Charge! a framework for higher-order separation logic in Coq

Lars Birkedal

Logic and Semantics Group
Dept. of Comp. Science, Aarhus University

(Joint work with J. Bengtson, J.B. Jensen, H. Mehnert, P. Sestoft, F. Sieczkowski)

April 2013

Separation Logic

- Program Logic a la Hoare Logic for reasoning about programs with pointers (or references to shared mutable data) [Reynolds, O'Hearn, . . ., 2000+]
- Main feature: it facilitates modular reasoning, formalized via so-called frame rule, using a connective called separating conjunction.

Hoare Logic - a recap

- Programming Language: imperative `while`-language
- Assertion Language: first-order logic w. equality
- Specifications for partial correctness:
  - $\{P\}C\{Q\}$
  - if $s \Vdash P$ and $C, s \to s'$, then $s' \Vdash Q$.
- Rules for deriving specifications (including rules for first-order logic).

Separation Logic

- Programming Language: as before but now with C-like pointers
- Specifications for partial correctness:
  - $\{P\}C\{Q\}$
  - if $s, h \Vdash P$ and $C, s, h \rightarrow s', h'$, then $s', h' \Vdash Q$.
- Assertion Language: first-order logic w. equality + BI connectives:
  - $s, h \Vdash emp$ iff h is the empty heap
  - $s, h \Vdash x \mapsto 5$ iff h is the singleton heap with one location $s(x)$ with value 5.
  - $s, h \Vdash P * Q$ iff h can be split into h_1 and h_2, with disjoint domains, such that $s, h_1 \Vdash P$ and $s, h_2 \Vdash Q$.
  - $s, h \Vdash P \mathrel{-\!\!*} Q$ iff ....

Example of "small" axiom

$$\overline{\{x \mapsto -\}\mathsf{dispose}(x)\{emp\}}$$

Modular Reasoning via Frame Rule:

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}}$$

(assuming $\mathit{modifies}(C) \cap \mathit{freevar}(R) = \emptyset$).

Higher-Order Separation Logic Example

- Developed HOSL in ESOP-2005 paper.

Stack ADT

$$
\begin{aligned}
&\mathsf{stackspec} = \\
&\exists \alpha : \mathrm{Type}.\exists \mathit{inv} : \alpha \times \mathsf{N}\,\mathrm{list} \to \mathrm{Prop}. \\
&\qquad \{\mathit{emp}\}\,\mathtt{new}()\,\{s : \alpha.\, \mathit{inv}(s, [])\} \times \\
&\qquad \forall s : \alpha.\forall x : \mathsf{N}.\forall l : \mathsf{N}\,\mathrm{list} \\
&\qquad\quad \{\mathit{inv}(s, l)\}\,\mathtt{push}(s, x)\,\{\mathit{inv}(s, x :: l)\} \times \\
&\qquad \forall s : \alpha.\forall x : \mathsf{N}.\forall l : \mathsf{N}\,\mathrm{list} \\
&\qquad\quad \{\mathit{inv}(s, x :: l)\}\mathtt{pop}(s)\{y : \mathsf{N}.\, \mathit{inv}(s, l) \wedge y = x\}.
\end{aligned}
$$

- Modularity: clients can use the spec without knowing anything about how the stack is implemented (since abstract in the inv predicate).
- Different stack implementations can meet this spec.

Charge!

Aims

- tool for verification of OO (Java / C#) code
- machine-checkable correctness proofs
- proofs similar to pen and paper proofs in separation logic
- automate tedious first-order reasoning where possible

Idea:

- test the theory on larger examples (hard to do with pen and paper)
- make it available for students to experiment with

Real-life test cases:

- from the C5 collection library for C#
  - uses interfaces heavily to parameterize modules on each other
  - ("interfaces" is the OO way of programming with unknown code)

Other approaches

When we started work on Charge:

- For similar prog. language and logic: Verifast / jStar
  - specialized tools,
  - Verifast: larger TCB (own specialized theorem prover). We focus more on modular reasoning about modular code, using facilities of Coq
  - jStar: no guarantee of soundness since based on user-declared theories
- Language designed arround type theory: Hoare Type Theory / Ynot [Nanevski, Morrisett, Birkedal, Chlipala, et. al.]
  - now we wanted to test ideas for logic for existing mainstream programming language
- C-like language: Appel / McCreight macros.
  - useful ideas that we build on, exposes the model of separation logic to user, reasoning about heaps directly, which we seeked to avoid

Overview of Charge!

