Formalización de un modelo del algebra de Kleene concurrente en Isabelle

Alvaro Cuestas

Marzo del 2024

¿Por qué hacer demostraciones con un asistente es diferente?

- Diálogo entre personas y asistentes
- Verificación automática y confiable ('esto sí es una prueba')
- Gaps, Goals, etc ('hoja de ruta')
- En algunos casos, automatización ('no comparable')
- Modularización, reusabilidad, mantenibilidad ('consideraciones estilo software')

¿Por qué hacer demostraciones con un asistente es diferente?

McCarthy 1961 (McCarthy sí la veía)

Checking mathematical proofs is potentially one of the most interesting and useful applications of automatic computers.

Computers can check not only the proofs of new mathematical theorems but also proofs that complex engineering systems and computer programs meet their specifications.

Proofs to be checked by computer may be briefer and easier to write than the informal proofs acceptable to mathematicians.

This is because the computer can be asked to do much more work to check each step than a human is willing to do, and this permits longer and fewer steps.

Relación con otros métodos formales

- Dificultad de la adquisición y la aplicación
- Disyuntiva: muchos programas chotos con *bugs* o pocos programas buenos sin *bugs*
- Peirce: Grado de automatización
- Peirce: El chequeo de tipos es el método formal de mayor aplicación y difusión
- 'Criterio del nicho': Sistemas críticos o sistemas muy usados (*kernels* de SO, compiladores)

Isabelle

- Mucha más automatización que otros asistentes
- Más parecido a la matematica informal / tradicional
  - Pocos / ningún *casteo*
  - Enunciados 'onda primer orden'
  - 'Tipos escondidos en la trastienda'
- AFP / Archivo de Pruebas Formales

Isabelle

- Lógica de alto orden clásica (no constructiva). Es un cálculo lambda con polimorfismo y tipos de datos algebraicos. Vendría a ser parecido a $F\omega$.
- *Kernel*: Hay un tipo de datos algebraico 'oculto' para los teoremas, y funciones que operan sobre ese tipo que no son accesibles para el usuario.
- 'Todo está parado sobre tortugas y las tortugas son *standard ML*'.
- Menos expresivo (eso es relevante para el grado de generalidad y reusabilidad de una demostración - probar un algoritmo *vs* dar una prueba parametrizada para una clase de algoritmos).

Algebras de Kleene

- Se usan para modelar programas.
- Su modelo canónico son los conjuntos de cadenas (que representan caminos de ejecución).
- Tiene operaciones parecidas a las operaciones sobre lenguajes regulares.
- Representa de forma homogénea a las propiedades y los programas (no como PDL)
- Un programa cumple ϕ si su ejecución termina en un estado en el que vale ϕ.

Algebras de Kleene

Son semianillos con propiedades adicionales, o sea, $(S, +, 0, \cdot, 1)$ tal que

- $+$ es conmutativo
- $\cdot$ distribuye sobre $+$ a ambos lados
- 0 es el neutro de la suma y el aniquilador del producto
- 1 es el neutro del producto

Álgebra de Kleene

Las propiedades adicionales que eso tiene que cumplir para ser un algebra de Kleene (o 'quantale') son:

- La suma es idempotente y se puede usar para definir un orden parcial sobre S: $a \leq b := a + b = b$
- Ese orden parcial tiene definido el supremo para cualquier subconjunto de S
- El producto distribuye sobre el supremo de cualquier subconjunto de S

Álebras de Kleene: Modelo canónico

- La suma es la unión de lenguajes
- El producto es la concatenación de lenguajes
- 0 es el conjunto vacío
- 1 es un lenguaje que tiene solamente la palabra vacía

Concretamente ¿Cómo representás un programa?

$$p\,;q := pq$$

$$\textbf{if } b \textbf{ then } p \textbf{ else } q := bp + \overline{b}q$$

$$\textbf{if } b \textbf{ then } p := bp + \overline{b}$$

$$\textbf{while } b \textbf{ do } p := (bp)^*\overline{b}$$

Algebras de Kleene: casi inservibles como modelo

- No modelan el contenido de la memoria ni ninguna información sobre variables o valores
- Si uno quiere verificar un sistema de software de verdad, usa un asistente con teoría de tipos! O una lógica del estilo de LTL
- No hacen ninguna relación entre conjuntos de trazas de ejecución y propiedades ('uno asume que dios sabe que propiedades valen para cada conjunto de ejecuciones')

Hoare et al. 2011 - Concurrent Kleene Algebra and its Foundations

Our model and its theorems may look elegant, but when applied to actual programs, they are far too weak to prove anything useful.

