A list-machine benchmark for mechanized metatheory

Andrew Appel, Robert Dockins, Xavier Leroy

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A list-machine benchmark for mechanized metatheory

Andrew W. Appel · Robert Dockins · Xavier Leroy

the date of receipt and acceptance should be inserted later

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 [4] 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. The statement of rationale by the POPLmark team (as of 26 June 2009) is,

> How close are we to a world in which mechanically verified software is commonplace? A world in which theorem proving technology is used routinely by both software developers and programming language researchers alike? One crucial step towards achieving these goals is mechanized reasoning about language metatheory. *[. . .]* To gauge progress in mechanizing programming language metatheory, we issue here a set of challenge problems chosen to exercise many aspects of programming languages that are known to be difficult to formalize.

A. W. Appel and R. Dockins
Princeton University, Princeton, NJ 08540, USA
E-mail: appel@princeton.edu, E-mail: rdockins@princeton.edu

X. Leroy
INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay, France
E-mail: xavier.leroy@inria.fr

The first POPLmark examples are all in the theory of $F_{<:}$ and emphasize the theory of binders (e.g., alpha-conversion).

As practitioners of machine-checked proof about real compilers, we have interests that are similar but not identical. We want to formally relate machine-checked proofs to actual implementations, not particularly to LaTeX documents. Amongst the three substantial software artifacts of conference paper, machine-checked proof, and executable implementation (in LaTeX, in a theorem-proving language, in a programming language, respectively), it is desirable that the connection between paper and proof be understandable by humans; but it is *necessary* that the connection between proof and program be generated or checked by machine, because of the size and complexity of those two components.

Furthermore, perhaps it is the wrong approach to "exercise aspects ... that are known to be difficult to formalize," (i.e., binders with alpha- and beta-conversion). Alpha-convertible binders with theorem prover support for alpha-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, 11]. 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 challenge is to represent these systems and prove soundness of the type system.

Since this benchmark is meant to illustrate what is useful to the **C**ompiler **I**mplementation **V**erification community, we call it a CIVmark. We have two implementations of the benchmark, one in Coq and one in Twelf metatheory, and we draw conclusions about the usability of these two systems.

In Table 1, we show what kinds of formal-methods activities are exercised by our list-machine benchmark; for each activity we contrast its importance for writing POPL papers and doing Compiler Implementation Verification (or just proof-carrying code). More ambitious members of the CIV community are proving the not just the soundness of type-checkers, but the correctness of compilers. In reading Table 1, these people should substitute the word "compiler" for "type-checker" and "correctness" for "soundness."

2 The list-machine benchmark

Our example is meant to compare mechanized metatheories on simple and useful tasks of importance to both communities, with a focus on compiler back-ends, that is, a typed assembly language.

We will first present the problem specification (sections 3 to 7), then the outline of a proof (sections 8 and 9), and then the representations of two solutions, one in the Twelf metatheory and one in Coq (sections 10 to 13). Wherever it was natural to do so, we chose the names and statements of lemmas to be the same in the two proofs. In cases where this would have distorted either the Twelf or Coq proof, we left the two proofs different.

Criterion	importance to: POPL	CIV/PCC
Represent[1] the operational semantics in the mechanized metatheory (MM)	+++	+++
Represent the type system in the MM	+++	+
Represent the type-checker algorithm in the MM	+	+++
Use the MM to simulate the execution of the type checker and the operational semantics on tiny examples, to debug them and gain understanding	+++	+
Derive formally, mechanically, and automatically an *efficient* implementation of the type-checker from the algorithm represented in the MM	+	+++
Prove termination properties of the type-checker	++	++
Prove the relation of the type-checker algorithm to the type system	++	+++
Prove soundness, i.e. that if a program type-checks then the operational semantics doesn't "get stuck" or perform illegal operations	+++	+++
Automatically translate inference rules and statements of lemmas from MM formalism to LaTeX.	+++	+

Table 1 Activities exercised by the list-machine benchmark

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]. Likewise, it is a convenient starting point to explore more advanced aspects of typed assembly languages: section 14 describes an extension of the list machine with indirect jumps.

3 Machine syntax

The machine manipulates only cons cells and nil—not numbers—but natural numbers are useful for discussing labels and variable names.

$n_1, n_2, \ldots : N$	natural numbers
$0 : N$	zero
$n+1 : N$	successor

$a_0, a_1, \ldots : A$	values
nil $: A$	the empty list
$\mathrm{cons}(a_1, a_2) : A$	list cell

Remark. Calling nil "the empty list" instead of "the null value" is already a prejudgment: this machine is a pointer machine, not a list machine, and it is only the type system that will impose a list discipline. A different type system for the same machine might not use a notion of lists at all.

The symbols a_i are metavariables that range over values; every value is either nil or the cons of two values. In this case, and in all other cases, we implicitly assume there is "no junk," i.e. that one can perform inductive reasoning over the syntax of values, naturals, etc.

$v_0, v_1, \ldots : V$	variables
$\mathbf{v}_0, \mathbf{v}_1 : V$	variable names

$r_0, r_1, \ldots : R$	stores
$r[v \mapsto a] : R$	bind a to v
$\{\} : R$	empty

[1] In this table, every use of the word "represent" is assumed to mean "represent as naturally as possible," and every use of the word "prove" is meant to mean "prove in a machine-checked way."

The symbols v_i are metavariables that range over variables; the variables themselves $\mathbf{v}_i$ are enumerated by the natural numbers.

In this section of the document when we write $r[v \mapsto a]$ we suggest that v is not in the domain of r; when doing proofs it may be necesary to formalize this notion. When we want to indicate the update operator we will write $r[v := a] = r'$.

$l_0, l_1, \ldots : L$		labels
$\mathbf{L}_0, \mathbf{L}_1$	$: L$	label names

The symbols l_i are metavariables that range over program labels; the labels themselves $\mathbf{L}_i$ are enumerated by the natural numbers.

$\iota_0, \iota_1, \ldots$	$: I$	instructions
jump l	$: I$	jump to label l
branch-if-nil v l	$: I$	if $v = nil$ go to l
fetch-field v 0 v'	$: I$	fetch the head of v into v'
fetch-field v 1 v'	$: I$	fetch the tail of v into v'
cons v_0 v_1 v'	$: I$	make a cons cell in v'
halt	$: I$	stop executing
ι_0 ; ι_1	$: I$	sequential composition

The semantics of instructions will be given by SOS rules, below. To keep the machine small, it features just enough instructions to construct, inspect, take apart and iterate over lists. General, recursive functions are not supported and would require various extensions to the machine: either primitive **call** and **return** instructions operating over a call stack, in the style of the Java Virtual Machine; or code pointers (see section 14) plus universal and existential quantification in the type system, to support CPS-conversion followed by closure conversion, in the style of Morrisett *et al* [13]; or ML-style recursive data types, to support CPS-conversion followed by defunctionalization, in the style of Danvy [7].

$p_0, p_1, \ldots : P$		programs
$\mathbf{L}_n : \iota;\ p$	$: P$	labeled block
end	$: P$	

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

4 Operational semantics

Machine states are pairs (r, ι) of the current instruction ι and a store r associating values to variables. We write $r(v) = a$ (pronounced[2] var-lookup(r, v, a)) to mean that a is the value of variable v in r.

The syntax hints that r is actually a mapping, but we will not take this for granted. Some MM formalizations will use relations everywhere, other will sometimes use functions. If functions are used, then the notation $r(v)$ makes sense; in this specification we will avoid assuming that functions are used and we will carefully use relational notation such as $r(v) = a$.

[2] By "pronounced" we mean that if an MM formalization requires an ASCII name, this is the name that should be used.

Similarly, the relation $r[v := a] = r'$ (perhaps pronounced var-set(r, v, a, r')), is written to hint that updating r with the binding $[v := a]$ yields a unique store r'.

The relation $p(l) = \iota$ (pronounced prog-lookup(p, l, ι)) looks up the mapping of label l in program p.

These three relations can either be specified as operators on mathematical mappings that therefore happen to satisfy the following rules, or on the other hand specified inductively as the least relations satisfying the following rules.

$$\frac{}{(r[v \mapsto a])(v) = a}\ \text{var-lookup1} \qquad \frac{v \neq v' \quad r(v') = a'}{(r[v \mapsto a])(v') = a'}\ \text{var-lookup2}$$

$$\frac{}{(r[v \mapsto a])[v := a'] = (r[v \mapsto a'])}\ \text{var-set1} \qquad \frac{v \neq v' \quad r[v' := a'] = r')}{(r[v \mapsto a])[v' := a'] = r'[v \mapsto a]}\ \text{var-set2}$$

$$\frac{}{\{\,\}[v := a] = \{\,\}[v \mapsto a]}\ \text{var-set3}$$

$$\frac{}{(\mathbf{L}_n : \iota;\ p)(\mathbf{L}_n) = \iota}\ \text{program-lookup1} \qquad \frac{n \neq n' \quad p(n') = \iota'}{(\mathbf{L}_n : \iota;\ p)(\mathbf{L}_{n'}) = \iota'}\ \text{program-lookup2}$$

Small-step and big-step relations. There is a small-step relation $(r, \iota) \overset{p}{\mapsto} (r', \iota')$ (pronounced "step"), parameterized by a program p, and the Kleene closure of this relation, $(r, \iota) \overset{p}{\mapsto}{}^{*} (r', \iota')$. We also derive a big-step relation $(p, r, \iota) \Downarrow$ (pronounced "run"); and a big-step relation $p\Downarrow$ on programs p (pronounced "runprog") that specifies a particular initial state.

