Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 95–108
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A List-machine Benchmark for Mechanized Metatheory
(Extended Abstract)

Andrew W. Appel[1]

Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08540, USA and INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay, France

Xavier Leroy[2]

INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay, France

Abstract

We propose a benchmark to compare theorem-proving systems on their ability to express proofs of compiler correctness. In contrast to the first POPLmark, we emphasize the connection of proofs to compiler implementations, and we point out that much can be done without binders or alpha-conversion. We propose specific criteria for evaluating the utility of mechanized metatheory systems; we have constructed solutions in both Coq and Twelf metatheory, and we draw conclusions about those two systems in particular.

Keywords: Theorem proving, proof assistants, program proof, compiler verification, typed machine language, metatheory, Coq, Twelf.

1 How to evaluate mechanized metatheories

The POPLmark challenge [3] aims to compare the usability of several automated proof assistants for mechanizing the kind of programming-language proofs that might be done by the author of a POPL paper, with benchmark problems "chosen to exercise many aspects of programming languages that are known to be difficult to formalize." The first POPLmark examples are all in the theory of $F_{<:}$ and emphasize the theory of binders (e.g., alpha-conversion).

Practitioners of machine-checked proof about real compilers have interests that are similar but not identical. We want to formally relate machine-checked proofs to actual implementations, not particularly to LaTeX documents. Furthermore,

[1] Email: appel@princeton.edu

[2] Email: Xavier.Leroy@inria.fr

perhaps it is the wrong approach to "exercise aspects ... that are known to be difficult to formalize." Binders and $\alpha\beta$-conversion are certainly useful, but they are not essential for proving real things about real compilers, as demonstrated in several substantial compiler-verification projects [9,10,13,6,7,8]. If machine-checked proof is to be useful in providing guarantees about real systems, let us play to its strengths, not to its weaknesses.

Therefore we have designed a down-to-earth example of machine-checked metatheory, closer to the semantics of typed assembly languages. It is entirely first-order, without binders or the need for alpha conversion. We specify the Structured Operational Semantics (SOS) of a simple pointer machine (cons, car, cdr, branch-if-nil) and we present a simple type system with constructors for list-of-τ and nonempty-list-of-τ. The benchmark explicitly covers the relationship of proofs about a *type system* to proofs about an executable *type checker.*

The challenge is to represent the type system, prove soundness of the type system, represent the type-checking algorithm, and prove that the algorithm correctly implements the type system. We have implemented the benchmark both in Coq and in Twelf metatheory, and we draw conclusions about the usability of these two systems.

We lack space to present here, but discuss in the full paper[1],

- how the needs of implementors (of provably correct compilers and provably sound typecheckers) differ from the needs of POPL authors addressed by the first POPLmark;
- a specification of the entire problem down to details such as the recommended ASCII names for predicates and inference rules;
- details of our Coq and Twelf solutions;
- more details about which subtasks were easy or difficult in Coq and Twelf;
- how easy it is to learn Twelf and Coq given the available documentation.

As well as a benchmark, the list machine is a useful exercise for students learning Coq or Twelf; we present the outlines of our solutions (with proofs deleted) on the Web [2].

2 Specification of the problem

Machine syntax. Machine values A are cons cells and nil.

$$a : A ::= \mathrm{nil} \mid \mathrm{cons}(a_1, a_2)$$

The instructions of the machine are as follows:

$$\begin{array}{lll} \iota : I ::= & & \\ & \mathbf{jump}\ l & \text{(jump to label } l\text{)} \\ \mid & \mathbf{branch\text{-}if\text{-}nil}\ v\ l & \text{(if } v = \mathit{nil} \text{ go to } l\text{)} \\ \mid & \mathbf{fetch\text{-}field}\ v\ 0\ v' & \text{(fetch the head of } v \text{ into } v'\text{)} \\ \mid & \mathbf{fetch\text{-}field}\ v\ 1\ v' & \text{(fetch the tail of } v \text{ into } v'\text{)} \\ \mid & \mathbf{cons}\ v_0\ v_1\ v' & \text{(make a cons cell in } v'\text{)} \\ \mid & \mathbf{halt} & \text{(stop executing)} \\ \mid & \iota_0\ ;\ \iota_1 & \text{(sequential composition)} \end{array}$$

In the syntax above, the metavariables v_i range over variables; the variables themselves $\mathbf{v}_i$ are enumerated by the natural numbers. Similarly, metavariables l_i range over program labels $\mathbf{L}_i$.