- Shallow embedding of HOSL in Coq.
- Support for program variables in assertions
- Step-indexed specification logic with quantifiers
- Nested triples, i.e. step-indexed specifications in assertions
- Hoare triple defined on semantic commands
- Building blocks for defining control flow constructs:

$$\textbf{id} \qquad \textbf{seq}\ \hat{c}_1\ \hat{c}_2 \qquad \hat{c}_1 + \hat{c}_2 \qquad \hat{c}^* \qquad \textbf{assume}\ P$$

- Instantiation to Java / C# like OO language.
- Static types replaced by specifications
- Tactics to automate part of the reasoning.
- Test applications
  - Pattern for interface specifications (since C5 relies on interfaces)
  - Snapshotable trees (datastructure with *sharing*)

Uniform Predicates

Definition (Separation Algebra)

A separation algebra is a partial, cancellative, commutative monoid $(\Sigma, \circ, \mathbf{1})$ where Σ is the carrier, $\circ$ is the monoid operator, and $\mathbf{1}$ is the unit element.

- Generalizes heaps, so we often refer to elements of Σ as heaps.
- $h_1 \sqsubseteq h_2$, if there exists an h_3 such that $h_2 = h_1 \circ h_3$.

Definition (Uniform Predicate)

$$UPred(\Sigma) \triangleq$$
$$\{p \subseteq \Sigma \times \mathbb{N} \mid \forall g, m.\ \forall h \sqsupseteq g.\ \forall n \leq m.\ (g, m) \in p \rightarrow (h, n) \in p)\}$$

Uniform Predicates, II

Lemma

UPred(Σ) *forms a complete BI algebra.*

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{true} & \triangleq & \Sigma \times \mathbb{N} & \mathit{false} & \triangleq & \emptyset \\
p \wedge q & \triangleq & p \cap q & p \vee q & \triangleq & p \cup q \\
\forall x : U.\, f & \triangleq & \bigcap_{x:U} f\ x & \exists x : U.\, f & \triangleq & \bigcup_{x:U} f\ x \\
p \to q & \triangleq & \{(h,n) \mid \forall g \sqsupseteq h.\, \forall m \leq n.\, (g,m) \in p \to (g,m) \in q\} & & & \\
p * q & \triangleq & \{(h_1 \circ h_2, n) \mid h_1 \# h_2 \wedge (h_1,n) \in p \wedge (h_2,n) \in q\} & & & \\
p \mathrel{-\!\!*} q & \triangleq & \{(h,n) \mid \forall m \leq n.\, \forall h_1 \# h.\, (h_1,m) \in p \to (h \circ h_1, m) \in q\} & & &
\end{array}
$$

For the quantifiers, U is of type *Type*, i.e. the sort of types in Coq, and f is any Coq function from U to *UPred*(Σ). This allows us to quantify over *any* member of *Type* in Coq.

Stacks and Assertions

Definition

- $stack \triangleq var \to val$.
- $s \simeq_V s' \triangleq \forall x \in V.\ s\ x = s'\ x$.
- Stack monad:

$$sm\ T \triangleq \{(f : stack \to T,\ V : \mathcal{P}(var)) \mid \forall s, s'.\ s \simeq_V s' \to f\ s = f\ s'\}$$

(intuition: V over-approx of free variables in f).

Definition

$$expr \triangleq sm\ val \qquad pure \triangleq sm\ Prop \qquad asn(\Sigma) \triangleq sm\ UPred(\Sigma)$$

Lemma

$asn(\Sigma)$ forms a complete BI algebra.

Stacks and Substitution

- Stack monad used to define map $f \mapsto \widehat{f}$
- If $f : T \to U$, then $\widehat{f} : sm\ T \to sm\ U$.
- Example
  - Given $R : val \to T \to UPred(heap)$, then
  - $\widehat{R} : sm\ val \to sm\ T \to sm\ UPred(heap)$, i.e.
  - $expr \to sm\ T \to asn(heap)$.
- Make lifting explicit in specifications since restricts how *program variables* behave under substitution.
  - $(\widehat{f}\ e)[e'/x] = \widehat{f}\ (e[e'/x])$ for any $f : val \to UPred(\Sigma)$,
  - But do NOT have $(g\ e)[e'/x] = g\ (e[e'/x])$ for any $g : expr \to asn(\Sigma)$, because $g\ e$ may have more free program variables than those appearing in e.
  - Hence in specs we typically quantify over functions into $UPred(\Sigma)$, which we then lift to $asn(\Sigma)$ as needed.

