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

Buscador de Modelos Acotados para el demostrador de teoremas PVS

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Leonardo Teren

Director: Carlos Gustavo López Pombo

Codirector: Mariano Miguel Moscato

Buenos Aires, 2021

BUSCADOR DE MODELOS ACOTADOS PARA EL DEMOSTRADOR DE TEOREMAS PVS

La demostración de teoremas en un ambiente interactivo, donde la guía de una persona es necesaria, es usualmente una actividad tediosa y muy propensa a fallas. Pequeños errores en la declaración original pueden provocar la pérdida de una cantidad considerable de tiempo por "forzar" al usuario a lidiar con la inútil tarea de intentar probar una propiedad que no es válida. Estos errores tienen un amplio rango de orígenes, pero son mucho más probables de ocurrir cuando la actividad de demostración forma parte de desarrollos formales de gran complejidad. El Sitema de Verificación de Protipos (conocido como PVS por su nombre en inglés) es un *framework* mecanizado para la especificación y verificación formal de artefactos en Lógica de Alto Orden.

Por otro lado, los analizadores automáticos han presentado un crecimiento pronunciado respecto a su poder de alcance en las últimas décadas. Nuevas técnicas pueden ser aplicadas ahora a problemas que no podían ser tratados en el pasado. Un ejemplo de esos analizadores es Kodkod: un buscador de modelos para Lógica Relacional de Primer Orden que, entre otros usos, sirve como *backend* para el Alloy Analyzer, desarrollado por el Massachusetts Institute of Technology.

En esta tesis se presentan los primeros pasos para desarrollar un buscador de contraejemplos para PVS basado en Kodkod. Se propone un enfoque preliminar para una de las características fundamentales para el método propuesto, que es la traducción entre un fragmento relevante del lenguaje de PVS y el de Kodkod. Además discutimos sobre el campo de aplicación de esta técnica automática en el contexto de la demostración de teoremas, sobre sus limitaciones y otras técnicas similares.

Palabras claves: Demostrador de teoremas, especificación de sistemas, buscador de modelos, contraejemplo, consistencia.

BOUNDED MODEL FINDER FOR PVS

Theorem proving in an interactive environment, where human guidance is needed, is usually a tedious and very error prone activity. Small mistakes in the original statement can provoke the waste of huge amount of time by deeming the user to deal with the unfruitful task of trying to prove a non-valid property. These mistakes have a wide range of sources, but they are much more probably to occur when the proving activity forms part of a formal development of great complexity. The Prototype Verification System (PVS) is a mechanized framework for formal specification and verification for higher-order logic artifacts.

On the other hand, automatic analyzers have shown a steep grown in power in the last few decades. New techniques can now be applied to problems that could not be treated in the past. One example of such analyzers is Kodkod: a model finder for Relational First Order logic that, among other uses, serves as the backend for the Alloy Analyzer, developed by the Massachusetts Institute of Technology.

In this thesis, the first steps in the developing of a counterexample finder for PVS based on Kodkod is presented. A preliminary approach to the key feature of the proposed method, the translation between relevant fragment of PVS and Kodkod grammars, will be introduced. Also, we will discuss the field of application of this automated technique in the theorem proving context, its limitations, and other similar techniques.

Keywords: theorem prover, systems specification, model finder, counterexample, consistency.

AGRADECIMIENTOS

A todos.

A mi viejo.

Índice general

1. INTRODUCCIÓN

En la mayoría de los sistemas de software la descripción del comportamiento esperado se realiza utilizando lenguaje coloquial, a esta descripción se la denomina especificación. Basándose en esa descripción se desarrolla el software. Luego, se ejecutan tests de manera tal de comprobar que los resultados obtenidos sean los deseados. En general, esto logra un buen balance entre el tiempo invertido en el diseño y desarrollo del software con la calidad obtenida. Pero también hay escenarios donde un error en la especificación del sistema puede generar un costo altísimo, por lo que es necesario invertir más tiempo y energía, no solo testeando el sistema en casos particulares, si no que además hay que verificar la corrección de la especificación, o sea, tratar de asegurar lo mejor posible que nuestra descripción del sistema es acertada, y que no se esté expresando erróneamente el comportamiento esperado. Para esto es necesario que la especificación deje de ser en lenguaje coloquial y pase a utilizarse un lenguaje formal para ésta, de manera tal que sea posible realizar demostraciones que prueben las propiedades que se esperan que sean cumplidas por el sistema. Esto último es el objeto de análisis de este trabajo.

Los sistemas llamados *críticos* son aquellos cuyas eventuales fallas pueden provocar disrupciones en las operaciones de una organización o inclusive tragedias tanto a nivel económico como humano. El sistema de navegación de las aeronaves modernas es un ejemplo habitual de este tipo de desarrollos. Al producir esta clase de artefactos es necesario contar con altos niveles de certeza acerca de su corrección. Para ello, un vasto espectro de herramientas y técnicas para el análisis de sistemas ha sido creado en las últimas décadas. Si solamente nos enfocamos en el grado de automatización ofrecido por ellas, el abanico va desde herramientas completamente automatizadas (*lightweight*) a completamente guiadas por el usuario (*heavyweight*), denominación otorgada por el grado de esfuerzo necesario para su aplicación. Una automatización completa tiene su precio, que usualmente se paga con limitaciones en el tipo de análisis que se puede hacer, o en la falta de escalabilidad de dichos análisis. *Model checking* (usando herramientas como SPIN[1]) y herramientas basadas en *SAT-solving* (usando, por ejemplo, Alloy Analyzer[2]) son ejemplos que se encuentran de ese lado del espectro.

Las herramientas guiadas por el usuario ofrecen técnicas de análisis más concluyentes, pero requieren de usuarios altamente entrenados y su aplicación suele consumir mucho. Ejemplos clásicos de este tipo de sistemas son los demostradores de teoremas interactivos, como el Prototype Verification System (PVS) [3], Isabelle[4] o Coq[5]. Dichas herramientas demandan un esfuerzo significativo por parte del usuario pues la construcción de demostraciones es una tarea no trivial que requiere de habilidades de deducción formal. La mayoría de estos demostradores de teoremas, como es el caso de los mencionados anteriormente, se basan en lógica de alto orden debido a su enorme poder expresivo, teniendo como contracara que la construcción de herramientas automáticas de búsqueda de modelos sea una imposibilidad.

La corrección de una especificación puede validarse verificando que ciertas propiedades se deducen formalmente de las definiciones y axiomas básicos que la describen. Se pueden utilizar técnicas de *model-checking* y *SAT-Solving* de manera tal de asistir con la

tarea. En ambos casos, las técnicas tienen limitaciones en la expresividad de los lenguajes que pueden analizar. Esto es particularmente claro en el caso de *SAT-Solving*, donde el lenguaje de especificación es la lógica proposicional y su utilización muchas veces se limita al análisis de fragmentos decidibles de una lógica. Si en cambio consideramos el uso de demostradores de teoremas, es posible utilizar lenguajes más expresivos; por ejemplo, hay muchos demostradores de teoremas para la lógica de primer orden. Estos incluyen técnicas para probar automáticamente algunos subconjuntos de propiedades pero, en términos generales, la demostración de propiedades requiere pasos creativos que deben necesariamente ser guiados por personas. Demostrar un teorema puede ser una tarea difícil y tediosa, lo que se ve potenciado por la eventualidad de cometer errores como puede ser la introducción de una hipótesis errónea. Esta situación es más común de lo deseado siendo muy desalentador por la enorme pérdida de tiempo que suele conllevar. Hay que notar que cada paso de la demostración que depende de una hipótesis incorrecta debe ser revisado al momento de trabajar en una nueva demostración.

A pesar de sus respectivas limitaciones, tanto los métodos automáticos como los guiados por el usuario, poseen cualidades que los hacen valiosos. Si logramos traducir nuestro modelo a la lógica proposicional (i.e. a una instancia del problema de *SAT-Solving*) o a un autómata (como suele ser el caso de *model-checking*), el modelo (probablemente más débil) que se obtiene puede ser analizado de forma automática. Vale resaltar que el resultado de analizar este modelo más débil puede ofrecer información parcial sobre el modelo original. Si existe un cálculo completo para nuestro lenguaje de especificación, entonces no hay necesidad de traducir nuestro modelo a uno más débil y así, probar una propiedad usando su cálculo asociado, permite concluir su validez en el modelo.

Existen varios ejemplos de herramientas automáticas que realizan construcciones de modelos finitos, dentro de cotas preestablecidas para los dominios del problema; siendo uno de los más conocidos Alloy [6]. Estas herramientas tienen por objetivo buscar contraejemplos a una propiedad dada. Si bien un resultado negativo a esta búsqueda no garantiza la ausencia de contraejemplos puesto que la exploración se realiza dentro de ciertas cotas, una respuesta positiva nos garantiza que la propiedad deseada no se deduce de la especificación propuesta.

Es posible complementar el análisis formal que proveen los demostradores de teoremas con el aporte que ofrece *SAT-solving*. Esto se logra realizando una traducción que obtenga como resultado una fórmula sirva como *input* para el *SAT-solver*. Esta interacción es muy valiosa para algunos aspectos de la demostración, como ser:

- Cuando durante una demostración usando el *bounded model-checker* se pueden detectar errores en la introducción de hipótesis
- Refinar secuentes detectando qué fórmulas no resultan de utilidad para llevar a cabo la demostración
- Proponer términos para utilizar como testigos de propiedades cuantificadas existencialmente

Una manera posible de obtener asistencia para una demostración es hacer una traducción de lógica de alto orden hacia lógica de primer orden. Y luego obtener demostraciones dentro del mundo sintáctico, en lógica de primer orden. Finalmente, y donde se encuentra el mayor desafío, se hace el pasaje de lógica de primer orden a lógica de alto orden. En

este apartado es donde podemos encontrar trabajos como los de Sledgehammer [7], [8] [**?**] [9] y [10]. Cabe destacar que en un principio hay una similitud con nuestro enfoque, ya que pasamos de lógica de alto orden a lógica de primer orden. Pero luego hay una intencionalidad distinta: nosotros utilizamos un buscador de modelos para generar instancias o contraejemplos, mientras que los *hammers* buscan una demostración.

Sinergia *lightweight-heavyweight*

Es usual partir de un conjunto de axiomas, querer probar una propiedad y que la demostración falle. Es un proceso que involucra trabajo manual de elevada complejidad y es común que se cometan errores. Otro escenario que puede suceder es que simplemente no sea posible hacer la demostración. En cualquiera de los dos casos se termina desperdiciando tiempo y esfuerzo.

Es posible combinar los métodos *lightweight* y *heavyweight* con el objetivo de alivianar la tarea de demostración de teoremas. Para esto se utiliza ambos enfoques de manera complementaria, es decir, la demostración se basa en un método *heavyweight*, pero a su vez se utilizan métodos *lightweight* para intentar encontrar un contraejemplo que invalide la demostración, o un modelo que asegure la consistencia de las hipótesis. De esta manera podemos evitar la pérdida de tiempo que se genera al intentar demostrar una propiedad que no se deduce de los axiomas.

Para lograr esta sinergia se pasa de una lógica más expresiva a una menos expresiva. Obviamente que esto trae aparejado un *tradeoff*: la asistencia provista en muchos escenarios es de ayuda, pero en otros seguimos teniendo incertidumbre. Por lo pronto, queremos trabajar con una lógica decidible, lo que conlleva a tratar universos finitos, haciendo que el análisis termine siendo algo parcial y no completo. Entonces, vale recalcar una vez más, que la imposibilidad de encontrar un modelo no indica que no exista, solamente tenemos la certeza de que no existe en el contexto definido. A pesar de las limitaciones ya mencionadas, siguen siendo beneficioso utilizar este enfoque. A continuación mencionaremos algunas herramientas relacionadas.

Dynamite[11] es un demostrador de teoremas para Alloy basado en PVS. Además, contiene extensiones que permiten disminuir el esfuerzo que una demostración requiere, analizando automáticamente nuevas hipótesis con la ayuda del Alloy Analyzer. A medida que se va avanzando en una demostración se van obteniendo fórmulas, y algunas de ellas pueden ser innecesarias para lo que se quiere demostrar, entonces contar con una forma de poder deshacerse de éstas es un gran beneficio para que los esfuerzos sean enfocados en las fórmulas que son útiles para la demostración; es en este escenario donde Alloy Analyzer puede ayudar refinando el conjunto de fórmulas. También es posible generar testigos para las fórmulas existencialmente cuantificadas.

Dynamite complementa el análisis parcial y automático que el Alloy Analyzer ofrece, con verificación semi-automática a través de la demostración de teoremas. Además mejora la experiencia de usuario, proveyendo una detección temprana de errores (además de los refinamientos de secuentes y la generación de testigos ya mencionados).

Isabelle es un asistente de demostración genérico. Permite expresar fórmulas matemáticas en un lenguaje formal y provee herramientas para probar esas fórmulas usando

un cálculo de deducción sintáctica. La aplicación principal es la formalización de demostraciones matemáticas y verificación de sistemas, que incluye la demostración de la corrección de hardware y software.

Nitpick[12] es un generador de contraejemplos para Isabelle/HOL que está diseñado para manejar fórmulas combinando tipos de datos (co)inductivos, predicados definidos (co)inductivamente y cuantificadores; y aproxima tipos infinitos usando subconjuntos finitos. Se basa en Kodkod, un buscador de modelos relacional que describiremos un poco más adelante. Conceptualmente es similar a Refute[13], del cual toma prestadas muchas ideas y fragmentos de código, pero se beneficia de las optimizaciones de Kodkod y un nuevo esquema de codificación.

Contribución

Cuando es necesario tener la mayor certeza posible sobre la correctitud de una especificación es necesario demostrar propiedades. PVS es uno de los *frameworks* que permiten realizar esta actividad, que es una tarea muy pesada en términos de esfuerzo y tiempo. En un intento por mejorar esta situación, desarrollamos una herramienta que ayuda a detectar más fácilmente la presencia de errores en el proceso de demostración, buscando un contraejemplo que cumpla con los axiomas de la especificación dada pero no con una propiedad que es requerida. Esto ahorra tiempo y esfuerzo de manera significativa en los escenarios donde un error fue introducido involuntariamente y por lo tanto la demostración sería imposible de completar.

Particularmente, el trabajo de esta tesis se enfoca en proveer ayuda en los puntos críticos de la demostración. Existen comandos de PVS que preservan la demostrabilidad, pero también hay otros que no lo hacen. Cuando se ejecuta uno de estos últimos se agrega información que potencialemente puede invalidar la demostrabilidad; un ejemplo de esto podría ser la instanciación de un existencial que contradice al menos un axioma de la especificación; y ese escenario sería un punto crítico. Vale la pena recalcar que nuestra herramienta tratar de minimizar los errores que pueda llegar a cometer la persona que está llevando a cabo la demostración, ya que asumimos que el demostrador que se utiliza no tiene errores.

Esta tesis se organiza de la siguiente manera: en el capítulo 2 presentamos los temas preliminares que son de interés para poder comprender mejor el trabajo realizado, el desarrollo realizado está presentado en el capítulo 3. En el capítulo 4 se ejemplifica el uso de la herramienta a través de un caso de estudio. En el capítulo 5 se discuten las conclusiones y el trabajo futuro se describe en el capítulo 6.

2. PRELIMINARES

2.1. *Prototype Verification System*

PVS provee un *framework* automatizado que se enfoca en el desarrollo y verificación de modelos conceptuales complejos. Sus características más relevantes son el lenguaje de especificación y el demostrador de teoremas interactivo que sirve de herramienta principal para dicha verificación.