Álgebras de Kleene: muy útiles para otras cosas

- Teorema folklórico (todo programa es equivalente a un programa con un solo *while*)
- Algoritmos eficientes para problemas con grafos y caminos
- Optimizaciones de ciclos verificadas formalmente (en un compilador)
- ¡Verificación de (algunas) propiedades de un programa en tiempo casi lineal!
- *Program Analysis and Verification based on Kleene Algebra in Isabelle/HOL*
- *Deciding Kleene Algebras in Coq*

(Ahora sí) Algebras de Kleene Concurrentes

- Agregan una operación de composición concurrente a la composición secuencial
- Dependiendo del modelo, se pueden modelar o no operaciones no atómicas / con duración
- Como álgebra, son 'dos algebras de Kleene juntas con la misma suma'

Álgebras de Kleene Concurrentes

Un álgebra $(S, +, 1, *, ;, 1)$ tal que...

- $(S, +, 1, *, 1)$ es un algebra de Kleene / un quantale
- $(S, +, 1, ;, 1)$ también
- Vale que $(a * b); (c * d) \leq (b; c) * (a; d)$

A eso le dicen ley de intercambio.

Es una especie de propiedad distributiva, pero mas débil.

Mezcla de cadenas y lenguajes

- En el modelo canónico a la composicion concurrente la interpretamos con la mezcla de lenguajes
- Uno se imagina que los símbolos de una cadena son operaciones, y que está ejecutando de forma concurrente operaciones de dos cadenas/programas, en cualquier orden

La definición recursiva de la mezcla de cadenas es esta:

$$\alpha \diamond \epsilon = \alpha$$

$$\epsilon \diamond \beta = \beta$$

$$a\alpha \diamond b\beta = a(\alpha \diamond b\beta) \cup b(a\alpha \diamond \beta)$$

Y para mezclar lenguajes, mezcla sus palabras de a pares

Estructuras algebraicas en Isabelle: *Locales*

- Isabelle tiene un mecanismo muy útil para definir estructuras algebraicas, que se llama *locales*.
- Mecanismo de abstracción con herencia y parametrización
- Definís la *signature* del *locale* (sus argumentos y constantes distinguidas), las hipótesis, y sus propiedades.
- Despues probás que algo es una instancia del *locale* mostrando que cumple las hipótesis, y con eso obtenés 'gratis' todas las propiedades del *locale* para esa instancia particular.

Definir *locales*

```
locale <name> :
    <other_locale_1>  arg_1  arg_2 +
    <other_locale_2>  arg_2  arg_1
 for
    arg_1 :: type_1  (<notation>)  and
    arg_2 :: type_2  (<notation>)
  assumes
    <axiom_1>  :  <statement>
    <axiom_2>  :  <statement>
  begin
    lemma <prop_1>  :
      <statement>
      <proof>
    lemma <prop_2>  :
      <statement>
      <proof>
  end
```

Instanciar *locales*

```
interpretation <name> :
    <locale>   arg_1   arg_2   arg_3
  proof
    show <axiom_1> <proof>
    show <axiom_2> <proof>
    show <axiom_3> <proof>
```

Ejemplos

```
locale idempotent_semiring =
  semiring_1 one times plus zero for
    one :: "'a" and
    plus :: "'a ⇒ 'a ⇒ 'a" (infixl "⊕" 70) and
    times :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 80) and
    zero :: "'a" +
  assumes
    plus_is_idempotent [simp] :
      " a ⊕ a = a "
begin

lemma plus_is_associative [ac_simps] : "(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c) "
  by (simp add:Groups.ac_simps)

lemma plus_is_commutative [ac_simps] : "a ⊕ b = b ⊕ a "
  by (simp add:Groups.ac_simps)

end
```

Ejemplos

```
locale natural_order_semiring =
  idempotent_semiring one plus times zero for
    one :: "'a" and
    plus :: "'a ⇒ 'a ⇒ 'a" (infixl "⊕" 70) and
    times :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 80) and
    zero :: "'a" ("0") and
    leq :: "'a ⇒ 'a ⇒ bool" (infixl "≼" 60) +
  assumes
    induced_natural_order :
      " a ⊕ b = b ⟷ a ≼ b "
begin

  lemma leq_is_reflexive : " a ≼ a "
    by (metis induced_natural_order plus_is_idempotent)

  lemma leq_is_transitive : " a ≼ b ∧ b ≼ c ⟹ a ≼ c "
    proof -
      have " a ≼ b ∧ b ≼ c ⟹ a ⊕ b = b ∧ b ⊕ c = c "
        by (metis induced_natural_order)
      then have " a ≼ b ∧ b ≼ c ⟹ (a ⊕ b) ⊕ c = c "
        by (auto)
      then have " a ≼ b ∧ b ≼ c ⟹ a ⊕ (b ⊕ c) = c "
        by (metis plus_is_associative)
      then show " a ≼ b ∧ b ≼ c ⟹ a ≼ c "
        by (metis induced_natural_order)
    qed
```

