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

Análisis de dataflow para mejorar la verificación de programas basada en SAT

Tesis de Licenciatura en Ciencias de la Computación

Bruno Esteban Cuervo Parrino

Directores: Juan Pablo Galeotti y Diego Garbervetsky

Buenos Aires, Mayo de 2011

Índice general

Resumen v

Agradecimientos vii

1. Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Descripción de la tesis . . . 1
- 1.3. Resumen de aportes . . . 2
- 1.4. Trabajo relacionado . . . 3

2. Herramientas preliminares 5
- 2.1. Verificación formal . . . 5
- 2.2. TACO . . . 5
- 2.3. Alloy . . . 6
- 2.4. Kodkod . . . 9
- 2.5. DynAlloy . . . 9
- 2.6. Análisis estático de programas . . . 10

3. TACO 13
- 3.1. Introducción . . . 13
- 3.2. Overview . . . 13
- 3.3. Planteo del problema . . . 17

4. Análisis de dataflow para el cómputo de cotas 21
- 4.1. Semántica concreta . . . 21
- 4.2. Semántica abstracta . . . 23
- 4.3. Correctitud y terminación . . . 25

5. Diseño 31
- 5.1. Construcción del CFG . . . 32
- 5.2. Framework de dataflow . . . 32
- 5.3. Análisis de cotas . . . 34
- 5.4. Loop unroll . . . 35

6. Evaluación experimental 37
- 6.1. Introducción . . . 37
- 6.2. Resultados . . . 38
- 6.3. Validez de los resultados . . . 42

7. Conclusiones y trabajo a futuro 45

Bibliografía 47

Resumen

La verificación acotada de programas basada en SAT-solving consiste en la traducción del programa a analizar, junto con las anotaciones provistas por el usuario, a una fórmula proposicional. Posteriormente, la fórmula obtenida es analizada en busca de violaciones a la especificación usando un SAT-solver. Esta técnica es capaz de probar la ausencia de errores hasta un scope dado.

SAT es un problema NP-completo, lo que implica que no se conoce un algoritmo que lo resuelva eficientemente, siendo su complejidad exponencial en la cantidad de variables proposicionales. Es por esto que es esperable que una representación lógica de programa que apunte a reducir el número de variables tenga un gran impacto en la performance y escalabilidad del análisis.

TACO es una herramienta que implementa la traducción de programas Java con anotaciones JML a Alloy, un lenguaje formal relacional que permite analizar automáticamente especificaciones buscando contraejemplos de aserciones con la ayuda de un SAT-solver. TACO introduce una técnica para computar los posibles valores asignados a las variables del programa, que llamamos cotas, en el estado inicial del programa.

En esta tesis mostramos cómo las técnicas de dataflow, típicamente utilizadas por los compiladores para optimizar programas, pueden también ser usadas como un medio para obtener traducciones optimizadas. En particular, definimos e implementamos un análisis de dataflow que, utilizando las cotas iniciales provistas por TACO, computa el conjunto de valores que pueden ser asignados a variables locales y de instancia en cada punto del programa. Esta información es utilizada para eliminar variables innecesarias en la fórmula proposicional resultante de la traducción del programa.

Nuestros experimentos muestran mejoras significativas en los tiempos de análisis, permitiendo tanto la verificación de casos de estudio que no podían ser analizados anteriormente como, en algunos casos, la extensión del scope del análisis.

Keywords: bounded program verification, dataflow analysis, SAT solving

Agradecimientos

A Juan Pablo y a Diego, por su trabajo, apoyo e inagotable paciencia antes, durante e incluso después de la realización de esta tesis, que no existiría sin ellos. También por sus consejos y acompañamiento, que llegaron mucho más allá del alcance de ella.

A Charly López Pombo y a Esteban Pavese, por haber aceptado y corregido este trabajo contrarreloj, y por haber invertido tiempo y dedicación gracias a los cuales el resultado es decididamente mejor.

Al Departamento de Computación de la facultad, por una educación excelente, y al grupo de RFM e Ingeniería de Software, por haberme recibido tan cálidamente durante la realización de este trabajo.

