Метод применения теории типов Мартина-Лёфа для верификации программных систем

Кирилл Таран

545 группа

26 мая 2014 г.

науч. рук.: Д.Ю. Булычев (кафедра с.п.)

рецензент: К. Соломатов (JetBrains)

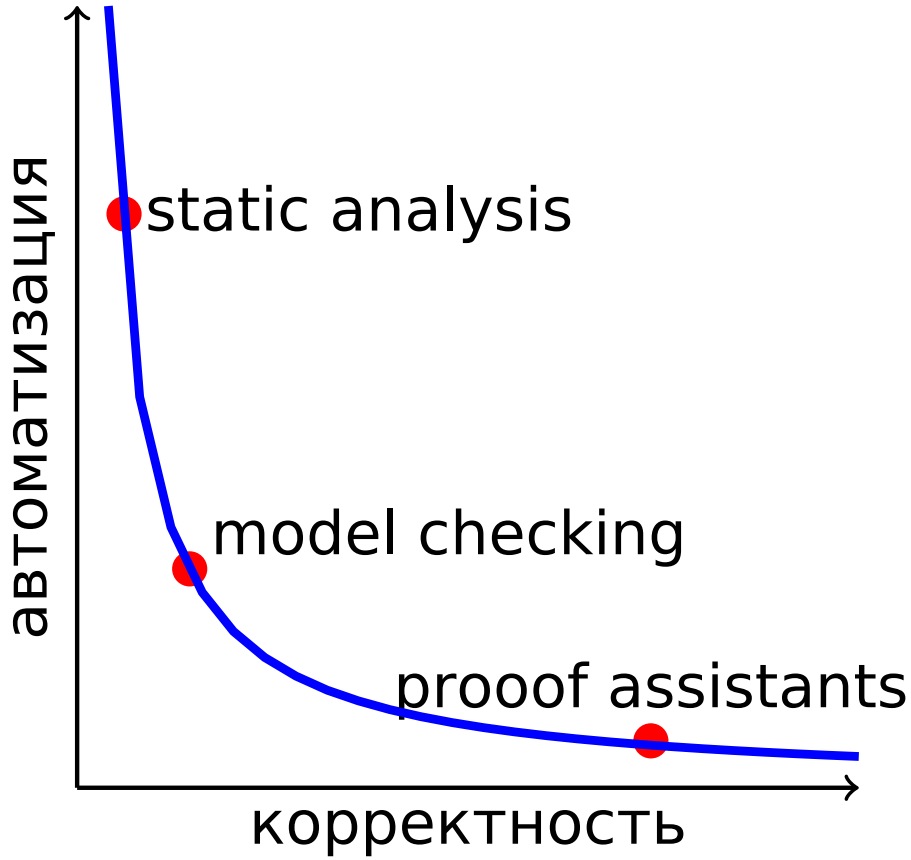
Методы верификации

Простые методы:

- тесты
- сборка мусора
- статическая типизация

Продвинутые методы:

- статический анализ
- model checking
- инструменты для построения доказательств



Теория типов Мартина-Лёфа

Включает языки спецификации
как программ, так и утверждений о них:

- ⋆ функциональный язык программирования
- ⋆ система типов — кодирование любых утверждений о программах

- \+ строгая математическая теория
- \+ имеет инструментальную поддержку — инструменты для построения доказательств (Coq, Agda)

- \+ можно формализовать любое понятие
- − слабая автоматизация — построение доказательств похоже на программирование

Постановка задачи

Цель:

- Создать подход на основе ТТМЛ для разработки программных систем с формально доказанными свойствами.

Выделены следующие задачи:

1. Предоставить сценарий использования инструментов, основанных на ТТМЛ.
2. Разработать средство генерации кода на промышленном языке.
   - ⋆ т.к. код ТТМЛ неэффективен
   - ⋆ т.к. программы на ТТМЛ трудно интегрировать с другими
3. Выполнить апробацию подхода.

1. Сценарий использования ТТМЛ

Синтез корректной системы вместо *анализа* уже реализованной системы:

- сразу строится *абстрактная модель* системы;
- нельзя изменить реализацию без изменения модели;

\+ не надо восстанавливать абстрактную модель по системе;

\− нужно строить систему по абстрактной модели.

Предлагаемый сценарий:

1. Описание абстрактной модели ядра системы на ТТМЛ.
2. Доказательство необходимых свойств.
3. Генерация промышленного кода ядра системы.
4. Интеграция ядра системы с менее критичной частью, реализованной на обычном языке.

2. Генерация кода

2.1. Обзор

В качестве целевого языка выбрана Java.

