Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

PPA - Un asistente de demostración para lógica de primer orden con extracción de testigos usando la traducción de Friedman

Tesis de Licenciatura en Ciencias de la Computación

Manuel Panichelli

Director: Pablo Barenbaum

Buenos Aires, 2024

PPA - UN ASISTENTE DE DEMOSTRACIÓN PARA LÓGICA DE PRIMER ORDEN CON EXTRACCIÓN DE TESTIGOS USANDO LA TRADUCCIÓN DE FRIEDMAN

La princesa Leia, líder del movimiento rebelde que desea reinstaurar la República en la galaxia en los tiempos ominosos del Imperio, es capturada por las malévolas Fuerzas Imperiales, capitaneadas por el implacable Darth Vader. El intrépido Luke Skywalker, ayudado por Han Solo, capitán de la nave espacial "El Halcón Milenario", y los androides, R2D2 y C3PO, serán los encargados de luchar contra el enemigo y rescatar a la princesa para volver a instaurar la justicia en el seno de la Galaxia (aprox. 200 palabras).

Palabras claves: Guerra, Rebelión, Wookie, Jedi, Fuerza, Imperio (no menos de 5).

PPA - A PROOF-ASSISTANT FOR FIRST-ORDER LOGIC WITH WITNESS EXTRACTION USING FRIEDMAN'S TRANSLATION

In a galaxy far, far away, a psychopathic emperor and his most trusted servant – a former Jedi Knight known as Darth Vader – are ruling a universe with fear. They have built a horrifying weapon known as the Death Star, a giant battle station capable of annihilating a world in less than a second. When the Death Star's master plans are captured by the fledgling Rebel Alliance, Vader starts a pursuit of the ship carrying them. A young dissident Senator, Leia Organa, is aboard the ship & puts the plans into a maintenance robot named R2-D2. Although she is captured, the Death Star plans cannot be found, as R2 & his companion, a tall robot named C-3PO, have escaped to the desert world of Tatooine below. Through a series of mishaps, the robots end up in the hands of a farm boy named Luke Skywalker, who lives with his Uncle Owen & Aunt Beru. Owen & Beru are viciously murdered by the Empire's stormtroopers who are trying to recover the plans, and Luke & the robots meet with former Jedi Knight Obi-Wan Kenobi to try to return the plans to Leia Organa's home, Alderaan. After contracting a pilot named Han Solo & his Wookiee companion Chewbacca, they escape an Imperial blockade. But when they reach Alderaan's coordinates, they find it destroyed - by the Death Star. They soon find themselves caught in a tractor beam & pulled into the Death Star. Although they rescue Leia Organa from the Death Star after a series of narrow escapes, Kenobi becomes one with the Force after being killed by his former pupil - Darth Vader. They reach the Alliance's base on Yavin's fourth moon, but the Imperials are in hot pursuit with the Death Star, and plan to annihilate the Rebel base. The Rebels must quickly find a way to eliminate the Death Star before it destroys them as it did Alderaan (aprox. 200 palabras).

Keywords: War, Rebellion, Wookie, Jedi, The Force, Empire (no menos de 5).

AGRADECIMIENTOS

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce sapien ipsum, aliquet eget convallis at, adipiscing non odio. Donec porttitor tincidunt cursus. In tellus dui, varius sed scelerisque faucibus, sagittis non magna. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Mauris et luctus justo. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Mauris sit amet purus massa, sed sodales justo. Mauris id mi sed orci porttitor dictum. Donec vitae mi non leo consectetur tempus vel et sapien. Curabitur enim quam, sollicitudin id iaculis id, congue euismod diam. Sed in eros nec urna lacinia porttitor ut vitae nulla. Ut mattis, erat et laoreet feugiat, lacus urna hendrerit nisi, at tincidunt dui justo at felis. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Ut iaculis euismod magna et consequat. Mauris eu augue in ipsum elementum dictum. Sed accumsan, velit vel vehicula dignissim, nibh tellus consequat metus, vel fringilla neque dolor in dolor. Aliquam ac justo ut lectus iaculis pharetra vitae sed turpis. Aliquam pulvinar lorem vel ipsum auctor et hendrerit nisl molestie. Donec id felis nec ante placerat vehicula. Sed lacus risus, aliquet vel facilisis eu, placerat vitae augue.