A program is a sequence of instruction blocks, each preceded by a label.

$$p : P ::= \quad \mathbf{L}_n : \iota;\ p \quad \mid \quad \mathbf{end}$$

Operational semantics. Machine states are pairs (r, ι) of the current instruction ι and a store r associating values to variables. We write $r(v) = a$ to mean that a is the value of variable v in r, and $r[v := a] = r'$ to mean that updating r with the binding $[v := a]$ yields a unique store r'. The semantics of the machine is defined by the small-step relation $(r, \iota) \overset{p}{\mapsto} (r', \iota')$ defined by the rules below, and the Kleene closure of this relation, $(r, \iota) \overset{p}{\mapsto}^* (r', \iota')$.

$$\frac{}{(r,\ (\iota_1;\iota_2);\iota_3) \overset{p}{\mapsto} (r,\ \iota_1;(\iota_2;\iota_3))}$$

$$\frac{r(v) = \mathrm{cons}(a_0, a_1) \quad r[v' := a_0] = r'}{(r, (\mathbf{fetch\text{-}field}\ v\ 0\ v'; \iota)) \overset{p}{\mapsto} (r', \iota)} \qquad \frac{r(v) = \mathrm{cons}(a_0, a_1) \quad r[v' := a_1] = r'}{(r, (\mathbf{fetch\text{-}field}\ v\ 1\ v'; \iota)) \overset{p}{\mapsto} (r', \iota)}$$

$$\frac{r(v_0) = a_0 \quad r(v_1) = a_1 \quad r[v' := \mathrm{cons}(a_0, a_1)] = r'}{(r, (\mathbf{cons}\ v_0\ v_1\ v'; \iota)) \overset{p}{\mapsto} (r', \iota)}$$

$$\frac{r(v) = \mathrm{cons}(a_0, a_1)}{(r, (\mathbf{branch\text{-}if\text{-}nil}\ v\ l; \iota)) \overset{p}{\mapsto} (r, \iota)} \qquad \frac{r(v) = \mathrm{nil} \quad p(l) = \iota'}{(r, (\mathbf{branch\text{-}if\text{-}nil}\ v\ l; \iota)) \overset{p}{\mapsto} (r, \iota')}$$

$$\frac{p(l) = \iota'}{(r,\ \mathbf{jump}\ l) \overset{p}{\mapsto} (r, \iota')}$$

A program p runs, that is, $p \Downarrow$, if it executes from an initial state to a final state. A state is an initial state if variable $\mathbf{v}_0 = \mathrm{nil}$ and the current instruction is the one at $\mathbf{L}_0$. A state is a final state if the current instruction is **halt**.

$$\frac{\{\,\}[\mathbf{v}_0 := \mathrm{nil}] = r \quad p(\mathbf{L}_0) = \iota \quad (r, \iota) \overset{p}{\mapsto}^* (r', \mathbf{halt})}{p \Downarrow}$$

It is useful for a benchmark for machine-verified proof to include explicit ASCII names for each constructor and rule. Our full specification [1] does that.

A type system. We will assign to each live variable at each program point a list type. To guarantee safety of certain operations, we provide refinements of the list type for nonempty lists and for empty lists. In particular, the **fetch-field** operations demand that their list argument has nonempty list type, and the **branch-if-nil**

operation refines the type of its argument to empty or nonempty list, depending on whether the branch is taken.

$$
\begin{array}{lll}
\tau : T ::= & & \\
 & \text{nil} & \text{(singleton type containing nil)} \\
 | & \text{list}\,\tau & \text{(list whose elements have type } \tau) \\
 | & \text{listcons}\,\tau & \text{(non-nil list of } \tau)
\end{array}
$$

An environment Γ is an type assignment of types to a set of variables. We define the obvious subtyping $\tau \subset \tau'$ among the various refinements of the list type, using a common set of first-order syntactic rules, easily expressible in most mechanized metatheories. We extend subtyping widthwise and depthwise to environments.

We define the least common supertype $\tau_1 \sqcap \tau_2 = \tau_3$ of two types τ_1 and τ_2 as the smallest τ_3 such that $\tau_1 \subset \tau_3$ and $\tau_1 \subset \tau_2$.

In the operational semantics, a program is a sequence of labeled basic blocks. In our type system, a *program-typing*, ranged over by Π, associates to each program label a variable-typing environment. We write $\Pi(l) = \Gamma$ to indicate that Γ represents the types of the variables on entry to the block labeled l.

