time evaluation. In the following section, we present a rich collection of (typable) examples, namely decision procedures, meaningful objects, fixed-points, and term rewriting systems. The type system can be extended with a *subtyping* relation, or with *bounded-polymorphism*, to capture the behavior of *structures-as-objects*, and object-oriented features. With respect to previous type systems for the (functional) Rho [CKL01b, CKL02, BCKL03, CLW04], structures *can now have different types*, *i.e.*, thanks to the following typing rule which is new with respect to the previous typed formulations of the Rewriting-calculus

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

Reminiscent of a record-types discipline, the type $\tau_1 \wedge \tau_2$ is suitable for heterogeneous structures, like lists, ordered sets, or objects. This enhancement gives a more flexible type discipline, where the product-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 with respect to previous typing disciplines on the Rewriting-calculus [CLW04], 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 main enhancement with respect to previous versions of the Rewriting-

calculus [BCKL03], is that here the elements of the structure are not forced to have the same type. This introduces a simple, although non-orthodox, notion of object-type.

The type system \vdash_A is *algorithmic*: the type rules are *deterministic* and suggest two *decidable procedures* for type-reconstruction and type-checking. We say that a set of rules specifies a deterministic typing algorithm if the type rules are *syntax-directed*, and each rule satisfies the *sub-formula property* (all the formulas appearing in the premise of a rule are sub-formulas 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 product-type containing all the components of the structure. As a simple example, if we apply a structure (with type $(b_1 \rightarrow b_2) \wedge (b_1 \rightarrow b_3)$) to an argument of type b_1 , we would obtain as result a structure-value of type $b_2 \wedge b_3$. 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 product-type into a function-type:

$$\begin{aligned} \text{arr}(\tau_1 \rightarrow \tau_2) &\triangleq \tau_1 \rightarrow \tau_2 \\ \text{arr}(\tau_1 \wedge \tau_2) &\triangleq \tau_3 \rightarrow (\tau_4 \wedge \tau_5) \left\{ \begin{array}{l} \text{if } \text{arr}(\tau_1) \equiv \tau_3 \rightarrow \tau_4 \\ \text{and } \text{arr}(\tau_2) \equiv \tau_3 \rightarrow \tau_5 \end{array} \right. \end{aligned}$$

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

$$\begin{array}{ccccccc} \Gamma \vdash_{\Gamma} ok & \Gamma \vdash_{\tau} \tau : ok & \Gamma \vdash_{\nu} A_{\nu} : \tau & & & & \\ \Gamma \vdash_{\rho} \rho : \Gamma' & \Gamma \vdash_{\sigma} \sigma : \Gamma' & \Gamma \vdash_{\mathfrak{p}} P : \tau & \Gamma \vdash_A A : \tau & & & \end{array}$$

which denote well-typed contexts, types, values, environments, stores, patterns, and terms, respectively. In the following, we let the symbol α range over $\mathcal{X} \cup \mathcal{K}$. The typing rules for patterns and terms are presented in Figure 9. Note that the following rule is admissible. It says that if A is a pointer to an object of type τ , then its access in memory, denoted by $!A$, has type τ .

$$\frac{\Gamma \vdash_A A : \tau \text{ ref}}{\Gamma \vdash_A !A : \tau} \text{ (Term·Deref)}$$

In what follows, we give a review of the most intriguing type-checking rules.

- (Patt·Start), (Term·Start) Those rules fetch from the context the correct type of variables and constants, respectively;
- (Patt·Struct), (Term·Struct) Those rules assign a product-type to a structure which records the type of both elements;
- (Patt·Algr), (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;

Pattern Rules

$$\begin{array}{c}
\frac{\Gamma_1, \alpha:\tau, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, \alpha:\tau, \Gamma_2 \vdash_{\mathbb{P}} \alpha : \tau} \text{(Patt·Start)} \\
\frac{\Gamma \vdash_{\mathbb{P}} P_1 : \tau_1 \quad \Gamma \vdash_{\mathbb{P}} P_2 : \tau_2}{\Gamma \vdash_{\mathbb{P}} P_1, P_2 : \tau_1 \wedge \tau_2} \text{(Patt·Struct)} \\
\frac{\text{arr}(\tau_1) \equiv \tau_2 \rightarrow \tau_3 \quad \Gamma \vdash_{\mathbb{P}} a \overline{P} : \tau_1 \quad \Gamma \vdash_{\mathbb{P}} P : \tau_2}{\Gamma \vdash_{\mathbb{P}} a \overline{P} P : \tau_3} \text{(Patt·Algr)}
\end{array}$$

Term Rules

$$\begin{array}{c}
\frac{\Gamma_1, \alpha:\tau, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, \alpha:\tau, \Gamma_2 \vdash_{\mathbb{A}} \alpha : \tau} \text{(Term·Start)} \\
\frac{\Gamma \vdash_{\mathbb{A}} A : \tau}{\Gamma \vdash_{\mathbb{A}} \text{ref } A : \tau \text{ ref}} \text{(Term·Ref)} \\
\frac{\Gamma \vdash_{\mathbb{A}} A : \tau \text{ ref} \quad \Gamma \vdash_{\mathbb{A}} B : \tau}{\Gamma \vdash_{\mathbb{A}} A := B : \tau} \text{(Term·Assign)} \\
\frac{\Gamma \vdash_{\mathbb{A}} A : \tau_1 \quad \Gamma \vdash_{\mathbb{A}} B : \tau_2}{\Gamma \vdash_{\mathbb{A}} A, B : \tau_1 \wedge \tau_2} \text{(Term·Struct)} \\
\frac{\text{arr}(\tau_1) \equiv \tau_2 \rightarrow \tau_3 \quad \Gamma \vdash_{\mathbb{A}} A : \tau_1 \quad \Gamma \vdash_{\mathbb{A}} B : \tau_2}{\Gamma \vdash_{\mathbb{A}} AB : \tau_3} \text{(Term·Appl)} \\
\frac{\text{Dom}(\Delta) = \text{Fv}(P) \quad \Gamma, \Delta \vdash_{\mathbb{P}} P : \tau_1 \quad \Gamma, \Delta \vdash_{\mathbb{A}} A : \tau_2}{\Gamma \vdash_{\mathbb{A}} P \rightarrow_{\Delta} A : \tau_1 \rightarrow \tau_2} \text{(Term·Abs)}
\end{array}$$

Fig. 9. Well Formed Pattern and Terms

- (Term·Abs) In this rule we note that the context Δ is charged in the premises, using the decidable function $\text{Fv}(P)$; 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 $\tau \text{ ref}$, 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 $\text{ref } A$, has type $\tau \text{ ref}$.