Semantic Commands

Definition (pre-command)

A *pre-command* $\tilde{c}$ relates an initial state to either a terminal state or the special **err** state:

$$precmd \triangleq \mathcal{P}(stack \times \Sigma \times ((stack \times \Sigma) \uplus \{\mathbf{err}\}) \times \mathbb{N})$$

We write $(s, h, \tilde{c}) \rightsquigarrow^n x$ to mean that $(s, h, x, n) \in \tilde{c}$.

Definition (Frame property)

A pre-command $\tilde{c}$ has the frame property in case the following holds. If $(s, h_1, \tilde{c}) \not\rightsquigarrow^n \mathbf{err}$ and $(s, h_1 \circ h_2, \tilde{c}) \rightsquigarrow^n (s', h')$ then there exists h'_1 such that $h' = h'_1 \circ h_2$ and $(s, h_1, \tilde{c}) \rightsquigarrow^n (s', h'_1)$.

Semantic Commands, II

Definition (Semantic command)

A semantic command satisfies the frame property and does not evaluate to anything in zero steps.

$semcmd \triangleq \{\hat{c} \in precmd \mid \hat{c}$ has the frame property $\wedge \forall s, h, x.\ (s, h, \hat{c}) \not\rightarrow^0\}$

The following are semantic commands:

$$\textbf{id} \qquad \textbf{seq}\ \hat{c}_1\ \hat{c}_2 \qquad \hat{c}_1 + \hat{c}_2 \qquad \hat{c}^* \qquad \textbf{assume}\ P \qquad \textbf{check}\ P$$

(The **check**-command works like the **id**-command if pure assertion P can be inferred.)

Specification Logic

Definition

$$spec \triangleq \{S \subseteq \mathbb{N} \mid \forall m, n.\ m \leq n \wedge n \in S \rightarrow m \in S\}$$

Gives an intuitionistic specification logic:

Lemma

$(spec, \subseteq)$ *is a cHA.*

Definition (Hoare Triple)

$$\begin{aligned}\{P\}\hat{c}\{Q\} \triangleq \{n \mid\ & \forall m \leq n.\ \forall k \leq m.\ \forall s, h. \\ & (h, m) \in P\ s \rightarrow (s, h, \hat{c}) \not\rightsquigarrow^{k} \mathbf{err} \wedge \\ & \forall h', s'.\ (s, h, \hat{c}) \rightsquigarrow^{k} (s', h') \rightarrow (h', m - k) \in Q\ s'\}\end{aligned}$$

Nested Specifications

Nested specifications for reasoning about higher-order / unknown code.

$$\mathit{FunI} : \mathit{spec} \to \mathit{UPred}(\Sigma) \triangleq \lambda S.\ \Sigma \times S.$$

Lemma

FunI is monotone, preserves implication, and has a left and a right adjoint.

So all specification logic connectives preserved.
(Could merge specifiation and assertion logic.)

Recursion

Step-indexing used for mutually recursive specifications.

$$\triangleright S \triangleq \{n+1 \mid n \in S\} \cup \{0\}$$

$$\frac{\Gamma \wedge \triangleright S \models S \qquad 0 \in \Gamma \rightarrow 0 \in S}{\Gamma \models S} \text{ Löb}$$

Instantiation to OO lang

Shallow embedding of expressions (recall $expr = sm\ val$); deep embedding of other syntax:

$$\begin{aligned}
\mathcal{P} &::= \mathcal{C}^* \qquad\qquad f \in (\text{field names}) \\
\mathcal{C} &::= \textbf{class}\ C\ f^*\ (m(\bar{x})\{c;\ \textbf{return}\ e\})^* \\
c &::= x := \textbf{alloc}\ C \mid x := e \mid x := y.f \mid x.f := e \mid x := y.m(\bar{e}) \\
&\quad \mid x := C::m(\bar{e}) \mid \textbf{skip} \mid c_1;\ c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \\
&\quad \mid \textbf{while}\ e\ \textbf{do}\ c \mid \textbf{assert}\ e
\end{aligned}$$

$$\Sigma = heap \triangleq (ptr \times field) \xrightarrow[\text{fin}]{} val$$

Semantics defined via (functional) relation $c \sim_{\text{sem}} \hat{c}$, relating syntactic command to semantic command.