Instruction typing. Individual instructions are typed by a judgment $\Pi \vdash_{\text{instr}} \Gamma\{\iota\}\Gamma'$. The intuition is that, under program-typing Π, the Hoare triple $\Gamma\{\iota\}\Gamma'$ relates precondition Γ to postcondition Γ'.

$$\frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi \vdash_{\text{instr}} \Gamma'\{\iota_2\}\Gamma''}{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1;\iota_2\}\Gamma''}$$

$$\frac{\Gamma(v) = \text{list}\,\tau \quad \Pi(l) = \Gamma_1 \quad \Gamma[v := \text{nil}] = \Gamma' \quad \Gamma' \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}(v : \text{listcons}\,\tau,\ \Gamma')}$$

$$\frac{\Gamma(v) = \text{listcons}\,\tau \quad \Pi(l) = \Gamma_1 \quad \Gamma[v := \text{nil}] = \Gamma' \quad \Gamma' \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}\Gamma}$$

$$\frac{\Gamma(v) = \text{nil} \quad \Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{branch-if-nil}\ v\ l\}\Gamma}$$

$$\frac{\Gamma(v) = \text{listcons}\,\tau \quad \Gamma[v' := \tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field}\ v\ 0\ v'\}\Gamma'} \qquad \frac{\Gamma(v) = \text{listcons}\,\tau \quad \Gamma[v' := \text{list}\,\tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{fetch-field}\ v\ 1\ v'\}\Gamma'}$$

$$\frac{\Gamma(v_0) = \tau_0 \quad \Gamma(v_1) = \tau_1 \quad (\text{list}\,\tau_0) \sqcap \tau_1 = \text{list}\,\tau \quad \Gamma[v := \text{listcons}\,\tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{cons}\ v_0\ v_1\ v\}\Gamma'}$$

Block typing. A *block* is an instruction that does not (statically) continue with another instruction, because it ends with a jump.

$$\frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi;\Gamma' \vdash_{\text{block}} \iota_2}{\Pi;\Gamma \vdash_{\text{block}} \iota_1;\iota_2} \qquad \frac{\Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi;\Gamma \vdash_{\text{block}} \textbf{jump}\ l}$$

Program typing. We write $\models_{\text{prog}} p : \Pi$ and say that a program p has program-typing Π if for each labeled block $l : \iota$ in p, the block ι has the precondition $\Pi(l) = \Gamma$ given in Π, that is, $\Pi;\Gamma \vdash_{\text{block}} \iota$. Moreover, we demand that $\Pi(\mathbf{L}_0) = \mathbf{v}_0 : \text{nil},\ \{\}$ and that every label l declared in Π is defined in p.

Type system vs. type checker. We have presented some relations defined by derivation rules and some defined informally. This is a bit sloppy, especially where a derivation rule refers to an informally defined relation; any solution to the benchmark must formalize this. We will use the notation $\models_{\text{prog}} p : \Pi$ to mean that program p has type Π in the (not necessarily algorithmic) type system, and the notation $\vdash_{\text{prog}} p : \Pi$ to mean that $p : \Pi$ is derived in some algorithmic type-checker. The full paper [1] outlines two such algorithmic type-checkers. One is written in pseudo-code and corresponds to a type-checker implemented in imperative or functional style. The other refines the derivation rules given above to make them fully syntax-directed and therefore amenable to an implementation as a logic program.

Sample program. The following list-machine program has three basic blocks. Variable $\mathbf{v}_0$ is initialized to nil as prescribed by the operational semantics. Block 0 initializes $\mathbf{v}_1$ to the list cons(nil, cons(nil, nil)) and jumps to block 1. Block 1 is a loop that, while $\mathbf{v}_1$ is not nil, fetches the tail of $\mathbf{v}_1$ and continues. The last instruction of block 1 is actually dead code (never reached). Block 2 is the loop exit, and halts.

$$\begin{array}{l}
p_{\text{sample}} = \\
\mathbf{L}_0 : \ \textbf{cons}\ \mathbf{v}_0\ \mathbf{v}_0\ \mathbf{v}_1;\ \textbf{cons}\ \mathbf{v}_0\ \mathbf{v}_1\ \mathbf{v}_1;\ \textbf{cons}\ \mathbf{v}_0\ \mathbf{v}_1\ \mathbf{v}_1;\ \textbf{jump}\ \mathbf{L}_1; \\
\mathbf{L}_1 : \ \textbf{branch-if-nil}\ \mathbf{v}_1\ \mathbf{L}_2;\ \textbf{fetch-field}\ 1\ \mathbf{v}_1\ \mathbf{v}_1;\ \textbf{branch-if-nil}\ \mathbf{v}_0\ \mathbf{L}_1;\ \textbf{jump}\ \mathbf{L}_2; \\
\mathbf{L}_2 : \ \textbf{halt}; \\
\textbf{end}
\end{array}$$

The program is well-typed with

$$\Pi_{\text{sample}} = \mathbf{L}_0 : (\mathbf{v}_0 : \text{nil},\ \{\}),\ \mathbf{L}_1 : (\mathbf{v}_0 : \text{nil},\ \mathbf{v}_1 : \text{list nil},\ \{\}),\ \mathbf{L}_2 : \{\},\ \{\}$$

3 Mechanization tasks

Implementing the "list-machine" benchmark in a mechanized metatheory (MM) comprises the following tasks:

1. Represent the operational semantics in the MM.
2. Derive the fact that $p_{\text{sample}} \Downarrow$. *The MM should conveniently simulate execution of small examples, so the user can debug the SOS and get an intuitive feel for its expressiveness.*

Soundness of a type system.

