

Specifying Imperative ML-like Programs Using Dynamic Logic ^{*}

Séverine Maingaud¹, Vincent Balat¹, Richard Bubel²,
Reiner Hähnle², and Alexandre Miquel³

¹ Laboratoire *Preuves, Programmes et Systèmes*
CNRS and Université Paris Diderot – Paris 7

² Department of Computer Science and Engineering
Chalmers University, Gothenburg

³ ENS Lyon, Université de Lyon,
LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA)

Abstract. We present a logical system suited for specification and verification of imperative ML programs. The specification language combines dynamic logic (DL), explicit state updates and second-order functional arithmetic. Its proof system is based on a Gentzen-style sequent calculus (adapted to modal logic) with facilities for symbolic evaluation. We illustrate the system with some example, and give a full Kripke-style semantics in order to prove its correctness.

Key words: ML, dynamic logic, program specification, program verification, KeY, AF2

1 Introduction

We present a logical system suited for specification and verification of imperative ML programs. Verification systems for functional programming languages have been traditionally investigated in the context of higher-order logical frameworks (e.g., Coq, Isabelle, HOL, ACL2, VeriFun, Elf), where structural induction is the central proof paradigm. To employ dynamic logic and symbolic execution constitutes a new departure which is motivated by the presence of reference types whose treatment is well understood in Hoare-style program logics. In our paper we show that dynamic logic is a suitable framework also for ML with references. Our specification language combines a generalisation of Hoare logics called dynamic logic (DL), explicit state updates, and second-order functional arithmetic (AF2) [9]. Its proof system is based on a Gentzen-style sequent calculus (adapted to modal logic) with facilities for symbolic evaluation.

Related Work. State-of-the-art verification systems based on dynamic logic are KIV [1] and KeY [3]. The idea of using updates to represent state changes in a

^{*} This work has partially been supported by the EU COST Action IC0701: Formal Verification of Object-Oriented Software.

dynamic logic setting originated also from KeY. We depart from KeY’s program logic, however, in two main aspects: (i) we use second-order dynamic logic to be able to deal with a functional language, thus bridging the gap between DL and AF2; (ii) memory allocation extends the domain of the store in contrast to the constant-domain assumption employed by KeY.

The proof assistant PAF! [2] is also a verification system for ML programs based on AF2 with symbolic evaluation, but it does not support verification of imperative ML. The verification tool WHY [5] for first-order imperative programs is a verification condition generator based on Dijkstra’s weakest precondition calculus. WHY is being adapted to higher-order programs [8] through the integration of effect polymorphism to previous work [10] on Hoare logics for call-by-value functional programs without states. In this setting the generated verification conditions are passed on to automatic theorem provers such as SMT solvers or to interactive proof systems like Coq or PVS. The Ynot system [4] uses Coq both as a theorem prover and as an imperative functional language thanks to a monadic formulation of separation logic.

Structure. The paper is structured as follows: Section 2 introduces basic concepts of dynamic logic and updates from a higher level perspective. Syntax and operational semantics of the supported imperative ML fragment are formally introduced in Section 3. Based on this Section 4 defines the logical and proof system. Section 5 provides the semantics of the logical systems. In Section 6 we present a detailed example illustrating the specification and verification of a program in our logical system. Section 7 concludes the paper with some final thoughts and future plans.

2 Dynamic Logic

Dynamic logic [6] can be seen as a class of modal logics suited for reasoning about imperative programs. Like Hoare logic it uses a specification language where the current program state is implicit. States are explicit only in the semantics, where they play the role of worlds of a Kripke frame, in the sense of modal logic.

The central idea is to introduce for each program p a separate modality (read ‘box p ’) $[p]$ whose accessibility relation in a Kripke frame corresponds exactly to the operational semantics of p : the formula $[p]B$ holds in a state s if the formula B holds in all states reachable by any execution of the program p . If p is deterministic (which we assume from now on) then there is at most one final state. Under this semantics the formula

$$A \rightarrow [p] B \tag{1}$$

expresses *partial correctness* of program p with respect to precondition A and postcondition B . Whenever A and B are first-order formulas (1) corresponds to the *Hoare triple* $\{A\} p \{B\}$ [7]. In contrast to Hoare logic, however, in dynamic logic modal operators with programs inside and propositional connectives can be arbitrarily nested which makes dynamic logic more expressive than Hoare logic.

In addition to the partial correctness modality there is a dual operator (read “diamond p ”) $\langle p \rangle$ defined as $\langle p \rangle B \leftrightarrow \neg[p] \neg B$. Using the diamond operator we can express *total correctness* of program p in dynamic logic: $A \rightarrow \langle p \rangle B$.

In contrast to higher-order logics, imperative programs are first-class citizens in dynamic logic and not modelled by (inductively defined) formulas. In consequence, the syntax and semantics of the underlying programs is fixed and one must define a specific dynamic logic for a given programming language. One advantage is that the programming language semantics is defined at the meta-level (as a property of Kripke frames) and needs not to be defined on the formula level. Likewise, programs can have any concrete syntax and need not follow a formula structure. This leads to a low formalization overhead and good readability when constructing proof obligations for program correctness which in turn is important for (i) handling complex target languages, (ii) achieving a high degree of automation, (iii) usability in interactive proofs.

Proof systems for dynamic logic do not proceed mainly via induction over the syntactic structure of programs, but by decomposition of programs and recording of intermediate (symbolic) states. If the application of decomposition rules follows the evaluation strategy of an interpreter of the underlying programming language, then this amounts to *symbolic execution*. The program-free part of dynamic logic is usually a standard first-order logic with sorts and interpreted symbols for arithmetic, arrays, etc. There is relatively strong automated reasoning support available for such logics. Two state-of-art software verification systems (KeY [3] and KIV [1]) with a very high degree of automation are based on dynamic logic and symbolic execution.

