iRho: An Imperative Rewriting-calculus

Luigi Liquori, Bernard Serpette

Under consideration for publication in Math. Struct. in Comp. Science

iRho: An Imperative Rewriting-calculus

Luigi Liquori

INRIA Sophia Antipolis, 2004 Route des Lucioles, FR-06902 France

Luigi.Liquori@inria.fr

Bernard Paul Serpette

INRIA Sophia Antipolis, 2004 Route des Lucioles, FR-06902 France

Bernard.Serpette@inria.fr

Received 26 January 2006; Revised 27 April 2007

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(Protheo Team 2005), Maude(Maude Team 2005), ASF+SDF(van Deursen, et al. 1996, Asf+Sdf Team 2005), OBJ*(Goguen 2005)) is a promising line of research unifying the logical paradigm with the functional paradigm.

Although rewriting-based languages are less popular than object-oriented languages such as Java(Sun 2005), C#(Microsoft 2005), 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 (Cirstea, et al. 2001a, Cirstea, et al. 2002, Cirstea, et al. 2004).

Pattern-matching has been widely used in functional and logic programming (ML(Milner,

et al. 1997, Cristal Team 2005), Haskell(Jones 2003), Scheme(R. Kelsey 1998), Curry(Hanus 1997), or Prolog(Kowalski 1979)); it is generally considered to be a convenient mechanism for expressing complex requirements about the function's argument, rather than a basis for an *ad hoc* paradigm of computation.

One of the 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 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 from 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 pair of two functions having the same pattern; when the pair 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 Cartesian-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) (Cirstea, et al. 2001b, Cirstea et al. 2004) integrates matching, rewriting, and functions in a uniform way; its abstraction mechanism is based on rewrite rule formation: in a term of the form $P \twoheadrightarrow A$, one abstracts over all the free variables of 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 $P \twoheadrightarrow A$ 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 to build 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 ML∪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 (like Ω, see Example 6). 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

[†] "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	$X \twoheadrightarrow Y \twoheadrightarrow (\mathsf{cons}\ X\ Y)$	$\lambda X.\,\lambda Y.\,\lambda Z.\,Z\ X\ Y$
car	$(\mathsf{cons}\ X\ Y) \twoheadrightarrow X$	$\lambda Z.\,Z\ (\lambda X.\,\lambda Y.\,X)$
cdr	$(\mathsf{cons}\ X\ Y) \twoheadrightarrow Y$	$\lambda Z.\,Z\ (\lambda X.\,\lambda Y.\,Y)$

Table 1. *Accessors and Destructors in Rho/Lambda Calculi*

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.

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 (Felleisen & Friedman 1989). 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 (Milner et al. 1997) 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(Xduce Team 2005), CDUCE(Frisch 2005)) or combining object-orientation and patterns with semi-structured data (*e.g.*, HYDROJ(Lee, et al. 2003)[‡]).

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) (Martin-Löf 1984, Milner 1986-87, Plotkin 1981, Tofte 1987, Kahn 1987, Plotkin 2004). 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 forgot 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 performance.

The static and dynamic semantics are suitable to be specified mathematically, to be implemented with high-level programming languages, *e.g.* Bigloo (Serrano 2005) (of the Scheme family), and to be certified with a modern and semi-automatic proof assistant, *e.g.* Coq (Logical Team 2005).

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

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 influenced the design of the semantics. The continuous interplay of mathematics and manual (*i.e.*, pen and paper) *vs.* mechanical proofs, and prototype 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 (Cristal, et al. 2003, Leroy 2005).

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.

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/mascotte/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, 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 (Cirstea et al. 2004) 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*, *i.e.*, pairs <pattern abstraction · environment>, to denote functional values value of free-variables recorded in the environment. This representation will avoid the use of *meta-substitution* everywhere.

2.1. *Functional Syntax*

Notational Conventions. We use the meta-symbols $\twoheadrightarrow$ (function- and type-abstraction), and "," (structure operator), and the implicit @ (application operator). We assume that

$$
\begin{array}{lll}
\tau & ::= & \mathsf{b} \mid \tau \twoheadrightarrow \tau \mid \tau \wedge \tau & \text{Types} \\
\Delta & ::= & \emptyset \mid \Delta, X{:}\tau \mid \Delta, a{:}\tau & \text{Contexts} \\
P & ::= & X \mid a\,\overline{P} \mid P\,,P & \text{Patterns} \\
A & ::= & a \mid X \mid P \twoheadrightarrow_{\Delta} A \mid A\,A \mid A\,,A & \text{Terms}
\end{array}
$$

Fig. 1. Rho's Syntax

the application operator @ associates to the left, while the other operators associate to the right. The priority of @ is higher than that of $\twoheadrightarrow$ which is, in turn, of higher priority than ",".

The symbols $A, B, C, \ldots$ range over the set $\mathcal{T}_A$ of terms, the symbols $X, Y, Z, \ldots$, $\mathsf{SELF}, \ldots$ ranges over the set $\mathcal{X}$ of variables ($\mathcal{X} \subseteq \mathcal{T}_A$), the symbols $a, b, c, \ldots, \mathsf{cons}$, $\mathsf{true}, \mathsf{false}, \mathsf{not}, \mathsf{and}, \mathsf{or}, \mathsf{dummy}, \ldots$ range over a set $\mathcal{K}$ of term-constants ($\mathcal{K} \subseteq \mathcal{T}_A$). The symbol P ranges over the set $\mathcal{P}$ of pseudo-patterns, ($\mathcal{X} \subseteq \mathcal{P}$). The symbol τ ranges over the set $\mathcal{T}_\tau$ of types, the symbol b ranges over the set of type-constants, the symbols Γ, Δ ranges over contexts. The symbols $A_\mathsf{v}, B_\mathsf{v}, C_\mathsf{v}, \ldots$ range over the set $\mathcal{V}al$ of values. We sometimes write $\overline{A}$ 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 b denotes basic types, the arrow-type $\tau_1 \twoheadrightarrow \tau_2$ is the type of pattern abstractions $P \twoheadrightarrow_\Delta A$, with Δ containing the types of the free variables of P, 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 (Klop 1980)). To retain confluence, Vincent van Oostrom (van Oostrom 1990) 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 $(a\,X\,X)$), 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 patterns (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 develop-

ment of our interpreter, and of the current mechanical development of the metatheory underneath iRho and not to theoretical problems (see (Barthe, et al. 2003)).

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.* $\lambda X{:}\tau.A \simeq X \twoheadrightarrow_{X:\tau} A$; 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:

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

— *(Application)* allows one to apply a pattern abstraction $P \twoheadrightarrow_{\Delta} A$ 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, $(\mathsf{car}\,(\mathsf{car}\,(\mathsf{cons}\,(\mathsf{cons}\,\mathsf{a}\,\mathsf{b})\,\mathsf{c})))$ will reduce to a;

Observe that compared with "non-strategic" implementations of the Rewriting-calculus (Cirstea et al. 2001a, Cirstea et al. 2001b, Cirstea et al. 2002, Barthe et al. 2003), the delayed matching-constraint $[P \ll_{\Delta} A].B$, becomes now just syntactic sugar for $(P \twoheadrightarrow_{\Delta} A)\,B$ (omitted from the source language but still present in the set of output values).