3. Represent the type system in the MM (define enough notation to represent the formula $\models_{\text{prog}} p : \Pi$ and inference rules from which type-soundness could be proved).
4. Represent in the MM an algorithm for least-common-supertype, that is, the computation $\tau_1 \sqcap \tau_2 = \tau_3$ producing τ_3 from inputs τ_1 and τ_2.
5. Using the type system, derive the fact that $\models_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$. *The MM should conveniently simulate type-checking of small examples, so the user can debug the type system and get a feel for its expressiveness.*
6. Represent the statement of the defining properties of least common supertypes, e.g., $\tau_1 \sqcap \tau_2 = \tau_3 \ \Rightarrow\ \tau_1 \subset \tau_3$.
7. Prove that the $\sqcap$ algorithm enjoys these properties.

8. Represent the statement of a soundness theorem for the type system. The informal statement of soundness is, "a well-typed program will not get stuck." A program state is not stuck if it *steps or halts*:

$$\frac{\models_{\text{prog}} p : \Pi \quad \text{initial}(p, r, \iota) \quad (r, \iota) \overset{p}{\mapsto}{}^{*} (r', \iota')}{(\exists r'', \iota''.\, (r', \iota') \overset{p}{\mapsto} (r'', \iota'')) \vee \iota' = \textbf{halt}} \text{ soundness}$$

9. Prove the soundness theorem. The full paper [1] outlines the principal lemmas of this proof, which is a standard argument by *type preservation* and *progress.*

Efficient type-checking algorithm.

10. Represent an asymptotically efficient type-checking algorithm $\vdash_{\text{prog}} p : \Pi$ in the MM. By efficient we mean that an N-instruction program with M live variables should type-check in $O(N \log M)$ time.
11. Using the type-checking algorithm, calculate $\vdash_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$. *The MM should simulate execution of algorithms on small inputs.*
12. Prove that the type-checking algorithm terminates on any program.
13. Demonstrate the type-checker on large-scale examples with good performance. Typically this will be done through an automatic translation to Prolog or ML which is then compiled by an optimizing compiler.
14. Prove that $\vdash_{\text{prog}} p : \Pi$ implies $\models_{\text{prog}} p : \Pi$. *That is, the type-checker soundly implements the type system.*

Writing the paper.

15. Use an automatic tool to generate readable LaTeX formulas for the SOS rules, the typing rules, and the statements of (not the proofs of) the least-common-supertype lemmas and soundness theorems. *Klein and Nipkow [6] demonstrate this very nicely in the Isabelle/HOL formalization of a Java subset compiler.*

4 A proof in Twelf metatheory

The Twelf system[12] is an implementation of the Edinburgh Logical Framework (LF). One can represent the operators of a logic as type constructors in LF, and proofs in that logic as terms in LF, and one can do proof-checking by type-checking the terms (considering them as derivations).

In Twelf one can prove *theorems* (proofs *in* a logic), or *metatheorems* (proofs *about* a logic). Either approach could be used for our benchmark. Our solution uses the usual approach in Twelf, which is metatheoretic.

In this case the logics in question are our operational semantics and our type system, and the metatheorem to be proved is type soundness: that is, if one can combine the inference rules of the type system to produce a derivation of type-checking, then it must be possible to combine the inference rules of the SOS to produce (only) non-stuck derivations of execution.

This approach is aggressively syntactic. Instead of saying that p is a mapping from labels to instructions, we give syntactic constructions that (we claim) represent such a mapping. One consequence of this style is that our $\models_{\text{prog}} p : \Pi$ is not just a semantic relation, but a syntactically derivable one expressed as Horn clauses. By carefully structuring the Horn clauses that define our relations so that we can identify "input" and "output" arguments, we can ensure that the logic-programming interpretation of our clauses is actually an algorithm. This input-output organization can be specified and mechanically checked in Twelf via `%mode` declarations. Our type system is then directly executable in Twelf.

Each clause in Twelf is named. When Twelf traces out, via Prolog-style backtracking, one or more derivations of a result by the successful application of clauses, it builds as well a derivation tree for each derivation.

In LF, one can compute as well on the derivation trees themselves. Suppose we write another Prolog program (set of clauses) that takes as input a derivation tree for type-checking, and produces as output a derivation tree for safe (non-stuck) execution. If this program is *total* (that is, terminates successfully on any input) then we have constructively proved that any well typed program is safe.