Functional Programs with References. Our programming language is an untyped version of *imperative ML* (IML). Imperative ML adds references (locations) with mutable content to the functional world. References are pointers to a fixed memory location. The value stored at that particular location can be accessed and changed by programs. Let r denote a reference: The IML fragment

$$r := 3; !r$$

consists of two (sequentially connected) expressions: the first expression $r := 3$ changes the content stored at the memory location referred to by r to 3; the second expression $!r$ looks up and evaluates to the value stored in r . Expressions composed by the semicolon operator are evaluated from left-to-right. The resulting value is the one of the last expression; the above IML fragment evaluates always to 3. More details are in Sect. 3.

When extending a functional language with references (and thus with a notion of state) one is has to deal with phenomena such as side-effects, aliasing, or sensitivity to evaluation order for functional correctness. For instance, the IML λ -expression

$$f := \lambda x, y. ((x := !x + 2; !x) + (y := !y * 5; !y))$$

(applied to arguments) has not only global visible side-effects (contents of references passed to x , y changed), but is also affected by aliasing and the evaluation

order: let r, s denote *distinct* references with *equal* content (say 3), then $(!f) r s$ evaluates to 20 and $(!f) r r$ evaluates to 30 (under left-to-right evaluation).

The specification language (logic) for IML programs needs not only to model the additional concepts faithfully, but must also ensure that the properties to be specified are actually expressible. For example, in a pure functional setting the formula $\forall x.(f x \leq g x)$ specifies that function g is an upper approximation of the program (function) f , but more thought is required in presence of side-effects where executing f might influence the evaluation of g .

Dynamic Logic. We sketch the basic concepts and ideas behind dynamic logic. A rigorous introduction of second-order dynamic logic for IML program is given in Sect. 4. Signature and syntax of dynamic logic are defined on top of an existing non-modal base logic (e.g., first-order or second-order logic). An important feature of first-order modal logics is the distinction between rigid and non-rigid function/predicate symbols. Rigid symbols are interpreted independent of a state, while the interpretation of non-rigid symbols is state-dependent. For instance, the interpretation of the IML dereferencing operator $!$ must obviously be state-dependent.

The inductive definition of DL syntax is fairly standard. Any formula of the underlying non-modal base logic is also a formula of its dynamic logic variant. Modalities are added to the syntax as follows: let p be an IML program, ϕ denote a DL formula then $[p]\phi$ and $\langle p \rangle \phi$ are DL formulas. An important restriction is that ML programs occurring as logical terms (i.e., outside a modality) must be state-independent and pure (side-effect free).

States in dynamic logic are not represented by an explicit datastructure passed as an extra argument to functions (predicates), but live solely on the semantical level. Formulas and terms are evaluated relative to a *Kripke structure* \mathcal{K} . Besides the elementary data domain and an interpretation for the rigid symbols the Kripke structure fixes also a set of states St , giving meaning to non-rigid symbols such as $!$, and a state transition relation $\tau : \Pi_{IML} \times St \times St$ that defines the semantics of IML programs. The cardinality of $\tau(\pi, s) = \{s' \mid \tau(\pi, s, s')\}$ is at most 1, because IML is deterministic.

Example 1. The DL formula

$$[\text{if } a > b \text{ then } max := a \text{ else } max := b] (!max \text{ as } x) x \geq a$$

specifies that if the program inside the first box modality terminates then in the final state the value stored at max is at least as large as the value of a . The construct “as x ”, introduced in Sect. 4, is a binder to recover the returned value.

Proof systems for DL typically use a sequent style calculus and follow the symbolic evaluation paradigm by realising a symbolic interpreter. The rule that handles assignment is often one of the most tricky ones and crucial for the efficiency of the verification process. Even for simple imperative languages the standard assignment rule requires renaming of locations and, in presence of aliasing, the introduction of several case distinctions. The update mechanism sketched in the following provides an elegant way to deal with this.

Update Mechanism. Influenced by *abstract state machines* and *generalized substitutions* (B method), the KeY verification system [3] introduced updates as a syntactical notion to represent symbolic state changes in dynamic logic.

An (elementary) *update* is an expression of the form $location := value$. By sequential composition of updates $u_1; u_2$ new (sequential) updates can be built. More complex update combinators are described in [11], but for our purposes elementary updates plus sequential composition is sufficient.

Let ξ denote a formula or term and u an update: then $\{u\}\xi$ is again a formula/term. The semantics of an elementary update is that of an assignment. In this paper we restrict the kind of term that may occur as location or as an assigned value to so-called *symbolic values*. Simply expressed, a symbolic value is a logical term or a program that has no side-effects and that is not state-dependent.

- Example 2.*
1. In the formula $\{l := v\}\phi$, the subformula ϕ is evaluated in a state where $!l$ has the value v .
 2. The formula $\{l := v1; l := v2\}\phi$ is equivalent to $\{l := v2\}\phi$, because the second update overwrites the effect of the first one.
 3. The update in $\{!l1 := 3; !l2 := !l1\}\phi$ is syntactically incorrect as the right side of the second update is state dependent and not a symbolic value according to the definition above.

During a sequent proof the updates accumulate in front of a symbolically executed program until execution terminates. Upon termination, the updates are applied to terms and formulas much like substitutions. This lazy application of updates helps efficiency, because automatic first-order simplification steps are applied eagerly *before* updates are substituted into formulas. This is particularly important in presence of aliasing, see Sect. 4.

3 Programming Language

We present the syntax and evaluation rules of a small untyped functional language with references. In this framework, we consider static typing as an attribute of the logic, and we ignore it when defining the operational semantics.

3.1 Syntax

Constants The language provides two kinds of constants: *integer constants* $n \in \mathbb{Z}$ and *location constants* $\ell \in \mathcal{L}$, where \mathcal{L} is an infinite set of symbols disjoint from \mathbb{Z} . Boolean values ‘true’ and ‘false’ are represented by the integer constants 1 and 0. In conditionals, we shall more generally consider that any value different from 0 represents the Boolean value ‘true’.

Primitive functions We assume a finite set of function symbols (notation: f, f', f_1 , etc.) representing elementary operations on data. Every function symbol f comes with an arity $k \geq 1$ and a total function $\tilde{f} : \mathbb{Z}^k \rightarrow \mathbb{Z}$ defining the corresponding operation. We assume that these primitives contain at least the usual arithmetic operations (+, −, *, /, etc).

