Introducción a la verificación formal

David Charte

15 de enero de 2018

Servidores seguros — Universidad de Granada

Introducción

Motivación

- Requisitos de seguridad
- Requisitos de temporalidad
- Bugs!

Índice

Proposiciones como tipos

Lógica intuicionista

Cada afirmación se debe probar de forma **constructiva**.

No asume **tercio excluso** ($p \vee \neg p$) ni eliminación de la doble negación $\Rightarrow$ No se puede demostrar por reducción al absurdo.

Utilidad: las definiciones partiendo de proposiciones bajo esta lógica son **computables**. No se puede definir

```
int n = if conjetura 1 else 0
```

si no podemos probar la conjetura o su negación.

Lógica de orden superior

Lógica de **primer orden**: cuantificar elementos

“existe x tal que x es mamífero y x tiene pico”

$$\exists \boldsymbol{x}(x \in M \land x \in P)$$

Lógica de **segundo orden**: cuantificar relaciones entre elementos

“existe una clase O de animales tal que para cada animal x, si x es mamífero y tiene pico entonces es de clase O”

$$\exists \boldsymbol{O} \forall x(x \in M \land x \in P \to x \in O)$$

Lógica de **orden superior** $= \bigcup_n$ lógica de n orden.

Utilidad: mayor expresividad.

Teoría de tipos

Estudia los sistemas formales que utilizan tipos para restringir las operaciones que se pueden aplicar a cada término.

Tipos de función: La función que lleva tipo `a` a tipo `b` es de tipo `(a -> b)`.

Hay una teoría de tipos intuicionista (Martin-Löf).

Tipos dependientes

Un tipo dependiente es un tipo cuya definición depende de un valor.

Ejemplo: “plantilla de tipos” `Matrix(m,n)`. Podemos definir la multiplicación de matrices

```
mult :: (Matrix(k,m), Matrix(m,n)) ->
                Matrix(k,n)
```

o la función que devuelve un vector de *n* elementos:

```
repNull :: n -> Vector(n)
```

¡El tipo de retorno depende del valor del parámetro!

Sistemas de cálculo

Son equivalentes (resuelven todo problema **efectivamente calculable**):

- Funciones recursivas generales (Gödel)
- Cálculo lambda (Church)
- Máquinas universales (Turing)

Es más sencillo razonar sobre estos sistemas que sobre un lenguaje de programación.

Cálculo lambda simplemente tipado: sólo incluye funciones y "tipos básicos", pero no es Turing-completo.

Correspondencia de Curry-Howard

Cada tipo se relaciona con una proposición (Curry, 1934).

Isomorfismo de Curry-Howard

Una proposición verdadera se identifica con un tipo del cual existe al menos un objeto, y una proposición falsa corresponde a un tipo para el cual es imposible construir un objeto.

Si encontramos un objeto del tipo que buscamos, entonces tenemos una demostración para nuestra proposición!

Correspondencia de Curry-Howard

Pareja / conjunción

Sólo hay elementos de tipo `(a,b)` si los hay de `a` y de `b`. De igual forma, $a \wedge b$ sólo se demuestra con una demostración para a y otra para b.

Correspondencia de Curry-Howard

Pareja / conjunción

Sólo hay elementos de tipo `(a,b)` si los hay de `a` y de `b`. De igual forma, $a \wedge b$ sólo se demuestra con una demostración para a y otra para b.

Alternativa / disyunción

Tipo `(a|b)` describe elementos que son de `a` o de `b`, se corresponde con $a \vee b$.

Correspondencia de Curry-Howard

Pareja / conjunción

Sólo hay elementos de tipo `(a,b)` si los hay de `a` y de `b`. De igual forma, $a \wedge b$ sólo se demuestra con una demostración para a y otra para b.

Alternativa / disyunción

Tipo `(a|b)` describe elementos que son de `a` o de `b`, se corresponde con $a \vee b$.

Función / implicación

`a -> b` indica entrada de un elemento de tipo `a` y salida de tipo `b`. Vistos a y b como proposiciones, existe tal función si y solo si se cumple $a \to b$.