To reason about this meta-program, we use (machine-checked) `%mode` declarations to explain what are the inputs and outputs of the derivation transformer. We also use (machine-checked) `%total` declarations to ensure that our meta-program has covered all the cases that may arise, and that our meta-program does not infinite-loop. We give an example of such a proof in section 6, items 6 and 7.

Twelf has an amazing economy of features. One does not have to learn a module system—because there is none—one just uses naming conventions on all one's identifiers. One does not have to learn how to use large libraries of lemmas and tactics, because there are no libraries of lemmas and tactics: but such libraries would not be so useful, because Twelf has few abstraction features, and no polymorphism. All proofs are done with the simple mechanism of proving the totality of metaprograms. There's a calculated gamble here: In return for the benefit of proving everything in one simple style, and rarely having to translate between abstractions, one trades away many things: there are some theorems that this notation cannot even express (because the quantifiers are nested too deep, for example); and there are some things that are provable but in a contrived way (expressing semantic properties only with inductive syntactic constructors), as illustrated below.

Our Twelf proof starts by defining inductively the notion of equalities and inequalities on natural numbers, labels, variables, type structure, and term structures. We give syntactic characterizations of well-formed environments (i.e., that do not map the same variable twice).

Sometimes it is tricky to make a properly inductive syntactic definition of a semantic property. For example, consider environment subtyping, semantically $\Gamma_1 \subset_{\mathrm{env}} \Gamma_2 \;\equiv\; \forall v.\; v \in \operatorname{dom}\Gamma_2 \;\Rightarrow\; (v \in \operatorname{dom}\Gamma_1 \;\wedge\; \Gamma_1(v) \subset \Gamma_2(v))$.

An "obvious" "inductive" definition uses the syntactic rules,

$$\frac{}{\Gamma \subset \{\,\}}\,a_1 \qquad \frac{\Gamma_1(v) = \tau' \quad \tau' \subset \tau \quad \Gamma_1 \subset_{\mathrm{env}} \Gamma_2}{\Gamma_1 \;\subset_{\mathrm{env}}\; v : \tau,\; \Gamma_2}\,a_2$$

The induction is (supposedly) over the size of the term to the right of the $\subset_{\mathrm{env}}$ symbol. However, this definition is not sufficiently inductive for useful properties (transitivity, reflexivity) to be provable—at least, we were not able to prove them.

The problem appears to be that Γ_1 does not decrease in rule a_2.

The following definition is properly inductive—we use Γ' instead of Γ_1 in the premise of rule b_2. Proving transitivity and reflexivity from this definition is easy; the difficulty is to avoid wasting time with the pseudo-inductive definition above.

$$\frac{}{\Gamma \subset_{\text{env}} \{\}}\, b_1 \qquad \frac{\Gamma_1 \doteq (v : \tau',\ \Gamma') \quad \tau' \subset \tau \quad \Gamma' \subset_{\text{env}} \Gamma_2}{\Gamma_1\ \subset_{\text{env}}\ v : \tau,\ \Gamma_2}\, b_2$$

5 A proof in Coq

The Coq system [5,4] is a proof assistant based on the Calculus of Inductive Constructions. This logic is a variant of type theory, following the “propositions-as-types, proofs-as-terms” paradigm, enriched with built-in support for inductive and coinductive definitions of predicates and data types.

From a user’s perspective, Coq offers a rich specification language to define problems and state theorems about them. This language includes (1) constructive logic with all the usual connectives and quantifiers; (2) inductive definitions via inference rules and axioms (as in Twelf’s meta-logic); (3) a pure functional programming language with pattern-matching and structural recursion (in the style of ML or Haskell).

For the list-machine benchmark, we used a combination of all three specification styles, following common practice in research papers on type systems. The inference rules for operational semantics and the type systems are transcribed directly as inductive definitions. Operations over stores, environments and program-typing, as well as least common supertypes and the type-checking algorithm are presented as functions. Finally, subtyping between environments $\Gamma \subset \Gamma'$ is defined by the propositional formula

$$\forall v, \forall t',\ \Gamma'(v) = t' \Rightarrow \exists t,\ \Gamma(v) = t \wedge t \subset t'$$

Unlike Twelf’s meta-theory, the logic of Coq provides rich forms of polymorphism. This enabled us to factor out the treatment of stores, environments, and program-typing by reusing an efficient, polymorphic implementation of finite maps as radix-2 search trees developed earlier by Leroy as part of the Compcert project [8].