Programs and values The syntactic category of *programs* (notation: p, p', p_1 , etc.) is defined by

$$\begin{aligned}
p ::= & x \mid n \mid \ell \mid f(p_1, \dots, p_n) \mid p = p' \\
& \mid \lambda x. p \mid p p' \mid (p_1, p_2) \mid \text{fst}(p) \mid \text{snd}(p) \\
& \mid \text{if } p \text{ then } p_1 \text{ else } p_2 \mid \text{ref } p \mid p := p' \mid !p
\end{aligned}$$

The set of free variables of a program p is written $FV(p)$, and the set of locations occurring in p is written $\text{loc}(p)$. We also use the shorthand $\text{let } x = p \text{ in } p'$ (local definition) for $(\lambda x. p') p$, the same program being more simply written $p; p'$ (sequence) in the case where $x \notin FV(p')$. The fixpoint combinator for call-by-value strategy can also be encoded as $\text{fix} \equiv \lambda f. (\lambda x. f (\lambda y. xxy)) \lambda x. f (\lambda y. xxy)$.

We call a *value* (notation: v, v', v_1 , etc.) any closed program that is generated from the following grammar:

$$v ::= n \mid \ell \mid (v_1, v_2) \mid \lambda x. p \quad (FV(p) \subseteq \{x\})$$

The set of all values is written \mathcal{V} . This set is equipped with an equivalence relation, written $v \sim v'$, that is used to implement the structural equality test. The definition of this relation will be given in Section 5.

3.2 Operational Semantics

Stores We call a *store* any partial function $s : \mathcal{L} \rightarrow \mathcal{V}$ whose domain, written $\text{dom}(s)$, is finite. A store may either represent the contents of the memory, or simply a set of local modifications (a ‘patch’). In this spirit, we define an operation of *asymmetric merge* between stores, written $s_1 \otimes s_2$ and defined by

$$\begin{aligned}
\text{dom}(s_1 \otimes s_2) &= \text{dom}(s_1) \cup \text{dom}(s_2) \\
(s_1 \otimes s_2)(\ell) &= s_2(\ell) && \text{if } \ell \in \text{dom}(s_2) \\
(s_1 \otimes s_2)(\ell) &= s_1(\ell) && \text{otherwise}
\end{aligned}$$

Intuitively, $s_1 \otimes s_2$ is the store obtained by applying the ‘patch’ s_2 to s_1 . This operation will be used in the semantics of the update mechanism in Sect. 5.

In what follows, we assume given an *allocation function* alloc that associates to every store s a new location $\text{alloc}(s) \in \mathcal{L}$ such that $\text{alloc}(s) \notin \text{dom}(s)$.

Evaluation contexts Evaluation contexts specify the strategy of evaluation. They are defined from the BNF:

$$\begin{aligned}
C ::= & () \mid f(v_1, \dots, v_n, C, p_1, \dots, p_m) \mid C = p \mid v = C \\
& \mid (C, p) \mid (v, C) \mid \text{fst}(C) \mid \text{snd}(C) \mid C p \mid v C \\
& \mid \text{if } C \text{ then } p_1 \text{ else } p_2 \mid \text{ref } C \mid !C \mid C := p \mid v := C
\end{aligned}$$

We assume that the programs p, p_1, p_2 occurring in the above definition are closed, so that evaluation contexts are closed objects. Similarly, we assume that the function symbols f are totally applied.⁴ We write $C(p)$ for the (closed) program obtained by substituting the (closed) program p to the hole $()$ in the evaluation context C .

⁴ According to this definition, arguments of functions are thus evaluated from the left to the right, as well as members of equalities, components of pairs, etc.

Evaluation An *evaluation state* is a pair $p \star s$ formed by a closed program p and a store s . The relation of one-step evaluation, written $p \star s \succ p' \star s'$, is the binary relation over evaluation states that is defined from the axioms of Figure 1, plus the ‘context’ rule

$$\frac{p \star s \succ p' \star s'}{C(p) \star s \succ C(p') \star s'}$$

We denote with \succ^* the reflexive-transitive closure of the relation \succ .

$$\begin{array}{l} (\lambda x. p) v \star s \succ [v/x] p \star s \\ \text{fst}(v_1, v_2) \star s \succ v_1 \star s \\ \text{snd}(v_1, v_2) \star s \succ v_2 \star s \\ \\ v = v' \star s \succ \begin{cases} 1 \star s & \text{if } v \sim v' \\ 0 \star s & \text{if } v \not\sim v' \end{cases} \\ \\ \begin{array}{l} v_1 v_2 \star s \succ 0 \star s \quad (\text{if } v_1 \text{ is not an abstraction}) \\ f(v_1, \dots, v_k) \star s \succ 0 \star s \quad (\text{if } v_i \notin \mathbb{Z} \text{ for some } i \in [1..k]) \\ \text{if } v \text{ then } p_1 \text{ else } p_2 \star s \succ 0 \star s \quad (\text{if } v \notin \mathbb{Z}) \end{array} \\ \\ \begin{array}{l} \text{fst}(v) \star s \succ 0 \star s \quad (\text{if } v \text{ is not a pair}) \\ \text{snd}(v) \star s \succ 0 \star s \quad (\text{if } v \text{ is not a pair}) \end{array} \quad \begin{array}{l} \text{ref } v \star s \succ \ell \star (s \otimes \ell \leftarrow v) \quad (\text{if } \ell = \text{alloc}(s)) \\ \ell := v \star s \succ 0 \star s \otimes \ell \leftarrow v \quad (\text{if } \ell \in \text{dom}(s)) \\ !\ell \star s \succ s(\ell) \star s \quad (\text{if } \ell \in \text{dom}(s)) \\ \\ v := v' \star s \succ 0 \star s \quad (\text{if } v \notin \text{dom}(s)) \\ !v \star s \succ 0 \star s \quad (\text{if } v \notin \text{dom}(s)) \end{array} \\ \\ \begin{array}{l} f(n_1, \dots, n_k) \star s \succ \tilde{f}(n_1, \dots, n_k) \star s \\ \text{if } n \text{ then } p_1 \text{ else } p_2 \star s \succ p_1 \star s \quad (\text{if } n \neq 0) \\ \text{if } 0 \text{ then } p_1 \text{ else } p_2 \star s \succ p_2 \star s \end{array} \end{array}$$