Su lenguaje de especificación está basado en el de la lógica clásica de alto orden enriquecida con tipos de datos, bajo una semántica estándar para esta clase de lógicas. El fragmento básico de la gramática para expresiones del lenguaje de PVS se describe en la figura 2.1. Nótese que en esta figura, así como en buena parte del resto de este documento, el término *expresión* se utiliza en una forma general en la que se incluye también a las fórmulas del lenguaje, dado que no son más que expresiones *booleanas*. El lenguaje de PVS dispone de otras características muy útiles para el desarrollo de especificaciones formales, como cláusulas condicionales (*if-then-else*) y declaraciones locales (*let-in*), pero ellas no son consideradas en este documento para facilitar la presentación del trabajo realizado.

Respecto de los tipos de datos, PVS provee ciertos tipos nativos, como *booleanos*, tipos numéricos y cadenas de caracteres (*strings*), entre otros. Además, PVS permite la declaración de tipos nuevos, ya sea combinando tipos existentes o sin una definición explícita. Los primeros son llamados tipos *interpretados* y los segundos *no interpretados*. Los tipos interpretados pueden generarse usando constructores de tipos usuales como los que permiten crear tipos funcionales, productos (también llamados *tuplas*), co-productos (también conocidos como *uniones disjuntas*) y registros. Adicionalmente, es posible incluir una restricción en la definición misma del tipo, expresándola como un predicado del lenguaje (*predicate subtyping*). La figura 2.2 muestra un fragmento de la gramática del lenguaje correspondiente a la declaración de tipos.

Cabe destacar que en la figura 2.2 se omiten características significativas del lenguaje de declaración de tipos de PVS que no son centrales para este trabajo, como por ejemplo la posibilidad de declarar tipos dependientes, subtipos estructurales, etcétera.

$$
\begin{array}{rl}
E ::= \neg E \mid E \wedge E & \text{(conectivos lógicos)} \\
\mid E = E & \text{(igualdad)} \\
\mid id \mid id(id, \cdots, id) & \text{(aplicación)} \\
\mid \forall (id : T) : E & \text{(cuantificación universal)} \\
\mid \exists (id : T) : E & \text{(cuantificación existencial)}
\end{array}
$$

Fig. 2.1: Fragmento de interés de la gramática de PVS para expresiones. El símbolo *id* representa a los identificadores del lenguaje y T a las expresiones que denotan tipos de datos.

$$
\begin{aligned}
\textit{type\_decl} ::= & \; id_T \texttt{ : TYPE = } T \\
T ::= & \; \texttt{bool} \mid \texttt{nat} \mid \texttt{int} \mid \cdots && \text{(tipos nativos)} \\
\mid & \; id_T && \text{(tipos definidos previamente)} \\
\mid & \; [T, \cdots, T \to T\,] && \text{(funciones)} \\
\mid & \; [T, \cdots, T\,] && \text{(productos)} \\
\mid & \; [T \texttt{ + } \cdots \texttt{ + } T\,] && \text{(co-productos)} \\
\mid & \; [\#\ id\ :\ T, \cdots, id\ :\ T\ \#] && \text{(registros)} \\
\mid & \; \{id\ :\ T \mid E\,\} && (\textit{predicate subtyping})
\end{aligned}
$$

Fig. 2.2: Fragmento de interés de la gramática de PVS para declaraciones de tipos. La expresión E para el caso de *predicate subtyping* debe ser de tipo Bool.

Por ejemplo, la siguiente expresión define el tipo `intfun` como el tipo de las funciones que tienen a los números enteros como dominio e imagen.

```
intfun: TYPE = [int -> int]
```

El apartado 2.1 muestra un ejemplo de una teoría PVS que modela el Principio del Palomar. Las líneas 4 y 5 declaran dos tipos de datos no interpretados, los cuales además son no vacíos debido al modificador `+` al final de la palabra reservada `TYPE`. La línea 7 declara un símbolo de función `pigeonhole` cuyo dominio son los elementos de tipo `Pigeon` y cuya imagen son los de `Hole`. El axioma `pigeonhole_is_injective` de las líneas 9-11 declara la inyectividad para la función `pigeonhole`, impiendo que haya más de un `pigeon` para un mismo `hole`. Por último, en las líneas 12-14 se declara un lemma que se llama `erroneous` diciendo que para cualquier par de elementos `p1` y `p2` del tipo `Pigeon` que sean distintos se deduce que el elemento del tipo `Hole` que le corresponde a `p1` es distinto que al que le corresponde a `p2`, lo cual es no es necesariamente cierto ya que en ningún momento se declara ese tipo de restricción en la teoría.

```
1  pigeonhole: THEORY
2  BEGIN
3
4      Pigeon: TYPE+
5      Hole: TYPE+
6
7      pigeonhole: [Pigeon -> Hole]
8
9      pigeonhole_is_injective : AXIOM
10         FORALL(p1, p2 : Pigeon):
11             pigeonhole(p1) = pigeonhole(p2) IMPLIES p1 = p2
12
13     erroneous: LEMMA
14         FORALL(p1,p2: Pigeon):
15         p1/=p2 IMPLIES pigeonhole(p1)/=pigeonhole(p2)
16
17 END pigeonhole
```

Listing 2.1: Ejemplo de teoría PVS.

$$\frac{\Gamma_1 \vdash \Delta_1 \qquad \cdots \qquad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta} R$$

Fig. 2.3: Esquema de regla de demostración.

Como puede verse, todas las características descriptas anteriormente determinan el alto grado de expresividad del lenguaje de PVS. De hecho, ellas permiten que PVS pueda soportar otras construcciones de tipos mediante facilidades implementadas como *azúcar sintáctico*. La declaración de tipos abstractos de datos (TAD) definidos inductivamente, como listas o árboles binarios, y tipos abstractos definidos co-inductivamente, como *streams*, se implementan de esta manera.

Debido al alto nivel de expresividad del lenguaje, en particular al soporte de *predicate subtyping*, el proceso de chequeo de tipos en PVS no puede ser completamente automático. Si bien en una primera etapa se detectan automáticamente errores sintácticos y gramaticales, el *type checker* puede emitir obligaciones de prueba que deben ser demostradas por el usuario para poder completar el proceso de chequeo de tipos. Estas obligaciones, llamadas *Condiciones de Corrección de Tipos* (TCC[1]), deben ser probadas utilizando el mismo demostrador provisto por PVS.

El demostrador

El módulo de demostración implementa un cálculo de secuentes especialmente adaptado al lenguaje de PVS. Como es usual en este tipo de cálculos, el demostrador mantiene una estructura que se conoce como *árbol de demostración*. Cada uno de sus nodos, llamados *secuentes*, contiene dos secuencias finitas de fórmulas. Si una fórmula aparece en la primera de las secuencias, se dice que es un *antecedente* del secuente, mientras que si aparece en la segunda, es llamada *consecuente*. La interpretación intuitiva de un secuente es que la conjunción de los antecedentes implica la disyunción de los consecuentes.

El usuario puede expandir el árbol aplicando un comando de demostración a alguna de sus hojas, lo que puede generar uno o más descendientes o cerrar la rama. Una rama se considera cerrada cuando todas sus hojas puedan ser trivialmente reconocidas como verdaderas, es decir, cuando la implicación subyacente de la hoja es trivialmente cierta. Este proceso de expansión continúa hasta que se consigue cerrar todas las ramas del árbol. Cada comando de demostración implementa una o más reglas de inferencia del cálculo. A continuación repasaremos las reglas básicas y daremos algunos ejemplos de cómo son implementadas en el demostrador de PVS. Para mayor información, recomendamos indagar en el manual correspondiente [14]. Las reglas de inferencia se muestran en diagramas como el presentado en la figura 2.3, donde las letras griegas mayúsculas representan listas finitas pero potencialmente vacías de fórmulas. Ese diagrama indica que si se aplica la regla R a una hoja del árbol de demostración de la forma $\Gamma \vdash \Delta$, se obtiene un árbol con n nuevas hojas $\Gamma_1 \vdash \Delta_1$ hasta $\Gamma_n \vdash \Delta_n$.

La forma de cerrar una rama del árbol de demostración es mediante la aplicación de alguna de las reglas axiomáticas descriptas en la figura 2.4. La primera de ellas puede aplicarse a un secuente con dos fórmulas A y B que sean equivalentes sintácticamente

[1] Por las siglas de su nombre en inglés: *Type-Correctness Condition*.

$$\frac{}{\Gamma, A \vdash B, \Delta} Ax(A \equiv B) \qquad \frac{}{\Gamma \vdash B, \Delta} Ax(B \equiv \top) \qquad \frac{}{\Gamma, A \vdash \Delta} Ax(A \equiv \bot)$$

Fig. 2.4: Reglas axiomáticas.

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \wedge \vdash \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \vdash \wedge$$

$$\frac{A, \Gamma \vdash \Delta \qquad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta} \vee \vdash \qquad \frac{\Gamma \vdash A, B\Delta}{\Gamma \vdash A \vee B, \Delta} \vdash \vee$$

$$\frac{B, \Gamma \vdash \Delta \qquad \Gamma \vdash A, \Delta}{A \supset B, \Gamma \vdash \Delta} \supset \vdash \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta} \vdash \supset$$

$$\frac{\Gamma \vdash A, \Gamma}{\Gamma, \neg A \vdash \Delta} \neg \vdash \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \vdash \neg$$

Fig. 2.5: Reglas proposicionales.

módulo el renombre de las variables ligadas que aparecen en ellas. Estas reglas también permiten cerrar ramas con hojas en las que uno de los antecedentes sea equivalente a una contradicción o uno de los consecuentes lo sea a una tautología. Las reglas axiomáticas son aplicadas automáticamente por PVS.

La figura 2.5 muestra las reglas para conectivos proposicionales: conjunciones, disyunciones, implicaciones y negaciones. Hay dos reglas para cada conectivo proposicional de la conjunción ($\wedge$), disyunción ($\vee$), implicación ($\supset$) y negación ($\neg$), que se corresponden con las ocurrencias en el antecedente y consecuente de estos conectivos. El demostrador de PVS aplica automáticamente las reglas referidas a la negación, $\neg \vdash$ y $\vdash \neg$, de modo que al usuario nunca se le presenta un secuente que contenga una negación entre sus fórmulas. Más adelante detallaremos otras implicancias de esta característica. Las reglas proposicionales que sólo generan un secuente nuevo, es decir: $\wedge \vdash$, $\vdash \vee$ y $\vdash \supset$, son implementadas mediante el comando `flatten`. Por otro lado, el comando `split` implementa las reglas de la figura 2.5 que dividen la rama en dos: $\vdash \wedge$, $\vee \vdash$ y $\supset \vdash$. PVS también cuenta con comandos de más alto nivel que realizan varias simplificaciones en un mismo paso; por ejemplo, el comando `prop` aplica todas las reglas proposicionales posibles sobre el secuente donde se ejecuta.

Las reglas para fórmulas con cuantificadores se muestran en la figura 2.6. La notación $A\{x \leftarrow t\}$ representa el resultado de sustituir todas las apariciones libres de x en A por t con la posibilidad de renombrar las variables ligadas en A para evitar capturar cualquier variable libre en t. En las reglas $\vdash \forall$ y $\vdash \exists$, a tiene que ser una constante nueva que no aparece en la conclusión del secuente. Como indican $\forall \vdash$ y $\vdash \exists$, las variables cuantificadas universalmente en un antecedente o las cuantificadas existencialmente en un consecuente pueden instanciarse con términos provistos por el usuario. Para ello, PVS provee una colección de comandos especialmente desarrollados con ese fin. Uno de los más básicos es `instantiate`, comando que acepta dos argumentos `fnum` y `exprs`. El primero de ellos indica la fórmula cuantificada que se quiere instanciar, ya sea un antecedente de la forma $(\forall x_1, \cdots, x_n : A)$ o un consecuente de la forma $(\exists x_1, \cdots, x_n : A)$. Por su parte, el argumento *exprs* debe usarse para proveer una lista de n términos $t_1, \cdots, t_n$ para que la fórmula cuantificada elegida sea reemplazada por $A[t_1 \leftarrow x_1, \cdots, t_n \leftarrow x_n]$ en el nuevo secuente. Hay que notar que cada término t_i se somete a un chequeo de corrección de tipos

$$\frac{\Gamma, A\{x \leftarrow t\} \vdash \Delta}{\Gamma, (\forall x : A) \vdash \Delta} \forall \vdash \qquad \frac{\Gamma \vdash A\{x \leftarrow a\}}{\Gamma \vdash (\forall x : A), \Delta} \vdash \forall$$

$$\frac{\Gamma, A\{x \leftarrow a\} \vdash \Delta}{\Gamma, (\exists x : A) \vdash \Delta} \exists \vdash \qquad \frac{\Gamma \vdash A\{x \leftarrow t\}}{\Gamma \vdash (\exists x : A), \Delta} \vdash \exists$$

Fig. 2.6: Reglas para fórmulas cuantificadas.

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash \Delta} Cut$$

Fig. 2.7: Regla de corte.

para asegurar que t_i sea compatible con el tipo de x_i; este proceso puede generar secuentes adicionales correspondientes a TCCs que deben ser probados por el usuario. PVS también provee una colección de distintos comandos enfocados en la implementación de las reglas de eliminación de cuantificaciones de sentido universal: $\vdash \forall$ y $\exists \vdash$. Algunos ejemplos de estos comandos son: `skolem`, `skosimp*`, `skeep`, etc. La mayoría de ellos aceptan como argumento un número de fórmula indicando el consecuente o antecedente que se desea procesar y una lista de identificadores nuevos que son juegan el rol del símbolo a en las reglas de la figura 2.6.

El cálculo de secuentes implementado en PVS incluye la *Regla de Corte* (Cut Rule) descrita en la figura 2.7, la cual puede usarse para incorporar una fórmula nueva A al secuente actual, lo que genera dos sub-ramas: en la primera se incorpora la fórmula A entre los antecedentes y en el segundo la misma fórmula se agrega como consecuente. Nótese que, debido a las reglas de equivalencia de la negación, los secuentes generados por la regla de corte pueden verse como una separación en casos de la demostración ya que el segundo secuente es equivalente a uno en el que $\neg A$ se incorpora como antecedente. Adicionalmente, la regla de corte puede aplicarse con otra intención: si pensamos a la fórmula A como una nueva hipótesis, en lugar de entenderla como la condición sobre la cual se separa en casos la rama de demostración, el segundo de los secuentes se puede interpretar como la obligación de demostrar que A se sigue del secuente original.

El más básico de los comandos de demostración provistos por PVS que implementa la regla de corte es el llamado `case`. Este comando materializa una versión más general de la regla, como lo muestra la figura 2.8. Si el secuente actual es de la forma $\Gamma \vdash \Delta$, entonces la regla (`case` $A_1 \cdots A_n$) genera genera $n + 1$ secuentes nuevos. Esto permite asumir que la fórmula o colección de fórmulas es correcta y luego demostrar que efectivamente son verdaderas. Como hemos señalado anteriormente, las fórmulas que son argumentos del comando se someten a un proceso de chequeo de tipos que puede llegar a generar secuentes adicionales que se corresponden con las TCCs resultantes de dicho proceso.

