Implementing a Program Logic of Objects in a Higher-Order Logic Theorem Prover

Martin Hofmann and Francis Tang*

Laboratory for Foundations of Computer Science
Division of Informatics
University of Edinburgh
Scotland, UK

Abstract. We present an implementation of a program logic of objects, extending that (AL) of Abadi and Leino. In particular, the implementation uses higher-order abstract syntax (HOAS) and—unlike previous approaches using HOAS—at the same time uses the built-in higher-order logic of the theorem prover to formulate specifications. We give examples of verifications, extending those given in [1], that have been attempted with the implementation. Due to the mixing of HOAS and built-in logic the soundness of the encoding is nontrivial. In particular, unlike in other HOAS encodings of program logics, it is not possible to directly reduce normal proofs in the higher-order system to proofs in the first-order object logic.

1 Introduction

The object-oriented (henceforth abbreviated as "OO") style of programming has shown to be exceptionally popular for developing large systems in a modular fashion. Despite its popularity, it is still lacking with regards to formal methods for verification.

This article is a foundational contribution towards the development of formal tools verification for OO languages. We have implemented a program logic for an object calculus, based on the logic from [1,2]. We have used the proof assistant LEGO[6] for historic reasons, though the techniques can be applied to other existing theorem provers, for example PVS and Isabelle/HOL. The encoding is notable for using:

- HOAS for encoding program syntax; and
- a direct embedding of the object logic into the metalogic.

The use of HOAS simplifies the encoding since we inherit variable scoping rules and alpha-conversion from the metalogic. Using the metalogic itself allows us to use the built-in features of the theorem prover. As a consequence of these two implementation decisions, soundness is non-trivial.

* Studentship funded by the EPSRC, UK.

J. Harrison and M. Aagaard (Eds.): TPHOLs 2000, LNCS 1869, pp. 268–282, 2000.

We give examples that have been attempted with the implementation. We considered the examples from Abadi and Leino as presented in [2]. Furthermore we extend their work with a new example based on the dining philosophers scenario.

Hereafter, the article is organised as follows. We first present a summary of the program logic from [2], giving the syntax of the object calculus and the verification axioms. We then present our implementation. Though the actual implementation is in LEGO, for expository purposes, we take as our metalanguage the fragment without universes, dependent and inductive types. We then give a selection of examples we have verified. A statement of the soundness property as well as the idea behind the proof are given in Section 5 but the details will be given elsewhere. Finally we conclude the article by giving some subjective views which have arisen from the work, and give a survey of related work.

2 The Abadi-Leino Logic

In [2], we are presented with an imperative, typed object language with subtyping but not recursive types. The language is given a syntax-directed operational semantics and also a Hoare-style verification logic for program correctness, which we will refer to simply as AL. The verification logic is proved to be sound but is also shown to be incomplete.

Objects are records of fields and methods. The only other types apart from objects are booleans and natural numbers. Each field is of primitive type or object type. Each method has exactly one bound variable denoting "self". The methods do not have any other formal parameters but arguments can be indirectly passed to them by first assigning to the fields in the object. There is no data abstraction nor an inheritance mechanism.

One record is a *subtype* of another if the one contains all fields of the other and for each method m in the other, a method whose return type is a subtype of that of m. The subtyping relation is said to be covariant with respect to method return values and invariant with respect to fields.

Method bodies are of the form $\varsigma(s)b$ where b is a program typically with free occurrences of s. Programs are built-up using constants (booleans and natural numbers), variables, and constructors for objects, let statements, conditional statements, field lookup, method invocation and field update. For field names $f_0, \ldots, f_k$, variables $x_0, \ldots, x_k$, method names $m_0, \ldots, m_l$, and method bodies $\varsigma(s)b_0, \ldots, \varsigma(s)b_l$, we have the program

$$[f_0{=}x_0, \ldots, f_k{=}x_k, m_0{=}\varsigma(s)b_0, \ldots, m_l{=}\varsigma(s)b_l]$$

which evaluates to a reference to an object with fields f_i and methods m_j. For a variable x and programs a and b, where b typically contains free occurrences of x, we have the program

$$\mathit{let}\ x{=}a\ \mathit{in}\ b\ .$$

The remaining constructions are standard and written *if* x *then* a_0 *else* a_1, $x.f$, $x.m$ and $x.f{:=}y$, where a_0 and a_1 are programs, x and y are variables, f is a field identifier and m is a method identifier.

The program *let* $x{=}a$ *in* b introduces the variable x. Execution of this program evaluates a, then b with occurences of x replaced with the previous result of evaluating a. Variables cannot be assigned to. However, a result can be an *object name* which does have state, and in this way mutable storage cells can be encoded as objects with one field. Since a is executed before b, we can define sequential composition $a; b$ in the usual way.

For arbitrary programs a with object type and field f, it is, in general, not possible to write $a.f$ with the intention to mean "evaluate a then look up f in the result." Such a program has to be encoded via the let construction as *let* $x{=}a$ *in* $x.f$. Similarly, it is in general not possible to write $a.f{:=}b$ nor $a.m$. This restriction simplifies the rules since evaluation of a variable is not side-effecting. Evaluation of an arbitrary program a is possibly side-effecting.

Since object terms evaluate to references, we also have the phenomenon of aliasing. For example, in the program

$$\mathit{let}\ x{=}a\ \mathit{in}\ (\mathit{let}\ y{=}x\ \mathit{in}\ b)\ ,$$

the variables x and y both refer to the same object during the excution of b: changes to x through field updates and method invocations also change y.

The semantics is given in terms of an abstract machine with a *stack* and a *store*. All terminating programs evaluate to a *result* in $\mathcal{R}$ which is the set of references (referred to as *object names* $\mathcal{H}$) and constants. An evaluation relation is introduced, written

$$S, \sigma \vdash a \rightsquigarrow v, \sigma'\ ,$$

meaning program a evaluated with stack S and initial store σ terminates with result v and final store σ'. We also have the set $\mathcal{F}$ of fieldnames and $\mathcal{M}$ of methodnames. A stack is a partial mapping from variables to *results*. A store is a mapping from object names to a pair of records: one of results indexed by field names, and one of method closures indexed by method names.