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

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

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

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

$$\frac{r(v) = \text{cons}(a_0, a_1)}{(r, (\textbf{branch-if-nil}\ v\ l; \iota)) \overset{p}{\mapsto} (r, \iota)}\ \text{step-branch-not-taken}$$

$$\frac{r(v) = \text{nil} \quad p(l) = \iota'}{(r, (\textbf{branch-if-nil}\ v\ l; \iota)) \overset{p}{\mapsto} (r, \iota')}\ \text{step-branch-taken}$$

$$\frac{p(l) = \iota'}{(r, \textbf{jump}\ l) \overset{p}{\mapsto} (r, \iota')}\ \text{step-jump}$$

$$\frac{(r,\iota) \overset{p}{\mapsto} (r',\iota') \quad (p,r',\iota') \Downarrow}{(p,r,\iota) \Downarrow} \text{ run-step} \qquad \frac{}{(p,r,\mathbf{halt}) \Downarrow} \text{ run-halt}$$

We say that a program p runs, that is, $p\Downarrow$, if it runs in the big-step relation from an initial state in which variable $\mathbf{v}_0 = \text{nil}$ and the current instruction is the one at $\mathbf{L}_0$.

$$\frac{\{\,\}[\mathbf{v}_0 := \text{nil}] = r \quad p(\mathbf{L}_0) = \iota \quad (p,r,\iota) \Downarrow}{p\Downarrow} \text{ run-prog}$$

5 A type system

What we specify in this section is a *type system*, not a *type-checking algorithm.* That is, in some places we give an abstract mathematical characterization of some operator (e.g., $\tau_1 \sqcup \tau_2$), and it will be necessary to derive an algorithm and prove that it is consistent with our characterization. We do it this way because different MMs have different ways of specifying algorithms (e.g., logic programs in Twelf, functional programs in Coq).

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.

$\tau_0, \tau_1, \ldots : T$		type
nil	$: T$	singleton type containing nil
list τ	$: T$	list whose elements have type τ
listcons τ	$: T$	non-nil list of τ

An environment Γ is an type assignment of types to a set of variables:

$\Gamma_0, \Gamma_1, \ldots : E$		env
$\{\,\}$	$: E$	empty var typing
$v : \tau,\ \Gamma$	$: E$	type attribution

Is Γ a mapping or is the the binding operator $v{:}\tau, \Gamma$ just a syntactic constructor? That is, how are we to make sense of forms such as $v{:}\tau, v{:}\tau', \{\,\}$ where the same variable v appears repeatedly? The answer depends on the style of specification used, and (presumably) will be made more clear in each MM formalization. In the specification below, we write $\Gamma(v) = \tau$ to denote that Γ associates type τ to variable v, and $\Gamma[v := \tau] = \Gamma'$ to mean that Γ' associates type τ to variable v and is otherwise identical to Γ.

We define subtyping among the various refinements of the list types:

$$\frac{}{\tau \subset \tau} \text{ subtype-refl}$$

$$\frac{}{\text{nil} \subset \text{list}\,\tau} \text{ subtype-nil}$$

$$\frac{\tau \subset \tau'}{\text{list}\,\tau \subset \text{list}\,\tau'} \text{ subtype-list}$$

$$\frac{\tau \subset \tau'}{\text{listcons}\,\tau \subset \text{list}\,\tau'} \text{ subtype-listmixed}$$

$$\frac{\tau \subset \tau'}{\text{listcons}\,\tau \subset \text{listcons}\,\tau'}\ \text{subtype-listcons}$$

We extend subtyping widthwise and depthwise to environments in the obvious way. (How the "obvious" way is specified actually depends on the choice of formalism; and see the discussion in section 12.2 that explains how the correct definition is not necessarily obvious.)

The least common supertype $\tau_1 \sqcup \tau_2 = \tau_3$ of two types τ_1 and τ_2 is 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* is a sequence of labeled environments, representing the types of the variables on entry to each basic block.

$\Pi_1, \Pi_2, \ldots : PT$		program typing
$\{\,\}$	$: PT$	empty program typing
$l : \Gamma,\ \Pi$	$: PT$	block typing

The block-typing $l : \Gamma,\ \Pi$ gives Γ as the types of the variables upon entry to label l, where Π is the rest of the program typing. We write $\Pi(l) = \Gamma$ to indicate that Π associates Γ with label l, and we write Π_0 for the empty program typing.

Because program typings may be given syntactically by the end user, it is important to ensure that they are well-formed—that Π maps each label to at most one Γ, and that each Γ maps each variable to at most one type. How this is guaranteed depends on the implementation. We suggest the notation $\vdash_{\text{env}} \Gamma$ to signify that Γ maps variables uniquely.

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

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

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

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

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

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

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

$$\frac{\begin{array}{cc}\Gamma(v_0) = \tau_0 & \Gamma(v_1) = \tau_1 \\ (\text{list}\,\tau_0) \sqcup \tau_1 = \text{list}\,\tau & \Gamma[v := \text{listcons}\,\tau] = \Gamma'\end{array}}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{cons}\ v_0\ v_1\ v\}\Gamma'}\ \text{check-instr-cons}$$

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

$$\frac{}{\Pi; \Gamma \vdash_{\text{block}} \textbf{halt}} \text{ check-block-halt}$$

$$\frac{\Pi \vdash_{\text{instr}} \Gamma\{\iota_1\}\Gamma' \quad \Pi; \Gamma' \vdash_{\text{block}} \iota_2}{\Pi; \Gamma \vdash_{\text{block}} \iota_1; \iota_2} \text{ check-block-seq}$$

$$\frac{\Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi; \Gamma \vdash_{\text{block}} \textbf{jump } l} \text{ check-block-jump}$$

Program typings. The judgment $\Pi \vdash_{\text{blocks}} p$ (pronounced "check-blocks Π p") means that the blocks p are well-typed in the program-typing Π.

$$\frac{\Pi(l) = \Gamma \quad \Pi; \Gamma \vdash_{\text{block}} \iota \quad \Pi \vdash_{\text{blocks}} p}{\Pi \vdash_{\text{blocks}} l : \iota;\ p} \text{ check-blocks-label}$$

$$\frac{}{\Pi \vdash_{\text{blocks}} \textbf{end}} \text{ check-blocks-empty}$$

In a complete program typing, the domain of Π must be a subset of the domain of p. A benchmark solution may simply say, $\text{dom}(\Pi) \subset \text{dom}(p)$ (as our Coq solution does, using an efficient library for manipulating finite functions on natural numbers). Or (as our Twelf solution does) one may use a stricter relation $\Pi \bowtie p$ that means that the labels of Π match those of p exactly, in strict numerical ascending order:

$$\frac{}{(l : \Gamma,\ \Pi_0) \bowtie (l : \iota;\ \textbf{end})} \text{ typing-dom-match1}$$

$$\frac{(\mathbf{L}_{n+1} : \Gamma',\ \Pi) \bowtie (\mathbf{L}_{n+1} : \iota';\ p)}{(\mathbf{L}_n : \Gamma,\ \mathbf{L}_{n+1} : \Gamma',\ \Pi) \bowtie (\mathbf{L}_n : \iota;\ \mathbf{L}_{n+1} : \iota';\ p)} \text{ typing-dom-match2}$$

Execution will start at the initial label $\mathbf{L}_0$, which will be bound to the initial environment $\Gamma_0 = (\mathbf{v}_0 : \text{nil},\ \{\})$.

Type-checking an entire program is therefore,

$$\frac{\Pi \vdash_{\text{blocks}} p \quad \Pi \bowtie p \quad \Pi(\mathbf{L}_0) = \Gamma_0}{\models_{\text{prog}} p : \Pi} \text{ check-program1}$$

(perhaps with the premise $\text{dom}(\Pi) \subset \text{dom}(p)$ instead of $\Pi \bowtie p$).

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.

6 Sample program

The following program has three basic blocks. Variable $\mathbf{v}_0$ is initialized to nil by the premises of the run-prog rule. Block 0 initializes $\mathbf{v}_1$ to the list $\mathrm{cons}(\mathrm{nil}, \mathrm{cons}(\mathrm{nil}, \mathrm{nil}))$ 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.

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

We claim the following typing for this program

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

7 Mechanization tasks

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

1. Represent the operational semantics in the MM.
2. Derive the fact that $p_{\mathrm{sample}}\Downarrow$. *Why is this useful? If the MM can conveniently simulate execution of small examples, then it is easier for the user to debug the SOS and get an intuitive feel for its expressiveness.*

Soundness of a type system.

3. Represent the type system in the MM. (One needs to define enough notation so that the formula $\models_{\mathrm{prog}} p : \Pi$ can be represented and type-soundness can be proved.)
4. Represent in the MM an algorithm for (or constructive proof of the existence of) least-common-supertype, that is, the computation $\tau_1 \sqcup \tau_2 = \tau_3$ producing τ_3 from inputs τ_1 and τ_2.
5. Using the type system, derive the fact that $\models_{\mathrm{prog}} p_{\mathrm{sample}} : \Pi_{\mathrm{sample}}$. *Why is this useful? If the MM can conveniently simulate type-checking of small examples, then it is easier for the user to debug the type system and get an intuitive feel for its expressiveness.*
6. Represent the statement of these properties of the chosen least-common-supertype algorithm:

$$\frac{\tau_1 \sqcup \tau_2 = \tau_3}{\tau_1 \subset \tau_3}\ \text{lub-subtype-left} \qquad \frac{\tau_1 \sqcup \tau_2 = \tau_3}{\tau_2 \subset \tau_3}\ \text{lub-subtype-right}$$

$$\frac{\tau_1 \sqcup \tau_2 = \tau_3 \quad \tau_1 \subset \tau_4 \quad \tau_2 \subset \tau_4}{\tau_3 \subset \tau_4}\ \text{lub-least}$$

7. Prove the lemmas *lub-subtype-left, lub-subtype-right,* and *lub-least. The first two lemmas are directly useful in the soundness proof; the last one is not, but is a reassuring completeness property.*

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*:

$$\text{step-or-halt}(p, r, \iota) \equiv (\exists r', \iota' . (r, \iota) \overset{p}{\mapsto} (r', \iota')) \vee \iota = \textbf{halt}.$$

$$\frac{\models_{\text{prog}} p : \Pi \quad \{\}[\mathbf{v}_0 := \text{nil}] = r \quad p(\mathbf{L}_0) = \iota \quad (r, \iota) \overset{p}{\mapsto}{}^{*} (r', \iota')}{\text{step-or-halt}(p, r', \iota')} \text{ soundness}$$

9. Prove the soundness theorem.

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 a program of N instructions, with maximum number number of live variables M, should type-check in $O(N \log M)$ time.
11. Using the type-checking algorithm, derive the fact that $\vdash_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$. *Why is this useful? If the MM can conveniently simulate type-checking of small examples, then it is easier for the user to debug the type system and get an intuitive feel for its expressiveness.*
12. Prove that the type-checking algorithm terminates on any program. *This is a nice property to have, especially if it is not difficult to establish.*
13. Demonstrate the type-checker on large-scale examples and show empirically that its efficiency is competitive with implementations in Prolog or ML. Specifically, some theorem-proving systems have the ability to translate their internal representation of computable functions into a Prolog or ML program that can be compiled by a high-performance compiler; this means that correspondence between the algorithm that is verified and the program that executes is established mechanically.
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 [9] have written an entire paper (formalization of a Java subset and of certain phases of a compiler for it) in which* every *formula given in the paper is the automatic translation of a statement whose proof has been mechanically checked in Isabelle/HOL. Therefore they avoid transcription errors and ameliorate version-control problems.*

Modularizing the proof.