Al grupo de "gente de la facu", con el que fue un placer cursar tantas materias, discutir ideas y proyectos y compartir todos estos años de carrera: Chapa, Facu, Fer, Giga, Leo R., Leo S., Luisito, Martín, Mati, Maxi, Nelson, Pablito, Palla, Serch, Tara, Vivi y Z. Y en especial, gracias a Javi y a Eddy por tantas madrugadas de trabajo juntos poniéndole ganas, y a Naty por habernos aguantado y complementado tan bien.

A mis compañeros de trabajo durante estos años, por haberme incentivado siempre a poner el estudio como una prioridad, y haberme ayudado a crecer a base de confianza y responsabilidad. A todos ellos, desde Mati por su insistente "¿y cuándo te recibís?", hasta Chris por haber sido el jefe que uno siempre quisiera tener.

A todos mis amigos por haberme acompañado siempre, en las buenas y en las malas, y por compartir con una sonrisa todas las etapas y actividades de la vida. Gracias a todos aquellos que, de forma metafórica o literal, están siempre presentes para sostener el otro extremo de la cuerda.

A mi *vieja*, por haber estado y estar, incondicionalmente, en todas.

¡Gracias!

Capítulo 1

Introducción

1.1. Motivación

Hoy en día, el software se encuentra en todos lados, desde celulares hasta transbordadores espaciales. El software no es sólo un producto negociable, sino una parte intrínseca de la vida moderna, y uno de los conductores principales de la economía global. Además de acrecentar día a día su presencia en todos los aspectos de la vida cotidiana, la complejidad del software crece de forma inquietante. Su tamaño ya no se mide en miles de líneas de código, sino en términos de millones de líneas de código. Este crecimiento viene acompañado indeseablemente de la presencia de defectos, conocidos comúnmente como *bugs*. Los defectos de software varían desde los que consideramos molestos y ocasionan pérdidas que pueden medirse en términos económicos [Bro75], hasta los de aplicaciones de misión crítica que pueden tener trágicas consecuencias.

La creciente importancia de la calidad del software en la economía y en la vida diaria demanda el desarrollo de técnicas y procesos de ingeniería más robustos para su construcción, y de herramientas más avanzadas para ayudar a los programadores a alcanzar una mayor calidad en los artefactos de software que producen. Escribir software correcto y confiable es una tarea muy difícil. Afortunadamente, tanto la academia como la industria han hecho progresos considerables en crear técnicas, herramientas y metodologías apuntadas a mejorar la calidad del software. En particular, el análisis automático de programas está cada vez más presente en las actividades relacionadas al desarrollo de software. Las herramientas automáticas se utilizan cada vez más para complementar las estrategias tradicionales de control de calidad [APM+07][BCLR04]. En la presente tesis seguimos el trabajo en esta área, con la intención de hacer un aporte en el campo de los métodos formales para mejorar la calidad del software.

1.2. Descripción de la tesis

El análisis de dataflow [Kil73] es una técnica de análisis estático ampliamente utilizada para la optimización y la comprensión de los programas. En esta tesis mostramos como también puede ser usado en la verificación formal como un medio para obtener optimizaciones en el proceso de traducción.

La verificación formal automatizada ha avanzado mucho en los últimos años, pero diversos problemas hacen que no se haya alcanzado todavía la capacidad necesaria para trabajar con software a gran escala. Entre ellos, el poder de las herramientas actuales de demostración automática que se necesitan en el proceso, si bien ha ido en aumento, sigue siendo limitado debido a la complejidad inherente del problema.

Dentro de las distintas técnicas existentes, este trabajo se centra en la verificación acotada, consistente en traducir apropiadamente el programa original a verificar, junto con su especificación, a una fórmula proposicional. Luego, un SAT-solver [ES04] permite encontrar una valuación de las variables proposicionales que representen una falla. Ya que SAT es un problema NP-completo [Coo71], no se conoce un algoritmo que lo resuelva eficientemente. Aunque en el peor caso la complejidad es exponencial, en la práctica los SAT-solvers modernos logran resultados mucho mejores en problemas cotidianos.

1.2.1. Estructura de la tesis

En el capítulo 2 introducimos los conceptos de verificación formal y análisis estático de código, y presentamos las herramientas preliminares que dan forma a este trabajo: TACO [Gal10], Alloy [Jac02], DynAlloy [FGLPA05] y Kodkod [TJ07].