The verification logic is used for deriving judgements of the form

$$\Gamma \Vdash a : A :: T$$

where Γ is a *context*, a is a program, A a *specification* and T is a *transition relation*. A context is a sequence $x_0{:}A_0, \ldots, x_k{:}A_k$ of variable/specification pairs. Transition relations describe the behaviour of executing a program. They play the rôle of the assertions p and q in a Hoare triple $\{p\}S\{q\}$ and use the pseudo-variables $(\grave{\sigma}, \mathit{all}\grave{\mathit{oc}})$ and $(\acute{\sigma}, \mathit{all}\acute{\mathit{oc}})$ to refer, respectively, to the store before and after execution, and r to refer the result. As a pair, (σ, alloc) can be considered as a (curried) partial function of type $\mathcal{H} \rightharpoonup (\mathcal{F} \cup \mathcal{M}) \rightharpoonup \mathcal{R}$. Formally σ is a total function of type $(\mathcal{H} \times (\mathcal{F} \cup \mathcal{M})) \to \mathcal{R}$ and *alloc* is a predicate over $\mathcal{H}$ that defines the domain of the partial function. A specification describes what

a result from executing a program can potentially do. It can be thought of as the *interface* of an object. Specifications are necessary because a result can be an object with methods which are in essence "thunked" functions in the sense of suspended computations. For compositionality of these verification judgements, it is important that we can deduce what potential behaviour an object has.

We follow [2] and introduce the abbreviation *Res* for creating transition relations, defined by

$$Res(e) \stackrel{\text{def}}{=} r = e \wedge (\forall x, y).(\acute{\sigma}(x,y) = \grave{\sigma}(x,y) \wedge (\acute{alloc}(x) \equiv \grave{alloc}(x))) \ .$$

The predicate $Res(e)$ is used to describe an execution where the result of evaluation is e and the store is not changed. For example, we have the constant rule

$$\frac{E \Vdash \diamond}{E \Vdash false : Bool :: Res(false)}$$

(where the judgement $E \Vdash \diamond$ simply means that E is a welldefined environment.) This rule states that evaluating a constant does not change the store and the result of evaluation is the constant itself. Another easy rule is that for field lookup:

$$\frac{E \Vdash x : [f{:}A] :: Res(x)}{E \Vdash x.f : A :: Res(\grave{\sigma}(x,f))} \ .$$

Note that in the premise, we have the simple judgement $E \Vdash x : [f{:}A] :: Res(x)$. To apply this rule to variables that have more fields or even methods, we must apply the subsumption rule that is defined below.

By allowing transition relations to be strengthened in a subspecification, the subtype relation is straight-forwardly extended to specifications to give a subspecification relation $<:$. The formal definition can be found in Sec. 3.3 of [2].

Using the subspecification relation, we have the important rule of subsumption for verification judgments:

$$\frac{\Vdash A <: A' \quad \Vdash_{fol} T \Rightarrow T' \quad E \Vdash a : A : T}{E \Vdash a : A' : T'}$$

provided A' and T' are wellformed. Here $\Vdash_{fol} \phi$ denotes provability in first-order logic augmented with the standard axioms for equality.

We also have the let rule. This rule is defined

$$\frac{\begin{array}{l} E \Vdash a : A :: T \quad E, x{:}A \Vdash b : B :: T' \\ E \Vdash B \quad E \Vdash T'' \text{ is a transition relation} \\ \Vdash_{fol} T[\check{\sigma}/\acute{\sigma}, \acute{alloc}/\acute{alloc}, x/r] \wedge T'[\check{\sigma}/\grave{\sigma}, \acute{alloc}/\grave{alloc}] \Rightarrow T'' \end{array}}{E \Vdash let\ x{=}a\ in\ b : B :: T''}$$

where the judgement $E \Vdash B$ means that B is a wellformed specification. Intuitively, we have $E \Vdash let\ x{=}a\ in\ b : B :: T''$ provided T'' is a consequence of T and T'. The substitutions $[\check{\sigma}/\acute{\sigma}, \acute{alloc}/\acute{alloc}, x/r]$ and $[\check{\sigma}/\grave{\sigma}, \acute{alloc}/\grave{alloc}]$ handle the

intermediate state that exists after executing a but before executing b and also the assignment of the result of a to variable x.

With the exception of the subsumption rule, the let rule is the only other rule that cannot be applied backwards. This is because the premise contains the following formula

$$\Vdash_{fol} T[\check{\sigma}/\acute{\sigma}, al\check{l}oc/al\acute{l}oc, x/r] \wedge T'[\check{\sigma}/\grave{\sigma}, al\check{l}oc/al\grave{l}oc] \Rightarrow T''$$

and the formulae on the left of the connective do not appear in the conclusion of the rule. This formula is necessary because

$$T[\check{\sigma}/\acute{\sigma}, al\check{l}oc/al\acute{l}oc, x/r] \wedge T'[\check{\sigma}/\grave{\sigma}, al\check{l}oc/al\grave{l}oc]$$

is not a transition relation since it possibly has free occurences of $\check{\sigma}$, $al\check{l}oc$) and x. In a higher-order setting, we can use existential quantification to bind such free variables. Note that this problem is not present in Hoare logic because the intermediate state is existentially quantified in the metalogic.

3 Implementation

We implement a logic that is based on that presented in [2]. In the implementation we: encode the language syntax using HOAS; and use the meta-logic itself as the logic for writing transition relations.

Since our meta-logic is higher-order, our logic differs from that of [2] in that transition relations are now higher-order formulae. We appropriately modify the premises in the subsumption and let rules to take into account this difference.

A convenient consequence of our use of higher-order logic is that we can derive new rules that are often easier to use.

3.1 Metalanguage

We now introduce the metalanguage which is used to present the implementation of the program logic. The metalanguage is based on a higher-order simply-typed lambda calculus.

For base types, we have the natural numbers nat, booleans bool, variables VV, fieldnames FN, methodnames MN, results EE, specifications SS and program terms PP. We have the usual type formers: propositions Prop, function space $\tau_1 \to \tau_2$, product space $\tau_1 \times \tau_2$, τ_1-indexed record $\mathsf{Rcd}_{\tau_1}^{\tau_2}$ with entries from τ_2, variables x, application $e_1\, e_2$ and abstraction $\lambda x^{\tau}.e_1$. The type of transition relations TR is defined to be $\mathsf{EE} \to (\mathsf{EE} \to \mathsf{FN} \to \mathsf{EE})^2 \to (\mathsf{EE} \to \mathsf{Prop})^2 \to \mathsf{Prop}$, where we write $\cdots \to A^2 \to \cdots$ as shorthand for $\cdots \to A \to A \to \cdots$.

We define a higher-order classical logic using the constants $\forall_\tau : (\tau \to \mathsf{Prop}) \to \mathsf{Prop}$ and $\supset\colon \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$ for universal quantification and implication respectively. We take the standard classical higher-order logic encodings of conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), existential quantification ($\exists$) and Leibniz equality $=^{\tau}\colon \tau \to \tau \to \mathsf{Prop}$.