Adicionalmente, el comando `lemma`, por su parte, permite agregar propiedades pre-

$$\frac{A_n, \ldots, A_1, \Gamma \vdash \Delta \quad A_{n-1}, \ldots, A_1, \Gamma \vdash A_n, \Delta \quad \cdots \quad A_1, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} (\texttt{case}\ A_1 \ldots A_n)$$

Fig. 2.8: Implementación de la regla de corte en PVS.

viamente definidas, como obligaciones de prueba o axiomas, a la rama actual de la demostración. Este comando admite como argumento el nombre de tal propiedad a incorporar. Es importante remarcar la diferencia entre los comandos `case` y `lemma`: al aplicar `lemma` se asume que la información que se incorpora a la demostración es verdadera, es decir, no es responsabilidad de la demostración actual justificar la validez de la fórmula que se introduce. Se podría decir que `lemma` es una versión de `case` en la que la rama donde se prueba la nueva hipótesis no se incluye en la demostración actual.

2.2. Kodkod

Kodkod [15] es un *constraint solver* basado en *SAT-solving* para lógica relacional que sirve para el análisis de diseños, verificación de código y generación de casos de tests. Para los problemas que son satisfacibles trabaja con un buscador de modelos finitos [16] y para los que son insatisfacibles con un *minimal unsatisfiable core extractor* [17]. Trabaja traduciendo un problema relacional a un problema de satisfacibilidad booleano, aplicando un *SAT-solver* sobre la fórmula resultante, y finalmente convirtiendo el resultado arrojado por el *SAT-solver* al dominio relacional.

Una fórmula en lógica relacional es una sentencia sobre un alfabeto de variables relacionales. Una constante relacional es un conjunto de tuplas tomado del universo del conjunto de átomos no interpretados. Un modelo, o una instancia, de una fórmula son una colección de constantes relacionales que hacen verdadera a la fórmula. Un motor que busca modelos de una fórmula en un universo finito es un buscador de modelos finito o, simplemente, un buscador de modelos.

Kodkod está basado en la lógica relacional de Alloy, esencialmente consiste de una lógica de primer orden extendida con operadores de cálculo relacional. La lógica de Alloy trata todo como una relación: los conjuntos como relaciones de aridad uno y los escalares como conjuntos *singletons*.

Sintaxis y Semántica

Un problema Kodkod consiste en una declaración de universo, un conjunto de declaraciones de relaciones, y una fórmula en la cual las relaciones declaradas aparecen como variables libres. Cada relación declarada especifica su aridad y sus cotas inferior y superior. La cota inferior contiene las tuplas que el valor que tome la variable en un modelo debe incluir necesariamente. La cota superior contiene las tuplas que el valor que tome la variable en un modelo puede o no incluir. Los elementos con los que se construyen las tuplas que formarán los valores que tomen las relaciones son tomados del universo del problema.

Para ilustrar esto, consideremos la formulación de la figura 2.9 del Principio del Palomar donde n palomas no pueden ser colocadas en $n-1$ huecos con cada paloma teniendo el hueco para si misma; vamos a tomar 3 palomas y 2 huecos. La primera línea indica el universo de átomos no interpretados, en este caso son 5. Arbitrariamente se eligen los primeros 3 para representar palomas y los últimos 2 para representar huecos. Como las fórmulas no pueden contener constantes, indicamos con la variable relacional $v :_k [C, C]$ con las mismas cotas inferiores y superiores con aridad k para cada constante C que necesita ser usada por una fórmula del problema. Las variables `Pigeon` y `Hole`,

$\{P1, P2, P3, H1, H2\}$

$\texttt{Pigeon} :_1 [\{\langle \texttt{P1}\rangle, \langle \texttt{P2}\rangle, \langle \texttt{P3}\rangle\}, \{\langle \texttt{P1}\rangle, \langle \texttt{P2}\rangle, \langle \texttt{P3}\rangle\}]$
$\texttt{Hole} :_1 [\{\langle \texttt{H1}\rangle, \langle \texttt{H2}\rangle\}, \{\langle \texttt{H1}\rangle, \langle \texttt{H2}\rangle\}]$
$\texttt{pigeonhole} :_2 [\{\}, \{\langle \texttt{P1}, \texttt{H1}\rangle, \langle \texttt{P1}, \texttt{H2}\rangle, \langle \texttt{P2}, \texttt{H1}\rangle, \langle \texttt{P2}, \texttt{H2}\rangle, \langle \texttt{P3}, \texttt{H1}\rangle, \langle \texttt{P3}, \texttt{H2}\rangle\}]$

$(\texttt{all p1, p2 : Pigeon} \mid \texttt{p1.pigeonhole} = \texttt{p2.pigeonhole} => \texttt{p1} = \texttt{p2})$

Fig. 2.9: Principio del palomar escrito en Kodkod para 3 palomas y 2 huecos

por ejemplo, sirven como identificadores para las constantes unarias $\{\langle P1\rangle\langle P2\rangle\langle P3\rangle\}$ y $\{\langle H1\rangle\langle H2\rangle\}$, que representan los conjuntos de todas las palomas y huecos respectivamente. La variable $pigeonhole \subseteq Pigeon \times Hole$ codifica la asignación de palomas a huecos. Su valor está restringido para ser una función inyectiva en la fórmula del problema.

Las producciones sintácticas, además de las declaraciones del universo y relaciones, definen una lógica relacional estándar con clausura transitiva, cuantificadores de primer orden y conectivos. Los operadores de la clausura ($\tilde{}$) y transposición ($\hat{}$) sólo pueden ser aplicados a expresiones binarias. Las expresiones de aridad cero y mixtas no están permitidas. La aridad de la variable de relación y sus cotas declaradas deben coincidir. El símbolo que denota al conjunto vacío constante, notado $\{\}$, es polimórfico, haciéndolo una cota válida en el contexto de cualquier declaración.

Kodkod evalúa recursivamente las declaraciones de relaciones y fórmulas respecto a un *binding* de variables a constantes. Se considera que un problema es verdadero respecto a un *binding* dado si y sólo si sus declaraciones y fórmulas son verdaderas bajo ese *binding*. Las expresiones de conjuntos son interpretadas como tuplas. Átomos, tuplas y constantes tienen sus interpretaciones estándar de teoría de conjuntos. Eso es, el significado de un átomo es su nombre, el significado de una tupla es una secuencia de átomos, y el significado de una constante es un conjunto de tuplas.

Análisis

El análisis de un problema Kodkod P involucra los siguientes cinco pasos:

1. Detección de simetrías de P.
2. Traducción de P a un *Compact Boolean Circuit*, CBC(P).
3. Computar SBP(P), a *symmetry breaking predicate* para P.
4. Transformar $CBC(P) \wedge SBP(P)$ a forma normal conjuntiva, CNF(P).
5. Aplicar un *SAT-solver* a CNF(P), y, si CNF(P) es satisfacible, interpretar el modelo como una instancia de P.

Cuando una problema no tiene un modelo para un universo dado, la mayoría de los buscadores de modelos simplemente informan que es insatisfacible en ese universo y no dan más información. Pero muchas aplicaciones necesitan saber la causa de la insatisfacibilidad, ya sea para tomar alguna acción correctiva o para chequear que el modelo no existe por las razones adecuadas. La falta de modelos no quiere decir que el análisis fue exitoso.

Si no existen modelos porque el problema está sobrecargado, o porque una propiedad es una tautología, se considera que el análisis falló por un error en la descripción del problema. La causa de una insatisfacibilidad de un problema dado, expresada como un subconjunto de restricciones de la especificación es insatisfacible en si mismo, y es llamado *unsatisfiable core*. Cada *unsatisfiable core* incluye una o más restricciones críticas que no pueden ser removidas sin dejar al resto del *core* satisfacible. Las restricciones que no son críticas, si existen, son irrelevantes para la insastisfacibilidad. *Cores* que solamente incluyen restricciones críticas se dicen que son minimales (por eso Kodkod es considerado un *minimal unsatisfiable core extractor* para los casos insatisfacibles).

Kodkodi

Kodkodi[18] es un front-end para Kodkod. Está diseñado para que la biblioteca Kodkod esté disponible para otros lenguajes de programación además de Java. Kodkodi toma la entrada del *standard input* y escribe su salida en el *standard output* (en caso de éxito) o *standard error* en caso de falla. Básicamente Kodkodi recibe una lista de problemas. A su vez, un problema consiste básicamente de 3 partes principales: un universo de especificación, un conjunto de especificaciones acotadas, y una fórmula *Kodkod* para chequear su satisfacibilidad. En esta tesis utilizamos Kodkod a través de Kodkodi.

3. BUSCADOR DE MODELOS PARA PVS USANDO KODKOD

Dada una especificación, supongamos que se quiere demostrar una propiedad, pero no se tiene la certeza de que sea válida. Cuando se hace esto formalmente, los primeros pasos suelen estar avocados a encontrar indicios de su validez. Uno de estos pasos, comúnmente utilizado en la práctica, se centra en explorar la posibilidad de que exista un modelo de las hipótesis que no satisfaga la propiedad de interés. En principio, esta metodología no tiene soporte bajo un único criterio de análisis. En el contexto de esta tesis, se propone la utilización de PVS para el desarrollo de la demostración formal, tarea que se realiza semi-automáticamente, dado que el lenguaje no es decidible. Por otro lado, proponemos la utilización de Kodkod como mecanismo que posibilita la exploración de un espacio finito de modelos finitos. En este contexto de automatización de la búsqueda de contraejemplos, la decisión de cuándo se da por agotada esta búsqueda puede estar determinada por dos causas: (1) el espacio de modelos a explorar resulta demasiado grande, haciendo que el proceso sea computacionalmente muy costoso, o (2) la imposibilidad de construir tal modelo debido a que los axiomas involucrados fuerzan su infinitud (más adelante en esta sección discutiremos más profundamente las implicancias de este caso).

Ahora bien, esta tarea de buscar contraejemplos como parte de la metodología que seguimos cuando queremos demostrar una propiedad, no solo se lleva a cabo al comienzo de la demostración. También puede suceder en otros momentos a lo largo del proceso de prueba. Por ejemplo, puede ocurrir cuando planteamos la utilización de un lema o cuando decidimos ignorar hipótesis que estimamos que ya no usaremos. Así, la búsqueda de contraejemplos es de utilidad en situaciones específicas de la demostración donde creemos que el paso dado podría modificar la *demostrabilidad* de la propiedad original. Cabe aclarar que por demostrabilidad nos referimos a la existencia de un artefacto deductivo, por ejemplo el árbol de pasos de demostración que constituye una prueba en cálculo de secuentes, que permita garantizar la validez de una fórmula. Estos momentos de la demostración que pueden poner en peligro la existencia de tal artefacto, son denominados *puntos críticos*. A continuación se enumeran cuatro situaciones puntuales que pueden considerarse puntos críticos de una demostración.

El primer escenario sucede cuando la demostración demanda cierta creatividad del usuario en la continuación de la demostración y se modifica el secuente de forma que pueda alterarse la demostrabilidad de la propiedad. Un ejemplo de esta situación se da en la instanciación de fórmulas que establecen propiedades existenciales. PVS provee una familia de comandos que permiten realizar esta tarea:

- `inst`: instancia una fórmula sin copiar
- `inst-cp`: copia e instancia una fórmula
- `inst?`: instancia una fórmula haciendo *matching*
- `instantiate`: instanciación primitiva
- `instantiate-one`: instanciar una fórmula existencial sin duplicación

Por ejemplo, el siguiente secuente es demostrable, ya que existen valores *booleanos* que la cumplen.

$$\begin{array}{ll} & \vdash \\ (1) & \exists(x, y, z : bool) : \neg x \vee (y \wedge z) \end{array}$$

Sin embargo, si se aplica el comando `(instantiate 1 ("false" "true" "false"))`, el secuente resultante que se muestra a continuación nos permite observar que la propuesta de testigo para el existencial hace que no se pueda cerrar la rama.

$$\begin{array}{ll} (-1) & true \\ & \vdash \\ (1) & true \wedge false \end{array}$$

Otro escenario posible ocurre cuando se generan TCCs y alguna de ellas no es demostrable. Recordemos que esto puede suceder cuando se está definiendo un subtipo, lo cual es básicamente aplicar un predicado sobre el tipo, y si éste se cumple sabemos que el elemento pertenece al subtipo. Esto se puede ilustrar de manera simple con el comando `name` que genera un *subgoal* donde la fórmula `expre = name` es agregada como un nuevo antecedente. Por ejemplo, cuando se aplica el comando `(name "exp1" "car(null)")` se genera una TCC no demostrable ya que `car(null)` trata de acceder al primer elemento de una lista nula.

Otro caso donde se puede romper la demostrabilidad es cuando se utiliza un comando que genera una colección de secuentes en los que alguno de ellos podría no ser demostrable. Recordemos que la regla de *Cut* puede ser entendida como una separación por casos dentro de una demostración de un secuente $\Gamma \vdash \Delta$, generando los *subgoals* $\Gamma, A \vdash \Delta$ y $\Gamma \vdash A, \Delta$. Supongamos que estamos realizando una demostración sobre una lista, y queremos ver que una propiedad α vale para toda lista no vacía l.

$$\begin{array}{ll} (-1) & cons?(l) \\ & \vdash \\ (1) & \alpha(l) \end{array}$$

El antecedente -1 establece la hipótesis de que la lista l contenga al menos un elemento. Podríamos dividir la demostración en dos casos: cuando la lista es vacía y cuando no lo es utilizando el comando `(case "null?(l)")` y terminamos obteniendo estas dos ramas:

$$\begin{array}{ll} (-1) & cons?(l) \\ (-2) & null?(l) \\ & \vdash \\ (1) & \alpha(l) \end{array} \qquad \begin{array}{ll} (-1) & cons?(l) \\ & \vdash \\ (1) & null?(l) \\ (2) & \alpha(l) \end{array}$$

donde es claro ver que la primera se cierra por el axioma de disyunción, ya que $cons?(l)$ y $null?(l)$ no pueden ser ambas ciertas. Por lo tanto se intenta demostrar la segunda rama que tiene dos alternativas: demostrar $\alpha(l)$, lo cual indicaría que la aplicación del comando `case` fue superflua, o demostrar $null?(l)$ lo cual no es posible ya que la hipótesis indica que la lista no está vacía.

Y finalmente el último caso se da cuando se aplica algún comando que modifique estructuralmente el secuente en forma arbitraria como lo son `hide` y `reveal` que permiten ocultar formulas que participan del secuente y revelar fórmulas previamente ocultadas. Estos comandos resultan de gran utilidad cuando enfrentamos demostraciones complejas

con secuentes que involucran gran cantidad de formulas, muchas de las cuales pueden resultar superfluas en la demostración de un *subgoal* específico. Debajo mostramos el efecto de aplicar el comando `hide`:

$$\begin{array}{ll} (-1) & P(a) \\ (-2) & P(b) \\ & \vdash \\ (1) & P(a) \wedge P(b) \end{array}$$

El resultado de la aplicación del comando `(hide -2)` en este secuente se muestra a continuación.

$$\begin{array}{ll} (-1) & P(a) \\ & \vdash \\ (1) & P(a) \wedge P(b) \end{array}$$

Aquí vemos que no se puede lograr la demostración ya que acabamos de esconder una de las hipótesis necesarias para cerrar la rama.

Es importante notar que la noción de punto crítico está íntimamente ligada a los momentos donde la persona que está realizando la demostración aplica su creatividad para poder avanzar. Consecuentemente, éstas son las ocasiones donde la probabilidad de cometer un error es más alta. Así, resulta de suma utilidad la posibilidad de poder comprobar automáticamente si se ha sacrificado la demostrabilidad de la fórmula a causa de un error cometido debido a esa demanda de creatividad por parte del usuario. Uno de los aportes fundamentales de esta tesis es la implementación de un comando de demostración para PVS que realiza este chequeo sobre las fórmulas del secuente actual, tratando de encontrar automáticamente un contraejemplo para ellas.