16. Insert an opaque abstraction boundary between the proof of soundness of the type system and the proof of soundness of the typechecking algorithm. *This task exercises the "software engineering" capabilities of the MM related to enforcing leak-free abstractions.*

Reading and maintaining the proof.

*. It is clearly desirable that the mechanized proofs are written to maximize readability of the logical arguments and maintainability when the type system evolves or a new, incompatible version of the proof assistant is released. These objectives are very difficult to quantify, and we will not attempt to do so in this benchmark.

8 Type checking algorithms

From the sketch of a type system given in section 5, one could specify a type-checking algorithm by giving a set of syntax-directed inference rules, by giving a functional program, by giving an imperative program, or by some combination of these techniques. The choice of style may be influenced by how the MM relates programs to proofs.

8.1 Type checking by syntax-directed rules

One of the ways to specify and implement a type-checker is by writing down a set of syntax-directed judgment rules. If the inference-rule style is chosen for implementation in the MM, we suggest that the following rules and notation should be used.

In the syntactic style, the notation $v : \tau,\ \Gamma$ is just the application of a three-argument constructor, and does not inherently guarantee that v is not in the domain of Γ. Within an algorithm one could arrange to preserve this property; but we want the end user to provide a (claimed) program-typing mapping each label to a Γ, so we need an algorithmic way to check that the user's Γ does not have multiple mappings for any variable. Therefore, we define a predicate to judge that an environment Γ is a (single-valued) function; to make the syntax-directed checker efficient, we insist that Γ map its variables in order of variable-number:

$$\frac{}{\vdash_{\text{env}} \{\,\}}\ \text{env-ok0} \qquad \frac{}{\vdash_{\text{env}} v : \tau,\ \{\,\}}\ \text{env-ok1}$$

$$\frac{n < n' \qquad \vdash_{\text{env}} \mathbf{v}_{n'} : \tau',\ \Gamma}{\vdash_{\text{env}} \mathbf{v}_n : \tau,\ (\mathbf{v}_{n'} : \tau',\ \Gamma)}\ \text{env-ok2}$$

It is helpful (though perhaps not necessary?) to make the rules properly inductive by introducing the notation

$$\Gamma \doteq (v : \tau,\ \Gamma')$$

which means that Γ is the disjoint union of a relation Γ' and a binding $v : \tau$. We will treat this formally as a four-place relation on Γ, v, τ, and Γ', pronounced *env-lookup*. We use the symbol $\doteq$ instead of $=$ to remind the reader that this is not syntactic equality, but really a computation with inputs Γ, v and outputs τ, Γ'. The rules for this relation are,

$$\frac{}{(v : \tau,\ \Gamma) \doteq (v : \tau,\ \Gamma)}\ \text{env-lookup1}$$

$$\frac{v \neq v' \quad \Gamma \doteq (v' : \tau',\ \Gamma')}{(v : \tau,\ \Gamma) \doteq (v' : \tau',\ (v : \tau,\ \Gamma))}\ \text{env-lookup2}$$

In order to make use of this relation, we adjust[3] most of the instruction-typing rules, as follows:

[3] In formulating a solution to this benchmark problem, it may be legitimate to make minor alterations of the type system, but one should then take care to make sure (informally) that the altered type system accepts at least all the programs that type-check in our specified type system.
On the other hand, one must not alter the dynamic semantics. The high-level claim is *our typechecker accepts 'enough' programs, and whatever programs it accepts are safe on the real machine.* The "real machine" is specified by the dynamic semantics, and one does not have a

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

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

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

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

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

$$\frac{\Gamma(v_0) = \tau_0\Gamma_0 \qquad \Gamma \doteq (v_1 : \tau_1,\ \Gamma_1) \quad (\text{list}\,\tau_0) \sqcup \tau_1 = \text{list}\,\tau \quad \Gamma[v := \text{listcons}\,\tau] = \Gamma'}{\Pi \vdash_{\text{instr}} \Gamma\{\textbf{cons}\ v_0\ v_1\ v\}\Gamma'}\ \text{check-instr-cons}'$$

The four-place relation $\Gamma[v := \tau] = \Gamma'$, pronounced "env-set($\Gamma, v, \tau, \Gamma'$)," has syntax-directed rules as follows,

$$\frac{}{(v : \tau,\ \Gamma)[v := \tau'] = (v' : \tau,\ \Gamma)}\ \text{env-set1}$$

$$\frac{v \neq v' \quad \Gamma[v' := \tau'] = \Gamma'}{(v : \tau,\ \Gamma)[v' := \tau'] = (\Gamma', v : \tau)}\ \text{env-set2}$$

$$\frac{}{\{\}[v := \tau] = (v : \tau,\ \{\})}\ \text{env-set3}$$

We give a syntax-directed view of environment subtyping as follows.

$$\frac{\Gamma = (v : \tau',\ \Gamma'') \quad \tau' \subset \tau \quad \Gamma'' \subset \Gamma'}{\Gamma \subset (v : \tau,\ \Gamma')}\ \text{env-sub1} \qquad \frac{}{\Gamma \subset \{\}}\ \text{env-sub2}$$

See section 12.2 for a discussion of the rule env-sub1.

Syntax-directed rules for least-common-supertype:

$$\frac{}{\tau \sqcup \tau = \tau}\ \text{lub-0} \qquad \frac{}{\text{list}\,\tau \sqcup \text{nil} = \text{list}\,\tau}\ \text{lub-1} \qquad \frac{}{\text{nil} \sqcup \text{list}\,\tau = \text{list}\,\tau}\ \text{lub-4}$$

choice in that. The proof that type-checking implies safety is the one formalized and machine-checked. But the claim that the type system accepts "enough" programs is not completely formalized within this benchmark. It is certainly untrue that all safe programs will type-check; it is true (but we do not formally prove) that there exists even one program that does type-check. Similarly, if the proposer of a benchmark solution has adjusted the type system, we expect (but do not demand a machine-checked proof of) relative completeness.

$$\frac{\text{list}\,\tau_1 \sqcup \text{list}\,\tau_2 = \tau_3}{\text{list}\,\tau_1 \sqcup \text{listcons}\,\tau_2 = \tau_3}\ \text{lub-2} \qquad \frac{\text{list}\,\tau_1 \sqcup \text{list}\,\tau_2 = \tau_3}{\text{listcons}\,\tau_1 \sqcup \text{list}\,\tau_2 = \tau_3}\ \text{lub-2b}$$

$$\frac{\tau_1 \sqcup \tau_2 = \tau_3}{\text{list}\,\tau_1 \sqcup \text{list}\,\tau_2 = \text{list}\,\tau_3}\ \text{lub-3}$$

$$\frac{}{\text{listcons}\,\tau \sqcup \text{nil} = \text{list}\,\tau}\ \text{lub-5} \qquad \frac{}{\text{nil} \sqcup \text{listcons}\,\tau = \text{list}\,\tau}\ \text{lub-6}$$

$$\frac{\tau_1 \sqcup \tau_2 = \tau_3}{\text{listcons}\,\tau_1 \sqcup \text{listcons}\,\tau_2 = \text{listcons}\,\tau_3}\ \text{lub-7}$$

Looking up a label in a program-typing, at the same time checking that the resulting Γ is well-formed:

$$\frac{\vdash_{\text{env}} (\Gamma)}{(l : \Gamma,\ \Pi)(l) = \Gamma}\ \text{block-typing-lookup1} \qquad \frac{l \neq l' \quad \Pi(l') = \Gamma'}{(l : \Gamma,\ \Pi)(l') = \Gamma'}\ \text{block-typing-lookup2}$$

8.2 Type checking by an imperative program

We now give an alternate presentation of the type-checking algorithm as imperative pseudocode, in the style of algorithm textbooks. Just as the presentation using syntax-directed rules is close to an implementation in a logic programming language, the pseudocode presentation in this section is close to an implementation in an ML-like language. The pseudocode raises an exception (denoted by `fail`) to report a type error and abort type-checking. The algorithm can also be written in a purely functional style: just replace this exception by explicit `Error` function results.

`check_env_sub` Γ_1 Γ_2:
  For each binding $v : \tau_2$ in Γ_2,
    Check that Γ_1 contains a binding $x : \tau_1$ and that $\tau_1 \subset \tau_2$; `fail` otherwise.

`typecheck_branch` Π Γ l:
  If $\Pi(l)$ is undefined, `fail`; otherwise, `check_env_sub` Γ $\Pi(l)$.