For predicates $P, Q : \tau \to \mathsf{Prop}$, we write $P \subseteq Q$ as shorthand for $\forall_\tau x.P(x) \supset Q(x)$. In particular, for sets A and B, the formula $A \subseteq B$ simply means than A is a subset of B, as expected. We define composition of $T : \mathsf{TR}$ and $U : \mathsf{EE} \to \mathsf{TR}$, written $T;U$ by

$$(T;U)(r, \grave{\sigma}, \acute{\sigma}, \grave{al\!loc}, \acute{al\!loc}) \stackrel{\text{def}}{=} \exists \check{r}, \check{\sigma}, \check{al\!loc}.\, T(\check{r}, \grave{\sigma}, \check{\sigma}, \grave{al\!loc}, \check{al\!loc}) \wedge U(\check{r}, r, \check{\sigma}, \acute{\sigma}, \check{al\!loc}, \acute{al\!loc}) \quad .$$

We have for constants: the normal constants for natural numbers (0, succ, $+, \ldots$); booleans ($\mathtt{ff}$, $\mathtt{tt}$, $\mathbf{and}$, $\ldots$); product types ($\pi_1^{\tau_1,\tau_2}$ $\pi_2^{\tau_1,\tau_2}$, $\langle -, - \rangle_{\tau_1,\tau_2}$; the record manipulation constants:

$$\mathsf{lookup}_{\tau_1,\tau_2} : \mathsf{Rcd}_{\tau_1}^{\tau_2} \to \tau_1 \to \tau_2 \qquad \mathsf{update}_{\tau_1,\tau_2} : \mathsf{Rcd}_{\tau_1}^{\tau_2} \to \tau_1 \to \tau_2 \to \mathsf{Rcd}_{\tau_1}^{\tau_2}$$
$$\mathsf{empty}_{\tau_1,\tau_2} : \mathsf{Rcd}_{\tau_1}^{\tau_2} \qquad \mathsf{domain}_{\tau_1,\tau_2} : \mathsf{Rcd}_{\tau_1}^{\tau_2} \to \tau_1 \to \mathsf{Prop} \quad .$$

We omit type annotations whenever possible, use parentheses with commas for (possibly) repeated application of terms, and infix notation where appropriate. We use the more succinct notation $[f_1{=}a_1, \ldots, f_k{=}a_k]$ for records, and for $\boldsymbol{a}$ of type $\mathsf{Rcd}_{\tau_1}^{\tau_2}$, we write a_e for $\mathsf{lookup}(\boldsymbol{a}, e)$. We write $f \in \mathsf{dom}(r)$ for $\mathsf{domain}_{\tau_1,\tau_2}(r, f)$.

We have two types of judgements: typing judgements $\Gamma \vdash e : \tau$ and validity judgements $\Gamma \vdash \phi$ provided $\Gamma \vdash \phi : \mathsf{Prop}$. We take the standard typing rules of simply typed lambda calculus and the standard logical rules and axioms of classical higher-order logic with Leibniz equality.

The intended interpretation of elements of record types are partial functions with finite domain. Thus equality over record types is the standard equality of partial functions, namely equality on their graphs. Thus we have a number of axioms on records that delineate this interpretation.

3.2 Program Logic

The remaining constants are those for the program logic: specification constructors $\mathsf{nat} : \mathsf{SS}$, $\mathsf{bool} : \mathsf{SS}$ and $\mathsf{obj} : \mathsf{Rcd}_{\mathsf{FN}}^{\mathsf{SS}} \to \mathsf{Rcd}_{\mathsf{MN}}^{\mathsf{EE}\to\mathsf{SS}\times\mathsf{EE}\to\mathsf{TR}} \to \mathsf{SS}$; specification subsumption $<: : \mathsf{SS} \to \mathsf{SS} \to \mathsf{Prop}$; program term constructors $\mathsf{false} : \mathsf{PP}$, $\mathsf{true} : \mathsf{PP}$, $\mathsf{let} : \mathsf{PP} \to (\mathsf{VV} \to \mathsf{PP}) \to \mathsf{PP}$, $\mathsf{obj} : \mathsf{Rcd}_{\mathsf{FN}}^{\mathsf{VV}} \to \mathsf{Rcd}_{\mathsf{MN}}^{\mathsf{VV}\to\mathsf{PP}} \to \mathsf{PP}$, $\mathsf{if} : \mathsf{VV} \to \mathsf{PP} \to \mathsf{PP} \to \mathsf{PP}$, $\mathsf{var} : \mathsf{VV} \to \mathsf{PP}$, $\mathsf{fsel} : \mathsf{VV} \to \mathsf{FN} \to \mathsf{PP}$, $\mathsf{minv} : \mathsf{VV} \to \mathsf{MN} \to \mathsf{PP}$ and $\mathsf{fupd} : \mathsf{VV} \to \mathsf{FN} \to \mathsf{VV} \to \mathsf{PP}$; the value constructors $\mathsf{bool}_{\mathsf{EE}} : \mathsf{bool} \to \mathsf{EE}$ and $\mathsf{nat}_{\mathsf{EE}} : \mathsf{nat} \to \mathsf{EE}$; and the formal symbol that represents the stack $\mathsf{var}_{\mathsf{EE}} : \mathsf{VV} \to \mathsf{EE}$. The constants $\mathsf{bool}_{\mathsf{EE}}$, $\mathsf{nat}_{\mathsf{EE}}$, and $\mathsf{var}_{\mathsf{EE}}$ can be seen as coercion functions and will therefore be omitted in this presentation.

Constant let gives our encoding of syntax its higher-order nature. As an illustration: program

let $x{=}$*true in if* x *then false else true*

has encoding

$$\mathsf{let}(\mathsf{true}, \lambda x.\mathsf{if}(x, \mathsf{false}, \mathsf{true})) \quad .$$

We generalise the encoding procedure and write $\ulcorner a \urcorner$ for the encoding of a.

Most important of all, we have the constant $[- : - :: -] : \mathsf{PP} \to \mathsf{SS} \to \mathsf{TR} \to \mathsf{Prop}$ and we encode the rules so that if

$$x_1{:}A_1, \ldots, x_n{:}A_n \Vdash a : A :: T$$

then we have

$$\vdash \forall x_1, \ldots x_n.[x_1 : \ulcorner A_1 \urcorner] \supset \cdots \supset [x_n : \ulcorner A_1 \urcorner] \supset [\ulcorner a \urcorner : \ulcorner A \urcorner :: \ulcorner T \urcorner]$$

where $[- : -]$ is defined by $[x : A] \stackrel{\text{def}}{=} [\mathsf{var}(x) : A :: \mathsf{Res}(x)]$.