Fig. 1. One step evaluation rules

Note that the evaluation rules given above (that are clearly deterministic) explicitly deal with ‘runtime errors’ (such as applying a value that is not a function, etc.) and return the arbitrary value 0 in this case. So that:

Lemma 1 (Determinism and progression). *For all evaluation states $p \star s$, there is at most one evaluation state $p' \star s'$ such that $p \star s \succ p' \star s'$. Moreover, this evaluation state $p' \star s'$ exists if and only if p is not a value.*

3.3 Well-formedness of stores

Let s be a store. A program (or a value) p is *well-formed* in the store s when $\text{loc}(p) \subseteq \text{dom}(s)$. The set of well-formed values in s is written \mathcal{V}_s . A *well-formed store* is a store s such that $s(\ell) \in \mathcal{V}_s$ for all $\ell \in \text{dom}(s)$. The set of well-formed stores is written \mathfrak{S} . Finally, an evaluation state $p \star s$ is said to be *well-formed* when s is a well-formed store and p is well-formed in s . Well-formedness of evaluation states is preserved by evaluation:

Lemma 2. *If $p \star s$ is a well-formed evaluation state and $p \star s \succ p' \star s'$, then $p' \star s'$ is a well-formed evaluation state too.*

4 Logical System

We present the syntax and the rules of a proof language designed to specify programs such as defined in Sect. 3. This proof language is based on an extension of Dynamic Logic (DL) with second-order quantifications, so that the language includes second-order functional arithmetic (AF2) [9] as well as the modalities of DL. The individuals manipulated by this logic are *symbolic values* that are formally defined below. Programs (actually: symbolic programs) may also appear inside formulas but restricted to specific positions as we shall see.

4.1 Symbolic Expressions

Location Names To reason efficiently about locations without mentioning them explicitly in the specification language, we introduce a new category of names, called *location names* and written α, β, γ , etc. Semantically, location names are characterized by three invariants:

1. A location name always refers to a concrete location.
2. The location referred to by a name is always allocated in the current store.
3. Two distinct location names refer to two distinct locations.

These invariants are essential to deal with problems of freshness and aliasing, and to ensure the absence of memory faults during evaluation (see Sect. 5).

Symbolic Programs Symbolic programs are defined in the same way as the programs introduced in Sect. 3. The only difference is that concrete locations are replaced by location names in the BNF. In this section p, q, p' , etc., denote symbolic programs instead of concrete programs.

The (capture-preserving) implicit substitution operation is defined as in the λ -calculus, and its result is written $[p'/x]p$. Note that in presence of side effects, this operation is not semantically sound, since the programs $[p'/x]p$ and $\text{let } x = p' \text{ in } p$ do not generally have the same operational semantics. A counter-example is given by the program $[!r/y](\lambda x. y) \equiv \lambda x. !r$, that does not behave the same way as the program $\text{let } y = !r \text{ in } \lambda x. y$. For this reason, we shall put severe restrictions on the use of this form of substitution in the logic.

Symbolic Values Symbolic values form a sub-class of the syntactic category of symbolic programs, that is defined from the following BNF:

$$\begin{aligned} v ::= & x \mid \alpha \mid n \mid f(v_1, \dots, v_n) \mid v_1 = v_2 \mid \lambda x. p \\ & \mid (v_1, v_2) \mid \text{fst}(v) \mid \text{snd}(v) \mid \text{if } v \text{ then } v_1 \text{ else } v_2 \end{aligned}$$

(Unlike concrete ML-values, symbolic values may be open as well as closed.)

Intuitively, symbolic values correspond to the programs that do not access the store, and whose form is simple enough to ensure termination. For this reason, every symbolic value unambiguously refers to a concrete value (provided we assign a value to every variable and a location to every location name).

Substitution of symbolic values v is thus a safe operation, since the program $[v/x] p$ has the same semantics as $\text{let } x = v \text{ in } p$.

Symbolic Evaluation of Symbolic Programs The class of symbolic programs comes with a congruence written $p \cong p'$ that expresses that the two programs p and p' are equivalent modulo zero, one or several steps of symbolic evaluation. This congruence is defined from the following rules:

$$\begin{array}{ll}
 f(n_1, \dots, n_k) \cong \tilde{f}(n_1, \dots, n_k) & \text{if } n \text{ then } p \text{ else } p' \cong p \quad (n \neq 0) \\
 (\lambda x. p) v \cong [\!/_x] p & \text{if } 0 \text{ then } p \text{ else } p' \cong p' \\
 \text{fst}((v_1, v_2)) \cong v_1 & v = v \cong 1 \\
 \text{snd}((v_1, v_2)) \cong v_2 & n = m \cong 0 \quad (n \neq m) \\
 & \alpha = \beta \cong 0 \quad (\alpha \neq \beta)
 \end{array}$$

Note that these rules can be applied in any context, even under λ -abstractions. In particular, we have $\lambda x. 1 + 1 \cong \lambda x. 2$ even though both members are values that are not further evaluated. The main reason for this design choice is that it makes the definition of the logical system conceptually and technically much more simple. (However, we shall see in Sect. 5.1 that this choice has subtle consequences on the semantics.)