Values and Environments. The set $\mathcal{V}al$ of *values*, and the set of environments $\mathcal{E}nv$ are defined below, where the last two forms are closures:

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

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

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

2.2. *The Rhosetta Stone*

The Rewriting-calculus is known to be a conservative extension of the Lambda-calculus (Cirstea et al. 2001a, Cirstea & Kirchner 2001). Nevertheless, it is typically presented using an infix notation, using as binder the meta-symbol "arrow" ($\twoheadrightarrow$), instead of the prefix notation using as a binder the meta-symbol "lambda" (λ) in conjunction with the meta-symbol "point" (.). 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

Context Syntax

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

Rewriting-like Syntax

$$A \quad ::= \quad P \twoheadrightarrow_\Delta A \mid \qquad A\,,B \qquad \mid \overbrace{((X_1\,,X_2) \twoheadrightarrow_\Delta X_{1,2})\,(A\,,B)}^{\mathsf{proj}} \mid \ldots$$

Lambda-like Syntax

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

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

Fig. 2. The Rhosetta (Functional) Stone

type. The rationale is:

$$\lambda X{:}\underbrace{X{:}\tau_1}_{\Delta}.A \quad \simeq \quad X \twoheadrightarrow_\Delta A \qquad \text{variables as patterns}$$

$$\lambda(f\,X\,Y){:}\underbrace{X{:}\tau_1, Y{:}\tau_2}_{\Delta}.A \quad \simeq \quad (f\,X\,Y) \twoheadrightarrow_\Delta A \quad \text{algebraic patterns}$$

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

$$\mathsf{Fv}(P \twoheadrightarrow_\Delta A) \quad \triangleq \quad \mathsf{Fv}(A) \setminus \mathsf{Fv}(P)$$

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.*

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

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

$$\mathsf{neg} \quad \triangleq \quad (\mathsf{true} \twoheadrightarrow \mathsf{false}\,, \mathsf{false} \twoheadrightarrow \mathsf{true})$$

$$\mathsf{if}\ \ A\ \ \mathsf{then}\, B\ \ \mathsf{else}\ \ C \quad \triangleq \quad (\mathsf{true} \twoheadrightarrow ((X \twoheadrightarrow B)\,\mathsf{dummy})\,, \mathsf{false} \twoheadrightarrow ((X \twoheadrightarrow C)\,\mathsf{dummy}))\,A$$

Observe that, because of the call-by-value strategy, the then and the else branches are wrapped in a dummy abstraction, and X is fresh in B and C, *i.e.*, $X \notin \mathsf{Fv}(B) \cup \mathsf{Fv}(C)$.

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.

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

$$\frac{}{\rho \vdash a \Downarrow_{\mathsf{val}} a}\ (\text{Red·Val}) \qquad \frac{X \in \mathsf{Dom}(\rho)}{\rho \vdash X \Downarrow_{\mathsf{val}} \rho(X)}\ (\text{Red·Var})$$

$$\frac{\rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \quad \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}}}{\rho \vdash A\,,B \Downarrow_{\mathsf{val}} A_{\mathsf{v}}\,,B_{\mathsf{v}}}\ (\text{Red·Struct})$$

$$\frac{}{\rho \vdash P \twoheadrightarrow_{\Delta} A \Downarrow_{\mathsf{val}} \langle P \twoheadrightarrow_{\Delta} A \centerdot \rho\rangle}\ (\text{Red·Fun})$$

$$\frac{\rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \quad \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \quad \vdash \langle A_{\mathsf{v}} \centerdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} C_{\mathsf{v}}}{\rho \vdash A\,B \Downarrow_{\mathsf{val}} C_{\mathsf{v}}}\ (\text{Red·Appl}_{\mathsf{v}})$$

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

$$\frac{}{\vdash \langle a\,\overline{A}_{\mathsf{v}} \centerdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} a\,\overline{A}_{\mathsf{v}}\,B_{\mathsf{v}}}\ (\text{Call·Algbr})$$

$$\frac{\vdash \langle A_{\mathsf{v}} \centerdot C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} D_{\mathsf{v}} \quad \vdash \langle B_{\mathsf{v}} \centerdot C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} E_{\mathsf{v}}}{\vdash \langle (A_{\mathsf{v}}\,,B_{\mathsf{v}}) \centerdot C_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} D_{\mathsf{v}}\,,E_{\mathsf{v}}}\ (\text{Call·Struct})$$

$$\frac{\rho \vdash \langle P \centerdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho' \quad \rho' \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}}}{\vdash \langle \langle P \twoheadrightarrow_{\Delta} A \centerdot \rho\rangle \centerdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} A_{\mathsf{v}}}\ (\text{Call·FunOk})$$

$$\frac{\nexists \rho'.\ \rho \vdash \langle P \centerdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho'}{\vdash \langle \langle P \twoheadrightarrow_{\Delta} A \centerdot \rho\rangle \centerdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{call}} \langle [P \ll_{\Delta} B_{\mathsf{v}}].A \centerdot \rho\rangle}\ (\text{Call·FunKo})$$

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

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

$$\frac{}{\rho \vdash \langle a \centerdot a\rangle \Downarrow_{\mathsf{match}} \rho}\ (\text{Match·Const})$$

$$\frac{\rho(X) = \bot}{\rho \vdash \langle X \centerdot A_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho[X \mapsto A_{\mathsf{v}}]}\ (\text{Match·VarNew}) \qquad \frac{\rho(X) = A_{\mathsf{v}}}{\rho \vdash \langle X \centerdot A_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho}\ (\text{Match·VarEq})$$

$$\frac{\rho_0 \vdash \langle A \centerdot A_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho_1 \quad \rho_1 \vdash \langle B \centerdot B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho_2}{\rho_0 \vdash \langle A \star B \centerdot A_{\mathsf{v}} \star B_{\mathsf{v}}\rangle \Downarrow_{\mathsf{match}} \rho_2}\ (\text{Match·Pair})$$

Fig. 3. Natural Functional Semantics

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. The semantics is defined via three judgments, of the shapes:

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

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, where the symbol $\star$ stands either for @ or ",". 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;
— (Red·Struct) This rule simple evaluates a structure to a structure-value;
— (Red·Fun) This rule evaluates a pattern abstraction to a closure;
— (Red·Appl$_{\mathsf{v}}$) 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 $\Downarrow_{\mathsf{call}}$ judgment;
— (Call·Algbr) This rule builds an algebraic-value under the shape of an application in weak head-normal form;
— (Call·Struct) This rule applies every element of the structure-value to the argument C_{v};
— (Call·FunOk) This rule first matches successfully P against B_{v}, and then evaluates the body of the pattern abstraction A in the new environment calculated by $\Downarrow_{\mathsf{match}}$;
— (Call·FunKo) This rule applies when the match of P against B_{v} fails: a failure-value is returned;
— (Call·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;
— (Match·Pair) Matching either an application or a structure (recall that $\star \in \{@\ ,\}$) 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 (Huet 1976) is implemented by a judgment $\rho \vdash \langle A \centerdot A_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho'$ 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 1 (On Failure-values and Exceptions). "Failure-values" $\langle [P \ll_{\Delta} A_{\mathsf{v}}].B \centerdot \rho \rangle$ denote failures occurring when we cannot find a correct substitution θ on the free variables of P such that $\theta(P) \equiv A_{\mathsf{v}}$; the environment ρ records the value of the free variables of B. Failure-values are obtained during the computation when a matching failure occurs.

They can in principle be discarded, or caught by an exception handler (see (Cirstea et al. 2002)) that can be implemented in the interpreter.

