Verification of Functional Program Components

Zoltán Horváth  Tamás Kozsik  Máté Tejfel

{hz,kto,matej}@inf.elte.hu
http://people.inf.elte.hu/{hz,kto,matej}/
Dept. of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary

NJSZT Szoftvertechnológiai Fórum, 7th February, 2007

---

1 Supported by ELTE IKKK (GVOP-3.2.2-2004-07-0005/3.0) and Stiftung Aktion Österreich–Ungarn (66őu2).
Outline

1. Introduction and motivation
2. Foundations
3. Temporal properties of functional programs
   - Object abstraction
   - Subtype marks expressing type invariants
4. CPPCC: Correctness of mobile components
Why functional programming?

- Clear program text – close to mathematical specification
- No assignments
- No side effects
- Relatively easy to prove correctness
- Ideal for trusted code
Motivation for using formal methods

- Sound concepts needed for distributed and parallel programs
- Verification of safety critical applications
- Safe usage of software components
- **Our focus**: machine verifiable mobile code
Need for trusted mobile code

Our programs often use code (applets, plug-ins etc.) written by somebody else.

- **Dangers:**
  - Viruses, attacks
  - Security holes in operating systems
  - Programming failures in safety critical software (embedded systems, control software of medical instruments)
  - Incomplete specifications, side effects

- **We need components with proven properties**
  - Resource consumption
  - Security
  - **Functionality**
The Certified Proved-Property-Carrying Code architecture (CPPCC)

Safe mobile code exchange with minimal run-time overhead.

Three main parties involved in the scenario:

1. Producer of the mobile code: adds proofs of properties
2. Receiver: executes code only after safety checks which ensure that the code satisfies the requirements specified in the receiver’s code
3. Certifying authority: reduces the work-load of the receiver, performs verification static-time
Overview of CPPCC

**code producer**
- source code properties proof

**certifier**
- source code properties proof
- verification compilation certification

**code repository**
- target code properties stamp

**code receiver**
- target code properties stamp
- ..
- ..
- ..
Our results in the FunVer project

- Extending Sparkle (the dedicated theorem prover for Clean) with support for temporal properties
- Expressing and proving temporal properties of a set of processes written in Clean
- Extending Clean dynamics with proven properties (CPPCC prototype)
- D-Clean (Distributed Clean)
Using the results

- Potential for FP in software industry
  - Embedded systems (Hume)
  - Telecommunication (Erlang)
  - FP components integrated into complex systems

- Moving results to mainstream languages / methodologies
  - C++, Java, B-method
Concepts

- Temporal properties about the states of distributed programs, for example: (subtype) invariants
- Formal proofs, machine verifiable by theorem provers
- Mobile components
  - Mobile expressions (functional code), in the FP language
  - Clean + dynamics (Mobile Haskell, JoCaml, etc.)
  - Java Virtual Machine code
- Property/proof carrying code architecture, type and semantical checks
A formal model of programming is required
The properties of the model impose constraints
  What applications can be developed
  What is possible to prove
  Our model: interleaving, branching-time temporal logic
Properties of the formal model

Specification of *problems* and developing the *solutions* of problems in case of *parallel and distributed systems*.

- An extension of a relational model of non-deterministic sequential programs
- Provide tools for stepwise refinement of problems in a FP approach
- Use the concept of iterative abstract program of UNITY
- The concept of solution is based on the comparison of the problem as a relation and the (static) behaviour relation of the program
UNITY-like temporal logic

- Convenient operators
  - Safety (invariant, unless)
  - Progress (ensures, leads-to)
  - Initial and final states (init, fixed points)

- Support for component-oriented approach
  (Composing specifications and programs)

- Example: resource scheduling
Dining philosophers

:: Philo = Thinking \mid Hungry \mid Eating

For all $i$ and $j$,

$$\neg (\text{neighbours}(i, j) \land \text{philo}_i = \text{Eating} \land \text{philo}_j = \text{Eating}) \in \text{inv}$$

philo$_i$ = Thinking unless philo$_i$ = Hungry

philo$_i$ = Eating ensures philo$_i$ = Thinking
Composing specifications and programs

- Certain properties of a system can be computed from properties of its components
- If a statement is invariant in all components, then it is invariant in the whole application
- Ability to reason about a system
  - even if certain components are not known
  - only their properties are known
- Components received as mobile code
A concept of state in pure functional languages

- No destructive assignments, variables are constants
- Advantage: referential transparency, equational reasoning, the occurrences of the same expression have the same value
- I/O: single reference to environment, referential transparency cannot be violated, environment represented as series of pure values
- State: abstract objects corresponding to series of values
To prove an invariant:

- one needs to check the initial value of objects and calculate the weakest precondition for all atomic actions;
- for all atomic actions we should calculate the substitution of the invariant using the state-transition function of the action;
- we should prove that all these $wp$-s hold, if the invariant holds (the truth of the invariant is reserved by each action).