`typecheck_instr` Π Γ ι:
  If $\iota = \iota_1; \iota_2$: return `typecheck_instr` Π (`typecheck_instr` Π Γ ι_1) ι_2.
  If $\iota =$ **jump** l: `fail`.
  If $\iota =$ **branch-if-nil** v l:
    If $\Gamma(v)$ is undefined, `fail`
    If $\Gamma(v) = \text{list}\,\tau$:
      `typecheck_branch` Π $\Gamma[v := \text{nil}]$ l and return $\Gamma[v := \text{listcons}\,\tau]$.
    If $\Gamma(v) = \text{listcons}\,\tau$:
      return Γ.
    If $\Gamma(v) = \text{listcons}$:
      `typecheck_branch` Π Γ l and return Γ
  If $\iota =$ **fetch-field** v n v':
    `fail` if $\Gamma(v)$ is undefined or not of the form $\text{listcons}\,\tau$.
    `fail` if $n \neq 0$ and $n \neq 1$.
    Return $\Gamma[v' := \tau]$ if $n = 0$, $\Gamma[v' := \text{list}\,\tau]$ if $n = 1$.
  If $\iota =$ **cons** v_0 v_1 v':
    `fail` if $\Gamma(v_0)$ or $\Gamma(v_1)$ is undefined.
    Compute $\tau = (\text{list}\,\Gamma(v_0)) \sqcup \Gamma(v_1)$.

If τ is of the form list τ', return $\Gamma[v' := \tau']$; otherwise, `fail`.

`typecheck_block` Π Γ ι:
If $\iota = \textbf{halt}$, success.
If $\iota = \iota_1; \iota_2$:
`typecheck_block` Π (`typecheck_instr` Π Γ ι_1) ι_2.
If $\iota = \textbf{jump}\ l$:
`typecheck_branch` Π Γ l.
Otherwise: `fail`.

`typecheck_blocks` Π p:
If $p = \textbf{end}$, success.
If $p = \mathbf{L}_n : \iota;\ p'$:
Let $\Gamma = \Pi(\mathbf{L}_n)$ (or `fail` if undefined).
`typecheck_block` Π Γ ι
`typecheck_blocks` Π p'.

`typecheck_program` Π p:
`typecheck_blocks` Π p
Check that $\Pi(\mathbf{L}_0) = \Gamma_0$ and that $\mathrm{dom}(\Pi) \subseteq \mathrm{dom}(p)$.

9 Proof outline

We outline here the principal lemmas of a soundness proof. Machine-checked derivations are not required to follow this outline; but to the extent that they do, it will be convenient if they use the same names for the principal lemmas.

The proof will rely on natural numbers with equality and inequalities; on equalities and inequalities of type and term structures; and the properties of (and operations on) stores and environments as mappings (partial functions). Appropriate lemmas must either be proved or imported from libraries.

Well-formed environments. The type-checker has some way of testing $\vdash_{\mathrm{env}} \Gamma$, i.e. that the environment syntax provided by the end user for some label in Π is well formed. One must prove that $\vdash_{\mathrm{env}} \Gamma$ implies that Γ is indeed a partial function.

Value-has-ty. The type-checker attributes types to variables, but surprisingly it never needs to attribute types to values. The soundness proof therefore needs to define this concept, typically by cases:

$$\frac{}{\mathrm{nil} : \mathrm{nil}} \qquad \frac{}{\mathrm{nil} : \mathrm{list}\,\tau}$$

$$\frac{}{\mathrm{cons}(a_0, a_1) : \mathrm{listcons}\,\tau} \qquad \frac{a : \mathrm{listcons}\,\tau}{a : \mathrm{list}\,\tau}$$

Var-set-type.

$$\frac{\Gamma\ \mathrm{env} \quad r : \Gamma \quad a : \tau \quad r[v := a] = r' \quad \Gamma[v := \tau] = \Gamma'}{r' : \Gamma'}\ \text{var-set-type}$$

Various progress lemmas.

$$\frac{\Pi \bowtie p \quad \Pi(l) = \Gamma}{\exists \iota'.\ p(l) = \iota'}\ \text{progress-typing-dom-match}$$

$$\frac{\models_{\text{prog}} p : \Pi \quad \Pi(l) = \Gamma}{\exists \iota'.\ p(l) = \iota'}\ \text{progress-check-program}$$

$$\frac{\Gamma(v) = \tau \quad r : \Gamma}{\exists a.\ \ r(v) = a\ \wedge\ a : \tau}\ \text{progress-env}$$

$$\frac{\models_{\text{prog}} p : \Pi \quad \Pi \vdash_{\text{instr}} \Gamma\{\iota\}\Gamma' \quad r : \Gamma}{\text{step-or-halt}(p, r, \iota)}\ \text{progress}$$

Lemmas about least-common-supertype.

$$\frac{\tau_1 \sqcup \tau_2 = \tau_3}{\tau_2 \sqcup \tau_1 = \tau_3}\ \text{lub-comm}$$

$$\frac{\tau_1 \sqcup \tau_2 = \tau_3}{\tau_1 \subset \tau_3}\ \text{lub-subtype-left} \qquad \frac{\tau_1 \sqcup \tau_2 = \tau_3}{\tau_2 \subset \tau_3}\ \text{lub-subtype-right}$$

Subsumption.

$$\frac{\tau_1 \subset \tau 2 \quad a : \tau_1}{a : \tau_2}\ \text{subsumption} \qquad \frac{\Gamma_1 \subset \Gamma 2 \quad r : \Gamma_1}{r : \Gamma_2}\ \text{subsumption-env}$$

Relation of static to dynamic semantics of the store.

$$\frac{\Gamma(v) = \tau \quad r : \Gamma \quad r(v) = a}{a : \tau}\ \text{preservation-env-lookup}$$

Looking up a well-typed block.

$$\frac{\Pi \vdash_{\text{blocks}} p \quad p(l) = \iota \quad \Pi(l) = \Gamma}{\Pi; \Gamma \vdash_{\text{block}} \iota \qquad \vdash_{\text{env}} \Gamma}\ \text{preservation-block-typing}$$

$$\frac{\models_{\text{prog}} p : \Pi \quad p(l) = \iota \quad \Pi(l) = \Gamma}{\Pi; \Gamma \vdash_{\text{block}} \iota \quad \vdash_{\text{env}} \Gamma}\ \text{preservation-program-typing}$$

Preservation.

$$\frac{\models_{\text{prog}} p : \Pi \quad \vdash_{\text{env}} \Gamma \quad r : \Gamma \quad \Pi; \Gamma \vdash_{\text{block}} \iota \quad (r, \iota) \overset{p}{\mapsto} (r', \iota')}{\exists \Gamma'.\ \ \vdash_{\text{env}} \Gamma' \ \wedge\ r' : \Gamma' \ \wedge\ \Pi; \Gamma' \vdash_{\text{block}} \iota'}\ \text{preservation}$$

$$\frac{\models_{\text{prog}} p : \Pi \quad \text{initial}(p, r_0, \iota_0) \quad (r_0, \iota_0) \overset{p}{\mapsto}{}^{*} (r, i)}{\exists \Gamma.\ \ \vdash_{\text{env}} \Gamma \ \wedge\ r : \Gamma \ \wedge\ \Pi; \Gamma \vdash_{\text{block}} \iota}\ \text{run-well-typed}$$

That which was to have been shown.

$$\frac{\models_{\text{prog}} p : \Pi \quad \text{initial}(p, r, \iota) \quad (r, \iota) \overset{p}{\mapsto}{}^{*} (r', \iota')}{\text{step-or-halt}(p, r', \iota')}\ \text{soundness}$$

10 Learning to use the MMs

An experienced practitioner of (nonmechanized) operational semantics may reasonably wonder, "How long does it take to learn how to use a mechanized metatheory?" (both the encoding of machines and type systems and the manipulation of the proof assistant) and "If I take the trouble to learn metatheory A, how long will it take to learn metatheory B?"

We did not perform a controlled experiment. The first author (Appel) had, at the time of attemping the benchmark, 6 years experience with nontactical proofs in higher-order logic represented in the Twelf system [1]; this style of proof is entirely unlike the Twelf *metatheory* discussed here, and the nontactical style is quite unlike the style of proof used in Coq. The third author (Leroy) had 6 years experience proving theorems in Coq. Consequently, neither author was able to definitively answer the first question, but Appel could attempt an answer to the second.

Twelf metatheory. In writing Twelf metatheory, one first must learn logic programming; any experience using Prolog would be helpful, but in any case a programming-language theoretician should be able to solve tasks 1 and 3 (represent the operational semantics, represent the type system) and tasks 2 and 5 ("execute" the operational semantics and the type system) in about a week. Appel started from this point. The next step is to learn how to use termination and totality in the metatheory, how to use logical tricks like contradiction, and so on. This took about 3 weeks to learn; the on-line documentation is barely adequate and at times frustrating, but at no time was it necessary to consult a metatheory expert. Therefore we estimate about 1 month to learn enough Twelf to complete all the tasks of this benchmark.

Coq. Appel started from zero on Coq; Leroy provided the Coq solution to this benchmark. Rather than repeating Leroy's work (for which the solution was too temptingly available), Appel defined (in Coq) the structured operational semantics of a von Neumann "list machine" with a memory and a heap-allocation pointer, wrote a program (in the executable part of Coq) to assemble the list-machine "assembly" language of this benchmark to the "machine" language, and proved the correctness of the translation. The techniques required are very similar to those needed for the type-soundness proof. It took about 3 weeks to learn enough Coq to do the assembler-correctness proof (meaning that, after 3 weeks, the proof wasn't finished but the issues changed from those of learning Coq to those of finding the right induction hypotheses). The on-line documentation (and the Coq'Art book [5]) are barely adequate and at times frustrating, but only three short (10-minute) consultations with the experts were necessary over the 3-week period.

Both Twelf and Coq have more sophisticated features which Appel did not attempt to learn in the trial period: Twelf has sophisticated techniques for handling binders in higher-order abstract syntax, and Coq supports programming with dependent types. This benchmark was deliberately chosen to exercise only the features most of interest to compiler-writers.