4.2 Updates

We employ a simplified form of update as compared to the general definition in [11]. Formally, updates (notation: u, u', u_1 , etc.) are defined as finite lists of pairs of symbolic values of the form $v := v'$:

$$u ::= \emptyset \mid u; v := v'$$

(Note that \emptyset acts a neutral element, hence $\emptyset; u \equiv u$.) The application of an update u to a symbolic program of the form $!v$ (where v is a symbolic value) is written $\{u\}!v$ and defined by

$$\begin{array}{l}
 \{\emptyset\}!v = !v \\
 \{u; v_1 := v_2\}!v = \text{if } v = v_1 \text{ then } v_2 \text{ else } \{u\}!v
 \end{array}$$

Note that the result of this operation is a symbolic program that can be simplified using the congruence rules of symbolic evaluation.

4.3 Formulas

Formulas (notation: A, B, C , etc.) are built from second-order variables (notation: X, Y, Z , etc.) that represent k -ary relations. We assume that every second-order variable comes with an arity which we indicate as a superscript when we introduce the variable. The syntax of formulas is the following:

$$\begin{array}{l}
 A ::= X(v_1, \dots, v_k) \mid A \rightarrow B \mid \forall x. A \mid \forall X^k. A \\
 \mid I(v) \mid \nu \alpha. A \mid [p \text{ as } x]A \mid \{u\}A
 \end{array}$$

(For simplicity, we consider a language based on implication and first- and second-order universal quantification, from which we easily recover other connectives and quantifiers.) We also provide the following constructs:

- A predicate constant I that transforms any symbolic value v into a formula $I(v)$ that is true if the concrete value denoted by v is a value different from 0.
- A construct $[p \text{ as } x] A$ that means: ‘if p evaluates to a value x , then A holds in the store affected by all the side effects performed by p ’. This construction is nothing but the box modality of DL that we transformed into a binder to recover the value computed by the program p . In particular, when A does not depend on x , we simply write $[p]A$.
- A construct $\{u\}A$ that means: ‘after updating the current store with the assignments in u , A holds’.
- A construct $\nu\alpha.A$ (ν -binder) that means: ‘after the allocation of a fresh address named α , A holds’.

The set of free variables (free names) of a formula A is written $FV(A)$ ($FN(A)$).

4.4 Symbolic Evaluation

The congruence defined in Sect. 4.1 over symbolic programs is extended to formulas which, together with a contextual closure, occur within formulas and with specific rules for decomposing boxes as well as for propagating updates and ν s throughout the structure of formulas (Fig. 2).

$$\begin{aligned} I(0) &\cong \perp \quad (\equiv \forall X.X) \\ I(n) &\cong \top \quad (\equiv \forall X.X \rightarrow X) \end{aligned} \quad n \neq 0$$

Decomposition of boxes

$$\begin{aligned} [C_{se}(p) \text{ as } x] A &\cong [p \text{ as } y] [C_{se}(y) \text{ as } x] A && y \notin FV(C_{se}(p), A, x) \\ [\text{ref } v \text{ as } x] A &\cong \nu\alpha.\{\alpha := v\}^{[\alpha/x]} A && \alpha \notin FN(A, v) \\ [v_1 := v_2] A &\cong \{v_1 := v_2\} A \\ [v \text{ as } x] A &\cong [^v/x] A \end{aligned}$$

Propagation of updates

$$\begin{aligned} \{u\}I(v) &\cong I(v) \\ \{u\}(A \rightarrow B) &\cong \{u\}A \rightarrow \{u\}B \\ \{u\}\forall x.A &\cong \forall x.\{u\}A && x \notin FV(u) \\ \{u\}\forall X.A &\cong \forall X.\{u\}A \\ \{u\}\nu\alpha.A &\cong \nu\alpha.\{u\}A && \alpha \notin FN(u) \\ \{u\}\{u'\}A &\cong \{u; u'\}A \\ \{u\}![v \text{ as } x] A &\cong [!\{u\}; v \text{ as } x] \{u\}A && x \notin FV(u) \end{aligned}$$

Propagation of ν s

$$\begin{aligned} \forall X^n. \nu\alpha.A &\cong \nu\alpha.\forall X^n. A \\ [p \text{ as } x] \nu\alpha.A &\cong \nu\alpha.[p \text{ as } x] A && \alpha \notin FN(p) \\ \nu\alpha.\nu\beta.A &\cong \nu\beta.\nu\alpha.A \end{aligned}$$

Fig. 2. Symbolic evaluation of formulas

Decomposition of boxes The decomposition of boxes has to take care of the evaluation order. The first rule splits a program inside a box in two pieces according to a given symbolic evaluation context C_{se} . (Symbolic evaluation contexts are defined as for evaluation contexts, replacing explicit locations with location names and explicit values with symbolic values.) Note that the enclosing symbolic evaluation context is not uniquely determined by the program within the box, and this rule can be used to decompose the very same box in many different ways. The next two rules deal with the creation of a reference (that introduces a ν -binder and an update) and with an assignment (that introduces an update). The last rule simply removes a box when the inner program is a symbolic value.

Propagation of updates Updates go down through the structure of formulas until they reach a box. An update can go through a box only when the inner program is of the form $!v$ (access to the contents of a reference), in which case the program is updated using the construction $\{u\}!v$ defined in Section 4.2. In all the other cases, the update is stuck in front of the box until this box is decomposed into smaller boxes using symbolic evaluation.

Propagation of ν s The ν -binder comes with quite standard propagation rules (we do not give them all). Note that there is a rule for commuting a ν -binder with second-order quantification, but no analogous rule for first-order quantification. The reason is that semantically, the domain of first-order quantification depends on the set of currently allocated locations, so that we have

$$\forall x.\nu\alpha.A \not\rightarrow \nu\alpha.\forall x.A$$

in general. We shall come back to this point in Sect. 5. Note also that in general a ν -binder cannot be dropped even when the name it binds does not occur in its scope, so we have $\nu\alpha.A \not\rightarrow A$ even if $\alpha \notin FN(A)$.

4.5 Deduction Rules

The language is equipped with a Gentzen-style sequent calculus. This system includes the standard rules for second-order logic: structural rules (weakening and contraction), axiom, cut, plus the standard left and right rules for implication, first- and second-order universal quantification. The specific rules of our system (see Fig. 3) include:

- Left and right rules for symbolic evaluation, expressing that computationally equivalent formulas (via symbolic evaluation) are logically equivalent.
- Necessitation rules for all modalities (ν -binder, updates, and boxes).