En el capítulo 3 detallamos el funcionamiento de TACO y presentamos la idea intuitiva detrás de nuestro análisis de dataflow. Utilizamos un ejemplo integrador con el objetivo de comprender todas las etapas y optimizaciones involucradas en el proceso de verificación.

En el capítulo 4 definimos y formalizamos nuestro análisis. A través de las definiciones de semántica concreta y abstracta, probamos su correctitud y terminación.

En el capítulo 5 presentamos cómo se lleva a la práctica la formalización del análisis, y se destacan los puntos particulares de la implementación, en el contexto de verificación acotada. Además, se presenta una mejora encontrada en el proceso de traducción de TACO.

En el capítulo 6 se evalúa experimentalmente nuestro análisis de dataflow, y se analizan los resultados.

Finalmente, en el capítulo 7 presentamos las conclusiones del presente trabajo, así como las alternativas de trabajo a futuro que de él se desprenden.

1.3. Resumen de aportes

La primera contribución de la tesis, que prepara el camino para el análisis de cotas, es la construcción de un framework genérico de dataflow para DynAlloy. A partir de este se puede implementar cualquier análisis de dataflow que se desee, con la intención de optimizar aún más el proceso de traducción, o de proveer al SAT-solver de información adicional que pueda ser inferida del código.

Presentamos la formalización de un análisis de dataflow para propagar valores a través del grafo de flujo de control del programa. Demostramos que el resultado del análisis es una sobre aproximación segura del comportamiento del programa. También demostramos que la fórmula proposicional obtenida por TACO utilizando nuestro análisis es equisatisfacible con respecto a la fórmula no optimizada.

El aporte principal es la construcción del análsis de dataflow que computa cotas para las variables locales y de instancia en todos los puntos del programa. Estas cotas son usadas para eliminar variables innecesarias con las que debe trabajar el SAT-solver. Esto lleva a una mejora del tiempo de análisis, y permite extender el scope de los casos de estudio.

Por último, como veremos más adelante, TACO utiliza loop unrolling para acotar la cantidad de iteraciones que ejecuta un ciclo. Con el objetivo de mejorar nuestro análisis de dataflow y obtener cotas más ajustadas en las pruebas iniciales, modificamos la forma en la que se efectúa el unrolling como parte del proceso de traducción, sin alterar el resultado de la verificación. Esta modificación no sólo resultó en una optimización de nuestro análisis, sino también en un significativo aumento de performance para TACO en sí mismo.

1.4. Trabajo relacionado

Los pasos iniciales en la verificación de programas pueden atribuirse a Floyd [Flo67] y Hoare [Hoa69]. Desde aquellos días se desarrollaron una gran cantidad de sistemas cuyo objetivo es asistir a la verificación automática de programas.

Hay mucho trabajo apuntado a mejorar la verificación de programas basada en SAT. En esta sección nos enfocaremos solamente en el trabajo relacionado más notable que utiliza, como esta tesis, análisis estático de código para facilitar el proceso de SAT-solving. Si bien nosotros nos centramos particularmente en el análisis de dataflow, otras técnicas como la ejecución simbólica son frecuentemente usadas con buenos resultados.

F-Soft [IYG+05] realiza un análisis de dataflow para computar rangos para las variables enteras y los punteros bajo la hipótesis de que las corridas tienen un largo acotado. El análisis de rango de F-Soft también tiene como objetivo reducir el número de variables proposicionales de la fórmula SAT resultante.

Saturn [XA07] comprime el tamaño de las fórmulas utilizando múltiples optimizaciones (como program slicing, por ejemplo). Saturn construye una sola fórmula SAT que es derivada al SAT-solver. En el nivel intraprocedural, el código secuencial es fielmente modelado (no se usan abstracciones), pero a diferencia de este trabajo, en el nivel interprocedural reemplaza llamadas a funciones con resúmenes generados automáticamente.

JForge [DYJ08] utiliza un análisis de dataflow para encontrar y eliminar ramas lógicamente no viables. La traducción del procedimiento a lógica de JForge utiliza una técnica de ejecución simbólica [Kin76], lo que significa que no se modela explícitamente un control flow graph.