4.1. Extra Typing Rules

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

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

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·Algr) needs a transformation step for product-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. All extra rules are presented in Figure 10.

Context Rules

$$\begin{array}{c}
\frac{}{\emptyset \vdash_{\Gamma} ok} \text{(Ctx-Axiom)} \\
\frac{\Gamma \vdash_{\Gamma} ok \quad \mathbf{b} \notin \text{Dom}(\Gamma)}{\Gamma, \mathbf{b}:ok \vdash_{\Gamma} ok} \text{(Ctx-Type)} \\
\frac{\chi \notin \text{Dom}(\Gamma) \quad \Gamma \vdash_{\tau} \tau : ok \quad \Gamma \vdash_{\Gamma} ok}{\Gamma, \chi:\tau \vdash_{\Gamma} ok} \text{(Ctx-Var/Const)}
\end{array}$$

Type Rules

$$\begin{array}{c}
\frac{\Gamma_1, \mathbf{b}:ok, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, \mathbf{b}:ok, \Gamma_2 \vdash_{\tau} \mathbf{b} : ok} \text{(Type-Start)} \\
\frac{\Gamma \vdash_{\tau} \tau : ok}{\Gamma \vdash_{\tau} \tau \text{ ref} : ok} \text{(Type-Ref)} \\
\frac{\Gamma \vdash_{\tau} \tau_1 : ok \quad \Gamma \vdash_{\tau} \tau_2 : ok}{\Gamma \vdash_{\tau} \tau_1 \rightarrow \tau_2 : ok} \text{(Type-Arrow)} \\
\frac{\Gamma \vdash_{\tau} \tau_1 : ok \quad \Gamma \vdash_{\tau} \tau_2 : ok}{\Gamma \vdash_{\tau} \tau_1 \wedge \tau_2 : ok} \text{(Type-Struct)}
\end{array}$$

Value Typing

$$\begin{array}{c}
\frac{\Gamma_1, a:\tau, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, a:\tau, \Gamma_2 \vdash_v a : \tau} \text{(Value-Start}_1\text{)} \\
\frac{\Gamma_1, \iota:\tau \text{ ref}, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, \iota:\tau \text{ ref}, \Gamma_2 \vdash_v \iota : \tau \text{ ref}} \text{(Value-Start}_2\text{)} \\
\frac{\Gamma \vdash_v A_v : \tau_1 \quad \Gamma \vdash_v B_v : \tau_2}{\Gamma \vdash_v A_v, B_v : \tau_1 \wedge \tau_2} \text{(Value-Struct)} \\
\frac{\Gamma \vdash_v a \bar{A}_v : \tau_1 \quad \Gamma \vdash_v B_v : \tau_2}{\Gamma \vdash_v a \bar{A}_v B_v : \tau_3} \text{(Value-Algr)} \\
\frac{\Gamma \vdash_{\rho} \rho : \Gamma' \quad \Gamma' \vdash_p P \rightarrow_{\Delta} A : \tau_1 \rightarrow \tau_2}{\Gamma \vdash_v \langle P \rightarrow_{\Delta} A \cdot \rho \rangle : \tau_1 \rightarrow \tau_2} \text{(Value-Clos)} \\
\frac{\Gamma \vdash_v A_v : \tau_1 \quad \Gamma \vdash_{\rho} \rho : \Gamma' \quad \Gamma' \vdash_p P \rightarrow_{\Delta} B : \tau_1 \rightarrow \tau_2}{\Gamma \vdash_v \langle [P \ll_{\Delta} A_v].B \cdot \rho \rangle : \tau_2} \text{(Value-Fail)}
\end{array}$$

Store Typing

$$\begin{array}{c}
\Gamma[l:\tau] \triangleq \begin{cases} \Gamma, \iota:\tau & \text{if } \iota \notin \text{Dom}(\Gamma) \\ \Gamma & \text{if } \iota:\tau \in \Gamma \end{cases} \\
\frac{\Gamma \vdash_{\Gamma} ok}{\Gamma \vdash_{\sigma} \emptyset : \Gamma} \text{(Store-Axiom)} \\
\frac{\Gamma \vdash_{\sigma} \sigma : \Gamma' \quad \Gamma'[l:\tau] \vdash_{\Gamma} ok \quad \Gamma \vdash_v \iota : \tau \text{ ref} \quad \Gamma \vdash_v A_v : \tau}{\Gamma \vdash_{\sigma} \sigma[l \mapsto A_v] : \Gamma'[l:\tau]} \text{(Store-Loc)}
\end{array}$$

Environment Typing

$$\begin{array}{c}
\Gamma[X:\tau] \triangleq \begin{cases} \Gamma, X:\tau & \text{if } X \notin \text{Dom}(\Gamma) \\ \Gamma & \text{if } X:\tau \in \Gamma \end{cases} \\
\frac{\Gamma \vdash_{\Gamma} ok}{\Gamma \vdash_{\rho} \emptyset : \Gamma} \text{(Env-Axiom)} \\
\frac{\Gamma \vdash_{\rho} \rho : \Gamma' \quad \Gamma'[X:\tau] \vdash_{\Gamma} ok \quad \Gamma \vdash_{\lambda} X : \tau \quad \Gamma \vdash_v A_v : \tau}{\Gamma \vdash_{\rho} \rho[X \mapsto A_v] : \Gamma'[X:\tau]} \text{(Env-Var)}
\end{array}$$

Fig. 10. Extra Typing Rules

5. Metatheory

In this section we present the main properties of iRho, as follows.

- 1 Natural Semantics for \Downarrow_{val} is deterministic;
- 2 Type-checking with \vdash_{Δ} 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.

The most crucial proofs have been done by a mechanical development using the proof assistant Coq [LS05]. Some of this development is presented in detail in Section 7. We start with a natural definition of free variables.