Adicionalmente, la posibilidad de buscar un modelo de un conjunto de fórmulas, además de proveer un mecanismo vital al momento de atravesar un punto crítico y ser capaz de brindar indicios de que no hemos alterado la demostrabilidad de nuestro secuente, ofrece una variedad de usos posibles de igual o mayor utilidad para el usuario de PVS. Un ejemplo claro de esto es la posibilidad de analizar la consistencia del conjunto de hipótesis frente a la introducción de hipótesis nuevas durante la demostración, que en PVS se consigue mediante la aplicación del comando `case`. Notemos que si la hipótesis introducida se contradice con las hipótesis del secuente, esa rama de la demostración sería trivial y no aportaría ninguna información sobre la propiedad original. A este escenario se suman usos más sutiles, como que el usuario desee realizar verificaciones más específicas. Por ejemplo, ver si una tesis en particular podría seguirse de un cierto subconjunto de las hipótesis o casos similares.

Por lo expresado anteriormente, nuestra propuesta se centra en la implementación de un comando de demostración PVS que puede ser invocado en cualquier punto de la prueba y que ofrece la mayor flexibilidad posible al usuario. El objetivo es que se puedan realizar análisis de existencia de modelos de cualquier subconjunto de fórmulas del secuente. Adicionalmente, se brinda a los usuarios la posibilidad de elegir la polaridad lógica con que cada fórmula se usa en la búsqueda de modelos: positiva (i.e. la fórmula lógica tal como aparece en el secuente) o negativa (i.e. la formula lógica que aparece en el secuente pero negada). Este grado de libertad es necesario debido a la idiosincracia propia de PVS que aplica automáticamente la regla de equivalencia de la negación (ver Fig. 2.5)

de modo tal que todas las fórmulas del secuente tengan polaridad positiva. Esto provoca que fórmulas negadas que forman parte de las hipótesis del problema aparezcan entre los consecuentes del secuente correspondiente. Equivalentemente, las tesis que originalmente son negaciones, aparecen con polaridad positiva entre los antecedentes.

En la sección 1 discutimos sobre la sinergia entre los métodos *lightweight* y *heavyweight*. Ahora queremos profundizar el tema y proponer un enfoque metodológico para el uso de nuestra herramienta durante una demostración. Ilustramos con el autómata de la figura 3.1 los posibles estados y transiciones. Cada nodo se divide en una parte izquierda y una derecha. La parte izquierda indica la situación respecto al conjunto de hipótesis, mientras que la derecha lo hace respecto al secuente en su totalidad. En la tabla 3.1 agregamos información sobre cómo interpretar los posibles estados del autómata y en la tabla 3.2 cuáles son las transiciones posibles.

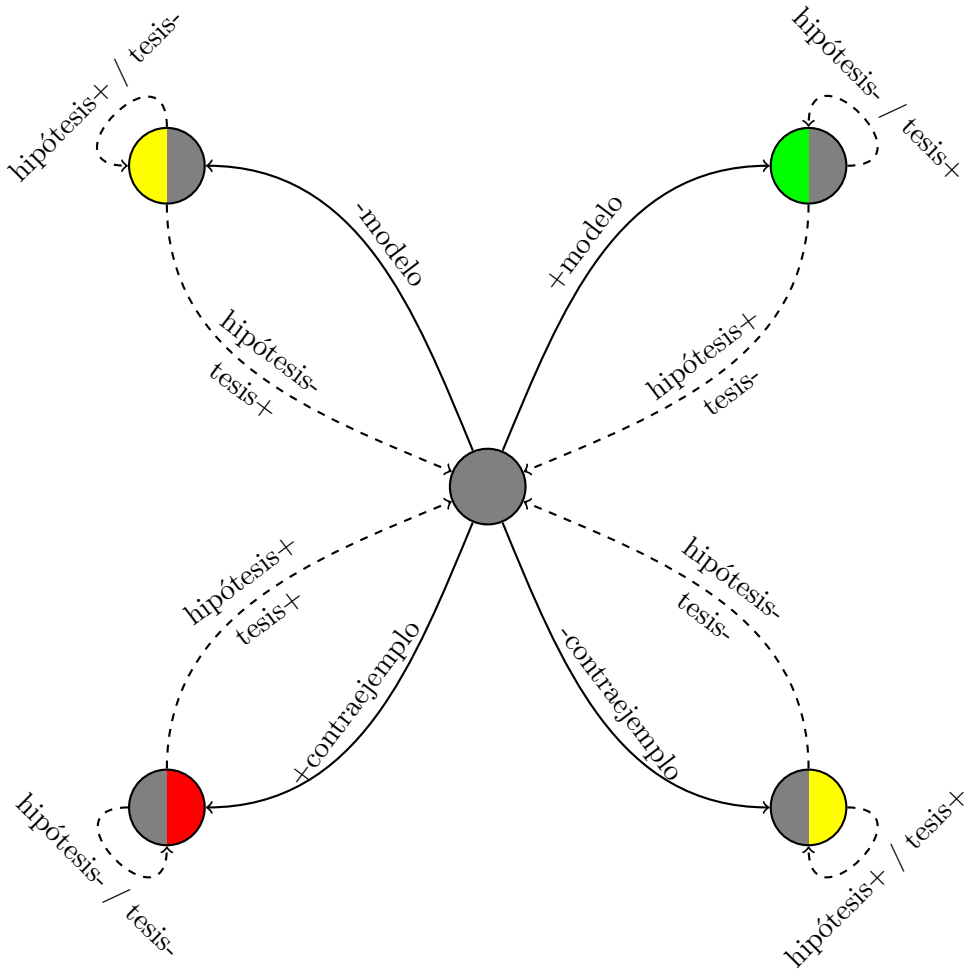

Fig. 3.1: Autómata representando demostración

Además, las flechas que indican transiciones pueden ser o no punteadas. La diferencia radica en que las flechas puntuadas además indican que se pasa a tener un nuevo secuente (ya que se agregan o quitan hipótesis o tesis, y por lo tanto se modifica el coloreo), mientras que las otras modifican el conocimiento que tenemos acerca de la validez o no de nuestro

secuente. Algunas de las transiciones no tienen mucho sentido, como cuando se agregan o quitan tesis, pero siguen siendo operaciones posibles por parte del usuario mediante los comandos de `hide` o `reveal`.

Supongamos que nos encontramos en el estado que tiene la parte izquierda de color verde y la derecha de color gris. Cuando pasamos a agregar una hipótesis, pasamos a desconocer la validez del modelo que teníamos para las hipótesis. Es por eso que se pasa al nodo gris, indicando que no tenemos información sobre la consistencia del conjunto de hipótesis. De manera análoga se pueden interpretar el resto de las transiciones puntuadas.

Si estamos en el nodo gris, eso indica que no hay información alguna respecto a la consistencia de las hipótesis o contraejemplo para el secuente. Por lo tanto es necesario usar nuestra herramienta para alguna de las dos cosas: encontrar un modelo (para las tesis) o un contraejemplo (para el secuente). Si se busca un modelo se pasa a estar en alguno de los nodos superiores, mientras que si se busca un contraejemplo se pasa a estar en alguno de los inferiores.

	◖	◗
	Sin información	Sin información
	Con modelo para hipótesis	-
	Sin modelo para hipótesis	Sin contraejemplo para el secuente
	-	Con contraejemplo para el secuente

Tab. 3.1: Posibles estados.

	+	-
hipótesis	Se agrega una hipótesis	Se esconde una hipótesis
tesis	Se agrega una tesis	Se esconde una tesis
modelo	Se encuentra un modelo para las hipótesis	No se encuentra un modelo para las hipótesis
contraejemplo	Se encuentra contraejemplo para el secuente	No se encuentra contraejemplo para el secuente

Tab. 3.2: Transiciones.

En PVS no es posible saber si una fórmula del secuente es parte de las hipótesis o de las tesis. Primero, porque PVS no lleva una trazabilidad de eso a medida que avanza la demostración, pero además el usuario puede darle a la fórmula la interpretación que quiera; por lo que no es un problema que se pueda resolver puramente desde el aspecto técnico. Asumamos por un momento que sí tenemos una forma de saber si una fórmula originalmente pertenecía al conjunto de hipótesis o al conjunto de tesis dentro del secuente. Se podría lograr una herramienta aún más útil que la que desarrollamos actualmente: tener estrategias específicas para algunos momentos puntuales de la demostración. La idea básica sería tratar de buscar un modelo o un contraejemplo teniendo en cuenta las nuevas hipótesis o tesis que tendría el secuente; pero recordemos que no encontrarlo (ya sea un modelo o contraejemplo, según corresponda) no da una certeza absoluta: quizás el universo de búsqueda no fue el adecuado y por eso el modelo/contraejemplo no fue encontrado.

Para todos los casos supongamos que contamos con un secuente de la forma $\Gamma \vdash \Delta$. Las potenciales estrategias podrían ser: instanciación, separación en casos, lemma *in-line*, esconder y revelar fórmulas.

Instanciación

Cuando usamos algún comando PVS para instanciar variables que estén cuantificadas universalmente en el antecedente o existencialmente en el consecuente, los cuales mencionamos al principio del capítulo, existiría la posibilidad hacer una verificación automática. Podríamos contar con una estrategia que trata de encontrar:

- modelo si se instanciaba una fórmula del conjunto de hipótesis
- un contraejemplo si se instancia una fórmula del conjunto de hipótesis

Separación en casos

Cuando se utiliza el comando `(case` α`)` para la separación en casos, el usuario tiene la intención de bifurcar la rama actual en los siguientes dos secuentes:

$$\begin{array}{ll} (-1) & \Gamma \\ (-2) & \alpha \\ & \vdash \\ (1) & \Delta \end{array} \qquad\qquad \begin{array}{ll} (-1) & \Gamma \\ (-2) & \neg\alpha \\ & \vdash \\ (1) & \Delta \end{array}$$

De todos modos, recordemos una vez más que PVS aplica automáticamente la regla de la negación por lo que en el secuente de la derecha α aparece en positivo como un consecuente. En un caso como este, podríamos tener una estrategia que verifique si el nuevo conjunto de antecedentes de cada rama, es decir $\Gamma \cup \{\alpha\}$ y $\Gamma \cup \{\neg\alpha\}$, son consistentes, lo que se puede comprobar encontrado un modelo para el secuente.

Lemma *in-line*

En esta ocasión queremos utilizar el comando `(case` α`)` para introducir una hipótesis nueva, por lo tanto, al aplicarlo pasamos a tener los siguientes secuentes.

$$\begin{array}{ll} (-1) & \Gamma \\ (-2) & \alpha \\ & \vdash \\ (1) & \Delta \end{array} \qquad\qquad \begin{array}{ll} (-1) & \Gamma \\ & \vdash \\ (1) & \alpha \\ (2) & \Delta \end{array}$$

En la rama de la izquierda tenemos que verificar que se mantenga la consistencia, por lo tanto lo que hay que buscar es un modelo. Para la rama de la derecha lo que podemos hacer es buscar un contraejemplo para la tesis α que fue introducida.

Esconder o revelar fórmulas

Al momento de esconder una fórmula (utilizando el comando `hide`) tenemos dos opciones: que sea parte de las hipótesis o de las tesis. Si lo que hacemos es esconder una tesis, no hay mucho más para hacer, ya que la demostrabilidad se mantiene intacta. Pero si se trata de una hipótesis, es un caso más interesante para analizar. Al esconder una fórmula que es parte del conjunto de hipótesis, podríamos estar rompiendo la demostrabilidad, y

que alguna de las tesis pase a no ser demostrable, por lo que buscar un contraejemplo sería de utilidad. Para esto podríamos tener una estrategia que recibe una lista de fórmulas a esconder y luego busca un contraejemplo.

Cuando usamos el comando `reveal` lo que sucede es que se vuelve a mostrar una fórmula que había sido escondida previamente, lo cual sería análogo a agregar una fórmula nueva. Cuando esto pasa tenemos dos alternativas: que la fórmula sea una hipótesis o una tesis. Por lo que podríamos tener una estrategia para asistir en estos casos. Para cuando se trata de una hipótesis, vamos a buscar un modelo que satisfaga todo el conjunto de hipótesis. Pero si se trata de una tesis lo que debemos buscar es un contraejemplo.

A continuación mostraremos con unos ejemplos cómo se utiliza nuestra estrategia y cómo se relaciona con el autómata de la figura 3.1.

3.1. Ejemplos de uso

Veamos ahora de manera un poco más concreta cómo se puede usar nuestra herramienta. Supongamos que tenemos la siguiente especificación PVS para el Principio del Palomar (que indica que n palomas no pueden ser colocadas en $n-1$ huecos):

```
pigeonhole: THEORY
BEGIN

  Pigeon: TYPE+
  Hole: TYPE+

  pigeonhole: [Pigeon -> Hole]

  pigeonhole_is_injective: AXIOM
    FORALL(h: Hole):
      EXISTS(p : Pigeon) :
        pigeonhole(p)=h

  lemma_to_prove: LEMMA
    FORALL(p1,p2: Pigeon):
      p1/=p2 IMPLIES pigeonhole(p1) /= pigeonhole(p2)

END pigeonhole
```

Queremos demostrar que el lema es válido, pero primero vamos a chequear si nuestra herramienta encuentra algún contraejemplo. Entonces iniciamos la demostración en PVS y tenemos que invocar la estrategia `find-model`:

```
  |-------
{1}   FORALL (p1, p2: Pigeon):
        p1 /= p2 IMPLIES pigeonhole(p1) /= pigeonhole(p2)

Rule? (find-model 1 :model-as-pvs? t :verbose? t)
```

Expliquemos un poco mejor de qué manera se está aplicando la estrategia [1]:

- el número de fórmula (`1` en el ejemplo) indica el subconjunto de las fórmulas del secuente sobre las que va a trabajar la estrategia,

[1] En la subsección 3.4 explicaremos con más detalles todos los parámetros de la estrategia `find-model`.

- el parámetro `model-as-pvs?` con valor `t` significa que en caso de encontrarse un modelo, se mostrará con la nomenclatura de PVS,
- el modo `verbose` en `t` sirve para mostrar más información en el *output*, ya que de otra forma solamente se indica si se encontró o no modelo pero no se incluye información respecto a ella.

Entonces el *output*[2] es el siguiente:

```
Rule? (find-model 1 :model-as-pvs? t :verbose? t)
Searching for counterexample...
Running Kodkod...
[...]
Counterexample found:

pigeonhole(?A1) = pigeonhole(?A0)
pigeonhole(?A2) = pigeonhole(?A0)
pigeonhole(?A3) = pigeonhole(?A0)
pigeonhole(?A4) = pigeonhole(?A0)
```

Lo que podemos ver es que el hueco asignado para la paloma `A1` es el mismo que el asignado para la `A0` (y lo mismo se repite para `A2`, `A3` y `A4`), contradiciendo lo que esperábamos. Por lo tanto la situación que tenemos es la de la figura 3.2, donde partimos sin tener información, para luego pasar a un estado donde indica que tenemos un contraejemplo.