Note that the generalized forms of the standard necessitation rules are allowed in our case because the programming language is deterministic and because values are normal forms (Lemma 1), so that the frame relation underlying each modality (including updates) is functional. The side condition of BOX-NCS is necessary because the evaluation of the inner program might not terminate. In this case, the hypothesis $[p \text{ as } x] \Gamma$ becomes vacuously valid (as we shall see in Sect. 5) while the empty conclusion is obviously not.

$$\begin{array}{c}
\frac{\Gamma, A' \vdash \Delta \quad A \cong A'}{\Gamma, A \vdash \Delta} \text{RWG} \qquad \frac{\Gamma \vdash A', \Delta \quad A \cong A'}{\Gamma \vdash A, \Delta} \text{RWD} \\
\\
\frac{\Gamma \vdash \Delta}{\nu\alpha. \Gamma \vdash \nu\alpha. \Delta} \nu\text{NCS} \qquad \frac{\Gamma \vdash \Delta}{\{u\}\Gamma \vdash \{u\}\Delta} \text{UPD-NCS} \\
\\
\frac{\Gamma \vdash \Delta}{\Delta \neq \emptyset \quad [p \text{ as } x]\Gamma \vdash [p \text{ as } x]\Delta} \text{BOX-NCS}
\end{array}$$

Fig. 3. Specific deduction rules

5 Semantics

We now build a Kripke model of the language where worlds are well-formed stores (simply called stores from now on). In this setting, each symbolic value is interpreted as a concrete value whereas each formula is interpreted as the set of stores in which the formula is true. The construction is standard, with some subtleties that will be explained in Sect. 5.1. The main feature of the model is that the domain of interpretation of the individuals (i.e. symbolic values) has to depend on the current store, because the values which make sense in a store s are those which are well-formed in s . (In particular, this property explains why we cannot commute first-order quantification with ν -binders.)

5.1 Invariance properties

Equivalence of values Both in IML and in the logical framework, two functional values $\lambda x. p$ and $\lambda x. p'$ are observationally equivalent when $p \cong p'$. To identify such values in the model, we introduce the relation of *equivalence of values*, written $v \sim v'$, as the least equivalence relation such that

- If $p \cong p'$, then $\lambda x. p \sim \lambda x. p'$.
- If $v_1 \sim v'_1$ and $v_2 \sim v'_2$, then $(v_1, v_2) \sim (v'_1, v'_2)$.

Invariance under automorphisms Similarly, the allocation order of locations is indistinguishable both for IML programs and for the logical framework. In order to ensure that the model is not sensitive to the allocation order either, we need to introduce the notion of invariance under all automorphisms.

Formally, an *automorphism* (of locations) is any bijection σ over the set \mathcal{L} of locations. An automorphism σ can be applied to a location, but also to a value (by applying σ to all the locations inside the value) as well as to a store. Formally, the store $\sigma(s)$ is defined by $\text{dom}(\sigma(s)) = \sigma(\text{dom}(s))$ and

$$\sigma(s)(\ell) = \sigma(s(\sigma^{-1}(\ell))) \qquad (\ell \in \text{dom}(\sigma(s)))$$

Propositional functions Let $f : \mathcal{V}^k \rightarrow \mathfrak{P}(\mathfrak{S})$ be a function from k -tuples of values to sets of (well-formed) stores. We say that f is:

- *compatible with the equivalence of values* when for all $v_1, \dots, v_k, v'_1, \dots, v'_k$ such that $v_1 \sim v'_1, \dots, v_k \sim v'_k$: $f(v_1, \dots, v_k) = f(v'_1, \dots, v'_k)$.
- *invariant under all automorphisms* when for all $v_1, \dots, v_k \in \mathcal{V}$, $s \in \mathfrak{S}$ and for all automorphisms σ : $s \in f(v_1, \dots, v_k)$ iff $\sigma(s) \in f(\sigma(v_1), \dots, \sigma(v_k))$.

The set of all functions $f : \mathcal{V}^k \rightarrow \mathfrak{P}(\mathfrak{S})$ that are both compatible with the equivalence of values and invariant under all automorphisms is written \mathcal{F}_P^k . In what follows, we shall interpret predicates variables of arity k (and formulas depending on k first-order variables) as elements of \mathcal{F}_P^k .

5.2 Interpreting symbolic values and updates

Valuations A *valuation* is a function ρ that maps each

- first-order variable x to a value $\rho(x) \in \mathcal{V}$;
- k -ary second-order variable X to a propositional function $\rho(x) \in \mathcal{F}_P^k$;
- location name α to a location $\rho(\alpha) \in \mathcal{L}$.

Moreover, we require that ρ is injective on location names: distinct location names are mapped to distinct locations. A valuation ρ is *well-formed* in a store s when $\rho(x) \in \mathcal{V}_s$ for all $x \in \text{dom}(\rho)$ and $\rho(\alpha) \in \text{dom}(s)$ for all $\alpha \in \text{dom}(\rho)$. This notion is clearly preserved by store extension.

Interpreting symbolic values Given a symbolic value \mathbf{v} and a valuation ρ , we denote by $\llbracket \mathbf{v} \rrbracket_\rho$ the unique value v such that $\mathbf{v}[\rho] \star s \succ^* v \star s$, where s is an arbitrary store. Note that such a value always exists—due to the restricted form of symbolic values—and that it is unique since evaluation is deterministic. Moreover, the value v does not depend on s , and the evaluation of the program $\mathbf{v}[\rho]$ that computes the value v does not modify the store.