Definition 1 (Free variables Fv).

$$\begin{array}{ll}
\text{Fv}(a) & \triangleq \emptyset & \text{Fv}(P \rightarrow_{\Delta} A) & \triangleq \text{Fv}(A) \setminus \text{Fv}(P) \\
\text{Fv}(X) & \triangleq \{X\} & \text{Fv}(AB) & \triangleq \text{Fv}(A) \cup \text{Fv}(B) \\
\text{Fv}(!A) & \triangleq \text{Fv}(A) & \text{Fv}(A, B) & \triangleq \text{Fv}(A) \cup \text{Fv}(B) \\
\text{Fv}(\text{ref } A) & \triangleq \text{Fv}(A) & \text{Fv}(A := B) & \triangleq \text{Fv}(A) \cup \text{Fv}(B)
\end{array}$$

Then, we prove that our natural semantics is deterministic.

Theorem 2 (Deterministic Semantics). For all term A , and environment ρ , and store σ :

- 1 If $\sigma_1 \cdot \rho \vdash A \Downarrow_{\text{val}} A_v \cdot \sigma_2$, and $\sigma_1 \cdot \rho \vdash A \Downarrow_{\text{val}} B_v \cdot \sigma_3$, then $A_v \equiv B_v$, and $\sigma_2 \equiv \sigma_3$;
- 2 If $\sigma_1 \vdash \langle A_v \cdot B_v \rangle \Downarrow_{\text{call}} C_v \cdot \sigma_2$, and $\sigma_1 \vdash \langle A_v \cdot B_v \rangle \Downarrow_{\text{call}} D_v \cdot \sigma_3$, then $C_v \equiv D_v$, and $\sigma_2 \equiv \sigma_3$;
- 3 If $\sigma \cdot \rho_1 \vdash \langle P \cdot A_v \rangle \Downarrow_{\text{match}} \rho_2$, and $\sigma \cdot \rho_1 \vdash \langle P \cdot A_v \rangle \Downarrow_{\text{match}} \rho_3$, then $\rho_2 \equiv \rho_3$.

Proof. By Coq. □

The type system presentation is syntax directed, hence it directly suggest how to build an algorithm, hence it enjoys the nice property of uniqueness of typing; a software prototype of a simple type-checker can be found in the web-appendix [LS05]).

Theorem 3 (Uniqueness of Typing). If $\Gamma \vdash_{\Delta} A : \tau$, then τ is unique.

Proof. By Coq. □

The following definition splits the typed context into two sub-contexts, the former recording types assigned to locations, and the latter recording types assigned to variables; it will be useful in the subject-reduction theorem.

Definition 4 (Coherence). The context Γ is coherent with a store σ , and an environment ρ , denoted by

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

if there exist two sub-contexts Γ_1 , and Γ_2 , such that $\Gamma_1, \Gamma_2 \equiv \Gamma$, and $\Gamma_1 \vdash_\sigma \sigma : \Gamma_1$, and $\Gamma_2 \vdash_\rho \rho : \Gamma_2$.

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

Theorem 5 (Subject-reduction for Open Terms). If $\sigma_1 \cdot \rho_1 \vdash A \Downarrow_{\text{val}} A_v \cdot \sigma_2$, and $\Gamma_1 \vdash_{\text{coh}} \sigma_1 \cdot \rho_1$, and $\Gamma_1 \vdash_A A : \tau$, then there exists Γ_2 , which extend Γ_1 , such that $\Gamma_2 \vdash_{\text{coh}} \sigma_2 \cdot \rho_1$, and $\Gamma_2 \vdash_v A_v : \tau$.

Proof. By Coq. □

The following result is crucial to state type soundness.

Theorem 6 (Subject-reduction for Closed Terms). If $\emptyset \cdot \emptyset \vdash A \Downarrow_{\text{val}} A_v \cdot \sigma$, and $\emptyset \vdash_A A : \tau$, then there exists Γ , such that $\Gamma \vdash_{\text{coh}} \sigma \cdot \emptyset$, and $\Gamma \vdash_v A_v : \tau$.

Proof. By Coq. □

The reduction rules for the operational semantics given in Figure 7 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 of the operational semantics shows that there are only few ways in which evaluation may get “stuck”:

- (\Downarrow_{val}) This judgment gets stuck when we access a variable not defined in the environment, when the evaluation of A in $A := B$ gives a fresh location (*i.e.*, not in the current used store), or if one premise in some judgment gets stuck;
- (\Downarrow_{call}) This judgment gets stuck when the we try to apply a location-value to a value (*e.g.* $\langle \iota \cdot A_v \rangle$), or if one premise in some judgment gets stuck;
- ($\Downarrow_{\text{match}}$) This judgment gets stuck when we try to match a pattern against a value with a different (unmatchable) shape, *e.g.* $\langle (PQ) \cdot (A, B) \rangle$, etc.

The following soundness theorem proves the absence of such errors in the evaluation of a well-typed closed expression.

Theorem 7 (Progress/Type-soundness).

Let A be a closed term such that $\emptyset \vdash_A A : \tau$ is derivable, and let $C[\cdot]$ be any context.

(Progress)

- if $A \equiv C[X]$, then there exists σ , and ρ , such that $\sigma \cdot \rho \vdash X \Downarrow_{\text{val}} \rho(X)$, and $\rho(X) \neq \perp$;
- if $A \equiv C[B := C]$, then there exists σ_1 , and ρ , such that $\sigma_1 \cdot \rho \vdash B \Downarrow_{\text{val}} \iota \cdot \sigma_2$, and $\iota \in \text{Dom}(\sigma_2)$;

(\Downarrow_{call}) If $A \equiv C[BC]$, then there exists σ_1 , and ρ , such that $\sigma_1 \cdot \rho \vdash B \Downarrow_{\text{val}} B_v \cdot \sigma_2$, and $B_v \neq \iota$;

($\Downarrow_{\text{match}}$) If $A \equiv C[BC]$, then there exists σ_1 , and ρ , such that $\sigma_1 \cdot \rho \vdash B \Downarrow_{\text{val}} \langle P \rightarrow_\Delta D \cdot \rho_1 \rangle \cdot \sigma_2$, and $\sigma_2 \cdot \rho \vdash C \Downarrow_{\text{val}} C_v \cdot \sigma_3$, and P will successfully match with C_v .