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 (Cirstea et al. 2004) for necessary extensions/enhancements of an operational semantics and for a suitable matching theory that automatically drops failure-values.

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 $\Downarrow_{\mathsf{val}}^{\mathsf{opt}}$ – resp. $\Downarrow_{\mathsf{val}}^{\mathsf{pex}}$ – resp. $\Downarrow_{\mathsf{val}}^{\mathsf{clean}}$, denote the optimistic – resp. pessimistic – resp. clean machine)

$$\frac{\vdots}{\emptyset \vdash (3 \twoheadrightarrow 3\,, 4 \twoheadrightarrow 4)\,4 \Downarrow_{\mathsf{val}}^{\mathsf{opt}} \langle [3 \ll 4].3 \,.\, \emptyset \rangle\,, 4}$$

$$\frac{\vdots}{\emptyset \vdash (3 \twoheadrightarrow 3\,, 4 \twoheadrightarrow 4)\,4 \Downarrow_{\mathsf{val}}^{\mathsf{pex}} \langle [3 \ll 4].3 \,.\, \emptyset \rangle}$$

$$\frac{\vdots}{\emptyset \vdash (3 \twoheadrightarrow 3\,, 4 \twoheadrightarrow 4)\,4 \Downarrow_{\mathsf{val}}^{\mathsf{clean}} 4}$$

We conclude with a simple example of two functional evaluations.

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

$$((a\,X) \twoheadrightarrow (3 \twoheadrightarrow 3)\,X)\,(a\,3) \quad \text{and} \quad ((a\,X) \twoheadrightarrow (3 \twoheadrightarrow 3)\,X)\,(a\,4)$$

and let

$$\begin{array}{rcl}
\odot & \equiv & \emptyset \vdash \langle ((a\,X) \twoheadrightarrow (3 \twoheadrightarrow 3)\,X) \,.\, \emptyset \rangle \Downarrow_{\mathsf{val}} \langle ((a\,X) \twoheadrightarrow (3 \twoheadrightarrow 3)\,X) \,.\, \emptyset \rangle \\
\triangle & \equiv & \emptyset \vdash (a\,3) \Downarrow_{\mathsf{val}} (a\,3) \\
\square & \equiv & \emptyset \vdash (a\,4) \Downarrow_{\mathsf{val}} (a\,4) \\
A_{\mathsf{v}} & \equiv & \langle [3 \ll 4].3 \,.\, \rho \rangle \qquad \rho_0 \triangleq [X \mapsto 3] \qquad \rho_1 \triangleq [X \mapsto 4]
\end{array}$$

The deduction trees are shown in Figure 4.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \langle a \centerdot a\rangle \Downarrow_{\mathsf{match}} \emptyset} \quad \dfrac{}{\emptyset \vdash \langle X \centerdot 3\rangle \Downarrow_{\mathsf{match}} \rho_0}}{\emptyset \vdash \langle (a\,X) \centerdot (a\,3)\rangle \Downarrow_{\mathsf{match}} \rho_0} \quad \dfrac{\dfrac{\dfrac{}{\rho_0 \vdash \langle 3 \centerdot 3\rangle \Downarrow_{\mathsf{match}} \rho_0} \quad \dfrac{}{\rho_0 \vdash 3 \Downarrow_{\mathsf{val}} 3}}{\vdash \langle\langle (3 \multimap 3) \centerdot \rho_0\rangle \centerdot 3\rangle \Downarrow_{\mathsf{call}} 3} \quad \dfrac{}{\rho_0 \vdash X \Downarrow_{\mathsf{val}} 3} \quad \dfrac{}{\rho_0 \vdash (3 \multimap 3) \Downarrow_{\mathsf{val}} \langle (3 \multimap 3) \centerdot \rho_0\rangle}}{\rho_0 \vdash (3 \multimap 3)\,X \Downarrow_{\mathsf{val}} 3}}{\vdash \langle\langle ((a\,X) \multimap (3 \multimap 3)\,X) \centerdot \emptyset\rangle \centerdot (a\,3)\rangle \Downarrow_{\mathsf{call}} 3}\ \odot\ \triangle}{\emptyset \vdash ((a\,X) \multimap (3 \multimap 3)\,X)\,(a\,3) \Downarrow_{\mathsf{val}} 3}$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \langle a \centerdot a\rangle \Downarrow_{\mathsf{match}} \emptyset} \quad \dfrac{}{\emptyset \vdash \langle X \centerdot 4\rangle \Downarrow_{\mathsf{match}} \rho_1}}{\emptyset \vdash \langle (a\,X) \centerdot (a\,4)\rangle \Downarrow_{\mathsf{match}} \rho_1} \quad \dfrac{\dfrac{\dfrac{\nexists \rho_2.\ \ \rho_1 \vdash \langle 3 \centerdot 4\rangle \Downarrow_{\mathsf{match}} \rho_2}{\vdash \langle\langle (3 \multimap 3) \centerdot \rho\rangle \centerdot 4\rangle \Downarrow_{\mathsf{call}} A_{\mathsf{v}}} \quad \dfrac{}{\rho_1 \vdash X \Downarrow_{\mathsf{val}} 4} \quad \dfrac{}{\rho_1 \vdash (3 \multimap 3) \Downarrow_{\mathsf{val}} \langle (3 \multimap 3) \centerdot \rho_1\rangle}}{\rho_1 \vdash (3 \multimap 3)\,X \Downarrow_{\mathsf{val}} A_{\mathsf{v}}}}{\vdash \langle\langle ((a\,X) \multimap (3 \multimap 3)\,X) \centerdot \emptyset\rangle \centerdot (a\,4)\rangle \Downarrow_{\mathsf{call}} A_{\mathsf{v}}}\ \odot\ \square}{\emptyset \vdash ((a\,X) \multimap (3 \multimap 3)\,X)\,(a\,4) \Downarrow_{\mathsf{val}} A_{\mathsf{v}}}$$

Fig. 4. Natural Deduction of $((a\,X) \multimap (3 \multimap 3)\,X)\,(a\,3)$ and $((a\,X) \multimap (3 \multimap 3)\,X)\,(a\,4)$

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

Fig. 5. iRho's Syntax

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 (Tofte 1987, Felleisen & Friedman 1989).

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 $\mathsf{ref}\, A$ is a referencing term (the-location-of); if A is a term of type τ, then $\mathsf{ref}\, A$ is a pointer to A of type $\tau\ \mathsf{ref}$;
— *(Assignment-terms)* The term $A := B$ is an assignment operator, which returns as result the value obtained by evaluating B (as in *e.g.* SmallTalk); other languages, *e.g.* OCaml, return a special value () of type unit.

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):

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

This is a nice feature with respect to functional core calculi, mixing imperative and functional features, as in Caml.
Sequencing can be also defined in iRho as follows (types are omitted):

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

Issues related to garbage collection are out of the scope of this paper: new locations created during reduction, via referencing ($\mathsf{ref}\, A$), will remain in the store forever. In principle, classical techniques of Ian Mason and Carolyn Talcott, and Greg Morrisset *et al.*(Mason & Talcott 1992, Morrisett, et al. 1995) could be applied to iRho.