En [TSJ06], los autores proponen un análisis basado en ejecución simbólica para inferir resúmenes de métodos para verificación modular.

En [SGKP10] se propone una técnica basada en análisis de dataflow (variable-definitions) para dividir el problema SAT en distintos sub-problemas.

En [BKS08], los autores reportan sobre el uso de análisis de dataflow y el model checker SAT-ABS [CKSY05]. En este caso, el análisis de dataflow es usado para sobre aproximar una relación de dependencia durante el análisis de concurrencia en programas de SystemC.

Capítulo 2

Herramientas preliminares

2.1. Verificación formal

Asegurar el buen funcionamiento de un sistema de software es una tarea compleja. El desarrollo de cualquier sistema no está completo sin verificar rigurosamente que la implementación es consistente con sus especificaciones. A lo largo de los últimos treinta a cuarenta años se han utilizado distintos enfoques con diversos grados de éxito [DHR+07]. El testing, para el cual hay muchos acercamientos, demostró ser útil para encontrar los errores más comunes. Sin embargo, aplicaciones comerciales que atraviesan estrictos procesos de testing son lanzadas al mercado con decenas de bugs. En las famosas palabras de Edsger Dijkstra: "el testing puede demostrar de manera convincente la presencia de bugs, pero nunca puede demostrar su ausencia" [Dij88].

La verificación formal consiste en probar o refutar la correctitud de un programa con respecto a una especificación del comportamiento esperado del mismo, utilizando métodos formales. La verificación formal automática es uno de los tópicos más activos en ciencias de la computación. De la misma forma que un compilador tiene la función de generar código ejecutable a partir de un programa, un verificador es responsable de chequear si un programa cumple con su especificación para todas las entradas, en todos los caminos posibles de ejecución.

Está probado que, dejando de lado hipótesis de finitud del epacio de estados posibles de programa, encontrar todos los posibles errores de ejecución, o más generalmente cualquier tipo de violación de una especificación en el resultado final de un programa, es indecidible: no hay un método efectivo que pueda responder siempre si un programa dado exhibe o no errores de ejecución [Tur37]. A pesar de esto, y al igual que con todos los problemas indecidibles, en la práctica igual se pueden dar respuestas aproximadas útiles.

2.2. TACO

La verificación acotada [DYJ08] es una técnica en la cual todas las ejecuciones de un procedimiento son examinadas exhaustivamente dentro de un espacio finito dado por una cota en el tamaño del heap, llamada *scope*, y un número de *loop unrolls*. El *scope* es el límite

del tamaño del dominio de datos durante el análisis. Por ejemplo, si estamos analizando una especificación de listas simplemente enlazadas que contienen objetos de tipo Data, el scope acota el número de objetos de tipo List, Node y Data a ser usados durante el análisis (para ilustrar, un scope posible sería 1 objeto List, 10 Node y 10 Data). *Loop unrolling* es una técnica que nos permite manejar ciclos, transformándolos en una secuencia de condicionales cuya cantidad es acotada por un límite provisto por el usuario. De esta forma, el análisis (estático) encontrará bugs que puedan ocurrir ejecutando hasta ese límite de iteraciones en tiempo de ejecución. Cabe mencionar que una interacción ocurre entre el scope y el número de loop unrolls. Esta es una situación normal bajo estas restricciones, e interacciones similares ocurren en otras herramientas [DCJ06][DVT07].

El scope de análisis es examinado en búsqueda de una traza de ejecución que viole la especificación dada. Si un modelo que exhiba un contraejemplo no es encontrado, la fórmula todavía podría tener un modelo en un scope más grande. Sin embargo, la utilidad del análisis se basa en que si hay un modelo para una fórmula que exhibe un contraejemplo, probablemente haya uno con pocos átomos que también lo exhiba [JD96].

Java Modeling Language (JML) [FLL+02] es un lenguaje de especificación de comportamiento para Java inspirado en la metodología de programación de diseño por contrato. Las anotaciones JML permiten a los programadores definitir el contrato de un método (usando construcciones como *requires, ensures, assignable, signals*, etc.), e invariantes. Un contrato puede incluir comportamiento normal y comportamiento excepcional, cuando se lanza una excepción.