(Type-soundness)

If $\emptyset \vdash_A A : \tau$, then $\emptyset \cdot \emptyset \vdash A \Downarrow_{\text{val}} A_v$, for some A_v .

Proof.

(Progress)

(\Downarrow_{val}) if A has a variable X as a sub-expression, then, by well-typedness of the sub-expressions, environment and store, $\rho(X) \neq \perp$. If A has an assignment $B := C$ as a sub-expression, then, by well-typedness of the sub-expressions, environment and store, B evaluates to $\iota \cdot \sigma_2$ and $\iota \in \text{Dom}(\sigma_2)$.

(\Downarrow_{call}) Similarly, if A has an application BC as a sub-expression, then by well-typedness of the sub-expressions, environment and store, the evaluation of B is not a location.

($\Downarrow_{\text{match}}$) Again, if A has an application BC as a sub-expression, then by well-typedness of the sub-expressions, environment and store, the evaluation of B leads to a closure-value $\langle P \rightarrow_{\Delta} D \cdot \rho_1 \rangle$, and the pattern P has a shape that can overlap with C_v (the latter obtained by the evaluation of C).

(Type-soundness)

Immediate from Progress and from Subject-reduction Theorem. □

We conclude with some decidability results.

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

- 1 **(Type-checking)** It is decidable if there exists a type τ such that $\emptyset \vdash_A A : \tau$;
- 2 **(Type-reconstruction)** It is decidable if, for a given τ , is it true that $\emptyset \vdash_A A : \tau$.

Proof.

1 Figure 11 gives the sketch of a recursive algorithm for building τ , or returning **false** if it does not exist.

2 We use the previous algorithm for type reconstruction (Figure 11). By uniqueness of typing, $\Gamma \vdash_A A : \tau$ if and only if τ is equivalent to the type found for A . □

Theorem 9 (Soundness and Completeness of iType). For a closed A , and a given Δ , $\text{iType}(\varepsilon; A) = \tau$, if and only if $\varepsilon \vdash_A A : \tau$ is derivable.

Proof. Both parts can be easily proved by induction on the structure of A . □

6. Examples

To simplify the derivations, types are omitted. The first example type-checks the above imperative term of Example 2. The second example deals with structures and normalized-types. The third example evaluates a more complicated imperative term, and the last example shows static and dynamic description of a simple functional fixed-point. When no ambiguity will arise we use the following syntactic sugar for multiple assignments:

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

$$\begin{aligned}
\text{iType}(A; \Gamma) &\triangleq \text{match } A \text{ with} \\
&X/a \quad \Rightarrow \tau \text{ if } X/a:\tau \in \Gamma \\
&A_1, A_2 \quad \Rightarrow \text{iType}(A_1; \Gamma) \wedge \text{iType}(A_2; \Gamma) \\
&\quad \text{if } \text{iType}(P; \Gamma, \Delta) \neq \text{false} \neq \text{iType}(A_1; \Gamma, \Delta) \\
&P \rightarrow_{\Delta} A_1 \Rightarrow \text{iType}(P; \Gamma, \Delta) \rightarrow \text{iType}(A_1; \Gamma, \Delta) \\
&\quad \text{if } \text{iType}(P; \Gamma, \Delta) \neq \text{false} \neq \text{iType}(A_1; \Gamma, \Delta) \\
&A_1 A_2 \quad \Rightarrow \tau_2 \\
&\quad \text{if } \text{iType}(A_1; \Gamma) = \tau_1 \rightarrow \tau_2 \text{ and } \text{iType}(A_2; \Gamma) = \tau_1 \\
&- \quad \Rightarrow \text{false} \\
\text{iTCheck}(A; \Gamma; \tau) &\triangleq \text{if } \text{iType}(A; \Gamma) = \tau \text{ then true else false}
\end{aligned}$$

Fig. 11. The Algorithms iType and iTCheck

$$\begin{array}{c}
\Gamma, \Delta \vdash_{\bar{A}} Y : \text{b ref} \\
\hline
\Gamma, \Delta \vdash_{\bar{A}} f : (\text{b ref} \wedge \text{b ref}) \rightarrow \text{b} \quad \Gamma, \Delta \vdash_{\bar{A}} !Y : \text{b} \\
\Gamma, \Delta \vdash_{\bar{A}} X, Y : \text{b ref} \wedge \text{b ref} \quad \Gamma, \Delta \vdash_{\bar{A}} X : \text{b ref} \quad \Gamma, \Delta \vdash_{\bar{A}} f : (\text{b ref} \wedge \text{b ref}) \rightarrow \text{b} \\
\hline
\Gamma, \Delta \vdash_{\bar{A}} (f(X, Y)) : \text{b} \quad \Gamma, \Delta \vdash_{\bar{A}} X := !Y : \text{b} \quad \Gamma, \Delta \vdash_{\bar{A}} \text{ref } 3, \text{ref } 4 : \text{b ref} \wedge \text{b ref} \\
\hline
\Gamma \vdash_{\bar{A}} ((f(X, Y)) \rightarrow_{\Delta} (3 \rightarrow_{\emptyset} X := !Y) !X) : \text{b} \rightarrow \text{b} \quad \Gamma \vdash_{\bar{A}} (f(\text{ref } 3, \text{ref } 4)) : \text{b} \\
\hline
\Gamma \vdash_{\bar{A}} ((f(X, Y)) \rightarrow_{\Delta} (3 \rightarrow_{\emptyset} X := !Y) !X) (f(\text{ref } 3, \text{ref } 4)) : \text{b}
\end{array}$$