Syntactic Hoare Triples

Definition

$$\{P\}c\{Q\} \triangleq \forall \hat{c}.\ c \sim_{\mathsf{sem}} \hat{c} \to \{P\}\hat{c}\{Q\}.$$

Definition

$$C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \triangleq \exists c, e.\\ wf(\bar{p}, r, P, Q, c) \wedge C::m(\bar{p})\{c; \textbf{return}\ e\} \in \mathcal{P}\\ \wedge \{P\}c\{Q[e/r]\},$$

(where *wf* expresses that method parameter names do not clash; that pre- and postconditions do not use any stack variables other than the method parameters and this (the postcondition may also use the return variable); the method body does not modify the values of the method parameters or this).

Example Proof Rules

- Rule for method call uses $\triangleright$, often used in connection with Löb induction.

$$\frac{\Gamma \models \triangleright C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \qquad |\bar{p}| = |y, \bar{e}|}{\Gamma \models \{y : C \wedge P[y, \bar{e}/\bar{p}]\}x := y.m(\bar{e})\{\exists v.\ Q[x, y[v/x], \bar{e}[v/x]/r, \bar{p}]\}}$$

- Frame Rule (uses notion of free vars for assertions)

$$\frac{\forall x \in \mathit{fv}\ R.\ c\ \mathit{does\ not\ modify}\ x}{\{P\}c\{Q\} \models \{P * R\}c\{Q * R\}}$$

- All proof rules proved sound in Coq.

Example Java program

```
interface ICell {
  /**
   * Should return the last
   * value passed to set() */
  int get();

  void set(int v);
}

static void proxySet(ICell c, int v) {
  c.set(v);
}
```

```
c := new Recell();
c.set(1);
proxySet(c, 2);
c.undo();
assert c.get() = 1;
```

Note: unknown code in *proxySet*.
Modularity point: specify and verify *proxySet* before knowing about *Recell* instance, and still be able to prove client program.

Recell implementation

```
interface IRecell extends ICell {
  void undo();
}
```

```
class Recell implements IRecell {
  Cell cell;
  int bak;
  Recell() {
    this.cell = new Cell();
  }
  int get() {
    return this.cell.get(); }
  void set(int v) {
    this.bak = this.cell.get();
    this.cell.set(v); }
  void undo() {
    this.cell.set(this.bak); }
}
```

Interfaces as specifications

ICell $C\ T\ R\ g\ s$ is a predicate in the specification logic.

$$
\begin{aligned}
\mathit{ICell} \triangleq\ & \lambda C : \mathit{classname}. \\
& \lambda T : \mathit{Type}. \\
& \lambda R : \mathit{val} \to T \to \mathit{UPred}(\mathit{heap}). \\
& \lambda g : T \to \mathit{val}. \\
& \lambda s : T \to \mathit{val} \to T. \\
& (\forall t : T.\ C{::}\mathrm{get}(\mathrm{this}) \mapsto \{\widehat{R}\ \mathrm{this}\ t\}\_\{\mathrm{x}.\ \widehat{R}\ \mathrm{this}\ t \wedge \mathrm{x} = g\ t\}) \wedge \\
& (\forall t : T.\ C{::}\mathrm{set}(\mathrm{this}, \mathrm{x}) \mapsto \{\widehat{R}\ \mathrm{this}\ t\}\_\{\widehat{R}\ \mathrm{this}\ (\widehat{s}\ t\ \mathrm{x})\}) \wedge \\
& (\forall t : T, v : \mathit{val}.\ g\ (s\ t\ v) = v)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{proxySet\_spec} \triangleq\ & \\
& \forall C, T, R, g, s.\ \mathit{ICell}\ C\ T\ R\ g\ s \to \\
& \forall t : T.\ \mathrm{proxySet}(\mathrm{c}, \mathrm{x}) \mapsto \{\mathrm{c} : C \wedge \widehat{R}\ \mathrm{c}\ t\}\_\{\widehat{R}\ \mathrm{c}\ (\widehat{s}\ t\ \mathrm{x})\}
\end{aligned}
$$

Cell instance

$$\begin{aligned} \mathit{Cell\_spec} \triangleq \exists R : \mathit{val} \to \mathit{val} \to \mathit{UPred}(\mathit{heap}).\ & \\ & \mathit{ICell}\ \mathrm{Cell}\ \mathit{val}\ R\ (\lambda v.\ v)\ (\lambda \_, v.\ v) \wedge \\ & \mathrm{Cell}{::}\mathrm{new}() \mapsto \{\mathit{true}\}\_\{\mathrm{ret}.\ \widehat{R}\ \mathrm{ret}\ \_\} \end{aligned}$$