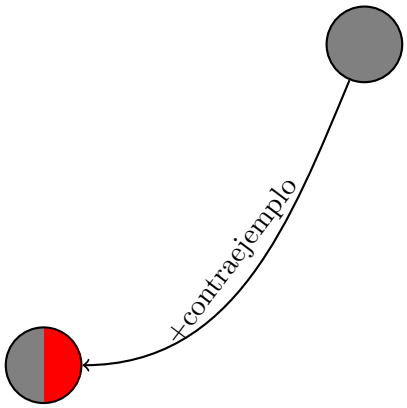

Fig. 3.2: Búsqueda de contraejemplo

En este punto nos podemos dar cuenta que hay un error. Revisando la especificación podemos notar que el axioma que fuerza la inyectividad está mal escrito (de hecho, está forzando sobreyectividad). Corregimos ese error y la especificación queda de la siguiente manera.

```
1 pigeonhole: THEORY
2 BEGIN
3
4   Pigeon: TYPE+
5   Hole: TYPE+
```

[2] En realidad se muestra lo que es relevante, erp en la práctica también se ve más información referente al funcionamiento interno de Kodkod, o algunas estadísiticas como el tiempo de parseo o de resolución del SAT-solver

```
6
7   pigeonhole: [Pigeon -> Hole]
8
9   pigeonhole_is_injective: AXIOM
10    FORALL(p1,p2: Pigeon):
11      pigeonhole(p1)=pigeonhole(p2) IMPLIES p1=p2
12
13  % This lemma is not valid if the axiom is commented out
14  lemma_to_prove: LEMMA
15    FORALL(p1,p2: Pigeon):
16        p1/=p2 IMPLIES pigeonhole(p1) /= pigeonhole(p2)
17
18 END pigeonhole
```

Reescribir la definición de un axioma es equivalente a quitar una hipótesis y agregar una nueva, lo cual representado en el autómata sería lo que se muestra en la figura 3.3:

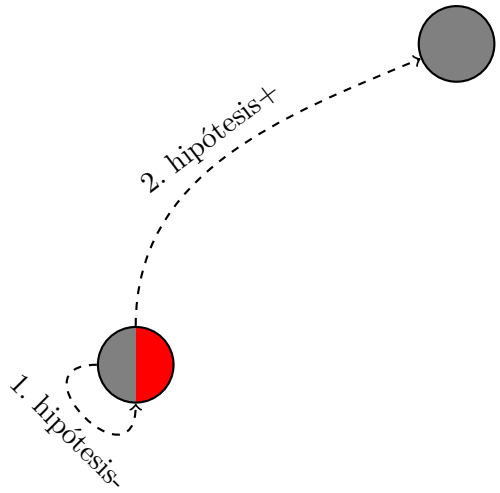

Fig. 3.3: Autómata representando demostración

Por lo tanto estamos nuevamente en un estado donde no sabemos si tenemos modelo o contraejemplo (nodo gris). Utilizamos nuevamente nuestra herramienta para que nos ayude a comprender mejor la situación, lo cual se representaría como lo muestra la 3.4.

```
Rule? (find-model 1 :model-as-pvs? t :verbose? t)
Searching for counterexample...
Running Kodkod...
[...]
Counterexample not found.
```

A continuación mostraremos cómo se realiza la traducción del universo de PVS al de Kodkod.

3.2. Traducción

Dada una especificación PVS queremos demostrar que una propiedad vale. Hay que evitar desperdiciar tiempo intentando hacer una demostración sobre algo que es falso, para esto utilizamos un buscador de modelos para que nos ayude a dirimir más rápidamente

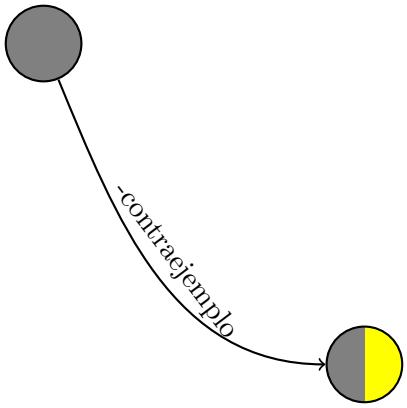

Fig. 3.4: Autómata representando demostración

si tiene sentido o no lo que estamos tratando de hacer. Para poder usar este buscador de modelos, que en nuestro caso es Kodkod, partimos del universo de PVS, aplicamos una traducción y obtenemos elementos de Kodkod los cuales son el *input* para el buscador de modelos.

Este es quizás el punto más importante de esta tesis. En esta sección se explica cómo se logra transformar especificaciones escritas en el lenguaje de PVS en fórmulas en el lenguaje de Kodkod. Explicado muy brevemente, lo que se hace es vincular tipos de datos con conjuntos de átomos y se sintetizan los axiomas y fórmulas del secuente en una sola fórmula Kodkod. Básicamente, un problema Kodkod se compone de las siguientes partes:

- un universo acotado,
- relaciones y conjuntos de átomos y
- una fórmula

También hay que agregar axiomas adicionales que son requeridos por ciertas estructuras de datos (e.g. listas) o fórmulas para forzar que ciertas relaciones se comporten como funciones. Estos detalles se explican más adelante.

Como el principal parámetro de entrada para Kodkod es una fórmula, lo primero que vamos a hacer es obtener una sola fórmula PVS que sintetice la información que tenemos en los axiomas y el secuente. Como hemos explicado en secciones anteriores, la búsqueda de contraejemplos puede no abarcar todas las fórmulas del secuente. Además, la polaridad con la que ellas aparecen puede cambiar debido a que la aplicación automática de la regla de la negación puede hacer aparecer una hipótesis entre los consecuentes y una tesis entre los antecedentes. Por esto, el primer paso de la traducción, que representaremos con el símbolo ϕ_S, toma como parámetros la lista de axiomas, el secuente y un parámetro extra que sirve para filtrar las fórmulas del secuente que participan de la traducción así como la polaridad con la que deben aparecer en el resultado.

$$\phi_S(\Psi, \Gamma \vdash \Delta, \langle \mathcal{I}_\Gamma, \mathcal{I}_\Delta \rangle) = \bigwedge_{\psi \in \Psi} \psi \wedge \bigwedge_{\gamma \in \Gamma|_{\mathcal{I}_\Gamma}} \gamma \wedge \bigwedge_{\delta \in \Delta|_{\mathcal{I}_\Delta}} \delta$$

donde $\mathcal{I}_\Gamma$ (y correspondientemente $\mathcal{I}_\Delta$) es un conjunto de números enteros cuyos módulos

están entre 0 y el tamaño de Γ (Δ) y donde, si A es una secuencia de fórmulas, $A|_{\mathcal{I}} = \{\gamma|(\exists i \in \mathcal{I}) :$ si $i \geq 0$, entonces $\gamma = A_i$ y, si no, $\neg\gamma = A_{-i}\}$.

El siguiente paso es *saltar* del mundo PVS al mundo Kodkod. Para ello separamos la traducción en dos: la traducción de tipos de datos no interpretados de PVS en conjuntos de átomos Kodkod, representada por la función τ_d, y la traducción de fórmulas PVS en fórmulas Kodkod, representada por la función τ_F.

Primero veamos lo que sucede con los tipos. Dado un tipo de datos T que aparece referenciado en alguno de los axiomas (Ψ) o en el secuente ($\Gamma \vdash \Delta$), la función de traducción τ_d se define de la siguiente manera.

$$\tau_d(T) = S_T$$

donde $S_T = \{a_1, \cdots, a_{\sigma(T)}\}$ es un conjunto de átomos y σ denota una función que indica, por cada tipo de dato T, el tamaño máximo para el conjunto de átomos correspondiente (S_T) que admitirán los modelos explorados por Kodkod. Naturalmente, f es un parámetro más de la traducción. Esto básicamente indica que para cada tipo no interpretado en PVS vamos a tener un conjunto de átomos en Kodkod que lo represente.

Lo que sucede con las funciones es un poco más complejo. La idea básica es que si tenemos una función que toma n parámetros, vamos a representarla en Kodkod como una relación $n+1$-aria, donde la coordenada extra representa al resultado de la función. A continuación vamos a formalizar esta noción. Sea S_{T_1} el conjunto de átomos en Kodkod que representa al tipo T_1 en PVS, S_{T_2} el que representa a T_2 y así sucesivamente. Si tenemos en PVS una función $f : T_1 \times \cdots \times T_n \to T_{n+1}$, la misma sería representada en Kodkod por una relación $r_f \subseteq S_{T_1} \times \cdots \times S_{T_n} \times S_{T_{n+1}}$.

Por último, nos queda detallar cómo es la la traducción de fórmulas τ_F, la cual transforma una fórmula PVS en una Kodkod. Para ello definimos $\tau_F : Form_{PVS} \longrightarrow Form_{Kodkod}$ de la siguiente forma:

$$\begin{aligned}
\tau_F(E = E) &:= \tau_F(E) = \tau_F(E) \\
\tau_F(f(E_1, \cdots, E_n)) &:= r_f(\tau_F(E_1), \cdots, \tau_F(E_n)) \\
\tau_F(\neg E) &:= \neg\tau_F(E) \\
\tau_F(E \wedge E) &:= \tau_F(E) \wedge \tau_F(E) \\
\tau_F(\forall(v : T) : E) &:= \forall(one\ v : S_T) : \tau_F(E) \\
\tau_F(\exists(v : T) : E) &:= \exists(one\ v : S_T) : \tau_F(E)
\end{aligned}$$

donde f es un símbolo de función, r_f es la relación en Kodkod que denota la función f de PVS como se explicó anteriormente y v es un símbolo de variable. Se muestra la definición de la traducción de fórmulas solamente sobre un conjunto mínimo de conectivos para sintetizar la explicación pero es linealmente extensible a otros elementos sintácticos como conectivos lógicos redundantes y otras facilidades sintácticas como expresiones condicionales *if-then-else*.

Ahora que fue explicada la traducción para fórmulas, a continuación detallaremos la traducción para un tipo abstracto de dato (TAD).

Traducción para un TAD

Cuando se introduce un nuevo TAD en PVS se genera automáticamente, durante el proceso de *typechecking*, una teoría que contiene declaraciones, definiciones y axiomas. Dado que no todos estos elementos son relevantes para el trabajo presentado en esta tesis, vamos a listarlos y aclarar cuáles tenemos en cuenta y cuáles no. Para ilustrar mejor esto, vamos trabajar con el ejemplo del tipo lista. La declaración sería de la siguiente forma:

```
list[t:TYPE] : DATATYPE
BEGIN
    null: null?
    cons(car: t, cdr: list) : cons?
END list
```

El tipo `list` es declarado como un tipo paramétrico para el tipo `t` con dos constructores: `null` y `cons`. El constructor `null` no toma argumentos. El predicado (*recognizer*) `null?` identifica a los elementos del tipo `list` que son `null`. El constructor `cons` toma dos argumentos donde el primero es del tipo `t` y el segundo es del tipo `list`. El predicado `cons?` es verdadero para las listas que son construidas utilizando `cons`, es decir, los listas que no son `null`. Hay dos *accessors* que corresponden a los dos argumentos de `cons`. Los *accessors* `car` y `cdr` pueden ser aplicados solo a listas que satisfacen el predicado `cons?` de manera tal que `car(cons(x, l))` es `x` y `cdr(cons(x, l))` es `l`. Notar que `car(null)` no es una expresión bien tipada ya que genera una obligación de demostración (TCC) inválida, pidiendo que `cons?(null)` sea verdadero.

PVS crea un axioma de *extensionalidad* para cada constructor constante (i.e., sin argumentos), que en el caso de la lista sería:

```
list_null_extensionality: AXIOM
    FORALL (null?_var: (null?), null?_var2: (null?)):
        null?_var = null?_var2;
```

Este axioma indica que cualquier instancia generada por un constructor constante representa el mismo elemento. La extensionalidad para los constructores con argumentos indica que los elementos son distinguibles por los *accessors*. En el caso de la lista sería:

```
list_cons_extensionality: AXIOM
    FORALL (cons?_var: (cons?), cons?_var2: (cons?)):
        car(cons?_var) = car(cons?_var2) AND cdr(cons?_var) = cdr(cons?
    _var2)
        IMPLIES cons?_var = cons?_var2;
```

El axioma eta indica que los valores obtenidos por los *accessors* para construir una nueva instancia resulta en la misma instancia. Con el tipo `list` sería:

```
list_cons_eta: AXIOM
    FORALL (cons?_var: (cons?)):
        cons(car(cons?_var), cdr(cons?_var)) = cons?_var
```

También se generan axiomas para el significado de los *accessors*, que en este caso sería uno para `car` y otro para `cdr`:

```
list_car_cons: AXIOM
    FORALL (cons1_var: T, cons2_var: list):
      car(cons(cons1_var, cons2_var)) = cons1_var;
```

```
4
5 list_cdr_cons: AXIOM
6     FORALL (cons1_var: T, cons2_var: list):
7       cdr(cons(cons1_var, cons2_var)) = cons2_var;
```

Es necesario que los *recognizers* sean capaces de caracterizar todos los elementos del tipo, para eso sería útil tener un axioma que indique todos los elementos están generados por los constructores (inclusión) y otro que indique los elementos generados por constructores distintos son disjuntos (disyunción). La teoría generada por el *typechecker* producía estos dos axiomas. Pero dado que la cantidad de fórmulas a generar para el axioma de disyunción es $\binom{n}{2}$ para un tipo que tiene n constructores, se dejó de utilizar puesto que esta propiedad se puede deducir lógicamente de la existencia de un orden bien fundado sobre los elementos del tipo a partir de mapearlos a los números naturales, y el hecho de que estos últimos son distintos. Como en nuestro caso no somos capaces de utilizar números naturales, no podemos utilizar este recurso y, consecuentemente, tenemos que generar el axioma de disyunción, aún a pesar del elevado costo mencionado anteriormente. Para el tipo lista los axiomas serían:

```
1 list_inclusive: AXIOM
2     FORALL (list_var: list): null?(list_var) OR cons?(list_var);
3
4 list_disjointness: AXIOM
5     FORALL (list_var: list): NOT(null?(list_var) AND cons?(list_var));
```

Para cada parámetro de tipo, se generan los operadores de alto orden *every* y *some*. Para el tipo `list` son:

```
1 every(p: PRED[T], a: list):  boolean =
2     CASES a
3         OF null: TRUE,
4             cons(cons1_var, cons2_var): p(cons1_var) AND every(p, cons2_var
    )
5     ENDCASES;
6
7 some(p: PRED[T], a: list):  boolean =
8     CASES a
9         OF null: FALSE,
10            cons(cons1_var, cons2_var): p(cons1_var) OR some(p, cons2_var)
11    ENDCASES;
```

Para *every* y *some* lo que se hace es traducirlos cuando existe una aplicación concreta en algún predicado, dando como resultado una definición de primer orden.

Aspectos ignorados por la traducción de TADs. Como mencionamos antes, existen algunos axiomas y definiciones que si bien son generados automáticamente durante el *typechecking* de un TAD, no resultan de utilidad durante el proceso de búsqueda de modelos y, por lo tanto, no son traducidos a Kodkod. Estos elementos son: *(1)* el esquema de inducción, *(2)* las funciones *reduce* (permiten asignar una medida a los términos, son generalmente utilizadas para probar la terminación de definiciones recursivas creadas por el usuario de la teoría), *(3)* la definición de subtérmino, y *(4)* axioma de buena fundación.

Párrafo aparte merece el axioma de buena fundación ya que su impacto no solo está en el terreno de lo estrictamente práctico, sino que también establece propiedades

metateóricas que limitan la aplicabilidad de la técnica.

Teorías paramétricas

Uno de los recursos sintácticos más apreciados en el desarrollo de software en todas sus etapas es el uso de definiciones de tipos o algoritmos en forma paramétrica. Estos son utilizados de manera que definen una familia estos cuyo comportamiento resulta relativamente independiente de la instanciación particular de dichos parámetros. Hay una enorme variedad de ejemplos de la utilización de este recurso; así como el comportamiento de una lista no se modifica al cambiar la naturaleza de los elementos que esta contiene, el de un algoritmo de ordenamiento tampoco lo hace, siempre y cuando el tipo de los elementos contenidos es la estructura posea una relación de orden total.