Fig. 12. Type-checking of Example 2

$$\begin{array}{c}
\Gamma, \Delta \vdash_{\bar{A}} X : \text{b ref} \quad \Gamma, \Delta \vdash_{\bar{A}} X : \text{b ref} \quad \Gamma, \Delta \vdash_{\bar{A}} X : \text{b ref} \\
\Gamma, \Delta \vdash_{\bar{A}} Y : \text{b ref} \quad \Gamma, \Delta \vdash_{\bar{A}} !Y : \text{b} \quad \Gamma, \Delta \vdash_{\bar{A}} Y : \text{b ref} \\
\hline
\Gamma, \Delta \vdash_{\bar{A}} X, !Y : \text{b ref} \wedge \text{b ref} \quad \Gamma, \Delta \vdash_{\bar{A}} X := !Y : \text{b} \quad \Gamma, \Delta \vdash_{\bar{A}} X, Y : \text{b ref} \wedge \text{b ref} \quad \Gamma, \Delta \vdash_{\bar{A}} Y : \text{b ref} \\
\hline
\Gamma \vdash_{\bar{A}} (X, Y) \rightarrow_{\Delta} X := !Y : (\text{b ref} \wedge \text{b ref}) \rightarrow \text{b} \quad \Gamma \vdash_{\bar{A}} (X, Y) \rightarrow_{\Delta} Y : (\text{b ref} \wedge \text{b ref}) \rightarrow \text{b} \\
\hline
\Gamma \vdash_{\bar{A}} ((X, Y) \rightarrow_{\Delta} X := !Y, (X, Y) \rightarrow_{\Delta} !Y) : (\text{b ref} \wedge \text{b ref}) \rightarrow \text{b} \wedge (\text{b ref} \wedge \text{b ref}) \rightarrow \text{b} \\
\text{arr}((\text{b ref} \wedge \text{b ref}) \rightarrow \text{b} \wedge (\text{b ref} \wedge \text{b ref}) \rightarrow \text{b}) \equiv (\text{b ref} \wedge \text{b ref}) \rightarrow (\text{b} \wedge \text{b}) \\
\Gamma \vdash_{\bar{A}} (\text{ref } a, \text{ref } b) : \text{b ref} \wedge \text{b ref} \\
\hline
\Gamma \vdash_{\bar{A}} ((X, Y) \rightarrow_{\Delta} X := !Y, (X, Y) \rightarrow_{\Delta} !Y) (\text{ref } a, \text{ref } b) : \text{b} \wedge \text{b}
\end{array}$$

Fig. 13. Type-checking of $((X, Y) \rightarrow_{\Delta} X := !Y, (X, Y) \rightarrow_{\Delta} Y) (a, b)$

6.1. Basic Examples

Example 3 (Type Checking). The type checking of the Example 2 is presented in Figure 12, where

$\Gamma \equiv \text{b:ok}, 3:\text{b}, 4:\text{b}, f:(\text{b ref} \wedge \text{b ref}) \rightarrow \text{b}$, and $\Delta \equiv X:\text{b ref}, Y:\text{b ref}$.

$$\begin{aligned}
\text{nnf1} &\triangleq \\
&\left(\begin{array}{ll}
\mathbf{p} & \rightarrow \mathbf{p}, \\
(\text{not } (\text{not } X)) & \rightarrow \text{!(SELF } X), \\
(\text{not } (\text{or } (X, Y))) & \rightarrow (\text{and } (\text{!(SELF } (\text{not } X))), \text{!(SELF } (\text{not } Y))), \\
(\text{not } (\text{and } (X, Y))) & \rightarrow (\text{or } (\text{!(SELF } (\text{not } X))), \text{!(SELF } (\text{not } Y))), \\
(\text{and } (X, Y)) & \rightarrow (\text{and } \text{!(SELF } X), \text{!(SELF } Y)), \\
(\text{or } (X, Y)) & \rightarrow (\text{or } (\text{!(SELF } X), \text{!(SELF } Y)))
\end{array} \right) \\
\text{nnf1} &\triangleq \\
&\left(\begin{array}{ll}
\mathbf{p} & \rightarrow \mathbf{p}, \\
(\text{not } (B_1, \text{ref } (\text{not } (B_2, X)))) & \rightarrow \text{!(SELF } (\text{!}X)), \\
(\text{not } (B_1, \text{ref } (\text{or } (B_2, X, Y)))) & \rightarrow (\text{and } (\text{ref false}, \\
& \text{!(SELF } (\text{ref } (\text{not } (\text{ref false}, X))), \\
& \text{!(SELF } (\text{ref } (\text{not } (\text{ref false}, Y))))), \\
(\text{not } (B_1, \text{ref } (\text{and } (B_2, X, Y)))) & \rightarrow (\text{or } (\text{ref false}, \\
& \text{!(SELF } (\text{ref } (\text{not } (\text{ref false}, X))), \\
& \text{!(SELF } (\text{ref } (\text{not } (\text{ref false}, Y))))), \\
(\text{and } (B, X, Y)) & \rightarrow \text{if } (\text{neg ref } B) \text{ then} \\
& (B, X, Y) := (\text{true}, \text{!(SELF } (\text{!}X)), \text{!(SELF } (\text{!}Y))) \\
& \text{else } (\text{and } (B, X, Y)), \\
(\text{or } (B, X, Y)) & \rightarrow \text{if } (\text{neg ref } B) \text{ then} \\
& (B, X, Y) := (\text{true}, \text{!(SELF } (\text{!}X)), \text{!(SELF } (\text{!}Y))) \\
& \text{else } (\text{or } (B, X, Y))
\end{array} \right)
\end{aligned}$$

Fig. 14. Imperative Encoding with(out) Sharing

Example 4 (Structures and Normalized-types). Let $\Gamma \equiv b:\text{ok}, a:b, b:b$ and $\Delta \equiv X:b \text{ ref}, Y:b \text{ ref}$. A derivation for

$$((X, Y) \rightarrow_{\Delta} X := !Y, (X, Y) \rightarrow_{\Delta} Y) (a, b)$$

is shown in Figure 13.

6.2. More Tricky Examples

Example 5 (Negation Normal Form).