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

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

Stores are partial functions from the set $\mathcal{L}$ of locations to the set of values *i.e.*, $\sigma \in \mathcal{S}tore \simeq [\mathcal{L}oc \Rightarrow \mathcal{V}al]_{\perp}$; we denote the extension of a store by $\sigma[\iota \mapsto A_{\mathsf{v}}]$ with the following meaning:

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

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:

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

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 $\Downarrow_{\mathsf{val}}$ and $\Downarrow_{\mathsf{call}}$, a store σ is given as input, and a (possibly modified) store σ' is returned as output. In the $\Downarrow_{\mathsf{match}}$ rule, a store σ is needed as input since our matching

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 6. Natural Imperative Semantics

algorithm allows to match a referencing terms $\mathsf{ref}\, A$ to a pointer-variable, such as in:

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

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

— (Red·{v, Appl$_{\mathsf{v}}$, Var, Fun, Struct}) All those rules essentially behave as in the functional case, with the exception that the store parameter is propagated over the judgments in the premises and in the conclusion;

— (Red·Ref) This rule first reduces A into a value, and then stores it into a "fresh" location ι;
— (Red·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 3 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:

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

This pattern-matching rule allows to consider the dereferencing term $!A$ as 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:

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

We conclude with a simple example of an imperative evaluation.

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

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

with $\sigma_0 \triangleq [\iota_0 \mapsto 3][\iota_1 \mapsto 4]$, and $\sigma_1 \triangleq \sigma_0[\iota_0 \mapsto 4]$, and $\rho_0 \triangleq [X \mapsto \iota_0][Y \mapsto \iota_1]$. The deduction tree is shown in Figure 7.

4. The Type System

In this section, we present a type system for iRho. As usual, we employ 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 probably 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 (Cirstea et al. 2001b, Cirstea et al. 2002, Barthe et al. 2003, Cirstea et al. 2004), structures *can now have different types*, *i.e.*, thanks to the following typing

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

$$\dfrac{\begin{array}{l} \sigma_0 \centerdot \emptyset \vdash \langle\langle ((a\,(X\,,Y)) \twoheadrightarrow (3 \twoheadrightarrow X := !Y)\,!X) \centerdot \emptyset \rangle \centerdot (a\,(\iota_0\,,\iota_1)) \rangle \Downarrow_{\mathsf{call}} 4 \centerdot \sigma_1 \\ \emptyset \centerdot \emptyset \vdash (a\,(\mathsf{ref}\,3\,, \mathsf{ref}\,4)) \Downarrow_{\mathsf{val}} (a\,(\iota_0\,,\iota_1)) \centerdot \sigma_0 \\ \emptyset \centerdot \emptyset \vdash ((a\,(X\,,Y)) \twoheadrightarrow (3 \twoheadrightarrow X := !Y)\,!X) \Downarrow_{\mathsf{val}} \langle ((a\,(X\,,Y)) \twoheadrightarrow (3 \twoheadrightarrow X := !Y)\,!X) \centerdot \emptyset \rangle \centerdot \emptyset \end{array}}{\emptyset \centerdot \emptyset \vdash ((a\,(X\,,Y)) \twoheadrightarrow (3 \twoheadrightarrow X := !Y)\,!X)\,(a\,(\mathsf{ref}\,3\,, \mathsf{ref}\,4)) \Downarrow_{\mathsf{val}} 4 \centerdot \sigma_1}$$

Fig. 7. Natural Deduction of $((a\,(X\,,Y)) \twoheadrightarrow (3 \twoheadrightarrow X := !Y)\,!X)\,(a\,(\mathsf{ref}\,3\,, \mathsf{ref}\,4))$

rule which is new with respect to the previous typed formulations of the Rewriting-calculus

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

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 (Cirstea et al. 2004), 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 (Barthe et al. 2003), is that here the elements of the structure are not forced to have the same type.

The type system $\vdash_{\mathsf{A}}$ is *algorithmic*: the type rules are *deterministic* and suggest two *decision 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 argu-

Pattern Rules

$$\frac{\Gamma_1, \alpha{:}\tau, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, \alpha{:}\tau, \Gamma_2 \vdash_{\mathsf{P}} \alpha : \tau}\ (\text{Patt·Start}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} P_1 : \tau_1 \quad \Gamma \vdash_{\mathsf{P}} P_2 : \tau_2}{\Gamma \vdash_{\mathsf{P}} P_1 \,, P_2 : \tau_1 \wedge \tau_2}\ (\text{Patt·Struct})$$

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

Term Rules

$$\frac{\Gamma_1, \alpha{:}\tau, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, \alpha{:}\tau, \Gamma_2 \vdash_{\mathsf{A}} \alpha : \tau}\ (\text{Term·Start}) \qquad \frac{\Gamma \vdash_{\mathsf{A}} A : \tau}{\Gamma \vdash_{\mathsf{A}} \mathsf{ref}\, A : \tau\ \mathsf{ref}}\ (\text{Term·Ref})$$

$$\frac{\Gamma \vdash_{\mathsf{A}} A : \tau\ \mathsf{ref} \quad \Gamma \vdash_{\mathsf{A}} B : \tau}{\Gamma \vdash_{\mathsf{A}} A := B : \tau}\ (\text{Term·Assign}) \qquad \frac{\Gamma \vdash_{\mathsf{A}} A : \tau_1 \quad \Gamma \vdash_{\mathsf{A}} B : \tau_2}{\Gamma \vdash_{\mathsf{A}} A \,, B : \tau_1 \wedge \tau_2}\ (\text{Term·Struct})$$

$$\frac{\begin{array}{l}\mathsf{arr}(\tau_1) \equiv \tau_2 \rightarrowtriangle \tau_3 \\ \Gamma \vdash_{\mathsf{A}} A : \tau_1 \quad \Gamma \vdash_{\mathsf{A}} B : \tau_2\end{array}}{\Gamma \vdash_{\mathsf{A}} A\,B : \tau_3}\ (\text{Term·Appl}) \qquad \frac{\begin{array}{l}\mathsf{Dom}(\Delta) = \mathsf{Fv}(P) \\ \Gamma, \Delta \vdash_{\mathsf{P}} P : \tau_1 \quad \Gamma, \Delta \vdash_{\mathsf{A}} A : \tau_2\end{array}}{\Gamma \vdash_{\mathsf{A}} P \rightarrowtriangle_{\Delta} A : \tau_1 \rightarrowtriangle \tau_2}\ (\text{Term·Abs})$$

Fig. 8. Well Formed Pattern and Terms

ment, 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 $(\mathsf{b}_1 \rightarrowtriangle \mathsf{b}_2) \wedge (\mathsf{b}_1 \rightarrowtriangle \mathsf{b}_3)$) to an argument of type b_1, we would obtain as result a structure-value of type $\mathsf{b}_2 \wedge \mathsf{b}_3$. To capture this behavior (which is a direct consequence of dispatching application into structures), we need the partial function arr on types, which transforms a product-type into a function-type:

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

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

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

$$\Gamma \vdash_{\Gamma} ok \qquad \Gamma \vdash_{\tau} \tau : ok \qquad \Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau$$
$$\Gamma \vdash_{\rho} \rho : \Gamma' \qquad \Gamma \vdash_{\sigma} \sigma : \Gamma' \qquad \Gamma \vdash_{\mathsf{P}} P : \tau \qquad \Gamma \vdash_{\mathsf{A}} A : \tau$$

which denote well-typed contexts, types, values, environments, stores, patterns, and terms, respectively. 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 8. 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.