6 Comparison of mechanized proofs

	Task	Twelf	Coq	
1.	Operational Semantics	126	98	lines
2.	Derive $p \Downarrow$	1	8	
3.	Type system $\models_{\text{prog}} p : \Pi$	167	130	
4.	$\sqcap$ algorithm	*	*	
5.	Derive $\models_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$	1	no	
6.	State properties of $\sqcap$	12	13	
7.	Prove properties of $\sqcap$	114	21	
8.	State soundness theorem	29	15	
9.	Prove soundness of $\models_{\text{prog}} p : Pi$	2060	315	
10.	Efficient algorithm	22	145	
11.	Derive $\vdash_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$	1	1	
12.	Prove termination of $\vdash_{\text{prog}} p : \Pi$	18	0	
13.	Scalable type-checker	yes	yes	
14.	Prove soundness of $\vdash_{\text{prog}} p : Pi$	347	141	
15.	Generate LaTeX	no	no	

We have implemented those tasks that are implementable in both the Twelf (metatheory) and Coq systems. The number of lines of code required is summarized in the table above. Total parsing and proof-checking time[3] was 0.558 seconds real time for Twelf, 2.622 seconds for Coq.

1. Operational semantics. Both Twelf and Coq make it easy and natural to represent inductive definitions of the kind found in SOS. In Coq one also has the choice of representing operations over mappings (e.g., lookup and update in stores) either as relations (defined by inductive predicates) or as functions (defined by recursion and pattern-matching).

2. Derive $p \Downarrow$. Twelf makes it very easy to interpret inductive definitions as logic programs. Therefore this task was trivial in Twelf. Coq does not provide a general mechanism to execute inductive definitions. However, the rules for the operational semantics were simple enough that (after some experimentation) we could use the proof search facilities of Coq (the `eauto` tactic) as a poor man's logic program interpreter. A more general method to execute inductive definitions in Coq, which we implemented also, is to define an execution function (61 lines), prove its correctness with respect to the inductive definition (35 lines), then execute the function. (Evaluation of functional programs is supported natively by Coq.)

3. Represent the type system. Easy and natural in both Twelf and Coq (with, as before, the choice in Coq of using the functional presentation of operations over mappings).

4. Least-upper-bound algorithm. Because the "type system" represented in Twelf is most straightforwardly done as a constructive algorithm, this was already done as part of task 3 in our Twelf representation. In Coq, while the type system itself is not algorithmic, we chose to specify the least-upper bound operation as a function from pairs of types to types. Therefore, the algorithm to compute least-

[3] Dell Precision 360, Linux, 2.8 GHz Pentium 4, 1GB RAM, 512kB cache.

upper bounds was already done as part of task 3 in the Coq development as well.

5. Derive an example of type-checking. Trivial to do in Twelf, by running the type system as a logic program. Not directly possible in Coq because the specification of the type system is not algorithmic: it uses universal quantification over all variables to specify environment subtyping.

6. State properties of least-upper-bound. Entirely straightforward in Coq. For example, here are the Coq statements of these properties:

```
Lemma lub_comm:          forall t1 t2, lub t2 t1 = lub t1 t2.
Lemma lub_subtype_left:  forall t1 t2, subtype t1 (lub t1 t2).
Lemma lub_subtype_right: forall t1 t2, subtype t2 (lub t1 t2).
Lemma lub_least:         forall t1 t3, subtype t1 t3 ->
          forall t2, subtype t2 t3 -> subtype (lub t1 t2) t3.
```

The correspondence with the mathematical statements of these properties is obvious.

In Twelf, stating the properties of least-upper-bound must be done in a way that seems artificial at first, but once learned is reasonably natural. The lemma

$$\frac{\tau_1 \sqcap \tau_2 = \tau_3}{\tau_1 \subset \tau_3}\ \text{lub-subtype-left}$$

is represented as a logic-programming predicate,

```
lub-subtype-left: lub T1 T2 T3 -> subtype T1 T3 -> type.
```

which transforms a derivation of `lub T1 T2 T3` into a derivation of `subtype T1 T3`. The "proof" will consist of logic-programming clauses over this predicate. To be a "proof" of the property we want, we will have to demonstrate (to the satisfaction of the metatheory, which checks our claims) that our clauses have the following properties:

`%mode lub-subtype-left +P1 -P2.` The *modes* of a logic program specify which arguments are to be considered inputs (`+`) and which are outputs (`-`). Formally, given any ground term (i.e., containing no logic variables) `P1` whose type is `lub T1 T2 T3`, our clauses (if they terminate) must produce outputs `P2` of type `subtype T1 T3` that are also ground terms.

`%total P1 (lub-subtype-left P1 P2).` We ask the metatheorem to check our claim that no execution of lub-subtype-left can infinite-loop: it must either fail or produce a derivation of `subtype T1 T3`; *and* we check the claim that the execution never fails (that all cases are covered). The use of `P1` in two places in our `%total` declaration is (in some sense) mixing the thing to be proved with part of the proof: we indicate that the induction should be done over argument 1 of lub-subtype-left, not argument 2.