An unless property can be proved in a similar way, using weakest precondition calculation (rewriting).

A property “$P$ unless($S$) $Q$” holds if for all $t$ atomic steps of $S$: $P \land Q \implies wp(t, P \lor Q)$
Proving properties of communicating programs

- Example: dining philosophers
  - one server process (resource scheduler)
  - several clients (resource consumers)
- State transition: a `next_event` function (state transitions are controlled by the server, a monitor-like solution)
- From the point of view of verification we simulate the program with a `process_events` function.
State space

:: Philo = Thinking | Hungry | Eating

- Local state of a client: a value of type Philo
- Local state of the server: a list of Philos,
- State transition: if a philosopher changes its local state, the server calculates the new local state values with the next_event function
next_event :: [Philo] Int -> ([Int],[Philo])

Arguments:
- the local state of the server
- the id of the client that changes its state

The result:
- the ids of the clients that can start eating
- the new local state of the server
The \texttt{process\_events} function

Recursively calls the \texttt{next\_event} function

\begin{verbatim}
process\_events:: [Philo] [Int] -> [Philo]
process\_events philos [] = philos

process\_events philos [id]
    | (id < 0) || (id >= length philos) = philos
    = snd (next\_event philos id)

process\_events philos [id : ids]
    # philos = process\_events philos [id]
    = process\_events philos ids
\end{verbatim}
Object abstraction

- We can consider the values of the different philos variables as different states of the same abstract object (global state).

- For this abstract object we can formalize and prove temporal properties.

- Example property: a safety property (unless) in the process_events function: if a client is hungry and its right neighbour is eating, then these two philos do not change state unless the neighbour starts thinking.
eval philos \rightarrow eval ids \rightarrow \\
(i \geq 0) \rightarrow (i < \text{length philos}) \rightarrow \\
[ \\
(\text{philos}!!i == \text{Hungry}) \land \\
(\text{philos}!!(\text{rightneighbour philos i}) == \text{Eating}) \\
\text{UNLESS}(\text{process_events philos ids}) \\
(\text{philos}!!(\text{rightneighbour philos i}) == \text{Thinking}) \\
]
Zoltán Horváth, Tamás Kozsik, Máté Tejfel

Verification of Functional Program Components
Type system with subtype marks
Formal reasoning about properties

Combining lightweight and heavyweight tools
- Lightweight: type system
- Heavyweight: proof system

Programming language (SENYV)
- Type system supporting subtype marks
- Proof system adapted to subtype marks
Subtype marks

- Annotations attached to types
- Denote type invariants
- E.g. let S denote “sorted”
- Expressing pre- and postconditions etc.

\[
\text{Insert :: Int} \rightarrow \text{List}\{S\} \rightarrow \text{List}\{S!\}
\]
\[
\text{Insert } e \text{ Nil} = \text{Cons } e \text{ Nil}
\]
\[
\text{Insert } e \text{ (Cons } x \text{ xs)} =
\]
\[
\quad \text{if } (e \leq x)
\]
\[
\quad \quad \text{(Cons } e \text{ (Cons } x \text{ xs)})
\]
\[
\quad \quad \text{(Cons } x \text{ (Insert } e \text{ xs))}
\]
Semantics of subtype marks

- Typing rules for subtype mark propagation
  - used by the type system
  - very simple typing rules: easy to use for an average programmer

- Bool-functions – used by the proof system
Semantics of subtype marks (cont’d)

- Each subtype mark corresponds to a predicate
- Sparkle: Bool functions written in Clean

\[
S :: !\text{List} \rightarrow \text{Bool}
\]
\[
S \text{ Nil} = \text{True}
\]
\[
S \left( \text{Cons} \ x \ \text{Nil} \right) = \text{True}
\]
\[
S \left( \text{Cons} \ x \ \text{xs}::\left( \text{Cons} \ y \ \text{ys} \right) \right) =
\]
\[
(x \leq y) \ \&\& \ (S \ \text{xs})
\]

\[
S : \text{List} \rightarrow \mathbb{L}
\]
\[
S(\text{list}) = (S \ \text{list} = \text{True})
\]
Believe-me mark

\[
\begin{align*}
\text{Insert} & : \text{Int} \rightarrow \text{List}\{S\} \rightarrow \text{List}\{S!\} \\
\text{Insert } e \text{ Nil} & = \text{Cons } e \text{ Nil} \\
\text{Insert } e \ (\text{Cons } x \ xs) & = \\
& \quad \text{IfL } (\text{LessEq } e \ x) \\
& \quad \quad (\text{Cons } e \ (\text{Cons } x \ xs)) \\
& \quad (\text{Cons } x \ (\text{Insert } e \ xs)) \\
\text{Sort} & : \text{List} \rightarrow \text{List}\{S\} \\
\text{Sort } \text{Nil} & = \text{Nil} \\
\text{Sort } (\text{Cons } x \ xs) & = \text{Insert } x \ (\text{Sort } xs)
\end{align*}
\]
Partial correctness of \texttt{Insert}