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

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·Algbr), (Term·Appl) Those rules deal with application. We discuss the application term-term, the pattern-pattern being similar. The application rule is the usual one can expect for an algorithmic version of a type system; note that before applying terms, we need to transform the type τ_1 of A into an arrow type, since it could happen that A is a structure containing more branches of the same domain type. The subject of the statement is $a\,\overline{P}\,P$, since $\overline{P}$ can be empty and, as usual, application associates to the left;
— (Term·Abs) In this rule we note that the context Δ is added to the premises, using the decidable function $\mathsf{Fv}(P)$; the context Γ gives types only for algebraic constants;
— (Term·Assign) This rule deals with assignment: the only possible choice is to assign to an expression A, of type τ ref , an object B of type τ;
— (Term·Ref) This rule says that, if an object A has type τ, then a pointer to this object, denoted by ref A, has type τ ref.

4.1. *Extra Typing Rules*

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

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

denoting well-formed contexts, types, values, environments, and stores. Those judgments are necessary when we encode iRho in the Logical Framework of Coq. The type rules of those five new judgments are really much more intuitive and they do not need any particular comment. It is worth noting that also rule (Value·Algbr) needs a transformation step for 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.

5. Metatheory

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

1 Natural Semantics for $\Downarrow_{\mathsf{val}}$ is deterministic;
2 Type-checking with $\vdash_{\mathsf{A}}$ is unique;
3 Subject-reduction holds (*i.e.*, types are preserved under reduction);

Context Type Rules

$$\frac{}{\emptyset \vdash_{\Gamma} ok}\ (\text{Ctx·Axiom}) \qquad \frac{\Gamma \vdash_{\Gamma} ok \quad \mathsf{b} \notin \mathsf{Dom}(\Gamma)}{\Gamma, \mathsf{b}{:}ok \vdash_{\Gamma} ok}\ (\text{Ctx·Type})$$

$$\frac{\chi \notin \mathsf{Dom}(\Gamma) \quad \Gamma \vdash_{\tau} \tau : ok \quad \Gamma \vdash_{\Gamma} ok}{\Gamma, \chi{:}\tau \vdash_{\Gamma} ok}\ (\text{Ctx·Var/Const})$$

Term Type Rules

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

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

Value Type Rules

$$\frac{\Gamma_1, a{:}\tau, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, a{:}\tau, \Gamma_2 \vdash_{\mathsf{v}} a : \tau}\ (\text{Value·Start}_1) \qquad \frac{\Gamma_1, \iota{:}\tau\ \mathsf{ref}\,, \Gamma_2 \vdash_{\Gamma} ok}{\Gamma_1, \iota{:}\tau\ \mathsf{ref}\,, \Gamma_2 \vdash_{\mathsf{v}} \iota : \tau\ \mathsf{ref}}\ (\text{Value·Start}_2)$$

$$\frac{\Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau_1 \quad \Gamma \vdash_{\mathsf{v}} B_{\mathsf{v}} : \tau_2}{\Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}}\,, B_{\mathsf{v}} : \tau_1 \wedge \tau_2}\ (\text{Value·Struct}) \qquad \frac{\begin{array}{l}\mathsf{arr}(\tau_1) \equiv \tau_2 \multimap \tau_3 \\ \Gamma \vdash_{\mathsf{v}} a\,\overline{A}_{\mathsf{v}} : \tau_1 \quad \Gamma \vdash_{\mathsf{v}} B_{\mathsf{v}} : \tau_2\end{array}}{\Gamma \vdash_{\mathsf{v}} a\,\overline{A}_{\mathsf{v}}\,B_{\mathsf{v}} : \tau_3}\ (\text{Value·Algbr})$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\rho} \rho : \Gamma' \\ \Gamma' \vdash_{\mathsf{p}} P \twoheadrightarrow_{\Delta} A : \tau_1 \multimap \tau_2\end{array}}{\Gamma \vdash_{\mathsf{v}} \langle P \twoheadrightarrow_{\Delta} A \bullet \rho \rangle : \tau_1 \multimap \tau_2}\ (\text{Value·Clos}) \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau_1 \quad \Gamma \vdash_{\rho} \rho : \Gamma' \\ \Gamma' \vdash_{\mathsf{p}} P \twoheadrightarrow_{\Delta} B : \tau_1 \multimap \tau_2\end{array}}{\Gamma \vdash_{\mathsf{v}} \langle\, [P \ll_{\Delta} A_{\mathsf{v}}].B \bullet \rho \rangle : \tau_2}\ (\text{Value·Fail})$$

Store Type Rules

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

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

$$\frac{\begin{array}{ll}\Gamma \vdash_{\sigma} \sigma : \Gamma' & \Gamma'[\iota{:}\tau] \vdash_{\Gamma} ok \\ \Gamma \vdash_{\mathsf{v}} \iota : \tau\ \mathsf{ref} & \Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau\end{array}}{\Gamma \vdash_{\sigma} \sigma[\iota \mapsto A_{\mathsf{v}}] : \Gamma'[\iota{:}\tau]}\ (\text{Store·Loc})$$

Environment Type Rules

$$\Gamma[X{:}\tau] \triangleq \begin{cases} \Gamma, X{:}\tau & \text{if } X \notin \mathsf{Dom}(\Gamma) \\ \Gamma & \text{if } X{:}\tau \in \Gamma \end{cases}$$

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

$$\frac{\begin{array}{ll}\Gamma \vdash_{\rho} \rho : \Gamma' & \Gamma'[X{:}\tau] \vdash_{\Gamma} ok \\ \Gamma \vdash_{\mathsf{A}} X : \tau & \Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau\end{array}}{\Gamma \vdash_{\rho} \rho[X \mapsto A_{\mathsf{v}}] : \Gamma'[X{:}\tau]}\ (\text{Env·Var})$$

Fig. 9. Extra Typing Rules

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 (Liquori & Serpette 2005). 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).

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

Then, we prove that our natural semantics is deterministic.

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

1 If $\sigma_1 \centerdot \rho \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \centerdot \sigma_2$, and $\sigma_1 \centerdot \rho \vdash A \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \centerdot \sigma_3$, then $A_{\mathsf{v}} \equiv B_{\mathsf{v}}$, and $\sigma_2 \equiv \sigma_3$;
2 If $\sigma_1 \vdash \langle A_{\mathsf{v}} \centerdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{call}} C_{\mathsf{v}} \centerdot \sigma_2$, and $\sigma_1 \vdash \langle A_{\mathsf{v}} \centerdot B_{\mathsf{v}} \rangle \Downarrow_{\mathsf{call}} D_{\mathsf{v}} \centerdot \sigma_3$, then $C_{\mathsf{v}} \equiv D_{\mathsf{v}}$, and $\sigma_2 \equiv \sigma_3$;
3 If $\sigma \centerdot \rho_1 \vdash \langle P \centerdot A_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho_2$, and $\sigma \centerdot \rho_1 \vdash \langle P \centerdot A_{\mathsf{v}} \rangle \Downarrow_{\mathsf{match}} \rho_3$, 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 (Liquori & Serpette 2005)).

Theorem 3 (Uniqueness of Typing). If $\Gamma \vdash_{\mathsf{A}} A : \tau_1$, and $\Gamma \vdash_{\mathsf{A}} A : \tau_2$, then $\tau_1 \equiv \tau_2$.