Hereafter, unless explicitly typed otherwise, (meta-)variables and decorated variants of: n have type nat, x and y have type VV; f have type FN; m have type MN; a have type PP; b have type $\mathsf{VV} \to \mathsf{PP}$; A have type SS; B have type $\mathsf{EE} \to \mathsf{SS}$; T have type TR; and U have type $\mathsf{EE} \to \mathsf{TR}$.

Subsumption axioms. The following axioms are for the subsumption relation.

$$\begin{array}{l}\forall_{\mathsf{Rcd}^{\mathsf{SS}}_{\mathsf{FN}}} \boldsymbol{A}, \boldsymbol{A'}.\forall_{\mathsf{Rcd}^{(\mathsf{EE}\to\mathsf{SS})\times(\mathsf{EE}\to\mathsf{TR})}_{\mathsf{MN}}} \boldsymbol{B}, \boldsymbol{B'}. \\ \quad (\mathsf{dom}(\boldsymbol{A'}) \subseteq \mathsf{dom}(\boldsymbol{A})) \supset \\ \quad (\forall f \in \mathsf{dom}(\boldsymbol{A'}).A_f = A'_f) \supset \\ \quad (\mathsf{dom}(\boldsymbol{B'}) \subseteq \mathsf{dom}(\boldsymbol{B}) \supset \\ \quad (\forall m \in \mathsf{dom}(\boldsymbol{B'}).\pi_1(B_m) <: \pi_1(B'_m)) \supset \\ \quad (\forall m \in \mathsf{dom}(\boldsymbol{B'}).\forall y.\pi_2(B_m)(y) \subseteq \pi_2(B'_m)(y)) \supset \\ \quad\quad \mathsf{obj}(\boldsymbol{A}, \boldsymbol{B}) <: \mathsf{obj}(\boldsymbol{A'}, \boldsymbol{B'})\end{array} \qquad \text{(ss\_obj)}$$

$$\mathsf{bool} <: \mathsf{bool} \qquad \text{(ss\_bool)}$$

$$\mathsf{nat} <: \mathsf{nat} \qquad \text{(ss\_nat)}$$

Program axioms. The remaining axioms are those of the program logic. They are encoded as follows.

$$\begin{array}{l}\forall a.\forall A'.\forall T'.\forall A.\forall T. \\ \quad (A <: A') \supset \\ \quad (T \subseteq T') \supset \\ \quad\quad [a : A :: T] \supset [a : A' :: T']\end{array} \qquad \text{(ws\_subs)}$$

$$[\mathsf{false} : \mathsf{bool} :: \mathsf{Res}(\mathtt{ff})] \qquad \text{(ws\_constf)}$$

$$[\mathsf{true} : \mathsf{bool} :: \mathsf{Res}(\mathtt{tt})] \qquad \text{(ws\_constt)}$$

$$\forall n.[\mathsf{nat}(n) : \mathsf{nat} :: \mathsf{Res}(n)] \qquad \text{(ws\_nat)}$$

As an example of functions over constants, for any binary natural number operation op, we have

$$\begin{array}{l}\forall x_0, x_1. \\ \quad [x_0 : \mathsf{nat}] \supset [x_1 : \mathsf{nat}] \supset \\ \quad\quad [op(x_0, x_1) : \mathsf{nat} :: \mathsf{Res}(op(x_0, x_1))]\end{array} \qquad \text{(ws\_natop)}$$

$$\forall x.\forall a_0, a_1.\forall B.\forall U.\forall B_0, B_1.\forall U_0, U_1.$$
$$[x : \mathsf{bool}] \supset$$
$$[a_0 : B_0(x) :: U_0(x)] \supset$$
$$(B_0(\mathtt{tt}) = B(\mathtt{tt}) \wedge U_0(\mathtt{tt}) \equiv U(\mathtt{tt})) \supset \qquad \text{(ws\_cond)}$$
$$[a_1 : B_1(x) :: U_1(x)] \supset$$
$$(B_1(\mathtt{ff}) = B(\mathtt{ff}) \wedge U_1(\mathtt{ff}) \equiv U(\mathtt{ff})) \supset$$
$$[\mathsf{if}(x, a_0, a_1) : B(x) :: U(x)]$$

$$\forall a.\forall b.\forall A'.\forall T''.\forall A.\forall T.\forall U.$$
$$[a : A :: T] \supset$$
$$(\forall x.[x : A] \supset [b(x) : A' :: U(x)]) \supset \qquad \text{(ws\_let)}$$
$$(T; U \subseteq T'') \supset$$
$$[\mathsf{let}(a, b) : B :: T'']$$

$$\forall x.\forall m.\forall B.\forall U.$$
$$[x : \mathsf{obj}([], [m{=}\langle B, U\rangle])] \supset \qquad \text{(ws\_minv)}$$
$$[\mathsf{minv}(x, m) : B(x) :: U(x)]$$

$$\forall x.\forall f.\forall A.\forall T.$$
$$[x : \mathsf{obj}([f{=}A], [])] \supset$$
$$(\forall r.\forall \grave{\sigma}, \acute{\sigma}.\forall \grave{al}oc, \acute{al}oc. \qquad \text{(ws\_fsel)}$$
$$T(r, \grave{\sigma}, \acute{\sigma}, \grave{al}oc, \acute{al}oc) \equiv \mathsf{Res}(\acute{\sigma}(x, f), r, \grave{\sigma}, \acute{\sigma}, \grave{al}oc, \acute{al}oc)) \supset$$
$$[\mathsf{fsel}(x, f) : A :: T]$$

$$\forall_{\mathsf{Rcd}^{\mathsf{VV}}_{\mathsf{FN}}} \boldsymbol{x}.\forall_{\mathsf{Rcd}^{\mathsf{VV}\to\mathsf{PP}}_{\mathsf{MN}}} \boldsymbol{b}.\forall_{\mathsf{Rcd}^{\mathsf{SS}}_{\mathsf{FN}}} \boldsymbol{A}.\forall_{\mathsf{Rcd}^{(\mathsf{EE}\to\mathsf{SS})\times(\mathsf{EE}\to\mathsf{TR})}_{\mathsf{MN}}} \boldsymbol{B}.\forall T.$$
$$(\mathsf{dom}(\boldsymbol{x}) = \mathsf{dom}(\boldsymbol{A}) \wedge \mathsf{dom}(\boldsymbol{b}) = \mathsf{dom}(\boldsymbol{B})) \supset$$
$$(\forall f \in \mathsf{dom}(\boldsymbol{x}).[\mathsf{var}(x_f) : A_f]) \supset$$
$$(\forall m \in \mathsf{dom}(\boldsymbol{b}).\forall y.[y : \mathsf{obj}(\boldsymbol{A}, \boldsymbol{B})] \supset [b_m(y) : \pi_1(B_m)(y) :: \pi_2(B_m)(y)]) \supset$$
$$\left(\begin{array}{l}\forall r.\forall \grave{\sigma}, \acute{\sigma}.\forall \grave{al}oc, \acute{al}oc.\\ \quad T(r, \grave{\sigma}, \acute{\sigma}, \grave{al}oc, \acute{al}oc) \equiv (\forall z.z \neq r \supset \grave{al}oc(z) \equiv \acute{al}oc(z)) \wedge\\ \qquad\qquad\qquad (\forall f \in \mathsf{dom}(\boldsymbol{x}).\acute{\sigma}(r, f) = x_f) \wedge\\ \qquad\qquad\qquad (\forall z.\forall w.z \neq r \supset \grave{\sigma}(z, w) = \acute{\sigma}(z, w))\end{array}\right) \supset$$
$$[\mathsf{obj}(\boldsymbol{x}, \boldsymbol{b}) : \mathsf{obj}(\boldsymbol{A}, \boldsymbol{B}) :: T]$$
$$\text{(ws\_obj)}$$