Índice general

1. INTRODUCCIÓN

2. DEDUCCIÓN NATURAL

3. EL LENGUAJE PPA

4. EL CERTIFICADOR DE PPA

5. EXTRACCIÓN DE TESTIGOS DE EXISTENCIALES

6. LA HERRAMIENTA `ppa`

En el Capítulo 3 (El lenguaje PPA) vimos a PPA como un lenguaje, desde el punto de vista de un usuario. En Capítulo 4 (El certificador de PPA) abordamos el funcionamiento interno, que expandimos en Capítulo 5 (Extracción de testigos de existenciales) con extracción de testigos. En este capítulo presentamos la herramienta de línea de comandos `ppa`, que permite procesar programas del lenguaje. Está implementada en Haskell.

6.1. Instalación

Para instalar `ppa` a partir de los fuentes, seguir los siguientes pasos.

1. Instalar Haskell, Cabal
2. Clonar el repositorio de `ppa` o descargarlo `https://github.com/mnPanic/tesis`
3. Instalar la herramienta con cabal

   ```
   tesis/ppa:~$ cabal install ppa
   ```

4. Verificar la instalación

   ```
   tesis/ppa:~$ ppa version
   ppa version 0.1.0.0
   ```

6.2. Interfaz y ejemplos

El ejecutable `ppa` cuenta con dos comandos: `check` (certificado y chequeo de programas) y `extract` (extracción de testigos). En el repositorio de los fuentes, el directorio `ppa/doc/examples` contiene ejemplos de programas. La extensión por convención de programas de `ppa` es `.ppa`.

6.2.1. `check` - Chequeo de programas

Permite certificar y chequear programas.

```
ppa check <in> <args>
```

Argumentos:

- El primer argumento posicional es el archivo que contiene el programa a certificar. Puede ser `-` para leer de *stdin*.
- `-out`, `-o` (*opcional*): Path para el archivo al cual escribir el certificado (con el sufijo `_raw.nk`) o `-` para usar *stdout*.

Primero lee, certifica y chequea el programa reportando el resultado. En caso de haber proporcionado un archivo de output, escribe el certificado de deducción natural a `<path>_raw.nk`.

Ejemplos:

```
$ ppa check doc/examples/parientes.ppa
Checking... OK!
```

```
$ ppa check doc/examples/parientes.ppa --out out
Checking... OK!
Writing...
Wrote raw to out_raw.nk
```

6.2.2. `extract` - Extracción de testigos

Permite ejecutar la extracción de testigos.

```
ppa extract <in> <args>
```

Argumentos:

- El primer argumento posicional es el archivo de entrada que contiene el programa. Puede ser `-` para leer de *stdin*.
- `-theorem`, `-t` (*obligatorio*): el nombre del teorema para el cual extraer el testigo
- `-terms`, `ts`: la lista de términos que indica cómo instanciar las variables cuantificadas universalmente. Debe contener la misma cantidad de términos que variables.
- `-out`, `-o` (*opcional*): Path para el archivo al cual escribir los certificados (con los sufijos `_raw.nk` y `.nj`) o `-` para usar *stdout*.

Primero lee y certifica, y chequea el programa. Luego, lo traduce con la traducción de Friedman, normaliza la demostración e intenta extraer un testigo, reportando cual fue el testigo extraído y cómo queda el término final con el testigo y los términos proporcionados para instanciar las variables cuantificadas universalmente.

Por ejemplo, si tenemos el programa de Figura 5.2

```
$ ppa extract doc/listings/extract/forall.ppa --theorem t --terms x
Running program... OK!
Translating... OK!
Checking translated... OK!
Extracted witness: v
of formula: p(x, v)
```

Y especificando output,

```
$ ppa extract doc/listings/extract/forall.ppa --theorem t --terms x -o out
Running program... OK!
Translating... OK!
Writing...
Wrote raw to out_raw.nk
Wrote translated+reduced to out.nj
Checking translated... OK!
Extracted witness: v
of formula: p(x, v)
```

6.3. Detalles de implementación

En esta sección contamos algunos detalles relevantes sobre la implementación, que deberían permitir navegar los fuentes para ver cómo fue implementado. Comencemos por la arquitectura de los módulos.

