Some Challenges for Future ITP

Andy Gordon
Microsoft Research

Workshop on Interactive Theorem Proving (ITP), University of Cambridge, August 2009
Typecheckers, Refinements, Provers

• Refinement (aka subset) types are a Hot Topic in PL design
  – \texttt{type} \texttt{pos} = x: \texttt{int} \{x > 0\}
  \texttt{val} \texttt{sqrt} : x: \texttt{real} \{x >= 0\} -> r: \texttt{real} \{x = r*r\}
  – Unifies behavioural types, security types, patterns, Hoare logic, etc
  – Typechecker generates logical goals, passes to automatic prover
  – But what if ATP fails, can we somehow appeal to ITP?

1. F7: Refinement Types for a Concurrent ML
2. Minim: Refinement Types for a Database Query Language
3. Some Observations, Some Challenges

**F7 – REFINEMENT TYPES FOR ML WITH CONCURRENCY (F#/OCAML)**
Problem of Verifying Protocol Code

• The problem of vulnerabilities in security protocols is remarkably resistant to the success of formal methods
• Perhaps, tools for verifying the actual protocol code will help
  – Csur (VMCAI’05), fs2pv (CSF’06), F7 (CSF’08), Aspier (CSF’09), etc etc
• Currently, fs2pv most developed, but hitting a wall
  – Translates libraries and protocol code from F#/OCaml to ProVerif
  – ProVerif does whole-program analysis of code versus symbolic attacker
  – Long, unpredictable run times on Cardspace (ASIACCS’08), TLS (CCS’08)
• Instead, we’re developing a compositional analysis for the fs2pv libraries and code, based on refinement types
Refined Types for Crypto APIs

val aes_encrypt: ( * AES CBC *)
  k:key →
  b:bytes{(SKey(k) ∧ CanSymEncrypt(k, b)) ∨ (Pub(k) ∧ Pub(b))} →
  e:bytes{IsEncryption(e, k, b)}

val aes_decrypt: ( * AES CBC *)
  k:key{SKey(k) ∨ Pub(k)} →
  e:bytes →
  b:bytes{∀p. IsEncryption(e, k, p) ⇒ b = p) ∧ (Pub(k) ⇒ Pub(b))} →

- APIs enriched with pre- and post-conditions in FOL
- Predicates declared by “equational” or “inductive” definitions
- Typechecker F7 relies on external SMT solver
Extended ML Interface, with *Refinement Type Annotations*

- `file.fs7`
- `file.fsi`
- `fsc`
- `Z3` (SMT Solver *Incomplete*)
- `F# Compiler`
F7 in Action
Performance on Larger Protocols

- F7’s compositional type-checking is scaling better than ProVerif’s whole-program analysis on these examples.
- Still, ProVerif can find attack traces; maybe ProVerif’s analysis can be modularized?

<table>
<thead>
<tr>
<th>Example</th>
<th>F# Program Modules</th>
<th>Lines of Code</th>
<th>F7 Typechecking Interface</th>
<th>Checking Time</th>
<th>FS2PV Queries</th>
<th>Verification Verifying Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptographic Patterns</td>
<td>1</td>
<td>158 lines</td>
<td>100 lines</td>
<td>17.1s</td>
<td>4</td>
<td>3.8s</td>
</tr>
<tr>
<td>Basic Protocol (Section 2)</td>
<td>1</td>
<td>76 lines</td>
<td>141 lines</td>
<td>8s</td>
<td>4</td>
<td>4.1s</td>
</tr>
<tr>
<td>Otway-Rees (Section 4.2)</td>
<td>1</td>
<td>265 lines</td>
<td>233 lines</td>
<td>1m.29.9s</td>
<td>10</td>
<td>8m 2.2s</td>
</tr>
<tr>
<td>Otway-Rees (No MACs)</td>
<td>1</td>
<td>265 lines</td>
<td>-</td>
<td>(Type Incorrect)</td>
<td>10</td>
<td>2m 19.2s</td>
</tr>
<tr>
<td>Secure Conversations (Section 4.3)</td>
<td>1</td>
<td>123 lines</td>
<td>111 lines</td>
<td>29.64s</td>
<td>-</td>
<td>(Not Verified)</td>
</tr>
<tr>
<td>Web Services Security Library</td>
<td>5</td>
<td>1702</td>
<td>475</td>
<td>48.81s</td>
<td>(Not Verified Separately)</td>
<td>20.2s</td>
</tr>
<tr>
<td>X.509-based Client Auth (Section 5.1)</td>
<td>+ 1</td>
<td>+ 88 lines</td>
<td>+ 22 lines</td>
<td>+ 10.8s</td>
<td>15</td>
<td>44m</td>
</tr>
<tr>
<td>Password-X.509 Mutual Auth (Section 5.2)</td>
<td>+ 1</td>
<td>+ 129 lines</td>
<td>+ 44 lines</td>
<td>+ 12s</td>
<td>18</td>
<td>51m</td>
</tr>
<tr>
<td>X.509-based Mutual Auth</td>
<td>+ 1</td>
<td>+ 111 lines</td>
<td>+ 53 lines</td>
<td>+ 10.9s</td>
<td>6</td>
<td>66m 21s</td>
</tr>
<tr>
<td>Windows Cardspace (Section 5.3)</td>
<td>1</td>
<td>1429 lines</td>
<td>309 lines</td>
<td>6m3s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Verification Times and Comparison with ProVerif
Three Observations