$$\forall x, y.\forall f.\forall_{\mathsf{Rcd}^{\mathsf{SS}}_{\mathsf{FN}}} \boldsymbol{A}.\forall_{\mathsf{Rcd}^{(\mathsf{EE}\to\mathsf{SS})\times(\mathsf{EE}\to\mathsf{TR})}_{\mathsf{MN}}} \boldsymbol{B}.\forall T.$$
$$[x : \mathsf{obj}(\boldsymbol{A}, \boldsymbol{B})] \supset$$
$$[y : A_f] \supset$$
$$\left(\begin{array}{l}\forall r.\forall \grave{\sigma}, \acute{\sigma}.\forall \grave{al}oc, \acute{al}oc.\\ \quad T(r, \grave{\sigma}, \acute{\sigma}, \grave{al}oc, \acute{al}oc) \equiv r = x \wedge \acute{\sigma}(x, f) = y \wedge\\ \qquad\qquad\qquad (\forall z.\forall w.\neg(z = x \wedge w = f) \supset\\ \qquad\qquad\qquad\quad \grave{\sigma}(z, w) = \acute{\sigma}(z, w)) \wedge\\ \qquad\qquad\qquad \grave{al}oc = \acute{al}oc\end{array}\right) \supset$$
$$[\mathsf{fupd}(x, f, y) : \mathsf{obj}(\boldsymbol{A}, \boldsymbol{B}) :: T]$$
$$\text{(ws\_fupd)}$$

4 Examples

We have attempted several examples in our implementation. Initially, we followed the development in [2] and introduced some abbreviations to make our programs more succinct. Furthermore we derived, using the existing axioms, theorems (or equivalently, derivable rules) for these syntactic abbreviations.

For example, in the pure language, field selection strictly has the form $x.f$ where x is a variable. We introduce an abbreviation so that for an arbitrary program a, the program $a.f$ is an abbreviation for $\mathit{let}\ x{=}a\ \mathit{in}\ x.f$, for x not free in a. This is encoded into our formal system by defining

$$\mathsf{fsel}'(a, f) \stackrel{\text{def}}{=} \mathsf{let}(a, \lambda x.\mathsf{fsel}(x, f)) \ .$$

We shall simply overload our existing notation and write fsel for fsel'. Crucially, we then prove the following theorem.

$$\begin{array}{l}
\vdash \forall a.\forall f.\forall A.\forall T', T. \\
\quad [a : \mathsf{obj}([f{=}A], []) :: T] \supset \\
\quad (\forall r.\forall \grave{\sigma}, \acute{\sigma}.\forall \grave{al}oc, \acute{al}oc. \\
\qquad T'(r, \grave{\sigma}, \acute{\sigma}, \grave{al}oc, \acute{al}oc) \equiv \\
\qquad \exists \check{r}, \check{\sigma}, \check{al}oc.T(\check{r}, \grave{\sigma}, \check{\sigma}, \grave{al}oc, \check{al}oc) \wedge \mathsf{Res}(\acute{\sigma}(\check{r}, f), r, \check{\sigma}, \acute{\sigma}, \check{al}oc, \acute{al}oc)) \supset \\
\qquad [\mathsf{fsel}(a, f) : A :: T']
\end{array}$$

(Note the use of the existential quantification in T' to account for the intermediate store and result of evaluating a.) This theorem allows us to directly derive judgments about programs using fsel without expanding its definition.

We continue to extend and overload the remaining program constructors and derive corresponding "higher-level" rules. Using these rules, we successfully prove the examples given in Sec. 4.1–2 of [2]. We then derive an easier-to-use let rule before attempting two larger examples: the greatest common divisor program (Sec. 4.3 in [2]), and an original example based on the dining philosophers scenario. We consider these two examples in more detail after the new let rule.

4.1 Reversible Let Rule

As mentioned in Sec. 2 the let rule is not reversible. In particular, whenever we apply the let rule, we must decide on what information to lose. Since most programs use the let constructor extensively, this quickly becomes cumbersome.

Using our implementation, we can derive the following substitute for the let rule.

$$\begin{array}{l}
\forall a.\forall b.\forall A'.\forall T''.\forall A.\forall T.\forall U. \\
\quad [a : A :: T] \supset \\
\quad (\forall x.[x : A] \supset [b(x) : A' :: U(x)]) \supset \qquad \text{(wsq_let)} \\
\quad (T; U \equiv T'') \supset \\
\qquad [\mathsf{let}(a, b) : B :: T'']
\end{array}$$

This rule does not lose any information. In proof derivations, it is particularly useful because information loss can be postponed until later using an explicit application of the subsumption axiom.