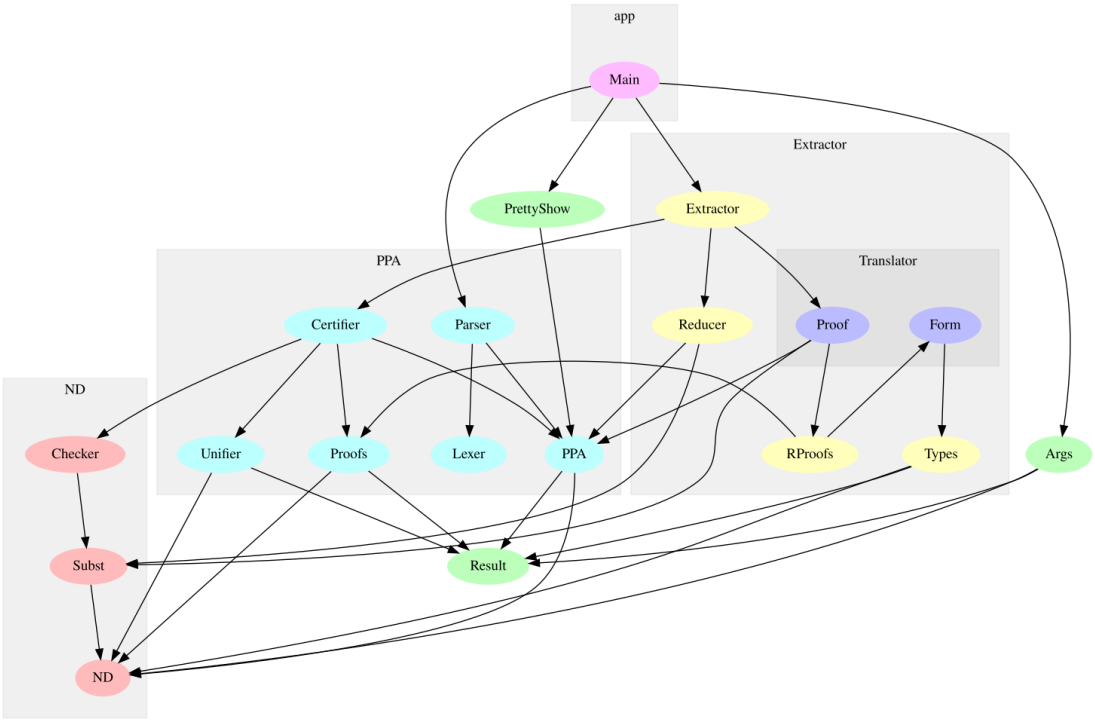

Fig. 6.1: Grafo de módulos del programa `ppa` generado con `graphmod`

Los más relevantes son,

- `ND`: Contiene el modelo central de deducción natural usado para los certificados. Fórmulas, términos y demostraciones. Tiene 3 sub módulos: `ND.Checker` (chequeador de demostraciones), `ND.Subst` (sustituciones sobre fórmulas y demostraciones) y `ND.ND` (el modelo en si). Ver Subsección 6.3.2 (Modelado de deducción natural).
- `PPA`: Contiene la implementación del lenguaje PPA por sobre deducción natural.
  - `PPA.Parser` y `PPA.Lexer` son el lexer y el parser respectivamente. Ver Subsección 6.3.1 (Parser y Lexer).
  - `PPA.Certifier` implementa el certificador, que genera demostraciones de deducción natural a partir de programas de PPA. Usa `PPA.Unifier` para unificación (usado en eliminación de $\forall$) y `PPA.Proofs` para demostraciones de equivalencias en ND (usado en DNF).
- `Extractor`: Contiene la lógica de extracción de testigos. Tiene varios sub módulos
  - `Extractor.Translator` implementa la traducción de Friedman en sus dos submódulos, `Extractor.Translator.Proof` para demostraciones y `Extractor.Translator.Form` para fórmulas.