TACO (Translation of Annotated COde) [Gal10] es una herramienta que traduce código Java con anotaciones JML a un problema SAT, utilizando Alloy como lenguaje intermedio. En la figura 2.1 podemos ver una representación de las distintas etapas involucradas en el proceso de traducción, para comprender cómo interactúan las herramientas presentadas en este capítulo. Como se puede apreciar, al iniciar la traducción se especifican tanto el scope como el número de loop unrolls. En el capítulo 3 veremos los detalles del funcionamiento de TACO.

2.3. Alloy

Alloy [Jac02] es un lenguaje de especificación formal, basado en lógica de primer orden [Jac00], que pertenece a la clase de los métodos formales referidos como orientados a modelos. Alloy está definido en términos de una semántica relacional simple. Su sintaxis incluye construcciones presentes frecuentemente en notaciones orientadas a objetos, y posee capacidades de análisis automático, que han hecho de Alloy una atractiva herramienta.

Alloy tiene sus raíces en el lenguaje de especificación Z [Spi89], aunque su sintaxis tiene más en común con lenguajes como OCL (Object Constraint Language) [Jac06]. Sin embargo, a diferencia de Z, fue diseñado con el objetivo de hacer a las especificaciones analizables automáticamente. Para esto fue desarrollado el Alloy Analyzer, que provee chequeo y simulación totalmente automáticos. El Alloy Analyzer se basa en SAT-solvers preexistenes, y busca exhaustivamente contraejemplos dentro de un scope finito de análisis.

La representación de sistemas de Alloy está basada en modelos abstractos. Estos modelos son definidos esencialmente en términos de dominios de datos, y operaciones entre estos do-

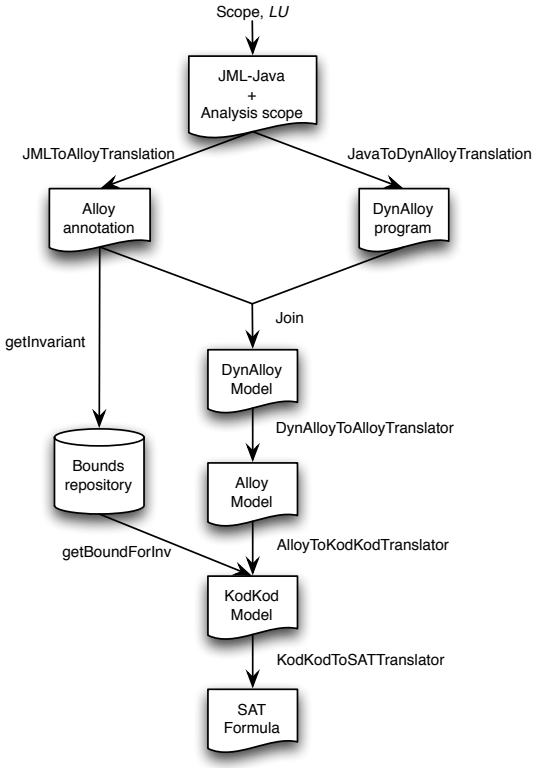


Figura 2.1: Proceso de traducción de TACO.

minios. En particular, se pueden usar dominios de datos para especificar el espacio de estados de un sistema o componente, y utilizar operaciones como un medio para la especificación de cambio de estado. En la figura 2.2 podemos ver una breve descripción de la sintaxis y gramática de Alloy.

En Alloy, la existencia de conjuntos disjuntos de átomos se especifica usando *signaturas*, que definen dominios de datos y sus estructuras. Alloy permite la definición de signaturas como subconjuntos del conjunto denotado por otra signatura "padre": esto es hecho mediante lo que se llama extensión de signatura. Los atributos de una signatura denotan relaciones. Por ejemplo, el atributo p de una signatura M puede representar una relación binaria de átomos de M a átomos de otra signatura N. Dado un conjunto m (no necesariamente singleton) de átomos de M, $m.p$ denota la imagen relacional de m bajo la relación denotada por p. Esto lleva a una visión relacional de la notación de punto, que es simple y elegante, y preserva la lectura intuitiva de la navegación, como en orientación a objetos.