This function (computing a negation normal form) is used in implementing *decision procedures*, present in almost all model checkers. The processed input is an implication-

free language of formulas with generating grammar:

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

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

(Imperative, I) this encoding uses a variable SELF which contains a pointer to the recursive code: here the recursion is achieved directly via pointer dereferencing, assignment and classical imperative fixed-point in order to implement recursion. Given the constant dummy, the function nnf1 is defined as in Figure 14: and the imperative encoding is the following:

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

(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:

$$\begin{aligned} \text{bool} &::= \text{true} \mid \text{false} \\ \text{flag} &::= \text{bool ref} \\ \psi &::= \text{ref } \phi \\ \phi &::= p \mid (\text{and}(\text{flag}, \psi, \psi)) \mid (\text{or}(\text{flag}, \psi, \psi)) \mid (\text{not}(\text{flag}, \psi)) \end{aligned}$$

Given the constant dummy, the function nnf2 is defined as in Figure 14 and the imperative encoding is the following:

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

(Typing The Imperative Encodings) If \mathbf{b} is the type of formulas ϕ , and $\mathbf{b} \text{ ref}$ is the type of the shared formulas ψ , and $\overset{6}{\wedge}\tau \triangleq \underbrace{\tau \wedge \dots \wedge \tau}_n$, and $\tau_1 \triangleq \mathbf{b} \rightarrow \mathbf{b}$, and $\tau_2 \triangleq \mathbf{b} \text{ ref} \rightarrow$

$\mathbf{b} \text{ ref}$, then it is easy to verify that the following judgments are derivable (we let $\Gamma_1 \triangleq \text{dummy}: \overset{6}{\wedge}\tau_1, \text{SELF}: \overset{6}{\wedge}\tau_1 \text{ ref}$, and $\Gamma_2 \triangleq \text{dummy}: \overset{6}{\wedge}\tau_2, \text{SELF}: \overset{6}{\wedge}\tau_2 \text{ ref}$):

$$(I) \quad \Gamma_1, X: \overset{6}{\wedge}\tau_1, \text{NNF}: \overset{6}{\wedge}\tau_1 \vdash \text{NNF}(\phi) : \overset{6}{\wedge}\mathbf{b}$$

$$(IS) \quad \Gamma_2, X: \overset{6}{\wedge}\tau_2, \text{NNF}: \overset{6}{\wedge}\tau_2 \vdash \text{NNF}(\psi) : \overset{6}{\wedge}\mathbf{b} \text{ ref}$$

Example 6 (Simple First-order Fixed-point [CLW04]). The type systems of iRho do not obey 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 $\Gamma \equiv$

$$\begin{array}{c}
\frac{\Gamma, \Delta \vdash_{\Delta} \text{fix} : (\mathbf{b} \rightarrow \mathbf{b}) \rightarrow \mathbf{b} \quad \Gamma, \Delta \vdash_{\Delta} X : \mathbf{b} \rightarrow \mathbf{b}}{\Gamma, \Delta \vdash_{\Delta} X : \mathbf{b} \rightarrow \mathbf{b}} \quad \frac{\Gamma, \Delta \vdash_{\Delta} X : \mathbf{b} \rightarrow \mathbf{b} \quad \Gamma, \Delta \vdash_{\Delta} (\text{fix } X) : \mathbf{b}}{\Gamma, \Delta \vdash_{\Delta} X (\text{fix } X) : \mathbf{b}} \quad \frac{\Gamma, \Delta \vdash_{\Delta} \text{fix} : (\mathbf{b} \rightarrow \mathbf{b}) \rightarrow \mathbf{b}}{\Gamma, \Delta \vdash_{\Delta} \Omega : \mathbf{b} \rightarrow \mathbf{b}} \\
\frac{\Gamma, \Delta \vdash_{\Delta} (\text{fix } X) : \mathbf{b} \quad \Gamma, \Delta \vdash_{\Delta} X (\text{fix } X) : \mathbf{b}}{\Gamma \vdash_{\Delta} \Omega : \mathbf{b} \rightarrow \mathbf{b}} \quad \frac{\Gamma, \Delta \vdash_{\Delta} (\text{fix } \Omega) : \mathbf{b}}{\Gamma \vdash_{\Delta} \Omega (\text{fix } \Omega) : \mathbf{b}} \\
\frac{\infty}{\vdots} \\
\frac{}{\emptyset \vdash \Omega (\text{fix } \Omega) \Downarrow_{\text{val}} \text{segmentation fault}}
\end{array}$$

Fig. 15. One Fixed-point

$\text{fix} : (\mathbf{b} \rightarrow \mathbf{b}) \rightarrow \mathbf{b}$, and $\Delta \equiv X : \mathbf{b} \rightarrow \mathbf{b}$. A derivation for $\Omega \triangleq (\text{fix } X) \rightarrow_{\Delta} (X (\text{fix } X))$ is shown in Figure 15. It is easy to verify that our interpreter diverges on this term.

Remark 3 (On let rec and Fixed-points). Fixed-points and `let rec` definitions are introduced using the well-known result of Nax Paul Mendler [MPC86, Men87]. 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 involved algebraic constructor `fix` (see Example 6) 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 [Gim98]), 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 to notice the “in-place update tricks” outlined by Guy Cousineau *et al.* [CCM87], and the more recent techniques due by Gérard Boudol and Pascal Zimmer [BZ02], and by Tom Hirschowitz *et al.* [HLW03], or the Peter Landin’s classical trick [Lan64].

7. Formalization in Coq

In the previous sections, we have given a mathematical presentation of `iRho` 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. Here we will just briefly discuss the most interesting aspects of this development.

```

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
| FunType : type -> type -> type
| ProdType : type -> type -> type
| RefType  : type -> type.
(* Types *)
Definition operator := boperator * type.
Inductive pattern   : Set := P0pe     : operator -> (list pattern) -> pattern
| PVar    : var -> type -> pattern
| PCons   : pattern -> pattern -> pattern
| PRef    : pattern -> pattern.
(* Patterns *)
Definition patterns := (list pattern).
Inductive expr      : Set := Ope      : operator -> expr
| 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.
(* Expressions *)
Inductive value     : Set := V0pe    : operator -> (list value) -> value
| Loc      : loc -> value
| Pair     : value -> value -> value
| Closure  : pattern -> expr -> env -> value
| Wrong    : pattern -> value -> expr -> env -> value.
(* Values *)

```

Fig. 16. Semantics Domains in Coq

The encoding of $iRho$ in Coq rephrases naturally the previous sections. Adequacy of the Coq encoding with respect to the mathematical presentation is proved by pen and paper.