- `Extractor.Reducer` implementa la normalización.
- `Extractor.Extractor` combina todo lo anterior para hacer la extracción de testigos de principio a fin, certificando, traduciendo y reduciendo. Usa `Extractor.RProofs` que es análogo a `PPA.Proofs` e implementa los lemas necesarios sobre demostraciones intuicionistas con negación relativizada.

6.3.1. Parser y Lexer

Una parte esencial de `ppa` es el compilador, que permite tomar un programa desde un archivo de texto y llevarlo a una representación estructurada, como un un tipo abstracto de datos, para el uso en el resto del programa. Se implementa en dos etapas.

- La primera es el análisis léxico o *lexer*, que convierte el texto plano en una lista de *lexemas* o *tokens*. Se puede implementar con expresiones regulares.
- Una vez que el programa ya fue convertido en *lexemas*, es procesado por el parser, que interpreta las estructuras sintácticas del lenguaje. Puede generar una representación intermedia como un tipo abstracto de datos.

Para el parsing en general hay dos caminos bien conocidos. O bien hacerlo a mano con alguna biblioteca de *parser combinators* como parsec, o usar un *parser generator*: un programa que genere un parser automáticamente a partir de una gramática en algún formato como EBNF. Nosotros decidimos, por familiaridad con el proceso, usar un parser generator. Los más conocidos históricamente son Lex (para el lexer) y Yacc (para el parser, *yet another compiler compiler*). En Haskell existen dos paquetes análogos: Alex para el lexer y Happy para el parser. Estos generan un lexer a partir de expresiones regulares y un parser respectivamente a partir de una gramática en un formato similar a EBNF.

En la Figura 6.2 se puede ver un extracto del archivo de input de Alex y en Figura 6.3 uno del de Happy de `ppa`. La implementación del parser solo construye términos de programas (definidos en `PPA.PPA`) y fórmulas (`ND.ND`) a partir de los cuales el programa opera.

6.3.2. Modelado de deducción natural

La parte más interesante del modelado son las demostraciones de deducción natural, que se pueden ver en la Figura 6.6. Una demostración está compuesta por la aplicación recursiva de reglas de inferencia, y su modelo omite todos los detalles que se pueden inferir durante su chequeo. De esa forma las demostraciones son más fáciles de escribir y generar, al costo de ser un poco más costosas de leer por un humano, cosa que de todas formas no deberíamos hacer. Por ejemplo, la introducción de la implicación, I→, modelada por `PImpI` no especifica cuál es la implicación que se está introduciendo, dado que durante el chequeo debería ser la fórmula actual a demostrar. Tampoco se explicita en cada regla, cual es el contexto de demostración. Se genera de forma dinámica.

```
tokens :-
    $white+                         ;
    "//".*                          ; -- comments
    "/*"(.|(\r\n|\r|\n))*"*/"       ; -- block comments
    \.              { literal TokenDot }
    \,              { literal TokenComma }
    \&              { literal TokenAnd }
    \|              { literal TokenOr }
    true            { literal TokenTrue }
    false           { literal TokenFalse }
    \-\>            { literal TokenImp }
    \<\-\>          { literal TokenIff }
    \~              { literal TokenNot }
    exists          { literal TokenExists }
    forall          { literal TokenForall }
    \(              { literal TokenParenOpen }
    \)              { literal TokenParenClose }
    axiom           { literal TokenAxiom }
    theorem         { literal TokenTheorem }
    proof           { literal TokenProof }
    end             { literal TokenEnd }
    \;              { literal TokenSemicolon }
    \:              { literal TokenDoubleColon }
    suppose         { literal TokenSuppose }
    thus            { literal TokenThus }
    hence           { literal TokenHence }
    have            { literal TokenHave }
    then            { literal TokenThen }
    by              { literal TokenBy }
    equivalently    { literal TokenEquivalently }
    claim           { literal TokenClaim }
    cases           { literal TokenCases }
    case            { literal TokenCase }
    take            { literal TokenTake }
    \:\=            { literal TokenAssign }
    st              { literal TokenSuchThat }
    consider        { literal TokenConsider }
    let             { literal TokenLet }

    \"[^\"]*\"          { lex (TokenQuotedName . firstLast) }

    (\_|[A-Z])[a-zA-Z0-9\_\-]*(\')*                     { lex TokenVar }
    [a-zA-Z0-9\_\-\?!#\$\%\*\+\<\>\=\?\@\^]+(\')*       { lex TokenId }
```