Existe una gran cantidad de trabajo realizado en lo que refiere a los aspectos lógicos que subyacen a la formalización de teorías paramétricas y los mecanismos disponibles para describir sus comportamientos. El área de investigación suele conocerse con el nombre de *Generic programming* y algunos de los trabajos de mayor relevancia en el área son [19, 20, 21, 22]. En el terreno de orden más práctico, todos los lenguajes de especificación, diseño y desarrollo cuentan con sintaxis específica que posibilita la definición de estructuras paramétricas en tipos, o incluso funciones. PVS no es una excepción y en su lenguaje de especificación pueden definirse este tipo de teorías. La sintaxis fue mostrada en forma parcial en la sección 2 y para una presentación más completa el lector es remitido a [23].

Si bien no nos detendremos en una presentación extensa sobre los beneficios de la utilización de este tipo de construcciones paramétricas, sí queremos mencionar que más allá de los ampliamente discutidos, y saludables, criterios de desarrollo que apuntan hacia la producción de componentes en forma modular y reutilizable, no se produce ningún incremento del poder expresivo en el lenguaje lógico. La razón detrás de esto es que las teorías paramétricas son recursos sintácticos que, una vez procesadas por el *type checker* de PVS, dan como resultado teorías cerradas producto del reemplazo de los parámetros formales de la teoría paramétrica, por lo los valores actuales sobre los que dicha teoría es utilizada, dando lugar a una teoría no paramétrica que resulta traducible a través de las funciones presentadas en los párrafos precedentes.

Debemos hacer una salvedad a lo dicho anteriormente; si bien las teorías paramétricas no representan ningún impedimento para la traducción que hemos implementado, esto solo vale en tanto los parámetros actuales sobre los que se instancien los parámetros formales sean de primer orden, es decir, que sean traducibles por la traducción implementada, ya que como mencionaremos más adelante, por el momento no hemos implementado la generación de dominios de alto orden.

3.3. Algunas limitaciones de nuestra herramienta

En esta sección, presentaremos algunas limitaciones de la implementación alcanzada y sugeriremos caminos posibles que permitan la extensión de la herramienta hacia los aspectos de interés que se mencionan.

Tipos Interpretados

Nuestra herramienta no contempla la traducción de tipos de datos interpretados, por ejemplo número de distinta naturaleza como son los enteros, los racionales o los reales. Esto es algo necesario para muchas ramas de la matemática, ingeniería y las ciencias en general: hacer demostraciones que involucran propiedades numéricas como ser desigualdades, uso de logaritmos, funciones trigonométricas como senos, cosenos y demás.

A continuación describimos algunos trabajos relacionados con el objeto de estudio de esta tesis que la complementan. En primer lugar, cabe destacar a MetiTarski [24] que es básicamente un demostrador de teoremas (Metis [25]) que fue modificado para invocar un procedimiento de decisión (QEPCAD [26]) para la teoría de cuerpos reales cerrados (RCF, de su nombre en inglés *real closed fields*), que resulta isomorfa a la teoría de primer orden de los números reales. La teoría de los RCF se centra en las igualdades y desigualdades de la adición, substracción y multiplicación. Por *real closed* se refiere a que todo número positivo tiene una raíz cuadrada. El procedimiento de decisión trabaja eliminando los cuantificadores de la fórmula suministrada; por ejemplo, $\exists x.ax^2 + bx + c = 0$ se reduce a

$$(a \neq 0 \wedge b^2 - 4ac \geq 0) \vee (a = 0 \wedge b \neq 0) \vee (a = b = c = 0).$$

Tarski probó que RCF es decidible en 1930, pero su procedimiento era impracticable, es por eso que se decide utilizar QEPCAD-B [27] que es una implementación avanzada de la descomposición algebraica cilíndrica (CAD por sus siglas en inglés), que es el mejor procedimiento de decisión disponible para la teoría completa de RCF.

El procedimiento de decisión implementado por MetiTarski simplifica las sentencias eliminando literales que son inconsistentes con otros hechos algebraicos, mientras que también elimina sentencias redundantes que se pueden deducir algebraicamente de otras, principalmente a través de reemplazar funciones por límites superiores e inferiores polinomiales y fue diseñado para probar desigualdades cuantificadas universalmente. Así, MetiTarski sirve como oráculo: puede indicar qué rama del árbol de demostración se puede descartar por no ser demostrable.

Quickcheck [**?**] es una herramienta que da soporte para formular y testear propiedades de programas en Haskell. Trabaja básicamente generando instancias random para luego chequear si las propiedades se cumplen o no en el programa. Esto sirvió de inspiración para crear *Random Testing* en PVS [28], que usa el mismo principio. Este enfoque sirve para atacar el problema de especificaciones mixtas (especificaciones que incluyen expresiones lógicas y numéricas). La idea básica es generar instancias *random* para los distintos tipos de datos, y la pieza fundamental de esto es el *random test generator*. Para cada tipo en PVS, un *random test generator* es creado para generar valores; cada invocación a este generador va producir un valor aleatorio para el tipo asociado. Los valores aleatorios son usados para instanciar una fórmula dada que luego es evaluada. Si se encuentra que es falsa, los valores del test son reportados, a modo de contraejemplo.

Para los tipos básicos los valores son generados usando la función `random` de Common Lisp. Los booleanos, tipos enumerados, etc. pueden ser manejados directamente con este enfoque. Para los tipos de datos básicos que no son acotados, como los enteros, una cota puede ser dada; por default es 100. Para los racionales se generan dos enteros random: el nominador y el denominador. Los racionales son también usados para generar reales aleatorios.

Los *random generators* para los subtipos presentan complicaciones. Actualmente se genera un valor para el supertipo, y luego se chequea si el resultado está en el subtipo evaluando en ese valor el predicado que especifica dicho subtipo. Si no es exitoso, se continúa haciendo esto hasta encontrar un valor adecuado o alcanzar un *timeout*. El buen funcionamiento de esta estrategia dependerá de cuán probable es que un valor aleatorio satisfaga el predicado del subtipo y cuán costoso, computacionalmente hablando, sea calcularlo. Para el caso de las funciones se memoriza si el argumento ya había sido visto, en caso positivo se devuelve el valor asociado. Pero si no, se genera uno aleatoriamente dentro de los valores de la imagen y se guarda el par para futuras invocaciones. Para los tipos de datos inductivos, básicamente hay un parámetro de tamaño utilizado para construir los elementos del tipo de dato para limitar las construcciones utilizando dicho parámetro como cota. Entonces si el tamaño es 4, listas de tamaño como máximo 4 pueden ser generadas (incluyendo `null`), y árboles con altura como máximo 4. Una lista de árboles también puede ser construida, donde la lista tiene como máximo 4 elementos, y cada elemento del árbol tiene como altura máxima 4.

Tipos (Co)Inductivos

Existen tipos de datos que modelan objetos infinitos, como puede ser el *stream* de datos que genera un sensor. A estos tipos de datos se los llama coinductivos. Plasmar esto utilizando lógica de alto orden no es problema, ya que el poder expresivo lo permite. En cambio, modelarlo es un poco más complejo. Se dice que dos objetos coinductivos son equivalentes si y sólo si conducen a las mismas observaciones, es decir que si se modelara con una máquina de estados, la secuencia de transiciones es la misma. A este concepto se lo llama bisimulación y se define coinductivamente. Nuestra herramienta aún no ofrece soporte para esta clase de tipos de datos y ciertamente deseamos poder proveer soporte para ellos.

En [12] los autores detallan el procedimiento por el cual Nitpick implementa una traducción de los tipos coinductivos a través de una formalización axiomática análoga a la de los tipos inductivos, pero evitando el uso de los correspondientes axiomas de aciclicidad de las estructuras para posibilitar una subclase de elementos de naturaleza infinita pero cuya representación resulta finita mediante ciclos; adicionalmente la traducción producirá predicados correspondientes a los principios de coinducción a fin de garantizar que la equivalencia es interpretada como una relación de bisimulación.

Cuantificación de Alto Orden

Otro de los elementos, siendo este de gran importancia, que no ha sido contemplado en nuestra traducción y que, por lo tanto, no se encuentra soportado por nuestra herramienta de búsqueda de contraejemplos, es la posibilidad de generar valores pertenecientes a dominios arbitrarios de alto orden. La lógica de alto orden de PVS resulta equipolente[3] a la presentación de la lógica de alto orden sobre tipos estructurados para funciones de Van Benthem [30].

La noción de modelo, y la consecuente definición de satisfacción de una fórmula por

[3] En el presente trabajo el término equipolente es utilizado con el sentido de equivalencia lógica que le da Tarski en [29, pp. 121].

un modelo para esta lógica, debe ser construida sobre dominios capaces de interpretar términos de todo posible orden; para hacer esto debe considerarse la superestructura de conjuntos $S^{\#} = \bigcup_n V^n(S)$ donde $V^1(S) = S$ y $V^{n+1}(S) = V^n(S) \cup \mathcal{P}(V^n(S))$, que a partir de los conjuntos en S, que ofrecen la posibilidad de interpretar símbolos de primer orden, proporciona soporte para los dominios de interpretación de orden superior.

Subtipos

Respecto a los subtipos, el soporte de nuestra es limitado. Solamente podemos indicar que un subtipo se encuentra dentro de otro tipo. *Predicate subtyping* es otra forma de especificar subtipos. Se trata de usar predicados para determinar la pertenencia o no del tipo. Actualmente no está soportado por nuestra herramienta, pero si solo consideráramos los tipos no interpretados y predicados que utilicen características soportadas actualmente, es una funcionalidad que se podría agregar sin mayores dificultades teóricas.

Búsqueda de contraejemplos y modelos infinitos

Como mencionamos anteriormente, la cuestión de los modelos infinitos merece un comentario aparte. Una situación posible cuando intentamos demostrar una propiedad es que los axiomas involucrados en el secuente en cuestión fuercen a que todo modelo sea necesariamente infinito. Ante esto, no hay nada que nuestra propuesta pueda ofrecer ya que no importa cuánto tiempo invirtamos en la construcción de un modelo, no podremos satisfacer las fórmulas que lo fuerzan a ser infinito. Así, la aplicabilidad de la técnica propuesta recae exclusivamente sobre la posibilidad que tengamos de modificar esta teoría para que admita modelos finitos; naturalmente, esta modificación implicará la eliminación de aquellos axiomas que, de una u otra forma, excluyen dichos modelos.

Si adoptamos este enfoque, el haber eliminado axiomas puede provocar que si bien encontremos un modelo de aquellos que mantuvimos y que resulte ser contraejemplo de la propiedad de interés, dicho modelo podría ser espurio pues no resulta extensible a un modelo que efectivamente satisfaga todos los axiomas de la teoría. De esta forma, será responsabilidad del usuario el analizar el modelo minuciosamente para determinar si éste refleja información de utilidad en relación a si la totalidad de los axiomas permiten demostrar la propiedad o no.

Si observamos lo dicho acerca de los aspectos ignorados en la traducción de TADs, es momento de retomar la consideración hecha al margen sobre el axioma de buena fundación. Cada vez que declaramos un TAD en PVS, al realizar el *typechecking*, PVS automáticamente genera un axioma que dice que hay una relación de orden estricta entre un término del TAD (i.e., una cadena finita de aplicaciones de los constructores) y sus subtérminos. Es fácil ver que este solo hecho obliga a que los modelos de las teorías generadas los TADs solo tengan modelos infinitos. Pensemos esto detenidamente, si un TAD admitiera un modelo finito, entonces bastaría con tomar un término de tamaño, al menos, uno más que la cardinalidad del conjunto del modelo para que tanto el término como alguno de sus subtérminos sean interpretados como el mismo objeto, violando el axioma de buena fundación.

Veamos un ejemplo de esta situación. Supongamos que modelamos los números naturales en PVS de la siguiente manera

```
Nat : BEGIN DATATYPE
    zero: zero?
    succ(n: Nat): notZero?
END Nat
```

y además tenemos la función `greaterThan` definida como:

```
greaterThan(n1 : Nat, n2 : Nat): RECURSIVE bool
= IF isZero?(n1) THEN FALSE
  ELSIF isZero?(n2)  THEN TRUE
  ELSE greaterThan(prev(n1), prev(n2))
  ENDIF
MEASURE n1 BY <<
```

que indica si `n1` es mayor a `n2`. El lema mostrado a continuación, donde indicamos que para todo `Nat` siempre hay uno mayor, es trivialmente cierto.

```
lemma_biggest: LEMMA
    FORALL(n_1 : Nat):
      (EXISTS (n_2 : Nat): greaterThan(n_2, n_1))
```

De todos modos, si buscamos un contraejemplo con nuestra herramienta obtenemos el siguiente resultado.

```
lemma_biggest :

  |-------
{1}   FORALL (n_1: Nat): (EXISTS (n_2: Nat): greaterThan(n_2, n_1))

Rule? (find-model 1 :verbose? t :model-as-pvs? t)
Searching for counterexample...
Running Kodkod...
[...]
Counterexample  found:

isZero?(zero)
notZero?(succ(zero))
prev(succ(zero)) = zero
greaterThan(succ(zero), zero)
```

El resultado que nuestra herramienta arroja es un contraejemplo válido aunque quizás no tan simple de interpretar. Lo que sucede es que para `succ(zero)` no hay un número natural que sea mayor, ya que el único par en el dominio de la función `greaterThan` en este modelo es `(succ(zero), zero)`. A pesar de ser correcto, este contraejemplo es algo que no tiene utilidad para el usuario ya que en nuestra especificación PVS no estamos limitando nunca el universo de los números naturales. Para que un modelo Kodkod cumpla nuestro lema debería ser infinito, pero eso es algo que no podemos manejar en nuestro buscador de contraejemplos ya que siempre nos manejamos dentro de un universo acotado.

3.4. La estrategia `find-model`: Implementación y uso

Del lado de la implementación de esta tesis, lo que tenemos principalmente es una estrategia de PVS llamada `find-model`, que cuenta con los siguientes parámetros:

- `fnums`: Este parámetro permite seleccionar qué fórmulas (ya sean del antecedente o del consecuente) se van a tener en cuenta al momento de buscar un modelo o contraejemplo.
- `univ-size`: Este parámetro permite elegir la cantidad de átomos máxima que va a tener el universo utilizado por Kodkod para hacer la instanciación del modelo.
- `exact-bounds?`: como ya describimos en la sección 10, este parámetro sirve para determinar si se evalúan todos los posibles tamaños hasta ese valor o sólo la cota superior. Las relaciones en Kodkod tienen una cota inferior (una instanciación parcial del problema) y una cota superior. Si este parámetro es `t` la cota inferior y superior son iguales. Esto es útil para cuando un error se presenta en un tamaño específico del universo y que no es el máximo
- `verbose?`: cuando es `t`, se muestra información extra al usuario.
- `model-as-pvs?`: si se encuentra un modelo, se muestra el mismo en formato PVS.
- `counter-example?`: cuando es `t` se busca un contraejemplo, si no se busca un modelo. Por default es `t`.
- `dry-run?`: cuando es `t`, solamente se realiza la traducción de la especificación de formato PVS a Kodkod, pero no se ejecuta la búsqueda de modelo.

4. CASO DE ESTUDIO

En este capítulo vamos a presentar distintos escenarios donde nuestra herramienta hace su mayor aporte. Como mencionamos en el capítulo 3, los puntos críticos son los momentos en que podemos llegar a introducir un error en la demostración y por lo tanto hacer imposible su prueba. El objetivo de este capítulo es ejemplificar cada uno de los casos identificados como puntos críticos. Para ello utilizamos como ejemplo central una teoría que especifica un árbol de sintaxis abstracta (*abstract syntax tree*) para expresiones de la lógica proposicional.