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

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

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 $\sigma_1 \centerdot \rho_1 \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \centerdot \sigma_2$, and $\Gamma_1 \vdash_{\mathsf{coh}} \sigma_1 \centerdot \rho_1$, and $\Gamma_1 \vdash_{\mathsf{A}} A : \tau$, then there exists Γ_2, which extend Γ_1, such that $\Gamma_2 \vdash_{\mathsf{coh}} \sigma_2 \centerdot \rho_1$, and $\Gamma_2 \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau$.

Proof. By Coq. □

The following result is crucial to state type soundness.

Theorem 6 (Subject-reduction for Closed Terms). If $\emptyset \centerdot \emptyset \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}} \centerdot \sigma$, and $\emptyset \vdash_{\mathsf{A}} A : \tau$, then there exists Γ, such that $\Gamma \vdash_{\mathsf{coh}} \sigma \centerdot \emptyset$, and $\Gamma \vdash_{\mathsf{v}} A_{\mathsf{v}} : \tau$.

Proof. By Coq. □

The reduction rules for the operational semantics given in Figure 6 readily suggest how an interpreter for iRho can be defined. Run-time errors for this interpreter would 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 two ways in which an evaluation may get "stuck"

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

($\Downarrow_{\mathsf{call}}$) This judgment gets stuck when we try to apply a location-value to a value (*e.g.* $\langle \iota \centerdot A_{\mathsf{v}} \rangle$), or if one premise in some judgment gets stuck;

($\Downarrow_{\mathsf{match}}$) This judgment gets stuck when we try to match a pattern against a value with a different (unmatchable) shape, *e.g.* $\langle (P\,Q) \centerdot (A\,, B) \rangle$, 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 $\emptyset \vdash_{\mathsf{A}} A : \tau$ is derivable, and let $\mathsf{C}[\cdot]$ be any iRho-context.

(Progress)

($\Downarrow_{\mathsf{val}}$) 1 if $A \equiv \mathsf{C}[X]$, then there exists σ_1, and ρ, such that $\sigma_1 \centerdot \rho \vdash X \Downarrow_{\mathsf{val}} \rho(X) \centerdot \sigma_2$, and $\rho(X) \neq \bot$;

2 if $A \equiv \mathsf{C}[B := C]$, then there exists σ_1, and ρ, such that $\sigma_1 \centerdot \rho \vdash B \Downarrow_{\mathsf{val}} \iota \centerdot \sigma_2$, and $\iota \in \mathsf{Dom}(\sigma_2)$;

($\Downarrow_{\mathsf{call}}$) If $A \equiv \mathsf{C}[B\,C]$, then there exists σ_1, and ρ, such that $\sigma_1 \centerdot \rho \vdash B \Downarrow_{\mathsf{val}} B_{\mathsf{v}} \centerdot \sigma_2$, and $B_{\mathsf{v}} \not\equiv \iota$;

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

(Type-soundness)

If $\emptyset \vdash_{\mathsf{A}} A : \tau$, then $\emptyset \centerdot \emptyset \vdash A \Downarrow_{\mathsf{val}} A_{\mathsf{v}}$, for some A_{v}.

Proof.

(Progress)

($\Downarrow_{\mathsf{val}}$)If A has a variable X as a sub-expression, then, by well-typedness of the sub-expressions, environment and store, $\rho(X) \neq \bot$. 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 $\iota \centerdot \sigma_2$ and $\iota \in \mathsf{Dom}(\sigma_2)$;

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

$$
\begin{array}{lll}
\mathsf{iType}(A;\Gamma) & \triangleq & \texttt{match } A \text{ with} \\
 & & \quad X/a \Rightarrow \tau \text{ if } X/a{:}\tau \in \Gamma \\
 & & \quad A_1\,,A_2 \Rightarrow \mathsf{iType}(A_1;\Gamma) \wedge \mathsf{iType}(A_1;\Gamma) \\
 & & \qquad \text{if } \mathsf{iType}(P;\Gamma,\Delta) \neq \mathsf{false} \neq \mathsf{iType}(A_1;\Gamma,\Delta) \\
 & & \quad P \twoheadrightarrow_\Delta A_1 \Rightarrow \mathsf{iType}(P;\Gamma,\Delta) \twoheadrightarrow \mathsf{iType}(A_1;\Gamma,\Delta) \\
 & & \qquad \text{if } \mathsf{iType}(P;\Gamma,\Delta) \neq \mathsf{false} \neq \mathsf{iType}(A_1;\Gamma,\Delta) \\
 & & \quad A_1\,A_2 \Rightarrow \tau_2 \\
 & & \qquad \text{if } \mathsf{iType}(A_1;\Gamma) = \tau_1 \twoheadrightarrow \tau_2 \text{ and } \mathsf{iType}(A_2;\Gamma) = \tau_1 \\
 & & \quad \_ \Rightarrow \mathsf{false} \\
\mathsf{iTCheck}(A;\Gamma;\tau) & \triangleq & \text{if } \mathsf{iType}(A;\Gamma) = \tau \text{ then true else false}
\end{array}
$$

Fig. 10. The Algorithms iType and iTCheck

($\Downarrow_{\mathsf{match}}$)Again, if A has an application $B\,C$ as a sub-expression, then by well-typedness of the sub-expressions, environment and store, the evaluation of B leads to a closure-value $\langle P \twoheadrightarrow_\Delta D \,.\, \rho_1\rangle$, 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 $\emptyset \vdash_{\mathsf{A}} A : \tau$;
2 **(Type-reconstruction)** It is decidable if, for a given τ, is it true that $\emptyset \vdash_{\mathsf{A}} A : \tau$.

Proof.

1 Figure 10 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 10). By uniqueness of typing, $\Gamma \vdash_{\mathsf{A}} A : \tau$ 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, $\mathsf{iType}(\varepsilon; A) = \tau$, if and only if $\varepsilon \vdash_{\mathsf{A}} A : \tau$ is derivable.

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

6. Examples

To simplify the derivations, some type 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:

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

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

Fig. 11. Type-checking of Example 2

6.1. *Basic Examples*

Example 3 (Type Checking). The type checking of the Example 2 is presented in Figure 11, where $\Gamma \equiv \mathsf{b}{:}ok, 3{:}\mathsf{b}, 4{:}\mathsf{b}, a{:}(\mathsf{b\ ref} \wedge \mathsf{b\ ref}) \multimap \mathsf{b}$, and $\Delta \equiv X{:}\mathsf{b\ ref}, Y{:}\mathsf{b\ ref}$.

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

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

is shown in Figure 12.