7. Prove properties of least-upper-bound. In Twelf this is done by writing logic-programming clauses that satisfy all the requirements listed above. For example, the following 9 clauses will do it:

```
-: lub-subtype-left lub-refl subtype-refl.
-: lub-subtype-left lub-1 subtype-refl.
-: lub-subtype-left (lub-2 P1) (subtype-list P2) <-
              lub-subtype-left P1 P2.
-: lub-subtype-left (lub-2b P1) (subtype-listcons P3) <-
              lub-subtype-left P1 P3.
-: lub-subtype-left (lub-3 P1) (subtype-list P2) <-
              lub-subtype-left P1 P2.
-: lub-subtype-left lub-4 subtype-nil.
-: lub-subtype-left lub-5 subtype-nil.
-: lub-subtype-left lub-6 (subtype-listcons subtype-refl).
-: lub-subtype-left (lub-7 P1) (subtype-listmixed P2) <-
              lub-subtype-left P1 P2.
```

These are not clauses of a type-checker, they are clauses *about* a type-checker, and serve only to "prove" the `%mode` and `%total` declarations.

In Coq, the proofs are done interactively by constructing proof scripts. For example, the proof of `lub_subtype_left` is:

```
induction t1; destruct t2; simpl; auto; rewrite IHt1; auto.
```

which corresponds to doing an induction on the structure of the first type `t1`, then a case analysis on the second type `t2`, then some equational reasoning.

There are 6 separate steps to the Coq proof, each takes just two or three tokens to write, and each takes some thought from the user. On the other hand, each of the 9 clauses of the Twelf proof, ranging in size from 6 to 16 tokens, also takes some thought. The time or effort required to build a proof is not necessarily proportional to the token count, but we report what measures we have.

8. State soundness theorem for the type system. In Coq, the statement is just ordinary mathematics. In Twelf, this is done, as above, by writing a logical predicate that relates a derivation of type-checking to a derivation of runs-or-halts, and then making the appropriate `%mode` and `%total` claims for the Twelf system to check.

9. Prove soundness of the type system. Writing such a logic program in Twelf takes more than 2000 lines; our full paper [1] explains this proof in more detail. The Coq proof of soundness is about 7 times shorter (300 lines). There are several reasons for Coq's superiority over Twelf here. The first is Coq's proof automation facilities, which were very effective for many of the intermediate proofs: once we indicated manually the structure of the inductions, Coq's proof search tactics were often able to derive automatically the conclusion from the hypotheses. A second reason is the use of non-algorithmic specifications, especially for environment subtyping, which are simpler to reason about. The last reason is the ability to reuse basic properties over mappings, such as the so-called "good variables" properties, instead of proving them over and over again.

Twelf lacks the ability to create and re-use abstract data types, so many clauses of the program and proof must be copied and edited. Twelf has some proof automation—the `%total` declaration calculates the structural induction automatically, and (if it fails) prints a report detailing the missing cases—but it does not automate the case analysis. [4]

10. Asymptotically efficient algorithm. In Twelf, the most straightforward representation of the type system, when run as an algorithm, takes quadratic time. This is because the rules for looking up labels in global environments Π involve a search of the length of Π for each lookup. In any Prolog system that permits the efficient dynamic assertion of new clauses, one can do lookup in constant time (the Prolog system uses hashing internally). Twelf supports dynamic clauses, so we can write a nice linear-time "type-checker" as a new logic program, reusing many of the Horn clauses that constitute the "type system."

In Coq, the type-checker is defined as a function from program typing and programs to booleans. Our solution uses intermediate functions for checking environment subtyping and for type-checking instructions and blocks. These functions return option types to signal typing errors, which are propagated in a monadic style. To avoid an n^2 algorithm, we represent environments and program typing as finite maps implemented by radix-2 search trees. Therefore, the typing algorithm has $O(n \log n)$ complexity.

11. Simulate the new algorithm. This is a trivial matter both in Twelf and in Coq. In Twelf, once again, we perform a one-line query in the logic-program interpreter. In Coq, we simply request the evaluation of a function application (of the type-checker to the sample program and program typing), which is also one line.

12. Prove termination of the type-checker. Twelf has substantial automated support for doing proofs of termination of logic programs (such as the type-checker) where the induction is entirely structural. This task was very easy in Twelf.

In Coq, this task was even easier: all functions definable in Coq are guaranteed to terminate (in particular, all recursions must be either structural or well-founded by Noetherian induction), so there was nothing to prove for this task.

13. Industrial-strength type-checker. Coq has a facility to automatically generate Caml programs from functions expressed in Coq. Automatic extraction of Caml code from the Coq functional specification of the type-checker produces code that is close to what a Caml programmer would write by hand if confined to the purely functional subset of the language.