4.1. Descripción

El caso de estudio se trata de un TAD que denota expresiones sobre la lógica proposicional. Para esto contamos con:

- un tipo de dato `exp` para modelar expresiones,
- una función `value` para evaluar expresiones y
- una función `sub` para sustituir una expresión por otra.

Las definiciones de estos elementos en lenguaje PVS pueden verse en el apartado 4.1.

```
exp: DATATYPE
BEGIN
    true_ : isTrue?
    false_: isFalse?
    not_(e1: exp): isNot?
    and_(e1: exp,e2: exp): isAnd?
END exp

value(e: exp): RECURSIVE bool
  = IF (isTrue?(e)) THEN TRUE
    ELSIF (isFalse?(e)) THEN FALSE
    ELSIF (isNot?(e)) THEN (NOT value(e1(e)))
    ELSE (value(e1(e)) AND value(e2(e)))
    ENDIF
MEASURE e BY <<

sub(e_1: exp, e_2: exp, e_3: exp): RECURSIVE exp
  = IF (isTrue?(e_1)) THEN true_
    ELSIF (isFalse?(e_1)) THEN false_
    ELSIF (isNot?(e_1)) THEN not_(sub(e_1, e_2, e_3))
    ELSE (and_(sub(e1(e_1), e_2, e_3), sub(e2(e_1), e_2, e_3)))
    ENDIF
MEASURE e_1 BY <<
```

Listing 4.1: Definiciones del tipo de datos y funciones.

4.2. Los escenarios

En esta sección vamos a proveer ejemplos concretos donde nuestra herramienta puede ser de utilidad. Es decir, podemos estar realizando alguna acción que introduzca un error y, por lo tanto, provoque que la demostración sea imposible de concretar.

El foco de las siguientes subsecciones es mostrar cómo nuestra herramienta ayuda a detectar errores. Para que sea más comprensible se optó por utilizar ejemplos simples, para que se entiendan los contraejemplos generados, y no escenarios complejos donde la búsqueda de contraejemplos es más valiosa, pero entender una instancia puede resultar más engorroso.

Modificación del secuente

Lo primero que vamos a mostrar es cuando ejecutamos un comando en la demostración que modifica el secuente y en consecuencia se altera la demostrabilidad de la propiedad con la que estamos trabajando.

Supongamos que queremos verificar que la siguiente fórmula es válida:

$$\forall(e, e\_prime : exp) : \exists(a, b : bool, e\_1 : exp) : \\ a \wedge value(e) \iff b \wedge value(sub(e, e\_prime, e\_1))$$

Entonces la fórmula escrita en lenguaje PVS sería:

```
1 modified_sequent: LEMMA
2     FORALL(e, e_prime : exp):
3         (EXISTS (a, b: bool, e_1: exp):
4             a AND value(e) IFF b AND value(sub(e, e_prime, e_1)))
```

Listing 4.2: definición del lema a probar en PVS

Para iniciar la demostración, el primer paso es eliminar el cuantificador universal, mediante la regla correspondiente ilustrada en 2.6, que se implementa en PVS con el comando `skeep`.

```
Rule? (skeep)
Skolemizing and keeping names of the universal formula in (+ -),
this simplifies to:
modified_sequent :

  |-------
{1}   EXISTS (a, b: bool, e_1: exp):
        a AND value(e) IFF b AND value(sub(e, e_prime, e_1))
```

Como tenemos un cuantificador existencial en el consecuente, podemos utilizar el comando `instantiate` para instanciar las variables ligadas con los valores que deseemos. Al realizar esto estamos modificando el secuente, y por lo tanto es una situación que identificamos como punto crítico.

```
Rule? (instantiate 1 ("TRUE" "FALSE" "true_"))
Instantiating the top quantifier in 1 with the terms:
 (TRUE FALSE true_),
this simplifies to:
```

```
modified_sequent :

  |-------
{1}   value(e) IFF FALSE
```

Luego se puede invocar al buscador de contraejemplos para validar, aunque sea parcialmente, si la instanciación no afectó la posibilidad de cerrar la demostración.

```
Rule? (find-model 1 :verbose? t :model-as-pvs? t)
Searching for counterexample...
Substitutions: e -> s0
Running Kodkod...

...

Counterexample found:

e = true_

...
```

Esto nos indica que cometimos un error al usar el comando `instantiate`, usando una instanciación errónea. Intentemos ahora ahora volver un paso atrás en la demostración usando `undo` para luego probar una instanciación distinta.

```
Rule? (undo)
This will undo the proof to:
modified_tcc :

  |-------
{1}   EXISTS (a, b: bool, e_1: exp):
        a AND value(e) IFF b AND value(sub(e, e_prime, e_1))

Sure? (Y or N) Y
modified_sequent :

  |-------
{1}   EXISTS (a, b: bool, e_1: exp):
        a AND value(e) IFF b AND value(sub(e, e_prime, e_1))

Rule? (instantiate 1 ("FALSE" "FALSE" "true_"))
Instantiating the top quantifier in 1 with the terms:
 (FALSE FALSE true_),
this simplifies to:
modified_sequent :

  |-------
{1}   FALSE IFF FALSE
```

Y ahora podemos observar que al hacer una instanciación diferente pudimos llegar a una fórmula que siempre es válida.

TCC no demostrable

Para el caso de una TCC no demostrable tuvimos que usar un poco más el ingenio. La realidad es que el *typechecker* de PVS captura los errores de chequeo de tipos más

triviales. Pero cuando la verificación no es tan directa podemos estar generando un error nuevo, y ahí es cuando nuestro buscador de modelos puede asistir.

Una manera posible de ilustrar el problema que queremos ver es utilizando subtipos. Para PVS, esto es básicamente tener un predicado que indica cuándo un elemento del tipo pertenece además al subtipo. Para exponer este tipo de errores, podemos usar el comando `name`, el cual le asigna un nombre a una expresión. Y esta expresión además va a tener un error de tipado.

```
Rule? (name "formula"
            "LET a: {x: exp | isTrue?(e2(x))} = true_ IN isTrue?(a)")
Letting formula name LET a: {x: exp | isTrue?(e2(x))} = true_ IN
                         isTrue?(a),
this yields  2 subgoals:
non_demo_tcc.1 :

{-1}  (LET a: {x: exp | isTrue?(e2(x))} = true_ IN isTrue?(a)) = formula
  |-------
[1]   FORALL (e, e_prime: exp):
        (EXISTS (a, b: bool, e_1: exp):
           a AND value(e) IFF b AND value(sub(e, e_prime, e_1)))
```

Esto nos genera dos *subgoals*. Para este ejemplo, centremos la atención en el segundo de ellos, al cual accedemos aplicando el comando `postpone`. Esta rama adicional fue generada por el *typechecker* y en ella se muestra una restricción de *tipado*.

```
Rule? (postpone)
Postponing non_demo_tcc.1.

non_demo_tcc.2T (TCC):

  |-------
{1}   isTrue?(e2(true_))
[2]   FORALL (e, e_prime: exp):
        (EXISTS (a, b: bool, e_1: exp):
           a AND value(e) IFF b AND value(sub(e, e_prime, e_1)))
```

No siempre es simple notar si esta clase de restricciones son demostrables o no, por lo tanto acudimos al buscador de modelos para verificarlo. La fórmula 1, generada por el *typechecker*, tiene un error de tipado, por lo cual no es posible encontrar una instancia válida y nuestra herramienta podría encontrar un contraejemplo.

```
Rule? (find-model 1 :model-as-pvs? t :vebose? t)
...
Counterexample not found.
...
```

Nuestra estrategia no pudo encontrar un contraejemplo, y esto en general sucede por dos razones: no existe contraejemplo alguno, o el espacio de búsqueda que se utiliza no es el adecuado. Pero en este caso sucedió algo diferente, y fue que partimos de una fórmula con un error de tipado (`e2(true_)`, que pide la segunda componente a la fórmula `true_`), y esto es algo que no puede manejar nuestra herramienta. Tenemos como precondición que las fórmulas estén bien tipadas. Pero por una cuestión práctica, PVS deja agregar expresiones durante la demostración. Es en ese momento donde el usuario debe probar que están bien tipadas, y por lo tanto se generan TCCs. En este caso puede ser engañoso

que no se encuentre contraejemplo, por lo que sería una buena idea además, tratar de buscar un modelo.

```
Rule? (find-model 1 :verbose? t :model-as-pvs? t :counter-example? nil)
Searching for instance...
Running Kodkod...
...
Instance not found.
...
```

Entonces, como tampoco fue posible encontrar un modelo, podemos sospechar que puede haber un error de tipado. Sin embargo, no lo podemos asegurar ya que existe la posibilidad que se esté trabajando con un universo que no es lo suficientemente grande.

Ahora discutiremos un escenario donde la respuesta de nuestra herramienta es concluyente. Supongamos que tenemos la siguiente conjetura que queremos demostrar:

```
1   counterexample_tcc: LEMMA
2     FORALL(e_: exp):
3        (FORALL (e1_: { e2_: exp | value(e2_) }):
4            value(and_(e1_,true_))) IMPLIES FALSE
```

Vamos a arrancar la demostración eliminando el cuantificador universal, usando su regla correspondiente mostrada en 2.6 que se implementa usando el comando `skeep`.

```
Rule? (skeep)
Skolemizing and keeping names of the universal formula in (+ -),
this simplifies to:
counterexample_tcc :

{-1}  (FORALL (e1_: {e2_: exp | value(e2_)}): value(and_(e1_,true_)))
  |-------
```

A continuación instanciamos con `e_` a la variable ligada por el cuantificador universal y pasamos a tener dos *subgoals*.

```
Rule? (inst -1 "e_")
Instantiating the top quantifier in -1 with the terms:
 e_,
this yields  2 subgoals:
counterexample_tcc.1 :

{-1}  value(and_(e_,true_))
  |-------
```

Ahora cambiamos de *subgoal* donde solo vamos a tener una tesis y utilizamos la herramienta para hacer el chequeo.

```
Rule? (postpone)
Postponing counterexample_tcc.1.

counterexample_tcc.2 (TCC):

  |-------
{1}   value(e_)

Rule? (find-model 1 :model-as-pvs? t :verbose? t)
Searching for counterexample...
```

```
...

Counterexample found:

and_(true_, not_(true_)) = e_
...
```

El contraejemplo generado es equivalente a `false_`. Entonces la fórmula del consecuente `value(e_)` no es válida.

Generación de *subgoals* no demostrables

Usando algunos comandos de PVS, como puede ser `case`, se generan *subgoals* nuevos, los cuales hay que probar para completar la demostración. Puede suceder que alguno de estos *subgoals* no sea demostrable, entonces aquí es cuando nuestra herramienta nos puede asistir.

Supongamos que contamos con el siguiente lema que queremos demostrar

```
1 lemma_case: LEMMA
2   FORALL(e_ : exp):
3     value(not_(and_(e_, e_)))=FALSE IMPLIES value(not_(and_(e_, true_)))
```

Como en los casos anteriores, el primer paso es eliminar el cuantificador universal (usando la regla de la figura 2.6) mediante el comando `skeep`.

```
Rule? (skeep)
Skolemizing and keeping names of the universal formula in (+ -),
this simplifies to:
lemma_case :

{-1}  value(not_(and_(e_, e_))) = FALSE
  |-------
{1}   value(not_(and_(e_, true_)))
```

Por lo que pasamos a tener una fórmula en el antecedente y otra en el consecuente. Queremos ver qué sucede cuando `e_` toma como valor `true_` y cuando toma como valor `false_`, por lo tanto utilizamos el comando `case`.

```
Rule? (case "e_ = true_")
Case splitting on
   e_ = true_,
this yields  2 subgoals:
lemma_case.1 :

{-1}  e_ = true_
[-2]  value(not_(and_(e_, e_))) = FALSE
  |-------
[1]   value(not_(and_(e_, true_)))
```

Si ahora ejecutamos nuestra estrategia obtenemos obtenemos:

```
Rule? (find-model * :model-as-pvs? t :verbose? t)
Searching for counterexample...
...
Counterexample found:
```

```
e_ = true_
...
```

y al encontrar un contraejemplo nos damos cuenta que demostrar la primera rama es imposible. Esto denota entonces que al utilizar el comando `case` introdujimos una hipótesis que no permite la demostrabilidad en la rama uno.

Modificación del secuente de manera arbitraria

Supongamos que estamos trabajando en una demostración compleja, la cual involucra una cantidad importante de fórmulas. La persona llevando a cabo esta tarea puede ocultar fórmulas que no considere relevantes en algún momento (utilizando el comando `hide`) para enfocarse en lo que cree útil. Y al hacer esto, puede incurrir involuntariamente en un problema, ya que se ocultó una fórmula que *a priori* no parecía hacer un aporte a la demostración, pero que en realidad sí era necesaria para continuar.

En ese momento es donde nuestra herramienta puede ser nuevamente útil. Vamos a intentar encontrar un contraejemplo que contradiga lo que queremos demostrar, que se genera por el hecho de ocultar una fórmula que es necesaria.

Si tenemos[1] el siguiente lema:

```
1    lemma_hide: LEMMA
2    FORALL(e_: exp): value(e_) IMPLIES value(not_(e_)) = FALSE
```

Listing 4.3: definición del lema a probar

Para comenzar la demostración, aplicamos la regla para el cuantificador universal que se ve en la figura 2.6, mediante el comando `skeep`

```
Rule? (skeep)
Skolemizing and keeping names of the universal formula in (+ -),
this simplifies to:
lemma_hide :

{-1}  value(e_)
  |-------
{1}   value(not_(e_)) = FALSE
```

y ahora ocultamos la hipótesis, quedándonos:

```
Rule? (hide -1)
Hiding formulas:  -1,
this simplifies to:
lemma_hide :

  |-------
[1]   value(not_(e_)) = FALSE
```

para luego invocar a la estrategia y chequear si se encuentra o no un contraejemplo:

```
Rule? (find-model 1 :verbose? 1 :model-as-pvs? t)
Searching for counterexample...
...
```

[1] Dado que lo que queremos mostrar es cómo ayuda la herramienta y no un caso donde el comando `hide` es útil, decidimos hacer un ejemplo pequeño de manera tal de disminuir la carga visual

```
Counterexample found:

e_ = false_
...
```

Y es efectivamente lo que sucede. Al haber ocultado una hipótesis, pasamos a tener menos restricciones y posible instanciar un contraejemplo.

Ahora revelemos la fórmula que escondimos, y luego invocamos a nuestra estrategia en búsqueda de un contraejemplo:

```
{-1}  value(e_)
  |-------
[1]   value(not_(e_)) = FALSE

Rule? (find-model * :model-as-pvs? t :verbose? t)
Searching for counterexample...
...
Counterexample not found.
...
```

Como era de esperar, al tener nuevamente la hipótesis como restricción no fue posible encontrar un contraejemplo.

5. CONCLUSIONES

Empezamos nuestro trabajo con un objetivo claro: asistir al usuario que está realizando una demostración en PVS. Para esto decidimos enfocar los esfuerzos para atacar las situaciones que generaban mayores problemas, y además queríamos hacerlo de la manera más automática posible. El principio entonces era identificar los puntos críticos de la demostración y realizar un chequeo automático. Es decir, quisimos hacer algo a análogo a lo que fue Nitpick para Isabelle. Entonces, la primera actividad era determinar si existía alguna herramienta que nos pueda facilitar esta tarea. Kodkod parecía ser una elección indicada ya que demostró ser un excelente *constraint solver* para lógica de primer orden. Además es un *setting* similar al de Isabelle/Nitpick donde también Kodkod es utilizado. Entonces la idea principal de la tesis pasó a ser la siguiente: partir de una especificación PVS, traducir al lenguaje Kodkod y finalmente verificar si existía o no un modelo. El problema más grande a atacar era la traducción entre PVS y Kodkod, así que decidimos enfocarnos en eso.