4.2 Greatest Common Divisor

The gcd program from [2], can be written using notation closer to that of popular OO languages, as follows. It is a simple exercise to translate this program into our formal language.

$$\begin{array}{l}\mathsf{calc\_gcd} \stackrel{\text{def}}{=} \lambda y.\\ \quad \begin{array}{lll}\texttt{if} & (y.f < y.g) & \{y.g = y.g - y.f; y.m()\}\\ \texttt{else if} & (y.g < y.f) & \{y.f = y.f - y.g; y.m()\}\\ \texttt{else} & & \{y.f\}\end{array}\\ \mathsf{gcd} \stackrel{\text{def}}{=} \mathsf{obj}([f{=}\mathsf{nat}(1), g{=}\mathsf{nat}(1)], [m{=}\mathsf{calc\_gcd}])\end{array}$$

This program creates an object with one method m, such that if the fields have nonzero values a and b, invoking m will reduce both fields to the gcd of a and b. This is the intuition behind the formal specification given in [2]. To prove the formal specification statement in our logic, we strengthen the transition relation given in [2]. We can then prove that gcd satisfies the stronger specification. The subsumption axiom can be used to return to the original statement. We introduce the constant $\mathsf{gcd} : \mathsf{EE} \to \mathsf{EE} \to \mathsf{EE}$ and add axioms consistent with its interpretation as the gcd function over natural numbers. And so we define

$$\begin{array}{l}U_{\mathsf{gcd}}(y) \stackrel{\text{def}}{=} \lambda r.\lambda \grave{\sigma}, \acute{\sigma}.\lambda \grave{al\!loc}, \acute{al\!loc}.\\ \quad (1 \leq \grave{\sigma}(y,f) \wedge 1 \leq \grave{\sigma}(y,g)) \supset\\ \qquad r = \acute{\sigma}(y,f) \wedge r = \acute{\sigma}(y,g) \wedge\\ \qquad r = \mathrm{gcd}(\acute{\sigma}(y,f), \acute{\sigma}(y,g)) \wedge\\ \qquad 1 \leq \acute{\sigma}(y,f) \wedge 1 \leq \acute{\sigma}(y,g)\\ \mathsf{Spec}_{\mathsf{gcd}} \stackrel{\text{def}}{=} \mathsf{obj}([f = \mathsf{nat}, g = \mathsf{nat}], [m = \langle \mathsf{nat}, U_{\mathsf{gcd}} \rangle]) \ .\end{array}$$

Using these definitions, we can prove

$$\vdash [\mathsf{gcd} : \mathsf{Spec}_{\mathsf{gcd}} :: T_{\mathrm{triv}}]$$

where T_{triv} is a trivial transition relation.

4.3 Dining Philosophers

Object oriented languages have shown to be particularly suitable for writing simulations. In the next example, we consider a simulation in an OO language for a formalisation of the dining philosophers scenario. The formalisation we choose is based on that presented in Roscoe's book [9], where a general description of the scenario can be found. Our implementation follows Roscoe's observation that the important events that we should model are when the forks get picked up and put down. To make the example more managable, we only consider the case for three philosophers at the table.

We simulate the scenario by creating an object for each fork, and an object for each philosopher. The philophers interact with the forks by invoking their

methods. In our example, two of the philosphers pick up their forks "left then right" and one picks up his forks "right then left." The resulting system is known not to deadlock. We prove this using a suitable formalisation of "does not deadlock."

Here is code to create a fork object.

$$\begin{aligned}&\mathsf{Fork} \stackrel{\text{def}}{=} \mathsf{obj}([\mathsf{on\_table}{=}\mathsf{true}],\\ &\qquad [\mathsf{try\_pick\_up}{=}\lambda s.\ \mathtt{if}\ \ (s.\mathsf{on\_table})\ \{s.\mathsf{on\_table} = \mathsf{false}; \mathsf{true}\}\\ &\qquad\qquad\qquad\qquad \mathtt{else}\ \{\mathsf{false}\}\\ &\qquad \mathsf{put\_down}{=}\lambda s.s.\mathsf{on\_table} = \mathsf{true}; \mathsf{false}])\end{aligned}$$

A philosopher object invokes the try_pick_up method to pick up a fork. The method returns true after updating the fork object's state if this is possible. It returns false if the fork is not on the table.

We introduce the following definitions for creating the two types of philosophers.

$$\begin{aligned}&\mathsf{phil\_tick} \stackrel{\text{def}}{=} \lambda s.\ \mathtt{if}\ \ (s.\mathsf{n\_forks} == 0\ \ \mathtt{and}\ \ s.\mathsf{hungry})\ \{\\ &\qquad\qquad \mathtt{if}\ \ (s.\mathsf{fork1.try\_pick\_up}())\ \{s.\mathsf{n\_forks} = 1; \mathsf{false}\}\\ &\qquad\qquad \mathtt{else}\ \ \{\mathsf{false}\}\\ &\qquad\qquad \}\ \mathtt{else\ if}\ \ (s.\mathsf{n\_forks} == 1\ \ \mathtt{and}\ \ s.\mathsf{hungry})\ \{\\ &\qquad\qquad \mathtt{if}\ \ (s.\mathsf{fork2.try\_pick\_up}())\\ &\qquad\qquad\quad \{s.\mathsf{n\_forks} = 2; s.\mathsf{hungry} = \mathsf{false}; \mathsf{false}\}\\ &\qquad\qquad \mathtt{else}\ \ \{\mathsf{false}\}\\ &\qquad\qquad \}\ \mathtt{else\ if}\ \ (s.\mathsf{n\_forks} == 2)\ \{\\ &\qquad\qquad s.\mathsf{fork2.put\_down}(); s.\mathsf{n\_forks} = 1; \mathsf{false}\\ &\qquad\qquad \}\ \mathtt{else}\ \{\\ &\qquad\qquad s.\mathsf{fork1.put\_down}(); s.\mathsf{n\_forks} = 0; s.\mathsf{hungry} = \mathsf{true}; \mathsf{false}\\ &\qquad\qquad \}\end{aligned}$$

$$\mathsf{LRPhil} \stackrel{\text{def}}{=} \lambda \mathit{fork}_l, \mathit{fork}_r.\mathsf{Phil}(\mathit{fork}_l, \mathit{fork}_r)$$

$$\mathsf{RLPhil} \stackrel{\text{def}}{=} \lambda \mathit{fork}_l, \mathit{fork}_r.\mathsf{Phil}(\mathit{fork}_r, \mathit{fork}_l)$$

$$\begin{aligned}&\mathsf{Phil} \stackrel{\text{def}}{=} \lambda \mathit{fork}_1, \mathit{fork}_2.\\ &\qquad \mathsf{obj}([\mathsf{hungry}{=}\mathsf{true}, \mathsf{n\_forks}{=}\mathsf{nat}(0),\\ &\qquad\quad \mathsf{fork1}{=}\mathsf{var}(\mathit{fork}_1), \mathsf{fork2}{=}\mathsf{var}(\mathit{fork}_2)],\\ &\qquad\quad [\mathsf{tick}{=}\mathsf{phil\_tick}])\end{aligned}$$

A philospher has four internal states: (1) he is hungry and is holding no forks; (2) he is hungry and he is holding one fork; (3) he is no longer hungry[1] and is holding two forks; and (4) he is not hungry and holding one fork. Each state transition corresponds exactly to a fork being either picked up or put down.

[1] Here we assume that the philosopher instantaneously eats as soon as he picks up the second fork and so is no longer hungry. The point is that the event corresponding to a philosopher eating is not important with respect to deadlock considerations.