Unfold definitions to get

$$\begin{aligned} \mathit{Cell\_spec} = & \\ & \exists R : \mathit{val} \to \mathit{val} \to \mathit{UPred}(\mathit{heap}). \\ & (\forall t : T.\ \mathrm{Cell}{::}\mathrm{get}(\mathrm{this}) \mapsto \{\widehat{R}\ \mathrm{this}\ t\}\_\{\mathrm{x}.\ \widehat{R}\ \mathrm{this}\ t \wedge \mathrm{x} = t\}) \wedge \\ & (\forall t : T.\ \mathrm{Cell}{::}\mathrm{set}(\mathrm{this}, \mathrm{x}) \mapsto \{\widehat{R}\ \mathrm{this}\ t\}\_\{\widehat{R}\ \mathrm{this}\ \mathrm{x}\}) \wedge \\ & \mathrm{Cell}{::}\mathrm{new}() \mapsto \{\mathit{true}\}\_\{\mathrm{ret}.\ \widehat{R}\ \mathrm{ret}\ \_\} \end{aligned}$$

Recell instance

$$\mathit{Recell\_spec} \triangleq$$

$$\exists R : \mathit{val} \to \mathit{val} \times \mathit{val} \to \mathit{UPred}(\mathit{heap}).$$

$$\mathit{ICell}\ \mathrm{Recell}\ (\mathit{val} \times \mathit{val})\ R\ \pi_1\ (\lambda(v, \_), v'.\ (v', v)) \wedge$$

$$\mathrm{Recell}::\mathrm{new}() \mapsto \{\mathit{true}\}\_\{\mathrm{ret}.\ \widehat{R}\ \mathrm{ret}\ (\_, \_)\} \wedge$$

$$(\forall v, b.\ \mathrm{Recell}::\mathrm{undo}(\mathrm{this}) \mapsto \{\widehat{R}\ \mathrm{this}\ (v, b)\}\_\{\widehat{R}\ \mathrm{this}\ (b, b)\})$$

Proof of client now possible

- even though *proxySet* verified independently (in fact before) of *Recell*.
- so OO style inheritance among interfaces can be specified in HOSL (no special features added to support inheritance)
- formalization borrows types, logic variable handling and higher-order features from Coq
  - avoids having to build ad-hoc copies of those features
  - program variable handling is more manual and interacts with higher-order featues (using the explicit lifting)

Tactics

- Challenge: we would like the user to reason in separation logic
- Coq tactics work on Coq logic
- Need some tactics to support reasoning in the shallowly embedded program logic.
- Our tactics [ITP-2012] work on any assertion logic, over general separation algebras.
- Implementation uses reflection for fragments of separation logic formulas
- Also uses hint databases (to add simplification rules to Coq database that the Coq auto-tactic can use)
- Most tactics require that formulas are of certain normal forms
- Net result is that user mostly has to give method specs and invariants.

Case Study: Snapshotable trees [VSTTE-2012]

- Algorithm internally uses a lot of sharing.
- Challenging for separation logic.
- We used approach based on ramification:
  - $\forall \varphi.\, \mathsf{inv}(\{\mathsf{Tree}(\mathtt{this}, \tau) \uplus \mathsf{Snap}(\ldots) \uplus \varphi\})$
- Suggested fictional separation logic.
- Development size (after VSTTE):