Idealmente nuestro buscador de modelos debería poder trabajar con cualquier demostración que se esté realizando en PVS. Pero esto no iba a ser posible por dos razones: la primera es que se trataba de un intento inicial por resolver este problema, entonces definir un alcance acotado iba a permitir enfocar mejor los esfuerzos y obtener un resultado mejor para escenarios específicos; y la segunda es que Kodkod tiene sus limitaciones, por lo que había que adaptarse a ellas.

Otro desafío que encontramos fue el de definir el alcance, es decir, qué elementos dentro de PVS íbamos a traducir a Kodkod. Para la la traducción –que se podría decir que es la parte fundamental de este trabajo– se determinó qué partes eran relevantes tratar y, a partir de eso, se comenzó a idear la implementación. Además, dentro del contexto los TADs, la traducción no se aplica de una manera ingenua y directa sino que, por ejemplo, se agrega el axioma de disyunción para facilitar y mejorar el desempeño de Kodkod. Cabe señalar que este axioma no se utiliza en PVS debido al alto costo computacional que impone al *typechecker*; se reemplaza por el orden bien fundado sobre los elementos del tipo, apareándolos con números naturales.

Como mencionamos anteriormente, PVS no representa a las fórmulas con negaciones, por lo tanto éstas terminan perteneciendo al antecedente o consecuente pero siempre de manera positiva. Además, PVS no rastrea si una fórmula es una hipótesis o una tesis, lo que finalmente termina limitando nuestra capacidad de automatización. Es decir, al agregar o quitar una fórmula no podemos saber si lo que hay que buscar es un modelo o un contraejemplo, es una decisión que tiene que tomar el usuario. Por otro lado, Kodkod es un buscador de modelos relacional. De esta manera se define un poco mejor el tipo de problemas que podemos manejar. Más específicamente: podemos trabajar con tipos de datos abstractos, pero nos quedan fuera de alcance otros tipos de datos, como los tipos interpretados.

Consideramos que el resultado fue exitoso dado que fuimos capaces de aplicar la técnica propuesta a casos de estudios no triviales. De todas maneras, entendemos que al

ser un primer intento, dejamos fuera muchas posibles mejoras para la herramienta, las cuales la harían más útil. Discutimos estos temas en el capítulo siguiente.

6. TRABAJO FUTURO

En términos generales, el trabajo futuro que se desprende de la presente tesis es avanzar sobre los elementos para los cuales PVS tiene soporte sintáctico, pero que no están cubiertos por la traducción presentada en este trabajo.

Soporte para cuantificación de alto orden

Como contamos anteriormente, trabajamos sobre una parte acotada del lenguaje de PVS. Una extensión natural del trabajo de esta tesis es ampliar el fragmento de lenguaje soportado. Para el caso de la cuantificación de alto orden lo que hay que intentar lograr es una traducción a la lógica relacional de primer orden. Como los SAT-solvers son especialmente sensibles a la codificación de los problemas, es necesario tener precaución a la hora de traducir funciones de alto orden. Algunos trabajos previos sobre técnicas similares indican que, siempre que sea posible, las funciones de alto orden deben ser traducidas a su equivalente en la lógica relacional de primer orden en lugar de ser expandidas a sus definiciones.

Soporte para tipos interpretados

Hay varios tipos de datos que nuestra implementación actual no soporta. Un enfoque para resolver esto podría ser utilizar otros *backends* que manejan estos tipos de datos (como ser *strings*, reales, enteros, *floats*, etc.). Entonces la tarea consistiría en identificar el tipo de dato que se está manejando y hacer una traducción adecuada para generar el *input* necesario para la herramienta. Pongamos un ejemplo concreto: si queremos manejar números enteros una opción posible es utilizar Random Testing. PVS cuenta con generadores de instancias *random* para cada tipo de dato interpretado. Entonces podemos obtener instancias, que pueden estar o no dentro de las restricciones que define nuestra teoría. Si la instancia es útil, es decir, es válida para la teoría, la podemos utilizar para construir un modelo.

Traducción de tipos inductivos y coinductivos

Una de las principales diferencias entre los modelos aceptados por una especificación de alto orden, como las que se pueden escribir en PVS, y los modelos explorados por Kodkod está dado por el salto cualitativo en sus cardinalidades. Esta diferencia toma particular relevancia en casos como los de la traducción de tipos inductivos. El problema que enfrentamos es el de lidiar con una naturaleza infinita en un universo finito. Por ejemplo, para cualquier número natural siempre es posible encontrar un sucesor. Entonces, el tipo de dato que corresponde a los números naturales no admite ningún modelo finito. Aún así, como hemos mostrado en secciones anteriores de este documento, muchos de los contraejemplos hallados en modelos finitos pueden reformularse en los modelos estándar de PVS y, por lo tanto, son de utilidad para validar la corrección de las especificaciones. De todos modos, otra importante extensión de este trabajo es la implementación de soporte para tipos coinductivos, lo que ha quedado fuera del alcance de esta tesis.

Otra posible extensión es evaluar diferentes técnicas de traducción de tipos inductivos. Por ejemplo, se puede utilizar una axiomatización basada en selectores y discriminadores, inspirada por el modelado de listas y árboles de Kuncak y Jackson [31]. Este tipo de vista podría ser más eficiente que la del constructor porque descompone los constructores de alta aridad en varios selectores de baja aridad.

Soporte de trazabilidad de hipótesis y tesis en un secuente

Como hemos mencionado, el uso del comando `find-model` requiere que el usuario sepa cuál es el rol de las fórmulas que aparecen en el secuente, es decir, debe tener en consideración cuáles se corresponden con hipótesis del problema original, o al menos fragmentos de ellas, y cuáles con la tesis, o fragmentos de ella. Es decir, si estamos abocados a la búsqueda de un contraejemplo para detectar si se ha alterado la demostrabilidad de nuestro secuente, nos encontramos en terreno firme pues toda la información del secuente debe ser considerada. Por el contrario, si deseamos someter a juicio la consistencia del conjunto de hipótesis para ver que la demostración no sea trivial, debemos saber cuáles de las fórmulas que aparecen como hipótesis en realidad son tesis que debieran estar negadas y cuáles de las fórmulas que aparecen en el consecuente, en realidad son hipótesis que debieran aparecer negadas.

PVS no provee ningún soporte de trazabilidad del rol que tenían asignado originalmente las fórmulas y, consecuentemente, cada vez que se aplica un comando que descompone una fórmula a partir de su estructura proposicional, se pierde información importante de su rol en la demostración. En este contexto, la posibilidad de hacer un uso potente del comando `find-model` recae en el conocimiento que el usuario de PVS tenga sobre el verdadero rol de cada una de las fórmulas que aparecen en el secuente. Una posible línea de extensión de este trabajo es la implementación de elementos que permitan saber efectivamente cuáles de las fórmulas del secuente son hipótesis y cuales son tesis de manera que dar soporte al usuario a la hora de decidir qué uso debe hacer de ellas.

Identificación de *unsat core*

Otra funcionalidad que podría ser útil implementar en un futuro es la de identificar qué parte de la demostración se corresponde con el *unsat core*. Es decir, el conjunto de restricciones que hacen insatisfacible el problema. Si se quita alguna de estas restricciones, el problema pasa a ser satisfacible. De esta manera, se pueden resaltar las fórmulas que generan el conflicto y poder enfocar los esfuerzos para comprender mejor dónde está el error.

Kodkod nos informa qué fórmulas pertenecen al *unsat core*. Pero al momento de querer entender con qué parte de la especificación de PVS se corresponde no alcanza con hacer la inversa de la traducción. Esto ocurre porque para algunos axiomas determinados de un TAD no realizamos una traducción directamente, si no que escribimos un axioma equivalente que luego sí traducimos. Es el caso del axioma de disyunción, el cual no es generado por PVS por tener un alto costo computacional y es reemplazado por otro que utilizan un mapeo a números enteros.

Para salvar esta situación, se debe identificar las fórmulas que son reemplazadas por otras antes de realizar la traducción. Entonces cuando hacemos la inversa de la traducción

para una fórmula que pertenece al *unsat core* podemos darnos cuenta si se trata de una de las fórmulas originales o no. Si no es una de las originales, podemos resaltar la que corresponde.

Mejora del *input* para Kodkod

Actualmente se realiza una traducción *naïve*, y por lo tanto se traslada gran peso computacional a Kodkod. Entonces el *input* generado para que Kodkod trate de buscar una instancia podría ser mejorado haciendo una traducción más eficiente.

Una primera área de mejora podría ser especificar mejor la cota inferior (los valores que debe incluir la instancia) y la superior (los valores que potencialmente puede incluir la instancia). Nuestra implementación actual pone como cota inferior el conjunto vacío y como cota superior el producto cartesiano (entre el dominio y la imagen), lo que genera un espacio de búsqueda demasiado grande y el cual casi nunca es necesario. Supongamos que se quiere trabajar con una identidad parcial, se podría definir como cota inferior el conjunto vacío y como cota superior $(A_1, A_2, \cdots, A_n - 1, A_n)$, es decir la diagonal de la matriz.

Bibliografía

[1] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[2] Daniel Jackson. *Software abstractions: logic, language, and analysis*. The MIT Press, 2006.

[3] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of the 11th. International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 148–752, Saratoga, NY, June 1992. Springer-Verlag.

[4] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A proof assistant for higher-order logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2002.

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series). Springer-Verlag, New York, NY, USA, 2004.

[6] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the 8th European software engineering conference held together with the 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 62–73, Vienna, Austria, 2001. ACM Press.

[7] Lawrence C. Paulson. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Renate A. Schmidt, Stephan Schulz, and Boris Konev, editors, *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*, volume 9 of *EPiC Series in Computing*, pages 1–10. EasyChair, 2010.

[8] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016.

[9] Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Autom. Reason.*, 61(1-4):423–453, 2018.

[10] Cezary Kaliszyk and Josef Urban. Hol(y)hammer: Online ATP service for HOL light. *Math. Comput. Sci.*, 9(1):5–22, 2015.

[11] Mariano Miguel Moscato, Carlos G. Lopez Pombo, and Marcelo F. Frias. Dynamite: A tool for the verification of alloy models based on pvs. *ACM Transactions on Software Engineering and Methodology*, 23(2):20:1–20:37, 2014.

[12] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of First International Conference on*

Interactive Theorem Proving (ITP 2010), volume 6172 of *Lecture Notes in Computer Science*, pages 131–146, Berlin, Heidelberg, July 2010. Springer-Verlag.

[13] Tjark Weber. *SAT-based finite model generation for higher-order logic.* PhD thesis, Technical University Munich, Germany, 2008.

[14] Sam Owre, Natarajan Shankar, John M. Rushby, and David Stringer-Calvert. *PVS prover guide.* Computer Science Laboratory, SRI International, version 7.1 edition, August 2020.

[15] Emina Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009.

[16] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orma Grumberg and Michael Huth, editors, *Proceedings of the 13th. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647, Braga, Portugal, April 2007. Springer-Verlag.

[17] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[18] Jasmin Christian Blanchette. *Clubbing Cods A User's Guide to Kodkodi.* Institut für Informatik, Technische Universität München, version 1.5.2 edition, Enero 2012.

[19] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert G.L.T. Meertens. Generic programming – an introduction –. In S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques, editors, *Proceedings of Third International School, Advanced Functional Programming - AFP'98 (Revised Lectures)*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, September 1999.

[20] Maarten M. Fokkinga and Johan Jeuring. *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, volume Part I. University of Utrecht, Utrecht, Netherlands, September 1992.

[21] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6(1):1–32, 1996.

[22] Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, 1992. Advisor: Lambert G.L.T. Meertens and Leo A.M. Verbeek.

[23] Sam Owre, Natarajan Shankar, John M. Rushby, and David Stringer-Calvert. *PVS language reference.* SRI International, version 7.1 edition, August 2020.

[24] Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic prover for the elementary functions. In Serge Autexier, John A. Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics, 9th International Conference, AISC 2008, 15th Symposium, Calculemus 2008, 7th International Conference, MKM 2008, Birmingham, UK, July 28 - August 1,*

2008. Proceedings, volume 5144 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 2008.

[25] J. Hurd. Metis first order prover. `http://gilith.com/software/metis/`, 2007.

[26] H. Hong. Qepcad — quantifier elimination by partial cylindrical algebraic decomposition. `https://www.usna.edu/Users/cs/wcbrown/qepcad/B/QEPCAD.html`.

[27] Christopher W. Brown. Qepcad b: a program for computing with semi-algebraic sets using cads. *ACM Sigsam Bulletin*, 37(4):97–108, 2003.

[28] Sam Owre. Random testing in pvs. In *Workshop on automated formal methods (AFM)*, volume 10, 2006.

[29] Dov Gabbay, Angus Macintyre, and Dana Scott, editors. *Introduction to Logic and to the Methodology of Deductive Sciences*, volume 24 of *Oxford Logic Guides*. Dover Publishing, 1941.

[30] J.F.A.K.van Benthem and Kees Doets. Higher-order logic. In Dov Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 1, pages 275–329. Kluwer Academic Publishers, second edition, 2001.

[31] Viktor Kuncak and Daniel Jackson. Relational analysis of algebraic datatypes. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 207–216. ACM, 2005.

Apéndice

Apéndice

A. COMANDOS DE PVS

La lista de comandos de PVS que generan TCCs se encuentran en el A.

- `case`
- `case*`
- `case-replace`
- `generalize`
- `inst`
- `inst-cp`
- `inst?`
- `instantiate`
- `instantiate-one`
- `name`
- `name-replace`
- `same-name`
- `lemma`
- `rewrite-lemma`
- `rewrite-with-fnums`
- `use`
- `apply-eta`
- `eta`
- `extensionality`
- `replace-eta`
- `replace-extensionality`
- `measure-induct`
- `measure-induct+`
- `measure-induct-and-simplify`
- `simple-measure-induct`
- `both-sides`
- `grind`

- `typepred`
- `typepred!`

Son básicamente los comandos que reciben una expresión como parámetro.

Nuestra modificación va a estar integrada internamente, eso va a provocar que se aplique no sólo a estos comandos, sino a todos aquellos que generen ramas TCC (sean estos definidos en otras bibliotecas o por el usuario).

B. NOTAS DE INSTALACIÓN

B.1. Prerrequisitos

El único prerrequisito para poder utilizar nuestra estrategia es tener instalado PVS (`http://pvs.csl.sri.com/download.shtml`).

B.2. Instalación

Los siguientes pasos son necesarios para que la estrategia `find-model` esté disponible en PVS:

1. Clonar el repositorio del proyecto (https://bitbucket.org/lteren/tesis/src/master/) en el directorio `<dir>`.
2. Colocar en un directorio definido en la variable `PATH` el archivo `run-kodkod.sh` (que se encuentra dentro de `<dir>/kodkodi-1.5.2`).
3. Configurar la variable de ambiente `PVS_LIBRARY_PATH` con la ruta absoluta de `<dir>`.

La estrategia `find-model` se encuentra definida en el archivo `patch-20291231-model-finding.lisp`, y como se encuentra dentro del directorio `<dir>/pvs-patches`, PVS la carga automáticamente de manera tal que esté disponible para usar.