Conclusion. Once you learn one system, you can move on to the next (even if it is quite different) with an investment of one month's time.

11 Comparison of mechanized proofs

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 following table.

	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.	$\sqcup$ algorithm	*	*	
5.	Derive $\models_{\text{prog}} p_{\text{sample}} : \Pi_{\text{sample}}$	1	no	
6.	State properties of $\sqcup$	12	13	
7.	Prove properties of $\sqcup$	114	21	
8.	State soundness theorem	29	15	
9.	Prove soundness of $\models_{\text{prog}} p : Pi$	2060	315	
10.	Efficient algorithm	22	130	
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	
16.	Separate soundness proof from specification	no	188	

Total parsing and proof-checking time[4] was 0.558 seconds real time for Twelf, 2.622 seconds for Coq.

In this section we will make a few remarks about each task, and in the later sections we will explain the principles of each MM proof in more detail.

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. 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. (See section 13.2 for details.) 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] Dell Precision 360, Linux, 2.8 GHz Pentium 4, 1GB RAM, 512kB cache.

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-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

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

is represented as a logic-programming predicate,

```
lub-subtype-left: unify 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 `unify 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 the 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:

```
Theorem safety:
  forall p s i pt,
  check_program pt p -> run_prog_finite p s i -> step_or_halt p s i.
```

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; section 12 explains this proof in more detail. The Coq proof of soundness (see section 13) 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 nonalgorithmic 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 again for each different kind of mapping.

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; we will explain this below. In Twelf one can also represent dynamic clauses, so it is a simple matter to replace some of the rules of the type system with ones that use a more efficient method to look up the typings of labels. The new logic program (most of the old type system, plus a few new rules) we call the "type checker;" it runs in time $O(n)$, in principle.[5]

In Coq, the type-checker is defined as a function from program typings 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; these errors are propagated in a monadic style (the error monad). To avoid an n^2 algorithm, we represent environments and program typings 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 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 (about 3 times shorter than the Twelf proof).

[5] This type-checkier will not really run in linear time in Twelf, but the Twelf program is automatically translatable to an efficient Prolog program, as we discuss later in the paper.

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.

16. Separate soundness proof from specification. In an extension of the list machine benchmark (see section 14), we used Coq's module system to enforce an abstraction boundary between 1) the soundness of the specification of the type system with respect to the machine's operational semantics and 2) the proof of soundness of the typechecking *algorithm* with respect to that specification. Twelf lacks a module system and cannot enforce a similar abstraction boundary.

12 A proof in Twelf metatheory

The Twelf system[15] 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. In this section we will explain a solution using 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) nonstuck 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[6]. 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.

Our specification of $\models_{\text{prog}} p : \Pi$, though executable, has quadratic running time, because the lookup of a label in the label-mapping Π takes time proportional to the number of labels mapped by Π. Therefore we also write a set of rules $\vdash_{\text{prog}} p : \Pi$ that executes more efficiently; we will describe the differences below.

Since we avoid using HOAS (higher-order abstract syntax) in our Twelf program, both our type system and our type-checker are trivially translatable to an executable Prolog program[7], and a good Prolog compiler can execute the type-checker with respectable performance.

Each clause in Twelf is named (we use the same names that are attached to the inference rules shown in this paper—see also Appendix A). When Twelf traces out,

[6] Actually, as dependently typed higher-order hereditary Harrop formulas, but for this simple benchmark we don't use the higher-order abstract syntax capabilities of Twelf.

[7] Dinghao Wu has implemented such a translator for an experiment running a large typed-assembly-language typechecker, specified in Twelf and executed either in SICStus Prolog or in Flit [18].

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 (nonstuck) 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.

12.1 Preliminaries to the Twelf proof

Equalities and inequalities on natural numbers. The type-checker takes type annotations from the user, attributing to each program label a formal-parameter list and within each formal-parameter list, attributing types to variables. To ensure that the user does not associate more than one parameter-list to a label, or more than one type to a variable, the type-checker requires that the labels and variables each be annotated in numerically increasing order. For this reason it will be necessary to reason about equality and inequalities (e.g., a less-than relation) of natural numbers, or their equivalent. Presumably a machine-checked derivation may use library definitions and lemmas for this, so we will not attempt to specify the names and forms of these lemmas here.

Equality and inequality of type and term structures. Similarly, it may be useful in most derivations to be able to use notions of equality, congruence, and inversion on the structure of types, values, and so on. Methods for handling congruences may be idiomatic to a mechanical proof assistant, and we will not overspecify the representation of equality or equivalence relations by axiomatizing them.

Store-has-type. A store r satisfies an environment Γ, written as $r : \Gamma$, if for every variable v in the domain of Γ, $r(v)$ has type $\Gamma(v)$.

If stores and environments are defined as *a priori* mappings, and if the type-checker uses the rules *env-lookup1, var-set2,* etc. defined in section 4 then it will be necessary to derive those rules.

However, in the syntactic style, the rules are taken as axiomatic, and it is necessary to define "good" environments (i.e., those that are mappings). The relation $v \notin \operatorname{dom}\Gamma$, pronounced "env-no-binding(v, Γ)" is used as an auxiliary in defining v env, pronounced "env-good(v)." Then we have lemmas,

$$\frac{r(v) = a \quad r(v) = a'}{a = a'}\ \text{var-lookup-uniq} \qquad \frac{r[v := a] = r'}{r'(v) = a}\ \text{var-set-lookup}$$

$$\frac{\Gamma\ \text{env} \quad \Gamma \doteq (v : \tau,\ \Gamma')}{\Gamma'\ \text{env} \quad v \notin \operatorname{dom}\Gamma'}\ \text{env-lookup-no-binding}$$

$$\frac{\Gamma\ \text{env} \quad \Gamma \doteq (v : \tau,\ \Gamma)' \quad v' \notin \operatorname{dom}\Gamma}{v' \notin \operatorname{dom}\Gamma'}\ \text{env-lookup-no-binding2}$$

$$\frac{\Gamma \text{ env} \quad \Gamma[v := \tau] = \Gamma'}{\Gamma' \text{ env}} \text{ env-set-good}$$

Environments are mappings. We define a syntactic identity relations $\tau_1 = \tau_2$ on types and $\Gamma_1 = \Gamma_2$ on environments. We do not necessarily assume that $\Gamma_1 \subset \Gamma_2 \wedge \Gamma_2 \subset \Gamma_1 \Rightarrow \Gamma_1 = \Gamma_2$.

$$\frac{\Gamma \doteq (v : \tau_1,\ \Gamma_1) \quad \Gamma \doteq (v : \tau_2,\ \Gamma_2)}{\tau_1 = \tau_2 \quad \Gamma_1 = \Gamma_2} \text{ env-lookup-uniq}$$

12.2 What is easy and what is difficult in Twelf?

Stephanie Weirich described [16] her experiences formalizing in Twelf metatheory a soundness proof for Featherweight Java. Our conclusions here are largely consistent with hers, except that we had no problems at all with induction over higher-order abstract syntax because we avoided using it.[8]

Easy. Since Twelf can directly execute operational specifications, we get a simulator for the SOS for free. This simulator is not very efficient; it is probably a factor of 100 slower than Prolog, and its asymptotic complexity may also be worse. Operational execution in Twelf is adequate for small examples, but not for relating proofs to real systems. However, it is very easy to translate first-order Twelf directly to Prolog, which is sufficiently efficient for real systems. (Our benchmark is entirely first-order.)

Easy. Twelf encourages a syntactic, clausal specification of the type system, so that our type system is identical to the type-checker and the type-checker can directly execute as a logic program in Twelf. Therefore, expressing algorithms and showing the relation of these algorithms to rule-based specification is straightforward and easy. This is useful for debugging small examples (but does not scale well except via translation to Prolog).

Easy. Twelf has automatic termination checking for logic programs, so it is extremely easy to do a machine-checked termination proof for the type-checker, especially when the proof proceeds by structural induction.

Easy. A totality proof in Twelf involves both case analysis (proving each case, making sure all cases are covered) and induction. One must prove each case by hand (which can be quite tedious), but Twelf automatically does the case analysis (and explains explicitly which cases are not covered, so that one can then prove these cases and try again). Twelf also does the induction analysis (and explains where well-founded induction fails) with a great deal of automation.

[8] For various reasons, we have deliberately chosen a very crude assembly language to formalize, with explicit variables and environments in place of any binding construct. In part this is designed to represent a low-level assembly language after register allocation; but it part it was to avoid the need for "users" of the list-machine benchmark to become experts on binders. Twelf has sophisticated features for inductive reasoning about HOAS, which can be effective in the hands of expert Twelf users; in the hands of mere expert programming-languages researchers, they can be troublesome. This was apparent in the discussion on the POPLmark list following Weirich's post of 17 August 2005.

Difficult. Since 1994 [17] we officially live in the era of syntactic, proof-theoretical soundness proofs upon structured operational semantics. The reality is that authors often use semantic reasoning ("Γ is a mapping") when it is convenient. Twelf forces a completely proof-theoretic style, so that semantic notations must be encoded as proof-theoretic arguments.

In many cases, proofs about SOS and rule-based type systems are extremely natural by structural induction over derivations. But not always: *sometimes, proof-theoretic induction is not entirely natural* and it takes some experience to learn how to do it. As an example, consider the following three definitions of environment-subtyping:

$$\Gamma_1 \subset_{\text{env}} \Gamma_2 \;\equiv\; \forall v.\, v \in \operatorname{dom}\Gamma_2 \;\Rightarrow\; (v \in \operatorname{dom}\Gamma_1 \;\wedge\; \Gamma_1(v) \subset \Gamma_2(v)).$$

Predicate logic. This definition is very natural, for one used to thinking in predicate logic, that is, in "mathematics." It quantifies over an infinite number of variables v, and we don't even have to think too hard about whether the infinity is countable or not—we know how to reason about quantification.

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

Pseudoinductive. This one looks like an inductive specification for which standard proof-theoretic techniques (structural induction) will be able to prove the obvious properties (transitivity, reflexivity). The induction is (supposedly) over the size of the term to the right of the $\subset_{\text{env}}$ symbol. However, this definition is not sufficiently inductive for useful properties to be provable (the first author gave up after two days). The problem appears to be that Γ_1 does not decrease in rule a_2.

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

Inductive This definition is properly inductive, since we use Γ' instead of Γ_1 in the premise of rule b_2. The obvious properties are provable without too much fuss. The difficulty is to avoid writing the pseudoinductive definition.

Tedious. Twelf does not have parametric polymorphism at the level of types. Effectively this means that once one defines a proof-theoretic setup to reason about Γ as a mapping, one must copy the whole thing to reason about Π as a mapping, then copy it again to reason about r and p as mappings.

Difficult. The proofs in Twelf metalogic are surprisingly long. Each case of the proof must be written out as a (Prolog-style) transformation on derivations; a typical case takes 5 to 20 lines to write down, and there are often a dozen cases in one lemma. Twelf is reasonably informative about explaining what cases are missing, and it is quite good at handling the structural induction (termination) argument automatically, but it does not write the cases for you. The soundness proof for $\models_{\text{prog}} p : \Pi$ took more than 2000 lines of Twelf.

Easy? 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." [14] and every version of the manual since 1998 contains this identical sentence.

Easy? Twelf has an amazing economy of features. 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, with the absence of polymorphism. One does not have to learn a module system—because there is none—one just uses naming conventions on all one's identifiers. All proofs are done with the simple mechanism of proving the totality of metaprograms. There is 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); there are some theorems that the notation can express but the system is too weak to prove (nontrivial metrics for wellfounded induction); and there are some things that are provable but in a contrived way (expressing semantic properties only with inductive syntactic constructors).

13 A proof in Coq

The Coq system [6,5] 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).

Proofs can be given directly as proof terms (expressions of the functional language that inhabit the proposition, viewed as a type), but this is impractical except for the simplest proofs. In practice, proofs are developed interactively using tactics that build incrementally the proof term behind the scene. These tactics range from the trivial (`intro`, which adds a λ-abstraction to the proof term) to rather complex decision procedures (`omega` for Presburger arithmetic; `congruence` for equational reasoning).

An original feature of Coq is that executable Caml code can be automatically extracted from functional specifications. This provides an efficient execution path for programs written and proved correct in Coq.

13.1 The specification

We now outline how the informal specification for the list-machine benchmark was expressed in Coq.

Mappings The first thing we noticed is that this specification, like many others related to programs and programming languages, cries out for finite mappings (functions with finite support from one type to another). We used the following presentation of finite mappings, taken from an earlier project:

```
Parameter map: Set -> Set.
Parameter empty: forall (A: Set), map A.
Parameter get: forall (A: Set), map A -> positive -> option A.
Parameter set: forall (A: Set), map A -> positive -> A -> map A.

Axiom get_empty:
  forall (A: Set) (n: positive), get A (empty A) i = None.
Axiom get_set_same:
  forall (A: Set) (n: positive) (m: map A) (x: A),
  get A (set A m n x) n = Some x.
Axiom get_set_other:
  forall (A: Set) (n1 n2: positive) (m: map A) (x: A),
  n1 <> n2 -> get A (set A m n1 x) n2 = get A m n2.

Notation "a # b" := (get _ a b) (at level 1).
Notation "a # b <- c" := (set _ a b c) (at level 1, b at next level).
```

The type `positive` used as index into mappings is the type of positive natural numbers in base-2 representation defined in the Coq standard library. These axioms and parameters were later realized (also by reusing an earlier development) by an implementation that represents mappings as radix-2 search trees. An additional operation over mapping was added to the signature: parallel traversal of two maps.

```
Parameter map_forall2:
  forall (A B: Set), (A -> option A -> bool) -> map A -> map A -> bool.
Axiom map_forall2_correct:
  forall (A B: Set) (pred: A -> option B -> bool) (m1: map A) (m2: map B)
         (i: positive) (x: A),
  map_forall2 A B pred m1 m2 = true -> m1#i = Some x ->
  pred x m2#i = true.
Axiom map_forall2_complete:
  forall (A B: Set) (pred: A -> option A -> bool) (m1: map A) (m2: map B),
  (forall (i: positive) (x: A), m1#i = Some x -> pred x m2#i = true) ->
  map_forall2 A B pred m1 m2 = true.
```

Thanks to the polymorphic typing of mappings and their operations, we can use them to represent stores (type `map value`), environments (type `map type`), and program typings (type `map environment`). Registers and labels are represented by the type `positive`. The predicates for lookup and update of stores (presented in section 4) and environments (section 5) are represented uniformly as equalities involving the `get` and `set` operations via their `#` notations. For instance, $\Gamma(v) = \tau$ becomes `Gamma#v = Some tau` in the Coq specification, and $\Gamma[v := \tau] = \Gamma'$ becomes `Gamma' = Gamma#v <- tau`.

Dynamic semantics The small-step and big-step semantics for the list machine are specified in relational style, as inductive predicates, closely following the informal specifications. The following excerpt should give the flavor:

```
Inductive step: program -> store -> instr -> store -> instr -> Prop :=
  | step_seq: forall p r i1 i2 i3,
    (*-----------------------------------------------------*)
      step p r ((i1 :: i2) :: i3) r (i1 :: i2 :: i3)

  | step_fetch_field_0: forall p r v1 v2 i a0 a1 r',
      r#v1 = Some (value_cons a0 a1) ->
      r' = r#v2 <- a0 ->
    (*-----------------------------------------------------*)
      step p r (instr_fetch_field v1 0 v2 :: i) r' i
  | ...
```

The action of looking up a label in a program, presented as a relation $p(l) = \iota$ in the informal specification, could have been expressed in Coq as an inductive predicate. However, we preferred to express it as a lookup function, because then it can be reused as is in the type-checking algorithm.

```
Fixpoint program_lookup (p: program) (l: label)
         {struct p} : option instr :=
  match p with
  | prog_end => None
  | prog_block l' i p' =>
      if label_eq l' l then Some i else program_lookup p' l
  end.
```

The type system The typing rules are also specified as inductive predicates, following very closely the informal specifications. For example:

```
Inductive check_instr: program_typing -> env -> instr -> env -> Prop :=
  | check_instr_branch_list: forall pt e v l t e1,
      e#v = Some (ty_list t) ->
      pt#l = Some e1 ->
      env_sub (e#v <- ty_nil) e1 ->
    (*-----------------------------------------------------*)
      check_instr pt e (instr_branch_if_nil v l) (e#v <- (ty_listcons t))
  | ...
```

The subtyping relation is also defined by an inductive predicate, but for the least upper bound operation, it was more convenient to use a recursive function that proceeds by pattern-matching on the two types. (Again, the recursive function can be reused in the type-checking algorithm.) The `env_sub` predicate for environment subtyping is defined by the following logic formula:

```
Definition env_sub (vt1 vt2: env) : Prop :=
  forall v t2,
  vt2#v = Some t2 -> exists t1, vt1#v = Some t1 /\ subtype t1 t2.
```

This illustrates that Coq specifications are not necessarily computable. In particular, they can involve quantifications over infinite sets.

The type-checking algorithm Coq specifications (terms of sort `Prop`) are not algorithms, in general. The proper way to describe an algorithm in Coq is as a function definition. If the algorithm is a decision procedure for a property (here, "is this program well-typed with respect to this program typing?"), the function returns a boolean (`true` or `false`) — not to be confused with the truth values `True` and `False` of the logic. The type-checking algorithm is therefore described by the following functions:

```
Fixpoint check_subtype (t1 t2: ty) {struct t1} : bool := ...
Definition check_env_sub (e1 e2: env) : bool :=
    map_forall2 ty ty ... e2 e1.
Definition typecheck_branch
    (pt: program_typing) (vt: env) (l: label): bool := ...
Fixpoint typecheck_instr (pt: program_typing) (e: env) (i: instr)
                         {struct i}: option env := ...
Fixpoint typecheck_block (pt: program_typing) (e: env) (i: instr)
                         {struct i}: bool := ...
Fixpoint typecheck_blocks (pt: program_typing) (p: program)
                         {struct p}: bool := ...
Definition typecheck_program
    (pt: program_typing) (p: program) : bool := ...
```

(The `struct` annotations on recursive function definitions indicate the argument over which the structural recursion proceeds.) The function definitions follow closely the pseudo-code given in section 8.2, but use booleans and option types to represent failures in a purely functional style.

13.2 Simulating execution and typing

Tasks 2 and 5 of the challenge are to simulate the execution and typing of a sample program using only the specifications of the operational semantics and the type system. After several unsuccessful attempts, we were able to coerce the Coq proof search facility into simulating executions, as follows. We first populate a so-called "hint database", that we name `exec`, with the following theorems and tactics:

```
Hint Resolve step_seq step_fetch_field_0 step_fetch_field_1
  step_cons step_branch_not_taken step_branch_taken
  step_jump run_halt run_step run_prog_intro: exec.
Hint Extern 1 (_ = _) => (simpl; reflexivity) : exec.
```

The `Hint Resolve` declaration tells Coq to try and apply the given theorems (which are the inference rules for the `step`, `run` and `run_prog` inductive predicates) when the current goal matches the conclusion of one of those theorems. The `Hint Extern` declaration tells Coq to try a proof by simplification and syntactic equality when the goal is an equality. Armed with this hint database, we can state that the sample program should execute, which becomes a proof goal, and use the proof search facility `eauto`:

```
Lemma run_prog_psample: run_prog psample.
Proof.
  unfold psample; eauto 50 with exec.
Qed.
```

(The `50` argument is the maximal depth for proof search. We can put a high value here because search trees for the `exec` hint database have essentially no branching.) The `eauto` tactic succeeds in proving the goal, taking about 0.2 s on a 3 GHz Pentium 4 machine. The sample program therefore evaluates safely.

This approach to simulating executions is not fully satisfactory for several reasons, however.

- The `eauto` tactic provides zero feedback when it fails to prove the goal. Significant trial and error is needed to find the right hints to put in the database. Coq does not provide a variant of `eauto` that would leave as subgoals the parts of the proof that it did not find. (This would at least provide feedback on the missing hints.)
- Even when `eauto` succeeds, it is difficult to know what proof derivation it constructed. The user can do `info eauto` to print a proof script indicating what theorems were applied at each step, but such proof scripts are hard to read.
- The form of unification (between goals and conclusions of theorems) performed by `eauto` is weak and does not expand definitions of names, in particular. This results in mysterious failures if the theorems are not written exactly as the tactic expects them. For instance, our initial definition of the `step` rules had updates over mappings in the conclusions of the rules, as in

```
| step_fetch_field_0: forall p r v1 v2 i a0 a1,
    r#v1 = Some (value_cons a0 a1) ->
  (*-------------------------------------------------*)
    step p r (instr_fetch_field v1 0 v2 :: i) (r#v2 <- a0) i
```

  Written this way, the `r#v2 <- a0` expression in the conclusion has type `map value` instead of the equivalent `store`. Therefore, `eauto` refuses to unify `r#v2 <- a0` with a logical variable of type `store`. We had to reformulate the rule with an additional equality hypothesis, as shown below, to enable `eauto` to work.