Fig. 6.2: Extracto del lexer de `ppa`

```
%token
    -- Enumeración de tokens del lexer

Prog    : Declarations

Declarations : Declaration Declarations
               | Declaration

Declaration : Axiom
            | Theorem

Axiom : axiom Name ':' Form

Theorem : theorem Name ':' Form proof Proof end

Proof   : ProofStep Proof
        | {- empty -}

ProofStep : suppose Name ':' Form
            | thus Form OptionalBy
            | hence Form OptionalBy
            | have Name ':' Form OptionalBy
            | then Name ':' Form OptionalBy
            | equivalently Form
            | claim Name ':' Form proof Proof end
            | cases OptionalBy Cases end
            | take var ':=' Term
            | let var
            | consider var st Name ':' Form by Justification

Cases   : Case Cases
        | {- empty -}

Case    : case Form Proof
        | case Name ':' Form Proof

OptionalBy : by Justification
OptionalBy : {- empty -}

Justification : Name ',' Justification
               | Name

Name    : id
        | name
```

Fig. 6.3: Extracto del parser de `ppa` (programas)

```
-- Resolución automática de conflictos shift/reduce
%right exists forall dot
%right imp iff
%left and or
%nonassoc not
%%

Form    : id TermArgs
        | Form and Form
        | Form or Form
        | Form imp Form
        | Form iff Form
        | not Form
        | exists var dot Form
        | forall var dot Form
        | true
        | false
        | '(' Form ')'

Term    : var
        | id TermArgs

TermArgs : {- empty -}
           | '(' Terms ')'

Terms   : Term
        | Term ',' Terms
```

Fig. 6.4: Extracto del parser de `ppa` (lógica de primer orden)

```
type VarId = String
type FunId = String
type PredId = String
type HypId = String

data Term
    = TVar VarId
    | TMetavar Metavar
    | TFun FunId [Term]
```

```
data Form
    = FPred PredId [Term]
    | FAnd Form Form
    | FOr Form Form
    | FImp Form Form
    | FNot Form
    | FTrue
    | FFalse
    | FForall VarId Form
    | FExists VarId Form
```

Fig. 6.5: Modelado de fórmulas y términos de LPO

```
data Proof =
    | PAx HypId
    | PAndI
        { proofLeft :: Proof
        , proofRight :: Proof
        }
    | PAndE1
        { right :: Form
        , proofAnd :: Proof
        }
    | PAndE2
        { left :: Form
        , proofAnd :: Proof
        }
    | POrI1
        { proofLeft :: Proof
        }
    | POrI2
        { proofRight :: Proof
        }
    | POrE
        { left :: Form
        , right :: Form
        , proofOr :: Proof
        , hypLeft :: HypId
        , proofAssumingLeft :: Proof
        , hypRight :: HypId
        , proofAssumingRight :: Proof
        }
    | PImpI
        { hypAntecedent :: HypId
        , proofConsequent :: Proof
        }
    | PImpE
        { antecedent :: Form
        , proofImp :: Proof
        , proofAntecedent :: Proof
        }
    | PNotI
        { hyp :: HypId
        , proofBot :: Proof
        }
    | PNotE
        { form :: Form
        , proofNotForm :: Proof
        , proofForm :: Proof
        }
    | PTrueI
    | PFalseE
        { proofBot :: Proof
        }
    | PLEM
    | PForallI
        { newVar :: VarId
        , proofForm :: Proof
        }
    | PForallE
        { var :: VarId
        , form :: Form
        , proofForall :: Proof
        , termReplace :: Term
        }
    | PExistsI
        { inst :: Term
        , proofFormWithInst :: Proof
        }
    | PExistsE
        { var :: VarId
        , form :: Form
        , proofExists :: Proof
        , hyp :: HypId
        , proofAssuming :: Proof
        }
```

Fig. 6.6: Modelado de reglas de inferencia para demostraciones

7. CONCLUSIONES