Interpreting updates Updates are interpreted as stores (intuitively: as ‘patches’ to the global memory). Given an update u and a valuation ρ , we define $\llbracket u \rrbracket_\rho$ by

$$\llbracket \emptyset \rrbracket_\rho = \emptyset \quad \text{and} \quad \llbracket u; \mathbf{v}_1 := \mathbf{v}_2 \rrbracket_\rho = \llbracket u \rrbracket_\rho \otimes \llbracket \mathbf{v}_1 \rrbracket_\rho \leftarrow \llbracket \mathbf{v}_2 \rrbracket_\rho$$

5.3 Interpreting formulas and sequents

We now define the relation of satisfaction $s \models A[\rho]$ (where s is a well-formed store and ρ a valuation that is well-formed in s) by letting:

$$\begin{aligned} s \models X(\mathbf{v}_1, \dots, \mathbf{v}_k)[\rho] & \text{ iff } s \in \rho(X)(\llbracket \mathbf{v}_1 \rrbracket_\rho, \dots, \llbracket \mathbf{v}_k \rrbracket_\rho) \\ s \models I(\mathbf{v})[\rho] & \text{ iff } \llbracket \mathbf{v} \rrbracket_\rho \neq 0 \\ s \models (A \rightarrow B)[\rho] & \text{ iff } s \models A[\rho] \text{ implies } s \models B[\rho] \\ s \models (\forall x.A)[\rho] & \text{ iff for all } v \in \mathcal{V}_s, s \models A[\rho; x \mapsto v] \\ s \models (\forall X^k.A)[\rho] & \text{ iff for all } f \in \mathcal{F}_P^k, s \models A[\rho; X \mapsto f] \\ s \models (\nu \alpha.A)[\rho] & \text{ iff } (s \otimes \text{alloc}(s) \leftarrow 0) \models A[\rho; \alpha \mapsto \text{alloc}(s)] \\ s \models (\{u\}A)[\rho] & \text{ iff } s \otimes \llbracket u \rrbracket_\rho \models A[\rho] \end{aligned}$$

$$s \models ([p \text{ as } x] A)[\rho] \quad \text{iff} \quad \text{for all } s' \in \mathfrak{S} \text{ and } v \in \mathcal{V}_{s'}, \\ p[\rho] \star s \succ^* v \star s' \text{ implies } s' \models A[\rho; x \mapsto v]$$

The interpretation immediately extends to sequents (notation: $s \models (\Gamma \vdash \Delta)[\rho]$), reading left hand-side commas as conjunctions, right hand-side commas as disjunctions and the symbol ‘ \vdash ’ as implication. Note that the formula $[p \text{ as } x] A$ is always valid when p does not terminate.

Theorem 1 (Correctness of the system) *If the sequent $\Gamma \vdash \Delta$ is derivable in the system, then for all well-formed stores $s \in \mathfrak{S}$ and for all valuations ρ that are well-formed in s , we have $s \models (\Gamma \vdash \Delta)[\rho]$.*

Theorem 1 relies on many intermediate lemmas that are not given here. Basically, these lemmas express that all the constructions of the programming language and of the logical framework are compatible with the equivalence of values and preserve the property of invariance under all automorphisms.

It is easy to check that $s \not\models \perp$. Hence, the formula \perp cannot be proved within our system, which is thus consistent.

6 Specification and Verification of a Recursive Function

We illustrate how to specify and verify a recursive function along a small example. Let us now consider the program cc defined by

$$cc \equiv \lambda n. \text{ let } c = (\text{let } r = \text{ref } 0 \text{ in } \lambda x. r := !r + 1 ; !r) \text{ in} \\ \text{let } aux = \text{fix } (\lambda fn. \text{ if } n = 0 \text{ then } c \ 0 \text{ else } (c \ 0 ; f \ (n - 1))) \text{ in} \\ aux \ n$$

This program takes a natural number n as an argument and calls $n + 1$ times the sub-program c that contains a local reference, before returning the result of the last call of c . (Here the argument 0 in $c \ 0$ plays the role of $()$ in ML.)

We intend to prove that for all natural numbers n , the result of $cc \ n$ is $n + 1$. Therefore, we need first to characterise natural numbers among all the possible values. For the characterisation we use their well-known second-order definition as given in [9] :

$$\text{Nat}(x) \equiv \forall X. (\forall y. (X(y) \rightarrow X(y + 1)) \rightarrow X(0) \rightarrow X(x))$$

To keep the presentation of the specification and verification readable we introduce the relativized quantification $\forall x : \text{Nat}. A$ as syntactic sugar for the formula $\forall x. (\text{Nat}(x) \rightarrow A)$. Finally, we can express the property of interest by the formula

$$\forall n : \text{Nat}. [cc \ n = n + 1 \text{ as } b] I(b).$$

A derivation Π_1 of this formula is shown in Fig. 4. Note that in the derivation, we use the obvious rules that can be derived from the definition of $\text{Nat}(x)$ as well as the (derived) induction rule:

$$\frac{\vdash A(0) \quad \text{Nat}(n), A(n) \vdash A(n + 1)}{\vdash \forall n : \text{Nat}. A(n)} \text{Ind}$$

We also introduce the following shortcuts:

$$\begin{aligned}
 \underline{c} &\equiv \text{let } r = \text{ref } 0 \text{ in } \lambda x. r := !r + 1 ; !r \\
 \underline{\text{aux}} &\equiv \text{fix } (\lambda fn. \text{if } n = 0 \text{ then } c \text{ } 0 \text{ else } (c \text{ } 0 ; f (n - 1))) \\
 \underline{cc} &\equiv \lambda n. \text{let } c = \underline{c} \text{ in let } \text{aux} = \underline{\text{aux}} \text{ in } \text{aux } n \\
 \underline{\text{aux}}' &\equiv \text{fix } (\lambda fn. \text{if } n = 0 \text{ then } (\lambda x. \alpha := !\alpha + 1 ; !\alpha) \text{ } 0 \\
 &\quad \text{else } (\lambda x. \alpha := !\alpha + 1 ; !\alpha) \text{ } 0 ; f (n - 1))
 \end{aligned}$$

To simplify the derivation of Fig 4, we denote by $\underline{\text{aux}}$ and $\underline{\text{aux}}'$ the functional values these programs reduce to. The specification is proved using an auxiliary lemma (L) stating that the property holds for any content of the reference: this lemma is proved (left premise of the Cut rule in II_1) by induction. Note that this lemma can be used only in a context in which the inner reference is visible. The bottom part of the proof partially evaluates the program $cc \ n$ to make the inner reference visible at the top level.