```
| step_fetch_field_0: forall p r v1 v2 i a0 a1 r',
    r#v1 = Some (value_cons a0 a1) ->
    r' = r#v2 <- a0 ->
  (*-------------------------------------------------*)
    step p r (instr_fetch_field v1 0 v2 :: i) r' i
```

On task 5 (typing the sample program), the `eauto` approach to simulation fails because there are subgoals of the form `env_sub e1 e2` that cannot be proved automatically: they involve infinite quantification over all variables. Appropriate lemmas and tactics over `env_sub` could possibly be developed and added as hints for `eauto`, but we did not pursue this approach.

13.3 The proofs

The proof of type soundness follows closely the outline given in section 9. The use of mappings to represent environments frees us from the obligation of asserting and proving well-formedness properties over environments and program typings: these are always well-formed by construction. The "good variable" properties of mappings (theorems `get_set_same` and `get_set_other`) are also very effective, enabling much more abstract reasoning and much more proof reuse than in the Twelf proof.

The proofs of the intermediate lemmas are straightforward structural inductions, either on the structure of a type or on a derivation of an inductively defined predicate given as assumption. While the inductions must be given explicitly to Coq (and sometimes this requires shuffling ∀ quantifiers and hypothesis in the statement of the lemma to put it in the form expected by the induction tactics), the remainder of the proofs (deriving the conclusion from the induction hypotheses) can be automated to a large extent using the `eauto` tactic described earlier. By simply adding all inference rules for execution and typing as hints for `eauto`, we were able to automate many of the proofs. This stands in sharp contrast with the Twelf proof, where inductions are discovered largely automatically by the termination checker, but the "plumbing" part of the proofs (deriving the conclusions from the hypotheses) must be given very explicitly as logic programs.

For extra credit, we used Coq's support for coinduction to specify and prove type safety in an alternate way. We define the following coinductive predicate:

```
CoInductive run_safely: program -> store -> instr -> Prop :=
  | run_safely_halt: forall p s,
      run_safely p s instr_halt
  | run_safely_step: forall p s i s' i',
      step p s i s' i' -> run_safely p s' i' ->
      run_safely p s i.
```

Intuitively, `run_safely p s i` means that the program `p`, started in state `s` and `i`, executes for a finite or infinite number of steps without ever getting stuck. We then prove the following type safety theorem using Coq's coinduction:

```
Theorem safety_coinductive:
  forall p pt, check_program pt p -> run_prog_safely p.
```

Grall and Leroy [12] provide background and more details on this class of uses of coinduction in operational semantics.

Turning to the type-checking algorithm, we proved the following theorem showing that it is correct with respect to the specification of the type system:

```
Theorem typecheck_program_correct:
  forall pt p, typecheck_program pt p = true -> check_program pt p.
```

The proof uses a number of supporting lemmas for the other `typecheck` functions, of the general form "if the function does not fail (and returns a result), then the corresponding predicate in the specification holds". The proofs are straightforward but somewhat boring arguments by case analysis on the execution of the `typecheck` functions.

For extra credit again, we also proved the completeness of the type-checking algorithm with respect to the type system:

```
Lemma typecheck_program_complete:
  forall pt p, check_program pt p -> typecheck_program pt p = true.
```

The proofs are slightly simpler than the proofs for correctness.

13.4 What is easy and what is difficult in Coq?

Difficult. The only difficult part of the challenge were tasks 2 and 5: simulating the execution and the typing of the sample program using only the specifications. Generally speaking, Coq does not offer an effective path for execution of inductive specifications. (But see Delahaye, Dubois and Étienne [8] for ongoing work in this direction.) The tricks described in section 13.2 worked for task 2, but it is likely that they do not scale up to more complex specifications.

Easy. Everything else in the challenge was very easy. It took Leroy about one day to translate the specifications into Coq and to do the proofs. The specifications changed several times afterwards, but adapting the Coq development never took more than one hour each time.

14 Indirect Jumps

We have also examined an extension to the list-machine (the list-machine 2.0) which allows indirect jumps. The interest in indirect jumps derives from a desire to handle more interesting control flow; for example, the standard call/return model of procedure calls requires some form of indirect jump to implement returns. For lack of space, we shall only describe the high-level details of this extension here.[9]

In the original machine, values and labels were distinct syntactic categories. To allow the machine to store and compute with labels, we include the labels into the syntax of values. We also remove the nil value; its role will instead be played by the label $\mathbf{L}_0$.

$l_0, l_1, \ldots : L$		labels	$a_0, a_1, \ldots$	$: A$	values
$\mathbf{L}_0, \mathbf{L}_1$	$: L$	label names	$\mathbf{L}_0, \mathbf{L}_1, \ldots$	$: A$	label names
			$\mathrm{cons}(a_1, a_2)$	$: A$	list cell

We also alter the instruction set by removing the static **jump** command, and replacing it with a **get-label** command and an indirect **jump**.

$\iota_0, \iota_1, \ldots$	$: I$	instructions
jump v	$: I$	indirect jump through v
get-label l v	$: I$	load label l into v
branch-if-nil v l	$: I$	if $v = \mathbf{L}_0$ go to l
fetch-field v 0 v'	$: I$	fetch the head of v into v'
fetch-field v 1 v'	$: I$	fetch the tail of v into v'
cons v_0 v_1 v'	$: I$	make a cons cell in v'
halt	$: I$	stop executing
ι_0 ; ι_1	$: I$	sequential composition

The instructions present in both machines have the same reduction rules as before, except that **branch-if-nil** checks for the distinguished label $\mathbf{L}_0$ rather than nil. The rules for the new instructions are given below.

[9] See `http://www.cs.princeton.edu/~appel/listmachine/2.0/` for more information about this extension and proofs.

$$\frac{r(v) = l \quad p(l) = \iota'}{(r, \textbf{jump}\ v) \overset{p}{\mapsto} (r, \iota')}\ \text{step-jump}$$

$$\frac{r[v := l] = r'}{(r, \textbf{get-label}\ l\ v; \iota) \overset{p}{\mapsto} (r', \iota)}\ \text{step-get-label}$$

14.1 Type systems

We can apply a type discipline nearly identical to the one from section 5 where we replace the typing rule for the direct **jump** with a rule which types the two instruction sequence **get-label** $v\ l$;**jump** v with the caveat that v must not appear in the environment for l.

$$\frac{\Pi(l) = \Gamma_1 \quad \Gamma \subset \Gamma_1 \quad v \notin \mathrm{Dom}(\Gamma_1)}{\Pi; \Gamma \vdash_{\mathrm{block}} \textbf{get-label}\ l\ v\ ;\ \textbf{jump}\ v}\ \text{check-block-jump2}$$

However, we gain significant expressive power if we move to a richer type discipline where we allow interesting uses of labels. In order to do this we need to introduce a type for *continuations*, that is, program labels. For technical reasons, we also add the top and bot types.

$\tau_0, \tau_1, \ldots$	$: T$	type
nil	$: T$	singleton type containing L_0
list τ	$: T$	list whose elements have type τ
listcons τ	$: T$	non-nil list of τ
cont Γ	$: T$	Label with environment Γ
top	$: T$	The universal type
bot	$: T$	The uninhabited type

Continuation types take a type environment and the interpretation of cont Γ is that the value is a label to which the machine may be safely jump if the register bank satisfies Γ. The continuation types significantly increase the complexity of the type system because they contain an entire type environment and because they are contravariant with respect to subtyping. Notice that the definition of types and the definition of type environments have become mutually inductive.

In addition to the subtyping rules from before, we add the subtyping rule for continuations and for top and bot:

$$\frac{}{\text{bot} \subset \tau}\ \text{subtype-bot}$$

$$\frac{}{\tau \subset \text{top}}\ \text{subtype-top}$$

$$\frac{\Gamma_2 \subset \Gamma_1}{\text{cont}\,\Gamma_1 \subset \text{cont}\,\Gamma_2}\ \text{subtype-cont}$$

As before, subtyping of type environments is interpreted both width- and depth-wise, in the "obvious" way. Just as the definitions of types and environments became mutually inductive, so too have the definitions of subtyping and environment subtyping become mutually inductive.

Since we have enhanced the definition of subtyping, we must also enhance the definition of least upper bounds. It happens that the original type system for the list machine naturally forms a lower semilattice; however, with the addition of continuation types we now have types which are "incompatible." We must artificially complete the space of types in order for us to be able to define least upper bounds. This requires adding the bot type and, surprisingly, the top type as well. The reason we also need top is because of the contravariance of the continuation type. In order to calculate $\mathrm{cont}\,\Gamma_1 \sqcup \mathrm{cont}\,\Gamma_2$ we must calculate the *greatest lower bound*, $\Gamma_1 \sqcap \Gamma_2$. This means the calculation of least upper bounds and greatest lower bounds must be mutually recursive with each other and with the same operations on type environments. This requires, in turn, that we must complete the upper part of the lattice with top just as we must complete the lower part of the lattice with bot.

It is worth noting that the distinguished label $\mathbf{L}_0$ is something of a corner case. It plays a dual role as both the nil value and as a legal program label; it makes sense to assign both the type nil and the type $\mathrm{cont}\,\Gamma$, where Γ is the type environment for $\mathbf{L}_0$. However, our subtyping rules do not make either of these types a subtype of the other. In particular, the glb of nil and $\mathrm{cont}\,\Gamma$ is bot, despite the fact that L_0 may inhabit both of these types. This means our syntactic notion of subtyping does not completely coincide with the "semantic" notion of subtyping and that types are only a lattice when considered as a syntactic system. We could fix the issue by making jumps to $\mathbf{L}_0$ illegal (forcing nil as its only legal type), but the fix seems contrived. We could instead fix the problem by defining subtyping as a judgment parameterized by program typings; this would allow us to know which Γ we should use to make nil a subtype of $\mathrm{cont}\,\Gamma$. However, complicating the judgment just to fix this small issue seems like overkill. We prefer to leave this little wart in as a reminder of the kinds of issues that can arise when working with realistic assembly languages.

Once we have modified subtyping and lubs in the required way, most of the typing system is unchanged from the original system (again, modulo the change from nil to $\mathbf{L}_0$); we need only modify the rule for **jump** and add rules for **get-label**:

$$\frac{\Gamma[v := \mathrm{nil}] = \Gamma'}{\Pi \vdash_{\mathrm{instr}} \Gamma\{\textbf{get-label}\ v\ \mathbf{L}_0\}\Gamma'}\ \text{check-instr-get-label-0}$$

$$\frac{\Pi(l) = \Gamma_1 \quad \Gamma[v := \mathrm{cont}\,\Gamma_1] = \Gamma'}{\Pi \vdash_{\mathrm{instr}} \Gamma\{\textbf{get-label}\ v\ l\}\Gamma'}\ \text{check-instr-get-label}$$

$$\frac{\Gamma(v) = \mathrm{cont}\,\Gamma_1 \quad \Gamma \subset \Gamma_1}{\Pi; \Gamma \vdash_{\mathrm{block}} \textbf{jump}\ v}\ \text{check-block-jump}$$

14.2 Soundness proofs

We have constructed soundness proofs for the list machine 2.0 in Coq for both the simplistic type system (which types only the **get-label** l v;**jump** v macro) and the full system with continuation types. Both proofs are done in the style of the "Very Modal Model" of Appel et al. [3], and are decidedly nonsyntactic. We also have a syntactic proof for the simple type system in Coq which is very similar to the proof described in section 13. Adam Chlipala has also contributed a semantic proof for the

simple system using an argument based on logical relations.[10] We have not attempted proofs of these systems in Twelf.

The simple type system (i.e., without continuations) presented no particular issues of interest, beyond what was described earlier in this paper. Most of the interesting issues with the proof of soundness for the advanced typechecker revolved around the mutual recursion between types and environments and with the contravariance of continuation types. Defining and demonstrating the termination of the subtype checking algorithm and of the lub/glb algorithms was especially troublesome.

In the future it would be interesting to do a syntactic proof of the type system with continuations for comparison purposes, and also to express the syntactic proof in Twelf's metalanguage.

14.3 Modularization

In our proof development for the 2.0 list machine, we pushed the idea of separating the soundness of the type *system* from the soundness of the typechecking *algorithm* to its conclusion by enforcing an abstraction boundary using Coq's module system. The type systems we examined were formulated as an abstract module type. The module types specify the rules of the type systems and the safety property the type system is supposed to satisfy. The typechecking algorithms are proved sound with respect to the module type. Implementations of the type system realize the module by providing concrete representations of the types, proofs of the rules, and a proof of soundness. Then any implementation of the module type can be combined with the algorithmic soundness proof to get an end-to-end proof of soundness for the typechecker.

This aggressively modularized development made it easy for us to experiment with different approaches to the soundness proof for the type system. We proved the simple type system sound using both proof-theoretic methods and a method based on the Very Modal Model. In addition, an external contributor (Adam Chlipala) was able to implement the module type in an entirely different way, using logical relations.

15 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 [17] 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.

[10] `http://adam.chlipala.net/poplmark/`

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.

A Summary of notation

In order to better compare different solutions to the benchmark, solutions should use common notation where possible. This table shows the recommended ASCII representations. The underscore _ or hyphen - may be used to separate words, depending on what the MM permits.

In some cases we show two ASCII representations. In most cases this is because our Twelf solution uses a syntactic relational style in a case where the Coq solution uses a semantic functional style, and no identical common notation is possible.

Notation	ASCII representation	page
A	value	3
nil	value-nil	3
cons(a_1, a_2)	value-cons a1 a2	3
V	var	3
$\mathbf{v}_n$	var# n *or* V n	3
R	store	3
$r[v \mapsto a]$	store-bind v a r *or* r#v<-a	3
$\{\}$	empty-store	3
$r(v) = a$	var-lookup r v a *or* r#v = Some a	4
$r[v := a] = r'$	var-set r v a r′ *or* r#v<-a = r′	4
L	label	4
$\mathbf{L}_n$	label# n *or* L n	4
I	instr	4
jump l	instr-jump l	4
branch-if-nil v l	instr-branch-if-nil v l	4
fetch-field v i v'	instr-fetch-field v i v′	4
cons v_0 v_1	instr-cons v0 v1	4
halt	instr-halt	4
ι_0 ; ι_1	i0 ,instr i1	4
P	block *and/or* program	4
end	prog-end	4
$l : \iota;\ p$	def-label l i ,block p *or* prog-block l i p	4
$p(l) = \iota$	program-lookup p l i *or* program-lookup p l = Some i	5
$(r, \iota) \overset{p}{\mapsto} (r', \iota')$	step p r i r′ i′*i*	5

Notation	ASCII representation	page
$(p, r, \iota) \Downarrow$	run p r i	5
$p\Downarrow$	run-prog p	5
step-or-halt(p, r, ι)	step-or-halt p r i	10
T	ty	6
nil	ty-nil	6
list τ	ty-list t	6
listcons τ	ty-listcons t	6
E	env	6
$\{\,\}$	env-empty	6
$v : \tau,\ \Gamma$	var-has-type v t g *or* g#v<-t	6
$\Gamma(v) = \tau$	g#v = Some t	6
$\Gamma \doteq (v : \tau,\ \Gamma')$	env-lookup g v t g′	11
$\Gamma[v := \tau] = \Gamma'$	env-set g v t g′ *or* g#v<-t = g′	6
$\vdash_{\text{env}} \Gamma$	env-ok g	7
$\tau \subset \tau'$	subtype t t′	6
$\tau_1 \sqcup \tau_2 = \tau_3$	lub t1 t2 t3 *or* lub t1 t2 = t3	7
PT	program-typing	7
Π_0	program-typing-empty	7
$l : \Gamma,\ \Pi$	block-has-type l g pi *or* pi#l<-g	7
$\Pi; \Gamma \vdash_{\text{block}} \iota$	check-block pi g i	8
$\Pi \vdash_{\text{blocks}} \Gamma$	check-blocks pi g	8
$\Pi \bowtie p$	typing-dom-match pi p	8
Γ_0	env0	8
$\models_{\text{prog}} p : \Pi$	check-program p pi	8
$\vdash_{\text{prog}} p : \Pi$	fast-check-program p pi *or* typecheck-program p pi = true	8
p_sample	sample-prog	9
Π_sample	sample-ty	9

References

1. Appel, A.W.: Hints on proving theorems in Twelf (2000). `http://www.cs.princeton.edu/~appel/twelf-tutorial`
2. Appel, A.W., Leroy, X.: List-machine exercise (2006). `http://www.cs.princeton.edu/~appel/listmachine/`
3. Appel, A.W., Melliès, P.A., Richards, C.D., Vouillon, J.: A very modal model of a modern, major, general type system. In: Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07), pp. 109–122. ACM Press (2007)
4. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLmark challenge. In: Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2005), *Lecture Notes in Computer Science*, vol. 3603, pp. 50–65. Springer-Verlag (2005). URL `http://plclub.org/mmm/`
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer-Verlag (2004)
6. The Coq proof assistant (1984–2010). Software and documentation available from `http://coq.inria.fr/`
7. Danvy, O.: Defunctionalized interpreters for programming languages. In: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, pp. 131–142. ACM Press (2008)
8. Delahaye, D., Dubois, C., Étienne, J.F.: Extracting purely functional contents from logical inductive types. In: Theorem Proving in Higher Order Logics (TPHOLs 2007), *Lecture Notes in Computer Science*, vol. 4732, pp. 70–85. Springer-Verlag (2007)
9. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Syst. **28**(4), 619–695 (2006)

10. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler. In: 3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), pp. 2–11. IEEE Computer Society Press (2005)
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
12. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation **207**(2), 284–304 (2009)
13. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Trans. Progr. Lang. Syst. **21**(3), 528–569 (1999)
14. Pfenning, F., Schuermann, C.: Twelf user's guide, version 1.4 (2002). `http://www.cs.cmu.edu/~twelf/guide-1-4`
15. Pfenning, F., Schürmann, C.: 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*, vol. 1632, pp. 202–206. Springer-Verlag, Berlin (1999)
16. Weirich, S.: Experience report with Twelf (2005). URL `http://lists.seas.upenn.edu/pipermail/poplmark/2005-August/000220.html`. E-mail to POPLmark mailing list, August 17
17. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1), 38–94 (1994)
18. Wu, D., Appel, A.W., Stump, A.: Foundational proof checkers with small witnesses. In: 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pp. 264–274. ACM Press (2003)