\texttt{Insert} :: \texttt{Int} -> \texttt{List}\{\texttt{S}\} -> \texttt{List}\{\texttt{S!}\}

\[\forall e :: \texttt{Int}. \forall xs :: \texttt{List}. \]

\[ (xs = \bot \lor S(xs)) \rightarrow (\texttt{Insert} e \hspace{1pt} xs = \bot \lor S(\texttt{Insert} e \hspace{1pt} xs)) \]

\[[e::\texttt{Int}][xs::\texttt{List}]\]

\[ (xs = _|_ \lor S \hspace{1pt} xs) \rightarrow (\texttt{Insert} e \hspace{1pt} xs = _|_ \lor S (\texttt{Insert} e \hspace{1pt} xs)) \]
Current work

- Subtype marks in C++ STL
- Implement subtype marks with C++ TMP
Correctness of mobile components

- Dynamically download, link and execute code
- Ensure the correctness of mobile code
- Formal reasoning is preferred
- Minimal client-side / run-time overhead
Requirements on mobile code

- It does not use too much resources
- It does not read or modify data unauthorised
- It implements the desired functionality
Solutions

- Full dynamic-time code verification just before the application of the code (static, structural and type correctness verification: well-formedness, data-flow analysis for illegal memory access, type of instruction arguments etc.)
- Trusting in the code producer unconditionally (with using a certificate mechanism, to check identity)
- Trusting in code integrity and performing run-time pattern-match for types (Clean dynamic)
The Certified Proved-Property-Carrying Code architecture (CPPCC)

Safe mobile code exchange with minimal run-time overhead.

Three main parties involved in the scenario:

1. Producer of the mobile code: adds proofs of properties
2. Receiver: executes code only after safety checks which ensure that the code satisfies the requirements specified in the receiver’s code
3. Certifying authority: reduces the work-load of the receiver, performs verification static-time
CPPCC overview

**code producer**
- source code properties proof
- target code properties stamp

**certifier**
- source code properties proof
- verification compilation certification
- target code properties stamp

**code repository**
- target code properties stamp
- code receiver
- target code properties stamp
Example

- Receiver: an application using resources
- Mobile code: resource scheduler (dining philosophers)
Transmission of verified mobile code

Producer of mobile code

Library of certified code

Prover

Network

Source code
Code
Type code
Properties
Proof

Certifier
Checker

Code
Type code
Properties
Certificate

Authorizer
Receiver of mobile code

Verification of Functional Program Components

Zoltán Horváth, Tamás Kozsik, Máté Tejfel
Producing verified mobile code

Operational definition  Type definitions  Properties

Compiler  Proof system

Type code  Abstract machine code  Source code  Properties (in a coded form)  Proofs
Certification of verified mobile code

<table>
<thead>
<tr>
<th>Type code</th>
<th>Abstract machine code</th>
<th>Source code</th>
<th>Properties (in a coded form)</th>
<th>Proofs</th>
</tr>
</thead>
</table>

Compilation checker  Correctness checker  Certifier

<table>
<thead>
<tr>
<th>Type code</th>
<th>Abstract machine code</th>
<th>Properties (in a coded form)</th>
<th>Certificate</th>
</tr>
</thead>
</table>
Executing the verified mobile code

Type code | Abstract machine code | Certificate | Properties (in a coded form)

Receiving program
Requirements
Type patterns

Authentication
Dynamic linker
Authorization
Type unifier
Application

Verification of Functional Program Components
CPPCC: B-method and Java bytecode

- **Projekt Packer**: Click’n’Prove logger, B4free, Click’n’Prove
  - B specification
  - B implementation proof
- **Certifier Server**: Write’n’Prove, B4free, jBTools, JARSigner
  - B specification
  - B implementation proof
- **Repository Server**: browser, web server
  - Java byte code
  - B specification
  - X.509 cert
- **CPPCC-API**: class loader
  - Java byte code
  - B specification
  - X.509 cert
Summary

- We have extended an existing proof tool for Clean with support for temporal properties and designed the proof tactics necessary to manipulate them.
- Subtype marks provide a way to annotate types with invariants, and establish a co-operation between a type checker and a proof system.
- Certified Proved Property Carrying Code framework: efficient verification of the correctness of mobile components.
Related projects

- Expressing and proving temporal properties of Clean programs
- Annotations for expressing subtype invariants
- Design of Distributed Clean
- Safe transformations: refactoring (Clean, Erlang)
- Safe destructive update of data structures