Correspondencia de Curry-Howard

Ejemplos:

- Tautología: $a \to a$ es cierto así que siempre existe una función de tipo `a -> a` (la identidad).
- Proyección: de una pareja podemos sacar el primer elemento (función tipo `(a,b) -> a`). Entonces, $a \wedge b \to a$ es verdadera.

Verificación formal

Verificación de modelos

Exploración **exhaustiva** de todos los estados y transiciones del modelo matemático asociado al sistema.

Aplicable principalmente a sistemas con un **número finito de estados**, (también a algunos infinitos que se puedan representar finitamente). No adaptable a grandes sistemas.

Las propiedades verificadas mediante esta técnica suelen venir descritas en una **lógica temporal**: “cada vez que ocurre x el sistema responde y”.

Verificación deductiva

Construir una **especificación formal** del comportamiento de un sistema. Realizar demostraciones para deducir que la implementación la cumple.

Generalmente las demostraciones se resuelven con

- **asistentes de demostración**,
- solucionadores de teorías de satisfacibilidad módulo,
- demostradores automáticos.

Asistentes de demostración

Programa que ayuda al desarrollo de demostraciones formales mediante **colaboración entre el humano y la máquina**.

Cada asistente incluye una teoría de tipos y generalmente utiliza **tácticas** para encontrar un elemento del tipo buscado (Curry-Howard: equivale a una demostración).

El humano escribe una guía para la demostración, completada y verificada por la máquina.

Asistentes de demostración

- **Agda** (Haskell): lógica de orden superior, tipos dependientes
- **Coq** (OCaml): lógica de orden superior, tipos dependientes
- **Isabelle** (ML): lógica de orden superior, tipos simples
- F*, HOL, PVS...

Ejemplo: Coq

```
(** Recursive polymorphic definition of trees *)
Inductive tree (X:Type) : Type :=
  | nilt : tree X
| node : X -> tree X -> tree X -> tree X.

(** Auxiliary lemmas about refl *)
Lemma refl_involutive : forall {X:Type} (t:tree X),
  refl (refl t) = t.
Proof.
  intros X t.
  induction t as [|a izq Hiz der Hdr].
    reflexivity.
    simpl. rewrite -> Hiz. rewrite -> Hdr. reflexivity.
Qed.
```

Fuente: `https://github.com/M42/recorridosArboles`

Aplicaciones

Aplicaciones en sistemas

Aplicable a kernels sencillos: derivados del **microkernel L4**.

Primeros intentos (1979): UCLA Secure Unix, Provably Secure Operating System. La verificación hacía el sistema **un orden de magnitud** más lento.

Actuales: PikeOS (real-time), VFiasco, seL4.

Secure Embedded L4 (seL4)

Primera demostración de corrección (200000 líneas para 7500 de código C) en 2009, sobre **Isabelle/HOL**.

Libre bajo GPLv2 en 2014.

Capas:

1. especificación abstracta (Isabelle)
2. especificación ejecutable (Haskell)
3. implementación (C) con prueba

La línea de código de seL4 es **25x más barata** que la del promedio de programas que verifican *Common Criteria*.

Aplicaciones en software

Compiladores certificados: CompCert C (Coq)

`https://github.com/coq/coq/wiki/ListofCoqPLProjects`

Criptografía verificada: HACL* (F*, parte de Project Everest)

`https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/`

Aplicaciones en hardware

Síntesis de hardware: Fe-Si HDL (Coq)

`https://link.springer.com/chapter/10.1007/978-3-642-39799-8_14`

Propiedad de vivacidad en procesadores: Bus *Runway* de HP (Isabelle) `https://rd.springer.com/chapter/10.1007/BFb0028385`

Procesador DLX verificado: Verified Architecture Microprocessor (PVS)

`https://link.springer.com/article/10.1007/s10009-006-0204-6`

Conclusiones

Conclusiones

La verificación formal presenta:

- Fuerte base teórica
- Coste computacional alto
- Mayor seguridad a menor coste económico
- Enfoque más frecuente: deductiva con asistentes de demostración

¡Gracias! ¿Preguntas?