Ejemplos

```
locale quantale =
  natural_order_semiring one plus times zero leq for
    one :: "'a" ("1") and
    plus :: "'a ⇒ 'a ⇒ 'a" (infixl "⊕" 70) and
    times :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 80) and
    zero :: "'a" ("0") and
    leq :: "'a ⇒ 'a ⇒ bool" (infixl "⊑" 40) and
    sup :: "'a set  ⇒ 'a" ("⨆") +
  assumes
    times_distributes_over_suprema :
      "  a ⊗ (⨆ A) = ⨆ ({ a ⊗ x | x. x : A }) " and
    is_complete_lattice :
      " ⋀ x A. (⋀ y . y : A ⟹ y ⊑ x) ⟹ (⨆ A) ⊑ x "

locale concurrent_kleene_algebra =
  sequential_quantale : quantale one plus seq zero leq sup +
  parallel_quantale : quantale one plus par zero leq sup
  for
    one :: "'a" ("1") and
    plus :: "'a ⇒ 'a ⇒ 'a" and
    seq :: "'a ⇒ 'a ⇒ 'a" (infixl ";" 70) and
    par :: "'a ⇒ 'a ⇒ 'a" (infixl "||" 60) and
    zero :: "'a" ("0") and
    leq :: "'a ⇒ 'a ⇒ bool" (infixl "⊑" 40) and
    sup :: "'a set ⇒ 'a" +
  assumes
    exchange_law :
      " (a || b) ; (c || d) ⊑ (b ; c) || (a ; d) "
begin

lemma one_is_seq_neuter_right [simp] :
  " x ; 1 = x "
```

Ejemplos

```
lemma par_is_commutative [simp] :
  " x || y = y || x "
proof -
  have
    " (x || y) ; (1 || 1) ⊑ (y ; 1) || (x ; 1) "
    by (metis exchange_law)
  then have
    simplify_ones :
    " (x || y ) ⊑ (y || x) "
    by (simp)
  then have " (y || x) ; (1 || 1) ⊑ (x ; 1) || (y ; 1) "
    using exchange_law by blast
  then have swapped_simplify_ones :
    " (y || x ) ⊑ (x || y) "
    by (simp)
  have  "(y || x ) ⊑ (x || y) ∧ (x || y ) ⊑ (y || x) "
    using swapped_simplify_ones simplify_ones by auto
  then show " (x || y) = (y || x) "
    using parallel_quantale.leq_is_antisymmetric by auto
qed
```

Ejemplos

```
interpretation union_concatenation_quantale :
  quantale "{ε}" "(∪)" "(;)" "{}" "(⊆)" "⋃"
proof
  show " ⋀ a A. a ; ⋃ A = ⋃ {a ; x |x. x ∈ A} "
    unfolding conc_def by auto
  show  " ⋀ x A. (⋀y. y ∈ A ⟹ y ⊆ x) ⟹ ⋃ A ⊆ x"
    by auto
qed

interpretation union_shuffle_quantale :
  quantale "{ε}" "(∪)" "(∥)" "{}" "(⊆)" "⋃"
proof
  show  " ⋀ α A. α ∥ ⋃ A =  ⋃ {α ∥ β | β. β ∈ A} "
    unfolding Shuffle_def by auto
  show  "  ⋀ x A. (⋀y. y ∈ A ⟹ y ⊆ x) ⟹ ⋃ A ⊆ x "
    unfolding Shuffle_def by auto
qed
```

Locales: Comentarios

- Documentación poco exhaustiva (pero existe *stackoverflow*)
- Nombrar las partes del *locale* sirve cuando cada parte hace mas de una cosa, o es argumento de varios *locales* en una misma definición
- Nombrar los argumentos de la firma hace más claro qué es cada cosa, y te ahorra andar inspeccionando varios archivos cuando la jerarquía de las defininciones está muy anidada

That's all folks