$$\begin{array}{c}
 \frac{}{\vdash I(1)} \text{Ax} \\
 \frac{}{\vdash \{\alpha := k + 1\}I(1)} \cong \\
 \frac{}{\vdash \{\alpha := k + 1\}[k + 1 = k + 1 \text{ as } b]I(b)} \cong \times 2 \\
 \frac{}{\vdash \{\alpha := k + 1\}[\alpha = k + 1 \text{ as } b]I(b)} \cong \\
 \frac{}{\vdash \{\alpha := k\}[(\alpha := k + 1; !\alpha) = k + 1 \text{ as } b]I(b)} \cong \\
 \frac{}{\vdash \{\alpha := k\}[(\alpha := !\alpha + 1; !\alpha) = k + 1 \text{ as } b]I(b)} \cong \quad [II_3] \\
 \frac{}{\vdash \{\alpha := k\}[\underline{\text{aux}}' \text{ } 0 = k + 1 \text{ as } b]I(b)} \cong \quad \vdots \\
 \frac{}{\vdash \forall k. \{\alpha := k\}[\underline{\text{aux}}' \text{ } 0 = k + 1 \text{ as } b]I(b)} \forall R \quad \frac{}{RH \vdash RS} \quad \frac{}{L, \text{Nat}(n) \vdash C} [II_2] \\
 \frac{}{\vdash \forall n : \text{Nat}. \forall k. \{\alpha := k\}[\underline{\text{aux}}' \text{ } n = n + k + 1 \text{ as } b]I(b)} \text{Ind} \quad \frac{}{L, \text{Nat}(n) \vdash C} \text{Cut} \\
 \frac{}{\text{Nat}(n) \vdash \{\alpha := 0\}[\underline{\text{aux}}' \text{ } n = n + 1 \text{ as } b]I(b)} \nu R \\
 \frac{}{\text{Nat}(n) \vdash \nu \alpha. \{\alpha := 0\}[\underline{\text{aux}}' \text{ } n = n + 1 \text{ as } b]I(b)} \nu R \\
 \frac{}{\text{Nat}(n) \vdash \nu \alpha. \{\alpha := 0\}[\lambda x. \alpha := !\alpha + 1; !\alpha \text{ as } c][\underline{\text{aux}} \text{ } n = n + 1 \text{ as } b]I(b)} \cong \\
 \frac{}{\text{Nat}(n) \vdash [\text{ref } 0 \text{ as } r][\lambda x. r := !r + 1; !r \text{ as } c][\underline{\text{aux}} \text{ } n = n + 1 \text{ as } b]I(b)} \cong \\
 \frac{}{\text{Nat}(n) \vdash [\underline{c} \text{ as } c][\underline{\text{aux}} \text{ as } \text{aux}][\text{aux } n = n + 1 \text{ as } b]I(b)} \text{Let } + \cong \\
 \frac{}{\text{Nat}(n) \vdash [\text{let } c = \underline{c} \text{ in let } \text{aux} = \underline{\text{aux}} \text{ in } \text{aux } n = n + 1 \text{ as } b]I(b)} \text{Let } \times 2 + \cong \\
 \frac{}{\text{Nat}(n) \vdash [\underline{cc} \text{ } n = n + 1 \text{ as } b]I(b)} \cong \\
 \frac{}{\vdash \forall n : \text{Nat}. [\underline{cc} \text{ } n = n + 1 \text{ as } b]I(b)} \forall R
 \end{array}$$

$$\begin{aligned}
 \text{where } RS &\equiv \forall k. \{\alpha := k\}[\underline{\text{aux}}'(n + 1) = n + k + 2 \text{ as } b]I(b) \\
 RH &\equiv \text{Nat}(n), \forall k. \{\alpha := k\}[\underline{\text{aux}}' \text{ } n = n + k + 1 \text{ as } b]I(b) \\
 L &\equiv \forall n : \text{Nat}. \forall k. \{\alpha := k\}[\underline{\text{aux}}' \text{ } n = n + k + 1 \text{ as } b]I(b) \\
 C &\equiv \{\alpha := 0\}[\underline{\text{aux}}' \text{ } n = n + 1 \text{ as } b]I(b)
 \end{aligned}$$

Fig. 4. II_1 : Derivation of the specification

The proof of the recursive step (II_3) is done using only $\cong, \forall R, \forall L$. The sequent $L, \text{Nat}(n) \vdash C$ is trivially proved (II_2) with instances of the $\forall L$ rule.

7 Conclusion

We have presented a system for specifying and verifying imperative ML programs, whose specification language combines dynamic logic with second-order logic à la AF2. This combination illustrates the flexibility of DL, that can be adapted to many programming languages (here: imperative ML) and to many logical frameworks (here: second-order logic), thus making them benefit of the strength of symbolic evaluation and of its deep impact in proof automation.

The next step is to test our system by implementing it, for instance as a component of KeY or within another logical framework. Another natural research direction would be the integration of a static type system at the level of the logic, following the spirit of the system of strong typing in PAF!

Acknowledgements Many thanks are due to Yann Régis-Gianas for stimulating discussions and valuable advices.

References

1. M. Balsler, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer-Verlag, 2000.
2. S. Baro. Introduction to PAF!, a proof assistant for ml programs verification. In *TYPES*, pages 51–65, 2003.
3. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.
4. A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, September 2009.
5. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
6. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, October 2000.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
8. J. Kanig and J.-C. Filliâtre. Who: A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, Edinburgh, Scotland, UK, August 2009.
9. J. L. Krivine. *Lambda-calculus, types and models*. Masson, 1993.
10. Y. Régis-Gianas and F. Pottier. A hoare logic for call-by-value functional programs. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th Intl. Conf., MPC 2008, Marseille, France*, volume 5133 of *LNCS*, pages 305–335. Springer, 2008.
11. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.