Similarly, Twelf programs (such as our type-checker) that don't use higher-order abstract syntax can be automatically translated to Prolog, and those that use HOAS can be automatically translated to lambda-Prolog. There are many efficient Prolog compilers in the world, and there is one efficient lambda-Prolog compiler.

14. Prove soundness of type-checker. Straightforward (though a bit tedious) both in Twelf and in Coq. Again, Coq's proof automation facilities result in a significantly shorter proof (3 times shorter than the Twelf proof).

[4] Supplying the case analysis automatically will be the job of the Twelf metatheorem prover. Unfortunately, it appears that the metatheorem prover does not work; the Twelf manual says, "The theorem proving component of Twelf is in an even more experimental stage and currently under active development" [11] and every version of the manual since 1998 contains this identical sentence. One doubts whether the last two words are accurate.

15. Generate LaTeX. Although both Coq and Twelf have facilities for generating LaTeX, neither has a facility that is sufficiently useful for the purposes of this benchmark.

7 Conclusion

Proofs of semantic properties of operational specifications can be aggressively "semantic," meaning that they avoid *all* proof-theoretic induction over syntax; denotational-semantic approaches and logical-relations models have this flavor. We have not discussed these approaches in this paper, but they can be successfully mechanized in Coq, in Isabelle/HOL, or in an object logic embedded in Twelf; however, it does not seem natural to mechanize semantic proofs in Twelf metatheory.

Or the proofs can be aggressively "syntactic," meaning that *only* proof-theoretic induction is used, and we avoid any attribution of "meaning" to the operators; the Wright-Felleisen notation [14] encourages this approach. Coq and Isabelle support this style, among others; Twelf metatheory supports *only* this pure proof-theoretic style. The advantages to using a pure style are that the metatheory itself can be much smaller and simpler—making it easier to learn and easier to reason about. Indeed, Twelf is a much simpler and smaller system than Coq.

Between these two extremes, it is possible to reason using a mix of semantic and syntactic reasoning. Authors who believe they are writing in a purely Wright-Felleisen style are often reasoning semantically about such things as environments and mappings. The Coq system supports the mixed style (or either of the two extremes) reasonably well. Therefore, it may be the case that specifications expressed in Coq are closer to what one would write in a research paper. Coq proofs can be substantially shorter than Twelf proofs, especially when experienced experts are manipulating the language of tactics. Therefore Coq may be a language of choice for those who do not want to commit in advance to a purely proof-theoretic style.

However, our benchmark does not exercise one of the main strengths of the Twelf system, the higher-order abstract syntax and related proof mechanisms. For syntactic theories that use binders and $\alpha\beta\eta$-conversion, the comparison might come out differently.

References

[1] Appel, A. W. and X. Leroy, *A list-machine benchmark for mechanized metatheory*, Research report 5914, INRIA (2006).

[2] Appel, A. W. and X. Leroy, *List-machine exercise* (2006), http://www.cs.princeton.edu/~appel/listmachine/.

[3] Aydemir, B. E., A. Bohannon, N. Foster, B. Pierce, D. Vytiniotis, G. Washburn, S. Weirich, S. Zdancewic, M. Fairbairn and P. Sewell, *The POPLmark challenge* (2005), http://fling-l.seas.upenn.edu/~plclub/cgi-bin/poplmark/.

[4] Bertot, Y. and P. Castéran, "Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions," EATCS Texts in Theoretical Computer Science, Springer-Verlag, 2004.

[5] *The Coq proof assistant* (1984–2006), software and documentation available from http://coq.inria.fr/.

[6] Klein, G. and T. Nipkow, *A machine-checked model for a Java-like language, virtual machine and compiler*, ACM Transactions on Programming Languages and Systems **28** (2006), pp. 619–695.

[7] Leinenbach, D., W. Paul and E. Petrova, *Towards the formal verification of a C0 compiler*, in: *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)* (2005), pp. 2–11.

[8] Leroy, X., *Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant*, in: *POPL'06: 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2006), pp. 42–54.

[9] Moore, J. S., *A mechanically verified language implementation*, Journal of Automated Reasoning **5** (1989), pp. 461–492.

[10] Moore, J. S., "Piton: a mechanically verified assembly-language," Kluwer, 1996.

[11] Pfenning, F. and C. Schuermann, *Twelf user's guide, version 1.4* (2002), http://www.cs.cmu.edu/~twelf/guide-1-4.

[12] Pfenning, F. and C. Schürmann, *System description: Twelf — a meta-logical framework for deductive systems*, in: *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, Lecture Notes in Computer Science **1632** (1999), pp. 202–206.

[13] Strecker, M., *Formal verification of a Java compiler in Isabelle*, in: *Proc. Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science **2392** (2002), pp. 63–77.

[14] Wright, A. K. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1994), pp. 38–94.