Además de especificar la estructura de los dominios de datos, por supuesto podemos también definir *operaciones* sobre los dominios definidos. Siguiendo el estilo de las especificaciones de Z, las operaciones en Alloy pueden ser definidas como expresiones, relacionando estados de los espacios de estados descritos por las definiciones de signaturas. Como cabe esperar, un modelo puede ser mejorado agregándole *propiedades* (axiomas). Estas propiedades son escritas como fórmulas lógicas, de manera muy similar al estilo de OCL. Las propiedades o restricciones en Alloy son definidas como *hechos*. Restricciones complejas pueden ser expresadas usando el considerable poder expresivo de la lógica relacional. Las *aserciones* son las

problem ::= decl*formula
decl ::= *var* : *typexpr*
typexpr ::=
type
| *type* → *type*
| *type* ⇒ *typexpr*

formula ::=
expr *in* expr (subset)
|!formula (neg)
| formula && formula (conj)
| formula || formula (disj)
| *all* *v* : *type* | formula (univ)
| *some* *v* : *type* | formula (exist)

expr ::=
expr + expr (union)
| expr & expr (intersection)
| expr − expr (difference)
|∼ expr (transpose)
| expr.expr (navigation)
| +expr (transitive closure)
| {*v* : *t* | formula} (set former)
| *Var*

Var ::=
var (variable)
| *Var*[*var*] (application)

M : formula → env → $Boolean$
X : expr → env → $value$
$env = (var + type) \to value$
$value = (atom \times \cdots \times atom)+$
$(atom \to value)$

$M[a\ in\ b]e = X[a]e \subseteq X[b]e$
$M[!F]e = \neg M[F]e$
$M[F\&\&G]e = M[F]e \wedge M[G]e$
$M[F\ ||\ G]e = M[F]e \vee M[G]e$
$M[all\ v : t \mid F] =$
$\bigwedge\{M[F](e \oplus v \mapsto \{\, x \,\})/x \in e(t)\}$
$M[some\ v : t/F] =$
$\bigvee\{M[F](e \oplus v \mapsto \{\, x \,\})/x \in e(t)\}$

$X[a + b]e = X[a]e \cup X[b]e$
$X[a\&b]e = X[a]e \cap X[b]e$
$X[a - b]e = X[a]e \setminus X[b]e$
$X[\sim a]e = \{\, \langle x, y\rangle : \langle y, x\rangle \in X[a]e \,\}$
$X[a.b]e = X[a]e; X[b]e$
$X[+a]e =$ the smallest r such that
$r; r \subseteq r$ and $X[a]e \subseteq r$
$X[\{v : t \mid F\}]e =$
$\{x \in e(t) \mid M[F](e \oplus v \mapsto \{\, x \,\})\}$
$X[v]e = e(v)$
$X[a[v]]e = \{\langle y_1, \ldots, y_n\rangle \mid$
$\exists x.\, \langle x, y_1, \ldots, y_n\rangle \in e(a) \wedge \langle x\rangle \in e(v)\}$

Figura 2.2: Gramática y semántica de Alloy.

propiedades deseadas de un modelo dado, que son usadas para validar especificaciones. Usando el Alloy Analyzer, es posible validar aserciones buscando posibles contraejemplos finitos, bajo las restricciones impuestas por la especificación del sistema.

Por último, cabe mencionar que el Alloy Analyzer es un *model finder*. Dada una fórmula lógica en lenguaje Alloy, intenta encontrar un modelo (una asignación de variables a valores) que satisfaga la fórmula. El Alloy Analyzer permite hacer dos tipos de verificación: simulación (a través del comando `run`) y chequeo (comando `check`). Una simulación permite ver los ejemplos (si los hay) que el Analyzer encuentra para los cuales el predicado a analizar es verdadero. Un chequeo es usado para encontrar contraejemplos: soluciones que satisfacen la especificación pero violan una aserción. Para esto último, si se quiere comprobar que el sistema descrito por la propiedad P posee la propiedad Q, el Alloy Analyzer niega la aserción $P \implies Q$ y busca un modelo de $P \wedge \neg Q$, que si es encontrado, representa un contraejemplo de la aserción.