Finally, we put the whole system together as follows by creating a "table" object.

$$\begin{aligned}\mathsf{Table} \stackrel{\text{def}}{=} \ & \mathsf{let}(\mathit{fork}_1 = \mathsf{Fork}, \mathit{fork}_2 = \mathsf{Fork}, \mathit{fork}_3 = \mathsf{Fork},\\ & \mathit{phil}_1 = \mathsf{LRPhil}(\mathit{fork}_1, \mathit{fork}_2),\\ & \mathit{phil}_2 = \mathsf{LRPhil}(\mathit{fork}_2, \mathit{fork}_3),\\ & \mathit{phil}_3 = \mathsf{RLPhil}(\mathit{fork}_3, \mathit{fork}_1),\\ & \mathsf{obj}(\\ & \quad [\mathsf{f1}{=}\mathit{fork}_1, \mathsf{f2}{=}\mathit{fork}_2, \mathsf{f3}{=}\mathit{fork}_3,\\ & \quad \mathsf{p1}{=}\mathit{phil}_1, \mathsf{p2}{=}\mathit{phil}_2, \mathsf{p3}{=}\mathit{phil}_3],\\ & \quad [\mathsf{tick1}{=}\lambda s.s.\mathbf{p1}.\mathsf{tick}(),\\ & \quad \mathsf{tick2}{=}\lambda s.s.\mathsf{p2}.\mathsf{tick}(),\\ & \quad \mathsf{tick3}{=}\lambda s.s.\mathsf{p3}.\mathsf{tick}()]))\end{aligned}$$

The table should be considered as a "black box" with three buttons, one for each of the tick methods. To complete the simulation, one must compose this program with another program that plays out the possible traces of the system.

With the dining philosopher scenario simulated by these code fragments, we can prove that this system does not deadlock, which we will now formalise. We say that a philosopher is *blocked* whenever he needs to pick up a fork to perform a state transition but cannot (exactly when the fork in question is not on the table.) The system is *deadlocked* precisely when all the philosophers on the table are blocked.

Given the store σ, we can determine whether any particular philosopher is blocked by inspecting the values of the fields of the philosopher and its forks. To assist our intuitions, we define the following predicates. "Philosopher p is holding fork *fork*",

$$\begin{aligned}\mathsf{is\_holding} \stackrel{\text{def}}{=} \lambda p.\lambda \mathit{fork}.\lambda\sigma.\ & \sigma(p, \mathsf{n\_forks}) = 1 \wedge \mathit{fork} = \sigma(p, \mathsf{fork1}) \vee\\ & \sigma(p, \mathsf{n\_forks}) = 2 \wedge \mathit{fork} = \sigma(p, \mathsf{fork1}) \vee\\ & \sigma(p, \mathsf{n\_forks}) = 2 \wedge \mathit{fork} = \sigma(p, \mathsf{fork2})\end{aligned}$$

"Philosopher p is waiting for fork *fork*,"

$$\begin{aligned}\mathsf{waiting\_for} \stackrel{\text{def}}{=} & \lambda p.\lambda \mathit{fork}.\lambda\sigma.\\ & (\sigma(p, \mathsf{n\_forks}) = 0 \wedge \sigma(\sigma(p, \mathsf{fork1}), \mathsf{on\_table}) = \mathtt{ff}\\ & \quad \wedge \mathit{fork} = \sigma(p, \mathsf{fork1}))\\ & \vee (\sigma(p, \mathsf{n\_forks}) = 1 \wedge \sigma(\sigma(p, \mathsf{fork2}), \mathsf{on\_table}) = \mathtt{ff}\\ & \quad \wedge \mathit{fork} = \sigma(p, \mathsf{fork2}))\ .\end{aligned}$$

Assuming that $F(t)$ is the set of forks, and $P(t)$ the set of philosophers on table t, for $p \in P(t)$, the predicate "philosopher p is blocked,"

$$\mathsf{blocked} \stackrel{\text{def}}{=} \lambda t.\lambda p.\lambda\sigma.\exists_{F(t)} \mathit{fork}.\mathsf{waiting\_for}(p, \mathit{fork}, \sigma)$$

and "all philosophers are blocked,"

$$\mathsf{all\_blocked} \stackrel{\text{def}}{=} \lambda t.\lambda\sigma.\forall_{P(t)}p.\mathsf{blocked}(p,\sigma) \ .$$

One way to prove that our system does not deadlock is to use the following fact. Let $\prec$ be a total order such that $\mathit{fork}_1 \prec \mathit{fork}_2 \prec \mathit{fork}_3$. It is the case that the order in which the philosophers pick up their forks respects $\prec$. It is then straightforward to prove that the system does not deadlock[2].

For table t, order relation orel and store σ, if we define $\mathsf{InvTable}$ by

$$\begin{aligned}\mathsf{InvTable}(t,\mathit{orel},\sigma) \stackrel{\text{def}}{=}\ & \forall_{P(t)}p.\mathsf{InvPhil}(p,\mathit{orel},\sigma) \wedge \\ & (\forall_{F(t)}f.\sigma(f,\mathsf{on\_table}) = \mathtt{ff} \supset \\ & \quad \exists_{P(t)}p.\mathsf{is\_holding}(p,f,\sigma)) \wedge \\ & \mathsf{f\_p\_relationship} \wedge \mathsf{f\_distinct} \wedge \mathsf{p\_distinct}\end{aligned}$$

where $\mathsf{f\_p\_relationship}$ states that the fork fields of the philosophers point to the intended forks, $\mathsf{f\_distinct}$ and $\mathsf{p\_distinct}$ state that the fork and philosopher objects are pairwise distinct and

$$\begin{aligned}\mathsf{InvPhil}(p,\mathit{orel},\sigma) \stackrel{\text{def}}{=}\ & \mathit{orel}\,(\sigma(p,\mathsf{fork1}),\sigma(p,\mathsf{fork2})) \wedge \\ & (\mathsf{state}_0(p) \vee \mathsf{state}_1(p) \vee \mathsf{state}_2(p) \vee \mathsf{state}_3(p))\end{aligned}$$

where each $\mathsf{state}_j(p)$ states the values of the $\mathsf{n\_forks}$ and hungry fields of philosopher p at the corresponding state. It follows that if we define