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:

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

We present two imperative encodings: in the first, the function is shared via a pointer and recursion is achieved via dereferencing. In the second, formulas are shared too with

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

$$\Gamma \vdash_{\mathsf{A}} ((X\,,Y) \multimap_{\Delta} X := !Y\ ,\ (X\,,Y) \multimap_{\Delta} !Y)\,(\mathsf{ref}\,a\,,\mathsf{ref}\,b) : \mathsf{b} \wedge \mathsf{b}$$

Fig. 12. Type-checking of $((X\,,Y) \multimap_{\Delta} X := !Y\,,(X\,,Y) \multimap_{\Delta} Y)\,(a\,,b)$

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 13: and the imperative encoding is the following:

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

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

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

Given the constant dummy, the function nnf2 is defined as in Figure 13 and the imperative encoding is the following:

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

(Typing The Imperative Encodings) If b is the type of formulas ϕ, and b ref is the type of the shared formulas ψ, and $\overset{n}{\wedge}\tau \triangleq \underbrace{\tau \wedge \ldots \wedge \tau}_{n}$, and $\tau_1 \triangleq \mathsf{b} \multimap \mathsf{b}$, and $\tau_2 \triangleq \mathsf{b\ ref} \multimap \mathsf{b\ ref}$, then it is easy to verify that the following judgments are derivable (we let

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

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

Fig. 13. Imperative Encoding with(out) Sharing

$\Gamma_1 \triangleq \mathsf{dummy}\colon \overset{6}{\wedge}\tau_1, \mathsf{SELF}\colon \overset{6}{\wedge}\tau_1\ \mathsf{ref}$, and $\Gamma_2 \triangleq \mathsf{dummy}\colon \overset{6}{\wedge}\tau_2, \mathsf{SELF}\colon \overset{6}{\wedge}\tau_2\ \mathsf{ref}$):

$$(\mathsf{I}) \qquad \Gamma_1, X\colon \overset{6}{\wedge}\tau_1, \mathsf{NNF}\colon \overset{6}{\wedge}\tau_1 \ \vdash\ \mathsf{NNF}(\phi) : \overset{6}{\wedge}\mathsf{b}$$

$$(\mathsf{IS}) \qquad \Gamma_2, X\colon \overset{6}{\wedge}\tau_2, \mathsf{NNF}\colon \overset{6}{\wedge}\tau_2 \ \vdash\ \mathsf{NNF}(\psi) : \overset{6}{\wedge}\mathsf{b}\ \mathsf{ref}$$

Example 6 (Simple First-order Fixed-point (Cirstea et al. 2004)). 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 $\Gamma \equiv \mathsf{fix}\colon(\mathsf{b} \twoheadrightarrow \mathsf{b}) \twoheadrightarrow \mathsf{b}$, and $\Delta \equiv X\colon\mathsf{b} \twoheadrightarrow \mathsf{b}$. A derivation for $\Omega \triangleq (\mathsf{fix}\,X) \twoheadrightarrow_\Delta (X\,(\mathsf{fix}\,X))$ is shown in Figure 14. 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 (Mendler, et al. 1986, Mendler

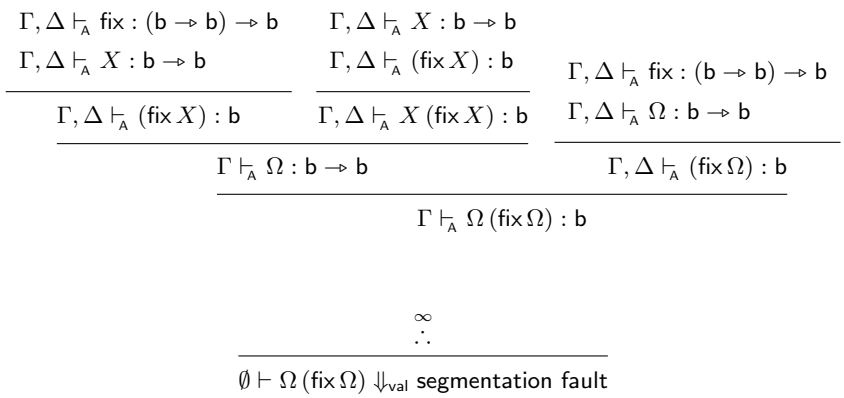

Fig. 14. One Fixed-point

1987). 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 (Gimenez 1998)), 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.* (Cousineau, et al. 1987), and the more recent techniques due by Gérard Boudol and Pascal Zimmer (Boudol & Zimmer 2002), and by Tom Hirschowitz *et al.* (Hirschowitz, et al. 2003), or the Peter Landin's classical trick (Landin 1964).

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.

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 $\multimap$-binder. Binders

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

Fig. 15. Semantics Domains in Coq

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

7.1. *Syntactic and Semantics Structures*

The signature of the encoding of iRho is therefore presented in Figure 15. 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;
— (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);

```
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. 16. Sketch of Natural Imperative Semantics in Coq

— 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 16. The web-appendix (Liquori & Serpette 2005) 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). We comment on the most interesting choices:

— (`evalApp`) This rule is an "ASCII-clone" of the (Red·Appl$_v$) natural semantic rule;

— (`evalRef`) This rule encodes the semantic rule (Red·Ref). Observe the use of the `sbrk` function, which extends a given store (partial function), via the (here omitted) auxiliary function `extend_store`;

— (`(evalDeref)`) This rule first verifies that the required location belongs to the store-

```
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 *)
...
 | toClosure : (S:storet)(e:env)(E:envt)
               (AbstractEnv S e E) -> (P:pattern)(B:expr)(t1,t2:type)
               (TypeCheckExpr E (Abs P B) (FunType t1 t2)) ->
               (TypeOf S (Closure P B e) (FunType t1 t2))
with AbstractEnv : storet -> env -> envt -> Prop :=    (* 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. 17. Sketch of Type-checking Rules in Coq

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. 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 17.

— (`tcPOpCons`), (`tcApp`) Those rules encode the type-checking rule for patterns (Patt·Algbr), and terms (Term·Appl), respectively: they make use of the function `NormalizeFunType` which behaves as a coercion to a functional type;
— (`tcRef`), (`tcDeref`), (`tcAssign`) Those rules encode the type rules (Type·Ref), (Type·Deref), and (Type·Assign): they just mimic the corresponding rules in natural semantics;
— (`aeExtend`) This rule is the counterpart of the judgment $\vdash_\rho$ (see 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;
— (`toClosure`) 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 (Liquori & Serpette 2005) 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 $\sigma \centerdot \rho \vdash A \Downarrow_{\mathsf{val}} A'_{\mathsf{v}} \centerdot \sigma'$, and $\sigma \centerdot \rho \vdash A \Downarrow_{\mathsf{val}} A''_{\mathsf{v}} \centerdot \sigma''$, then $A'_{\mathsf{v}} \equiv A''_{\mathsf{v}}$, and $\sigma' \equiv \sigma''$;

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

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

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

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

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

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

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.

We experimented a "pattern"[§] (in the sense of *"The Gang of Four"* (Gamma, et al. 1994)) called DIMPRO, a.k.a. Design-IMplement-PROve, to design safe software, which

[§] *"A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts..." (Riehle & Züllighoven 1996).*

respects *in toto* its mathematical and functional specifications. This pattern is familiar to anyone who has knowledge of proof-assistants.

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 (Abrial 1996)). All three phases have the same status, and each influences the other.

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 software interpreter has been 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* (Common Criteria Consortium 2005) (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 (Stratego Team 2003) is an implementation of an early version of the uRho (Cirstea & Kirchner 2001), written in the strategic language Stratego(Stratego Team 2005). The implementation tests strategic programming with higher-order functional programming;

— Rogue(Stump & Schürmann 2005) is another implementation of a dialect of the uRho (Cirstea & Kirchner 2001): 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 (Stump, et al. 2002) infrastructure;

— JRho (Faure & Moreau 2002) is a Java prototype of uRho (Cirstea & Kirchner 2001), using the TOM pattern-matching compiler (Moreau, et al. 2003).

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 (Cirstea et al. 2002); this would lead 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 (Hanus 1997).
— 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 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 (Boudol & Zimmer 2002).