A well-known problem we have to deal with is the encoding of the \rightarrow -binder. Binders are known to be difficult to encode in proof assistants; our encoding was essentially based on *closures*, *i.e.*, pairs $\langle \text{pattern abstraction} \cdot \text{environment} \rangle$. 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.

7.1. Syntactic and Semantics Structures

The signature of the encoding of $iRho$ is therefore presented in Figure 16. We comment on the most interesting choices:

- An *ad hoc* type `var` is introduced for variables; the only terms which can inhabit `var` are the variables in the logical framework. Thus, α -equivalence on terms is immediately inherited from the metalanguage, together with induction and recursion principles;
- 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;

```

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. 17. Sketch of Natural Imperative Semantics in Coq

- (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 also by an inductive datatype.

7.2. 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 17. The web-appendix [LS05] 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 on the most interesting choices:

- (`evalApp`) This rule is an “ASCII-clone” of the $(\text{Red}\cdot\text{Appl}_v)$ natural semantic rule;

```

Inductive TypeCheckPattern : envt -> pattern -> envt -> type -> Prop :=(* Type-check for patt. *)
...
| 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 *)
...
| tcClosure : (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 := (* Coh. env-type via coh. 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 := (* Coh. 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 := (* Coh. env-type-store *)
[e:env][s:store][E:envt][S:storet] ((AbstractEnv S e E) /\ (AbstractStore S s S)).

```

Fig. 18. Sketch of Type-checking Rules in Coq

- (`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;
- (`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 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`).

7.3. Type System

The encoding of the type system is rather straightforward, again as a consequence of our DIMPRO pattern. The encoding is comprised of 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 18.

- (`tcPopCons`), (`tcApp`) Those rules encode the type-checking rule for patterns (`Patt·Algr`), 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_p (see Subsection 4.1) 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 Section 5) bridges the gap from mathematical presentation to the encoding;
- (`tcClosure`) This rule faithfully encodes rule (`Value·Clos`) (see Subsection 4.1);
- (`FixAbstract`) This definition is crucial to establish a coherence relation between (un)typed environments and (un)typed stores.

7.4. 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 Section 5 for the the full metatheory, and to [LS05] for its complete mechanical counterparts.

Theorem 10 (Coq’s 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) (* 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)
  (match P v s e e1) -> (e2:env)                                     (* algo pattern-matching *)
  (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 11 (Coq’s Compile-Time Galleria).

```

1 Lemma NormalizeFunType_deterministic: (t,t1:type)                    (* arr-function is deterministic *)
  (NormalizeFunType t t1) -> (t2:type)
  (NormalizeFunType t t2) -> (t1=t2).

```

```

2 Lemma TypeCheckPattern_deterministic: (E:envt)(P:pattern)(E1:envt)(t1:type)
  (TypeCheckPattern E P E1 t1) -> (E2:envt)(t2:type) (* algo type-check pattern *)
  (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_match: (P:pattern)(v:value)(s:store)(e,e':env)
  (match P v s e e') -> (E:envt)(S:storet) (* SR for open expr match*)
  (FixAbstract e s E S) -> (E1:envt)(t1:type)
  (TypeCheckPattern E P E1 t1) -> (TypeOf S v t1) -> (AbstractEnv S e' E1).
5 Lemma open_subject_reduction: (A:expr)(e:env)(s:store)(v:value)(s2:store)
  (eval A e s v s2) -> (E:envt)(S:storet) (* SR for open expr eval *)
  (FixAbstract e s E S) -> (t:type)
  (TypeCheckExpr E A t) -> (EX S2:storet | ((Coherent s S s2 S2) /\ (TypeOf S2 v t))).
6 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, and types are isomorphic in “mathematics” and in `Coq`, the adequacy result comes directly as a matter of fact. To resume, we label with a “ \surd ” all theorems proved by the proof assistant `Coq`.

(Determinism) ^{\surd} If $\sigma \cdot \rho \vdash A \Downarrow_{\text{val}} A'_v \cdot \sigma'$, and $\sigma \cdot \rho \vdash A \Downarrow_{\text{val}} A''_v \cdot \sigma''$, then $A'_v \equiv A''_v$, and $\sigma' \equiv \sigma''$;

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

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

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

(Type-soundness) If $\emptyset \vdash_A A : \tau$, then $\emptyset \cdot \emptyset \vdash A \Downarrow_{\text{call}} A_v$, for some A_v ;

(Type-checking) It is decidable if there exists a type τ such that $\emptyset \vdash_A A : \tau$;

(Type-reconstruction) It is decidable if, for a given τ , is it true that $\emptyset \vdash_A A : \tau$.

8. Conclusions, Related, Future

In this paper, we have presented a formal development of the theory of `iRho`, a typed rewriting-based calculus featuring term-rewriting, pattern-matching on imperative terms, structures, functions, and side-effects. We mix rewriting (for rule-based languages), with functions (for functional languages), structures (for logic-like languages) and safe imperative structures, all “glued” by a pattern-matching algorithm that takes into account the imperative features. 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 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`.

During development we often had the feeling that the mathematical design was driven both by the machine assisted certification and by the software implementation, and that the feedback between those three phases (usually considered distinct) was crucial in order to make safe software and safe theory.

The lesson learned with `iRho` was that the hand of the math’s designer must be in strict

contact with the hand of the software’s designer, which, in turn, must be in strict contact with the hand of the proof’s certifier. Our recipe probably suggests a new schema, or “pattern”, in the sense of “*The Gang of Four*” [GHJT94], for design-implement-certify safe software. This could be subject of future work. Our software interpreter 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* [Com05] (from EAL5 to EAL7), or level five of the *Capability Maturity Model*, *CMM*. We schedule in our agenda to “formalize” our novel pattern DIMPRO, in the folklore of “design pattern”, hoping that it would be useful to the community developing safe software for crucial applications.

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

- RhoStratego [Str03] is an implementation of an early version of the uRho [CK01], written in the strategic language Stratego [Str05]. The implementation tests strategic programming with higher-order functional programming;
- Rogue [SS05] is another implementation of a dialect of the uRho [CK01]: 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 [SBD02] infrastructure;
- JRho [FM02] is a Java prototype of uRho [CK01], using the TOM pattern-matching compiler [MRV03].

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, we identify the following.