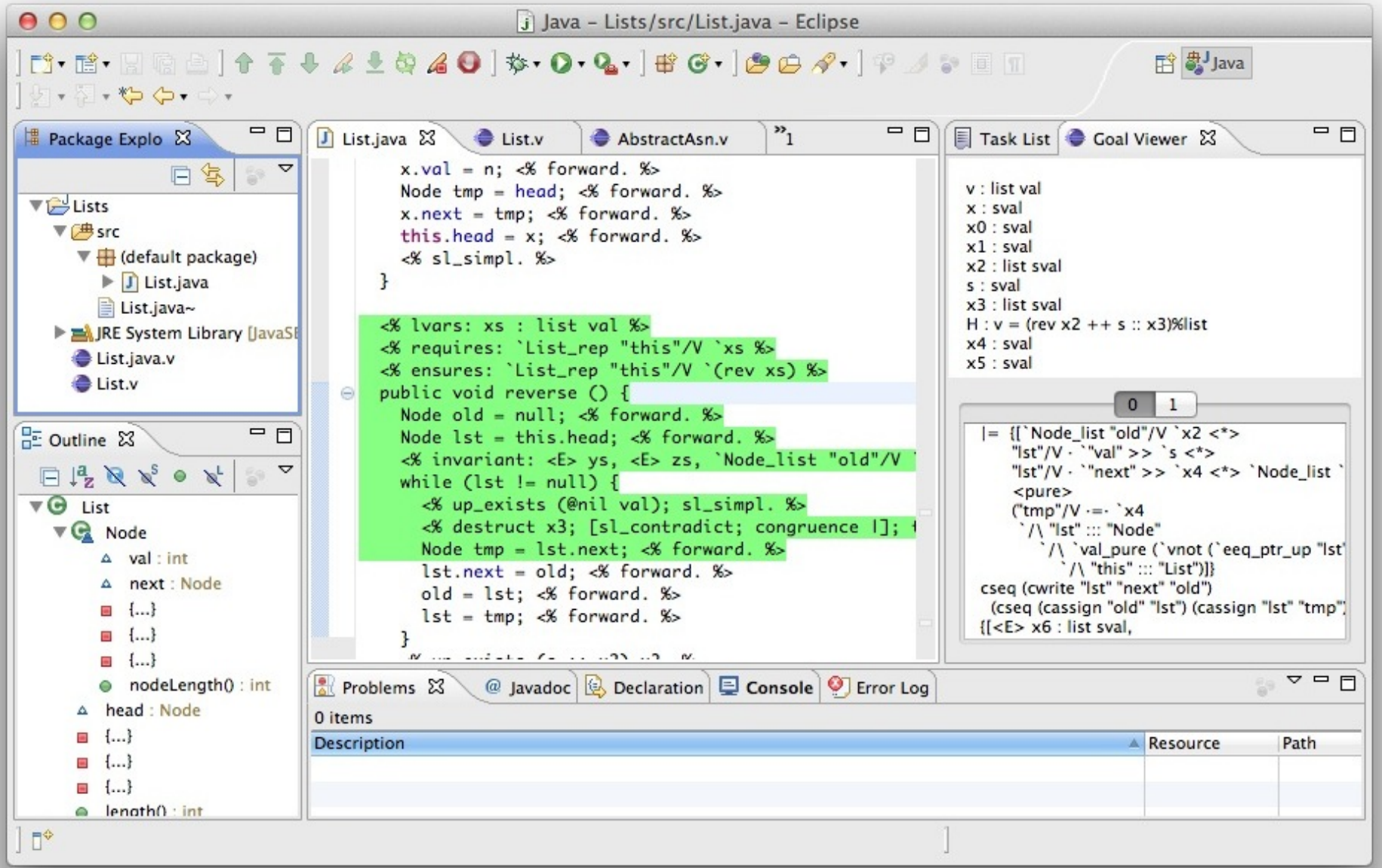
	Code spec	Code proof	Code total	Incl. model
pre-tactics	226	3325	3713	5083
post-tactics	156	671	963	2202

- So a dramatic reduction in proof size due to tactics, but still non-trivial effort.

Some general lessons

- For complicated model structures (as needed for snapshotable trees, e.g.) that are not already part of Coq library, most of the effort seems to be on the mathematical model!
- Observe: unlike most work in prog. lang., we are not only using Coq for meta-theory, but would also like to use the embedded logic.
- Need more efficient ways of reasoning in embedded logics!

Kopitiam — Eclipse plug-in for Java and Coq



Selected Ongoing / Future

- Extend Charge! with fictional separation logic
- Extend Charge! to experiment with higher-order frame rules
- Formalize the specification of Joins library in Charge!
- Extend type theory with guarded recursion to make it easier to formalize step-indexed models (cf. LICS 2011 and LICS 2013 papers)

Advertising!

- I recently moved to Aarhus, starting new group in Logic and Semantics.
- Open position as Associate Professor, announced on types mailing list.
- Also looking for postdocs and PhD students!
- Please talk to me later if you know of potentially interested candidates.

References

- J. Bengtson, J.B. Jensen, F. Sieczkowski, and L. Birkedal: Verifying object-oriented programs with higher-order separation logic in Coq. [ITP-2011]
- J. Bengtson, J.B. Jensen, and L. Birkedal: Charge! a framework for higher-order separation logic in Coq. [ITP-2012]
- H. Mehnert, F. Sieczkowski, L. Birkedal, and P. Sestoft: Formalized verification of snapshotable trees: Separation and sharing. [VSTTE-2012]