2.4. Kodkod

Alloy tiene como *backend* a Kodkod, un model finder basado en SAT [TJ07]. Una característica distintiva de Kodkod es que permite definir instancias parciales en las especificaciones. Luego son aplicadas técnicas de optimización en la tradución de lógica relacional a booleana, aprovechando las instancias parciales provistas por el usuario, y un algoritmo general de ruptura de simetrías que funciona en presencia de instancias parciales arbitrarias [Tor09].

Esta característica de permitir la utilización de instancias parciales es esencial para nuestro trabajo, ya que las cotas calculadas a partir del análisis de dataflow son justamente inyectadas como tales. Es por esto que, como veremos más adelante, el dominio de nuestro análisis son las relaciones de Alloy. Las cotas declaradas para estas son aprovechadas para optimizar la traducción a lógica proposicional, simplificando el problema SAT resultante.

2.5. DynAlloy

DynAlloy es una extensión de Alloy para describir propiedades dinámicas de sistemas usando acciones [FGLPA05]. Como se puede apreciar en la figura 2.1, es un lenguaje intermedio utilizado por TACO en la traducción a Alloy. La razón detrás de esta extensión es que se requieren acciones para representar cambios de estados (es decir, para describir las relaciones entre datos de entrada y salida). Al contrario del uso de predicados para este propósito, las acciones tienen un significado de entrada/salida reflejado en su semántica, y pueden ser compuestas para formar acciones más complejas, usando construcciones conocidas de lenguajes imperativos de programación. Vale la pena mencionar que tanto la sintaxis como la semántica de DynAlloy fueron fuertemente motivadas por la lógica dinámica [HKT00], y la conveniencia de esta para expresar aserciones de correctitud parcial.

En la figura 2.3 podemos ver un fragmento relevante de la gramática de DynAlloy. Este fragmento corresponde a los programas DynAlloy que son la salida del traductor *JavaToDynAlloy* en TACO. DynAlloy extiende a Alloy al permitir al usuario escribir aserciones de correctitud parcial de la forma {formula}*program*{formula}. Vale la pena notar que las

construcciones típicas de programación estructurada pueden ser descritas usando estas construcciones lógicas básicas. Por ejemplo, `if B then P else Q fi` puede ser escrito como el programa DynAlloy $B?; P + (\neg B)?; Q$. De manera similar, `while B do P od` se puede expresear como $(B?; P)^*; (\neg B)?$.

formula	::=	... | {formula} *program* {formula}	"partial correctness"
program	::=	$v := expr$	"copy"
	|	$v.f := expr$	"store"
	|	*skip*	"skip action"
	|	$formula?$	"test"
	|	$program + program$	"non-deterministic choice"
	|	$program; program$	"sequential composition"
	|	$program^*$	"iteration"
	|	$\langle program \rangle(\overline{x})$	"invoke program"
expr	::=	**null**	"null value"
	|	v	"variable"
	|	$v.f$	"field access"

Figura 2.3: Fragmento de la gramática de DynAlloy.

Ya que el diseño de Alloy fue profundamente influenciado por la intención de producir un lenguaje automáticamente analizable, DynAlloy también fue diseñado con el objetivo de que sus especificaciones puedan ser analizadas automáticamente. La traducción de aserciones de correctitud parcial a fórmulas de primer orden de Alloy es realizada por DynAlloy aplicando "weakest liberal precondition" [DS90]. Las fórmulas de Alloy generadas, que pueden ser largas y bastante difíciles de comprender, no son visibles al usuario final, que sólo accede a la especificación declarativa de DynAlloy.

Es justamente al nivel de DynAlloy que se realiza nuestro análisis de dataflow, debido principalmente a dos motivos. Por un lado, su estructura similar a cualquier lenguaje imperativo de programación lo hace adecuado para definir e implementar el análisis en cada una de sus operaciones básicas. Por el otro, su cercanía a Alloy, que es en definitiva el nivel en el que se basa el dominio del análisis, reduce la complejidad de las traducciones necesarias para pasar de uno al otro, sin tener que trabajar directamente al nivel de la lógica relacional.