Acknowledgment. The authors would like to thank all the members of the Protheo Team in Nancy for their invaluable comments and interactions on Rewriting-calculus. The authors also warmly thank Barry Jay for the fruitful X-discussions, with $\mathsf{X} \in \{@, \mathsf{LondonPub}, \mathsf{WSRewriting}\}$. 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* and by the AEOLUS FET Global Computing Proactive, *Algorithmic Principles for Building Efficient Overlay Computers.*

References

J. R. Abrial (1996). *The B-Book: Assigning Programs to Meanings.* Cambridge University Press.

Asf+Sdf Team (2005). 'The Asf+Sdf Meta-Environment Home Page'. `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment`.

G. Barthe, et al. (2003). 'Pure Pattern Type Systems'. In *Proc. of POPL*. The ACM Press.

G. Boudol & P. Zimmer (2002). 'Recursion in the Call-by-Value Lambda-Calculus'. In *In Proc. of FICS*, Note Series NS-02-2. BRICS.

H. Cirstea & C. Kirchner (2001). 'The Rewriting Calculus — Part I *and* II'. *Logic Journal of the Interest Group in Pure and Applied Logics* **9**(3):427–498.

H. Cirstea, et al. (2001a). 'Matching Power'. In *Proc. of RTA*, vol. 2051 of *LNCS*, pp. 77–92. Springer-Verlag.

H. Cirstea, et al. (2001b). 'The Rho Cube'. In *Proc. of FOSSACS*, vol. 2030 of *LNCS*, pp. 166–180.

H. Cirstea, et al. (2002). 'Rewriting Calculus with(out) Types'. In *Proc. of WRLA*, ENTCS.

H. Cirstea, et al. (2004). 'Rho-calculus with Fixpoint: First-order system'. In *Proc. of TYPES*. Springer-Verlag.

Common Criteria Consortium (2005). 'The Common Criteria Home Page'. `http://www.commoncriteria.org`.

G. Cousineau, et al. (1987). 'The Categorical Abstract Machine'. *Science of Computer Programming* **8**(2):173–202.

Cristal, et al. (2003). 'Concert: Compilateurs Certifiés'. ARC INRIA 2003-2004, `http://www-sop.inria.fr/lemme/concert`.

Cristal Team (2005). 'OCaml Home page'. `http://www.ocaml.org/`.

G. Faure & P. Moreau (2002). 'Jrho: a Java Implementation of the Rho Calculus'. `http://elan.loria.fr/Soft/jrho-0.1.tar.gz`.

M. Felleisen & D. P. Friedman (1989). 'A Syntactic Theory of Sequential State'. *Theoretical Computer Science* **69**(3):243–287.

A. Frisch (2005). 'The Cduce Home Page'. `http://www.cduce.org`.

E. Gamma, et al. (1994). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley.

E. Gimenez (1998). 'Structural Recursive Definitions in Type Theory'. In *Proc. of ICALP*, pp. 397–408.

J. Goguen (2005). 'The OBJ Family Home Page'. `http://www.cs.ucsd.edu/users/goguen/sys/obj.html`.

M. Hanus (1997). 'A Unified Computation Model for Functional and Logic Programming'. In *Proc. of POPL*, pp. 80–93. The ACM Press.

T. Hirschowitz, et al. (2003). 'Compilation of Extended Recursion in Call-by-Value Functional Languages'. In *In Proc. of PPDP*. The ACM Press.

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

S. P. Jones (2003). *Haskell 98 Language and Libraries*. Cambridge University Press. also as a special issue in The Journal of Functional Programming, Volume 13, Number 1, 2003.

G. Kahn (1987). 'Natural Semantics'. In *Proc. of STACS*, vol. 247 of *LNCS*, pp. 22–39. Springer-Verlag.

J. Klop (1980). *Combinatory Reduction Systems*, vol. 127. CWI. Ph.D. Thesis.

R. Kowalski (1979). *Logic for Problem Solving*. Artificial Intelligence Series, North Holland.

P. J. Landin (1964). 'The Mechanical Evaluation of Expression'. *The Computer Journal* **6**:308–320.

K. Lee, et al. (2003). 'HydroJ: Object-Oriented Pattern Matching for Evolvable Distributed Systems'. In *Proc. of OOPSLA*. The ACM Press.

X. Leroy (2005). 'Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. '. Submitted.

L. Liquori & B. Serpette (2005). 'Web Appendix of this paper'. `http://www-sop.inria.fr/mascotte/Luigi.Liquori/iRho`.

Logical Team (2005). 'The Coq Home Page'. `http://coq.inria.fr`.

P. Martin-Löf (1984). *Intuitionistic Type Theory*, vol. 1 of *Studies in Proof Theory*. Bibliopolis, Naples.

I. A. Mason & C. L. Talcott (1992). 'References, Local Variables and Operational Reasoning'. In *In Proc. of LICS*, pp. 66–77.

Maude Team (2005). 'The Maude Home Page'. `http://maude.cs.uiuc.edu/`.

N. P. Mendler (1987). *Inductive Definition in Type Theory*. Ph.D. thesis, Cornell University, Ithaca, USA.

N. P. Mendler, et al. (1986). 'Infinite Objects in Type Theory'. In *Proc. of LICS*, pp. 249–255.

Microsoft (2005). 'The C# Home Page'. `http://msdn.microsoft.com/vcsharp/`.

R. Milner (1986-87). 'CS 3 Language Semantics'. Course notes, LFCS, University of Edinburgh.

R. Milner, et al. (1997). *The Definition of Standard ML (Revised)*. The MIT Press.

P. Moreau, et al. (2003). 'The Tom Home Page'. `http://tom.loria.fr/`.

J. G. Morrisett, et al. (1995). 'Abstract Models of Memory Management'. In *In Proc. of FPCA*, pp. 66–77. The ACM Press.

G. D. Plotkin (1981). 'A Structured Approach to Operational Semantics'. Tech. Rep. DAIMI FN-19, Aarhus University.

G. D. Plotkin (2004). 'The Origins of Structural Operational Semantics.'. *J. Log. Algebr. Program.* **60-61**:3–15.

Protheo Team (2005). 'The Elan Home Page'. `http://elan.loria.fr`.

J. R. e. R. Kelsey, W. Clinger (1998). 'Revised5 Report on the Algorithmic Language Scheme'. *Higher-Order and Symbolic Computation* **11**(1). Also in ACM SIGPLAN Notices, Vol. 33, No. 9, 1998.

D. Riehle & H. Züllighoven (1996). 'Understanding and Using Patterns in Software Development'. *In Theory and Practice of Object Systems* **2**(1):3–13.

M. Serrano (2005). 'The Scheme Bigloo Home Page'. `http://www.sop.inria.fr/mimosa/fp/bigloo/`.

Stratego Team (2003). 'The Rho Stratego Home Page'. `http://www.stratego-language.org/twiki/bin/view/Stratego/RhoStratego`.

Stratego Team (2005). 'The Stratego Home Page'. `http://www.stratego-language.org`.

A. Stump, et al. (2002). 'CVC: A Cooperating Validity Checker'. In *CAV*. System Description.

A. Stump & C. Schürmann (2005). 'The Rogue Home Page'. `http://www.cse.wustl.edu/~stump/rogue.html`.

Sun (2005). 'Java Technology'. `http://java.sun.com/`.

M. Tofte (1987). *Operational Semantics and Polymorphic Type Inference*. Ph.D. thesis, LFCS, University of Edinburgh.

A. van Deursen, et al. (1996). *Language Prototyping*. World Scientific.

V. van Oostrom (1990). 'Lambda Calculus with Patterns'. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam.

Xduce Team (2005). 'The Xduce Home Page'. `http://xduce.sourceforge.net`.