Желательна *source-to-source* трансляция, а не просто JVM:

- jvm-байткод получаем автоматически;
- существуют фреймворки, которые требуют Java, например GWT транслирует Java в Javascript.

Существует стандартный плагин Coq $\rightarrow$ OCaml/Haskell, тогда достаточно транслятора OCaml/Haskell $\rightarrow$ Java.

Подобных удовлетворительных трансляторов нет:

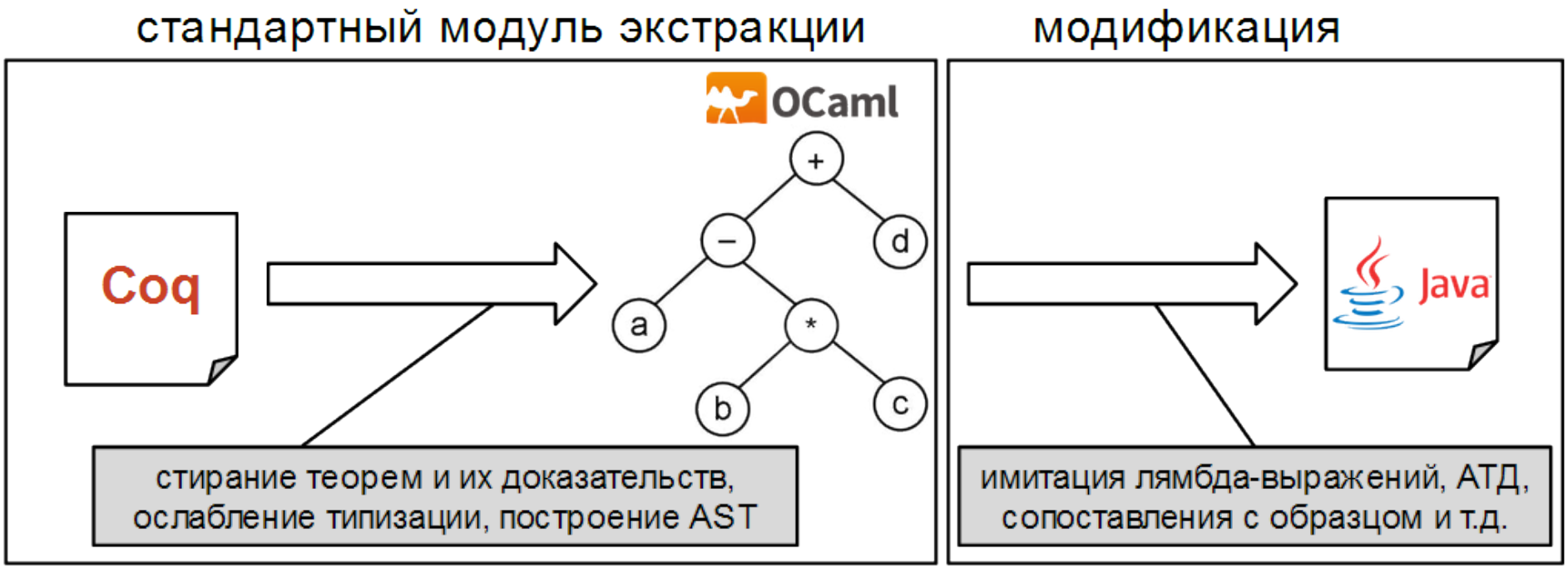
- “OCaml-Java” — jvm-байткод, не поддерживается;
- старые версии “UHC” — jvm-байткод, не поддерживается.

2. Генерация кода

2.2. Схема транслятора

Трансляция кода на “функциональном” языке программирования в “императивный”:

- трансляция алгебраических типов данных
- трансляция функций как значений первого порядка



2. Генерация кода

2.3. Пример генерации

Coq

```
Definition cnot (control : bool) :=
  if control then negb
             else id.
```