$$\begin{aligned}\mathsf{Spec}_{\mathsf{Table}} \stackrel{\text{def}}{=} \mathsf{obj}(\ & [\mathsf{f1}{=}\mathsf{Spec}_{\mathsf{Fork}}, \mathsf{f2}{=}\mathsf{Spec}_{\mathsf{Fork}}, \mathsf{f3}{=}\mathsf{Spec}_{\mathsf{Fork}}, \\ & \mathsf{p1}{=}\mathsf{Spec}_{\mathsf{Phil}}, \mathsf{p2}{=}\mathsf{Spec}_{\mathsf{Phil}}, \mathsf{p3}{=}\mathsf{Spec}_{\mathsf{Phil}}], \\ & [\mathsf{tick1}{=}\mathsf{TR}_{\mathsf{tick}}, \mathsf{tick2}{=}\mathsf{TR}_{\mathsf{tick}}, \mathsf{tick3}{=}\mathsf{TR}_{\mathsf{tick}}])\end{aligned}$$

and

$$\mathsf{TR}_{\mathsf{tick}} \stackrel{\text{def}}{=} \lambda s, r.\lambda\grave{\sigma},\acute{\sigma}.\lambda\grave{\mathit{alloc}},\acute{\mathit{alloc}}.\mathsf{InvTable}(s,\grave{\sigma}) \supset \mathsf{InvTable}(s,\acute{\sigma})$$

we can prove in our logic,

$$\vdash [\mathsf{Table} : \mathsf{Spec}_{\mathsf{Table}} :: \lambda r.\lambda\grave{\sigma},\acute{\sigma}.\lambda\grave{\mathit{alloc}},\acute{\mathit{alloc}}.\mathsf{InvTable}(r,\prec,\acute{\sigma})] \ .$$

That is, $\mathsf{InvTable}$ is an *invariant* of the system. It is an invariant in the sense that it holds immediately after the table object is created, and it is invariant with respect to the actions of the three "buttons." Of course, $\mathsf{Spec}_{\mathsf{Fork}}$ and $\mathsf{Spec}_{\mathsf{Phil}}$ are specifications that are strong enough to describe the behaviour of fork and philosopher objects respectively.

Furthermore, we can prove, for table t, philosopher p, forks fork, fork' and store σ,

$$\mathsf{blocked}(p,\sigma) \supset \mathsf{is\_holding}(p,\mathit{fork},\sigma) \supset \sigma(p,\mathsf{fork1}) = \mathit{fork} \tag{1}$$

$$\mathsf{is\_holding}(p,\mathit{fork},\sigma) \supset \mathsf{waiting\_for}(p,\mathit{fork}',\sigma) \supset \sigma(p,\mathsf{fork2}) = \mathit{fork}' \tag{2}$$

$$\begin{aligned}&\mathsf{InvTable}(t,\mathit{orel},\sigma) \supset \forall_{F(t)}\mathit{fork}. \\ &\quad \sigma(\mathit{fork},\mathsf{on\_table}) = \mathtt{ff} \supset \exists_{P(t)}p.\mathsf{is\_holding}(p,\mathit{fork},\sigma)\end{aligned} \tag{3}$$

[2] This is in fact a special case of Roscoe's rules for avoiding deadlock in [9].

and

$$\mathsf{InvTable}(t, orel, \sigma) \supset \forall_{P(t)} p.\ orel\,(\sigma(p, \mathsf{fork1}), \sigma(p, \mathsf{fork2}))\ . \tag{4}$$

Assuming this, it is straight forward to prove the corollary

$$\mathsf{InvTable}(t, orel, \sigma) \supset \neg \mathsf{all\_blocked}(t, \sigma)\ , \tag{5}$$

as required.

5 Soundness

Similar to AL we have the following soundness property.

Theorem 1. *Assume that* $\emptyset, \emptyset \vdash a \rightsquigarrow v, \sigma'$ *is provable. For boolean* b*, if*

$$\vdash [\ulcorner a \urcorner : bool :: \lambda r.\lambda\grave{\sigma}, \acute{\sigma}.\lambda\grave{al\acute{l}oc}, \grave{al\acute{l}oc}.r = b]$$

then $v = b$.

We prove this, or rather an appropriate generalisation involving open programs and assumptions of the form $[x_i : A_i]$ where x_i are free program variables, by induction on the operational semantics, along the lines of the soundness proof in [2]. Complication arises from the fact that the predicates $[- : -]$ and $[- : - :: -]$ can appear in the transition relations, as can the constants for program constructions and specifications. We overcome this by assigning trivial meanings to these when they appear in transition relations. For example, $[- : - :: -]$ can be interpreted as constant true. The details will appear in an expanded version of this article.[3]

6 Conclusions and Related Work

Our implementation differs from other work not only because we use HOAS but also because we embed the logic in the metalogic directly. Primarily for the purpose of "language analysis," Nipkow et al.[4] have encoded an OO language in Isabelle/HOL[5] using a "deep embedding" for the syntax. Similarly, Honsell in [3] encodes the syntax of Dynamic Logic in a first-order style. Such encodings allow justification arguments to be given by induction over the syntax. In [7], Reddy presents an OO, Algol-like language IA$^+$ based on Reynold's Idealized Algol[8] and its specification logic. Language IA$^+$ uses HOAS and its specification logic is higher-order but programs can only create objects on the stack, not in the heap. In contrast, the language of AL creates objects in the heap (global store).

Our design decision to use HOAS has allowed us to quickly and succinctly implement a verification system since we inherit scoping rules and alpha conversion

[3] The originally submitted version of this article indicated a semantical soundness proof. In the meantime, we have realised that this more direct approach is possible.

from the metalanguage. Directly embedding the logical connectives results in a system that can take full advantage of the features of the underlying theorem prover.

The use of a theorem prover has shown to be invaluable for keeping track of the many assumptions that occur during the verification of nontrivial examples. However, in practice, it is still difficult to verify, using LEGO, even small examples such as those presented in this article. One often gets too involved trying to prove "trivial" subgoals. Since our implementation does not use any specific features of LEGO nor constructive type theory, we may find that using a theorem prover with more automation, such as PVS or Isabelle/HOL, would result in a more usable verification tool.

Acknowledgements

The authors would like to thank David von Oheimb, Jan Jürjens and anonymous referees for their comments on earlier versions of this article.

References

[1] Martín Abadi and Rustan Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, New York, N.Y., 1997.

[2] Martín Abadi and Rustan Leino. A logic of object-oriented programs. SRC Research Reports SRC-161, Compaq SRC, September 1998. Revised version of [1].

[3] F. Honsell and M. Miculan. A natural deduction approach to dynamic logic. *Lecture Notes in Computer Science*, 1158, 1996.

[4] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*. IOS Press, 2000.

[5] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[6] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[7] U. S. Reddy. Objects and classes in algol-like languages. In *Foundations of Object-oriented Languages*, January 1998.

[8] John C. Reynolds. Idealized algol and its specification logic. In Danielle Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

[9] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.