- To add an exception handling mechanism, following the lines of [CKL02]; this would leads to modify static and dynamic semantics.
- To add 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.
- To 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.
- To analyze, perhaps using abstract interpretation or static analysis techniques, the possibility to statically catch some pattern-matching failures.
- To enhance our matching algorithm with residuation and narrowing, in the style of the functional-logic programming language Curry by Michael Hanus [Han97].
- To add some *ad hoc* XML primitives to iRho.
- To 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.

- To conceive, following the “design pattern” jargon, the pattern DIMPRO.
- To 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 [BZ02].

Acknowledgment. The authors would like to thank all the members of the Protheo Team in Nancy for their comments and interactions on Rewriting-calculus. Liquori visited the University of Sussex, Brighton, and he would like to thank his hosts Matthew Hennessy and Vladimiro Sassone, and the whole Department of Informatics for the ideal working conditions they provided. Finally, the authors are warmly grateful to Matt Wall, Germain Faure, Daniel Dougherty and Steffen van Bakel for the careful reading of the paper, and all anonymous referees for their extremely useful comments and suggestions. This work is supported by the French grant CNRS *ACI Modulogic*.

References

- J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- Asf+Sdf Team. The Asf+Sdf Meta-Environment Home Page, 2005. <http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment>.
- G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems. In *Proc. of POPL*. The ACM Press, 2003.
- 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.
- G. Cousineau, P.-L. Curien, and M. Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8(2):173–202, 1987.
- Cristal, Foc-CNAM, Lemme, Mimosa, Miró, and Oasis. Concert: Compilateurs Certifiés, 2003. ARC INRIA 2003-2004, <http://www-sop.inria.fr/lemme/concert>.
- 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.
- H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.
- H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.
- H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In *Proc. of WRLA, ENTCS*, 2002.
- H. Cirstea, L. Liquori, and B. Wack. Rho-calculus with Fixpoint: First-order system. In *Proc. of TYPES*. Springer-Verlag, 2004.
- Common Criteria Consortium. The Common Criteria Home Page, 2005. <http://www.commoncriteria.org>.
- Cristal Team. OCaml Home page, 2005. <http://www.ocaml.org/>.
- M. Felleisen and D. P. Friedman. A Syntactic Theory of Sequential State. *Theoretical Computer Science*, 69(3):243–287, 1989.
- G. Faure and P.E. Moreau. Jrho: a Java Implementation of the Rho Calculus, 2002. <http://elan.loria.fr/Soft/jrho-0.1.tar.gz>.
- A. Frisch. The Cduce Home Page, 2005. <http://www.cduce.org>.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides (The Gang of Four). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- E. Gimenez. Structural Recursive Definitions in Type Theory. In *Proc. of ICALP*, pages 397–408, 1998.

- J. Goguen. The OBJ Family Home Page, 2005. <http://www.cs.ucsd.edu/users/goguen/sys/obj.html>.
- M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of POPL*, pages 80–93. The ACM Press, 1997.
- 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.
- G. Huet. *Résolution d'équations dans les langages d'ordre 1,2, ..., ω* . Ph.d. thesis, Université de Paris 7 (France), 1976.
- S. P. Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003. also as a special issue in The Journal of Functional Programming, Volume 13, Number 1, 2003.
- G. Kahn. Natural Semantics. In *Proc. of STACS*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.
- J.W. Klop. *Combinatory Reduction Systems*, volume 127. CWI, 1980. PhD Thesis.
- R. Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series, North Holland, 1979.
- P. J. Landin. The Mechanical Evaluation of Expression. *The Computer Journal*, 6:308–320, 1964.
- X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. . Submitted, 2005.
- K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-Oriented Pattern Matching for Evolvable Distributed Systems. In *Proc. of OOPSLA*. The ACM Press, 2003.
- Logical Team. The Coq Home Page, 2005. <http://coq.inria.fr>.
- L. Liquori and B. Serpette. Web Appendix of this paper, 2005. <http://www-sop.inria.fr/mirho/Luigi.Liquori/iRho>.
- P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984.
- Maude Team. The Maude Home Page, 2005. <http://maude.cs.uiuc.edu/>.
- N. P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, USA, 1987.
- 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.
- Microsoft. The C# Home Page, 2005. <http://msdn.microsoft.com/vcsharp/>.
- R. Milner. CS 3 Language Semantics. Course notes, LFCS, University of Edinburgh, 1986-87.
- N. P. Mendler, P. Panangaden, and R. L. Constable. Infinite Objects in Type Theory. In *Proc. of LICS*, pages 249–255, 1986.
- P.E. Moreau, C. Ringeissen, and M. Vittek. The Tom Home Page, 2003. <http://tom.loria.fr/>.
- I. A. Mason and C. L. Talcott. References, Local Variables and Operational Reasoning. In *In Proc. of LICS*, pages 66–77, 1992.
- R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- G. D. Plotkin. A Structured Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.
- G. D. Plotkin. The Origins of Structural Operational Semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.
- Protheo Team. The Elan Home Page, 2005. <http://elan.loria.fr>.
- J. Rees (eds.) R. Kelsey, W. Clinger. Revised5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998. Also in ACM SIGPLAN Notices, Vol. 33, No. 9, 1998.

- D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. In *Theory and Practice of Object Systems*, 2(1):3–13, 1996.
- A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In *CAV*, 2002. System Description.
- M. Serrano. The Scheme Bigloo Home Page, 2005. <http://www.sop.inria.fr/mimoso/fp/bigloo/>.
- A. Stump and C. Schürmann. The Rogue Home Page, 2005. <http://www.cse.wustl.edu/~stump/rogue.html>.
- Stratego Team. The Rho Stratego Home Page, 2003. <http://www.stratego-language.org/twiki/bin/view/Stratego/RhoStratego>.
- Stratego Team. The Stratego Home Page, 2005. <http://www.stratego-language.org>.
- Sun. Java Technology, 2005. <http://java.sun.com/>.
- M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, LFCS, University of Edinburgh, 1987.
- A. van Deursen, J. Heering, and P. Klint. *Language Prototyping*. World Scientific, 1996.
- V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.
- Xduce Team. The Xduce Home Page, 2005. <http://xduce.sourceforge.net>.