Java

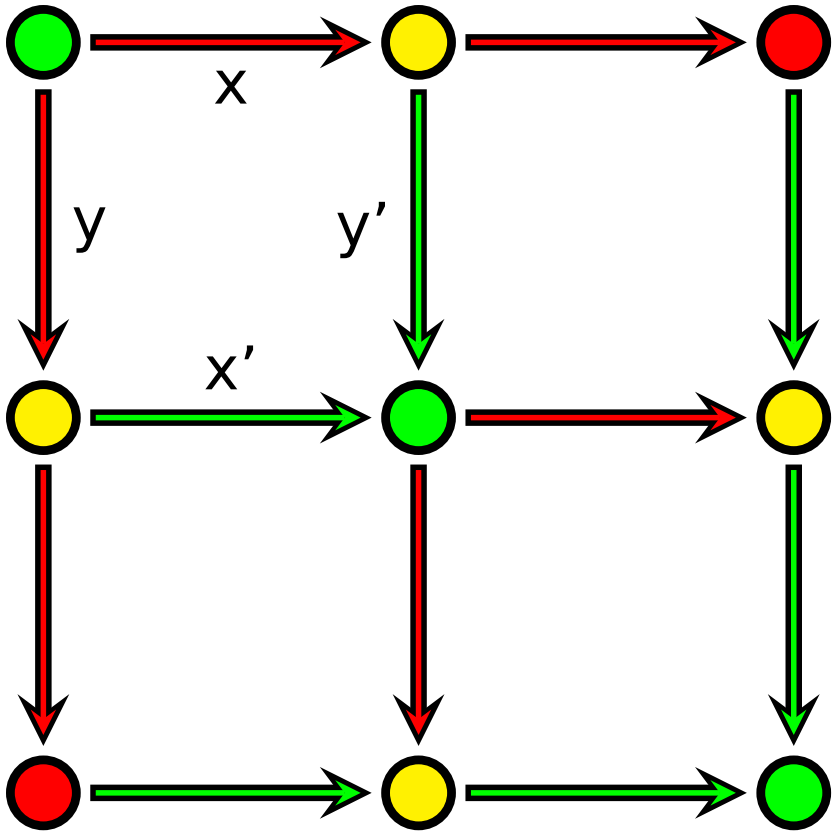
```
public static class cnot {
  public static Function<bool,bool> apply(final bool arg1) {
    Function<bool,bool> var0 = null;
    switch (((bool)arg1).tag) {
      case true: {
        final bool.true var1 = (bool.true)arg1;
        var0 = negb.apply();
        break;
      }
      case false: {
        final bool.false var3 = (bool.false)arg1;
        var0 = id.apply();
        break;
      }
    }
    return (Function<bool,bool>)var0;
  }
}
```

3. Апробация подхода

3.1. Алгоритм операционных преобразований

Алгоритм “оптимистичной” синхронизации:

- изменения применяются сразу же
- все изменения модели представляются операциями
- синхронизация — преобразование чужих изменений через свои



$$y' \circ x = x' \circ y$$

\+ не требует остановки для синхронизации

\+ локальные операции применяются мгновенно

− система становится консистентной только через какое-то время

3. Апробация подхода

3.2. Упрощённая модель сети

Coq

```
Inductive Message : Type :=
  | Do  : list cmd -> Message
  | Ack : Message.

Record Node := {
  data    : model;
  history : list cmd;
  input   : {
    size  : nat &
    queue Message size
  }
}.

Record Network := {
  first  : Node;
  second : Node
}.

Separate Extraction
  Network Node Message.
```

Java

```
public static abstract class Message {
    public Tag tag;
    public static enum Tag { Do, Ack }
    public static final class Do extends Message {
        public final list<cmd> field0;
        public Do(list<cmd> arg0) {
            field0 = arg0;
            tag = Tag.Do;
        }
    }
    public static final class Ack extends Message {
        public Ack() { tag = Tag.Ack; }
    }
}
```

3. Апробация подхода

3.3. Пример доказанного свойства

утверждение

рано или поздно сеть придёт в консистентное состояние

скрипт-доказательство

метапрограмма, работающая с контекстом: гипотезами и леммами

Coq

```
Definition stabilization :=
  transitive receive_rel.

Definition life :=
  transitive transition.

Theorem stability : forall net, exists net',
  stabilization net net' /\ consistent net'.
Proof.
  intros net N. inversion_clear N as [d L].
  remember (stabilization_ready net) as SR.
  inversion SR; inversion_clear H as [S R].
  remember
    (apptransitive L (stabilization_life S))
    as L'.
  assert (N : natural x) by (
    unfold natural; exact (ex_intro _ d L')).
  exact (ex_intro _ x (conj S
    (consistency x (conj N R)))).
Qed.
```

Результаты

1. Предоставлен сценарий использования инструментов на основе ТТМЛ.

2. Разработано средство генерации кода на промышленном языке программирования:
   - ⋆ стандартный модуль экстракции Coq модифицирован для генерации Java-кода по спецификации системы.

3. Выполнена апробация подхода на алгоритме операционных преобразований:
   - ⋆ формализована модель алгоритма;
   - ⋆ доказаны ключевые свойства:
     - ▶ равенство данных после работы алгоритма;
     - ▶ завершаемость алгоритма.