• We need some way to justify our assumptions
• ATP is mono-tactical ITP
  – Input via obscure parameters, patterns, repetition
  – Output partly via timing channel
• A lesson learnt from crypto formalisms is that it’s better to start from code and extract logical model, than the converse
  – Think of the C++ “don’t pay if you don’t use” principle
  – F# is in-the-box with Visual Studio 2010 – what will happen?
  – But with some exceptions, this is reverse of tooling I’ve seen for ITP
MINIM – REFINEMENT TYPES FOR A DATABASE QUERY LANGUAGE
Semantic Subtyping with an SMT Solver

• Since summer 2008, we’ve been collaborating on the design and implementation of typing for a new database language, M

• M is a data-oriented first-order functional language, combining refinement types \((T \ where \ e)\) and typecase \((e \ in \ T)\)
  – A novel combination, useful eg for database integrity constraints

• Our research contributions include:
  – Semantics for M in first-order logic: expressions are terms; types are predicates; (semantic) subtyping is valid implication
  – MSRC Minim checker relies on SMT solver (Z3) to decide subtyping

• Semantic subtyping adds value in key Oslo scenarios (eg DSLs)
  – So, engaging to enhance Oslo codebase with Minim algorithms
  – And, building reference implementation for post-PDC version of M
Accessing Tagged Unions

U is the type of tagged data, where the tag determines the type of the data

A notorious problem is forgetting to check the tag, but Minim catches this

To type-check the else-branch y.data, we know !(y.tag), and must show the type of y, which is (U where value==y), is a **subtype** of the record type {data: Text;}

We check subtyping via a semantics of types in logic, and ask Z3 the following: “if !(y.tag) and y satisfies (U where value==y), does y satisfy {data: Text;}”
The standard M typechecker relies on standard **structural subtyping**; Structural rules do not work well for the rich type system of M and fail to catch even simple errors like this one, caught by Minim’s **semantic subtyping**.

Semantic subtyping effectively checks code manipulating the syntax trees of Domain Specific Language, an important application area for M.
Three Challenges

In the context of Fancy Type Systems, three reasons to use ITP:
1. To Mechanize the Metatheory for the Masses (the POPLmark Challenge)
2. To check that FOL theories used in refinement formulas are sound
3. To help out the ATP during type-checking

• **Challenge 2:** Steal UI ideas from modern programming and testing environments (as if proofs were programs!)
  – Hover, Pause, F5

• **Challenge 3:** Conversely, can typecheckers steal ideas from ITP to “make the common case easy, and the rare case possible”
  – Annotate code with tactics to help typechecker (cf Why/Caduceus and HOL-Boogie)
  – Least common denominator tactic language? How about an ITP Systems Comp?
(DEFPRED (Man x))
(DEFPRED (Mortal x))

(BG_PUSH (Man Socrates)) ; add to background theory

(Man Socrates) ; purple formulas proved by Z3
(Mortal Socrates) ; red formulas not proved

(BG_PUSH (FORALL (x) (IMPLIES (Man x) (Mortal x)))))

(Man Socrates)
(Mortal Socrates)

; squiggles updated behind scenes by running Z3
Proof by Testing

```csharp
[TestMethod]
public void Test1() { ValidExpected("(EQ 0 0)"); }

[TestMethod]
public void Test2() { ValidExpected("(NOT (AND (In_Integer v) (In_Logical v)))"); }

[TestMethod]
public void Test3() { ValidExpected("(EXISTS (x) (EQ x 0))"); }
```
Resources

- Umbrella project, Cryptographic Verification Kit
  http://research.microsoft.com/cvk

- F7: refinement types for F#
  http://research.microsoft.com/F7

- Lectures on Principles and Applications of Refinement Types

- Microsoft “Oslo” Developer Center
  http://msdn.microsoft.com/oslo

- Z3: an efficient SMT solver