2.6. Análisis estático de programas

El análisis estático de programas ofrece técnicas de tiempo de compilación para predecir aproximaciones seguras y computables para el conjunto de valores o comportamientos que surgen dinámicamente al ejecutar un programa [NNH99]. Una de sus aplicaciones principales es permitir a los compiladores generar código para evitar computaciones redundantes, por ejemplo reusando resultados disponibles, o evitando computaciones superfluas, por ejemplo de resultados que se sabe no son necesarios o son ya conocidos en tiempo de compilación. Dentro de las aplicaciones más recientes está la validación de software, para reducir la posibilidad de

comportamiento indeseado [NNH99]. Común a estas aplicaciones es la necesidad de combinar información de diferentes partes del programa. En esta sección hacemos sólo una somera introducción al tema, pues veremos más adelante todos los detalles pertinentes.

Una cuestión recurrente detrás de todos los enfoques del análisis de programas es que para que sean computables, sólo pueden proporcionar respuestas aproximadas. Por ejemplo, consideremos un lenguaje simple de statements y el siguiente programa, donde `S` es un statement que no contiene una asignación a `y`:

```
read(x); (if x>0 then y:=1 else (y:=2; S)); z:=y
```

Intuitivamente, los valores de `y` que llegan a `z:=y` serán 1 ó 2. Ahora, supongamos que un análisis afirma que el único valor de `y` que alcanza `z:=y` es 1. Mientras que esto parece intuitivamente incorrecto, es de hecho correcto en el caso donde `S` no termina para $\texttt{x} \leq 0$ e $\texttt{y}=2$. Pero ya que no es *decidible* en el caso general si `S` termina o no [Tur37], normalmente no esperamos que nuestro análisis intente detectar esta situación. Así que en general, esperamos que el análisis de programa produzca un conjunto más grande (sobre aproximación) de posibilidades de las que sucederían durante la ejecución del programa. Esto significa que es aceptable un análisis que afirma que los valores de `y` que alcanzan `z:=y` están entre 1, 2 ó 42, aunque claramente preferimos uno que dé la respuesta más ajustada: que los valores están entre 1 ó 2.

Esta noción de aproximación segura se puede ilustrar pensando que si la solución exacta es el conjunto de valores $\{d_1, \ldots, d_n\}$, donde el conjunto de todos los valores posibles es $\{d_1, \ldots, d_N\}$, con $N > n$, entonces una respuesta segura es de la forma $\{d_1, \ldots, d_{n+m}\}$, con $m \geq 0$ y $n + m \leq N$. Nótese que aunque el análisis no dé información totalmente precisa, puede igualmente dar información útil: saber que el valor de `y` es uno entre 1, 2 y 42 antes de la asignación `x:=y` todavía nos dice que `z` es positivo y que `z` entra en 1 byte de almacenamiento, por ejemplo.

2.6.1. Dataflow

El análisis de dataflow, contenido dentro del análisis estático de programas, está compuesto por técnicas que derivan información acerca de los posibles conjuntos de valores calculados en los distintos puntos de un programa [ALSU06]. *Reaching definitions*, uno de los ejemplos canónicos del análisis de dataflow, fue el que intuitivamente usamos en la sección 2.6. La ejecución de un programa puede ser vista como una serie de transformaciones de los valores de todas las variables del programa. Cada ejecución de una operación intermedia de código transforma un estado de entrada en un nuevo estado de salida.

Un grafo de flujo de control (*control flow graph*, o *CFG*) es una representación, usando notación de grafos, de todos los caminos que pueden ser recorridos en un programa durante su ejecución. En análisis de dataflow, este es usado para determinar las partes del programa a las cuales un valor particular asignado a una variable puede propagarse. Cuando analizamos el comportamiento de un programa, debemos considerar todas las posibles trazas de ejecución (o caminos) a través del grafo de flujo que la ejecución del programa puede tomar. Entonces se extrae, de todos los posibles estados de programa en cada punto, la información que se necesita para el problema del análisis de dataflow particular que se quiere resolver. En los

análisis más complejos, debemos considerar los caminos que saltan entre los distintos grafos de flujo de varios procedimientos, a medida que se ejecutan instrucciones de *call* y *return*.

Capítulo 3

TACO

3.1. Introducción

TACO introduce dos técnicas para optimizar el tiempo de análisis de verificación acotada basada en SAT [GRLPF10]. Primero, fuerza una representación canónica del heap de memoria de Java removiendo simetrías de referencias de objetos. Segundo, y como una consecuencia de la canonización del heap, determina de antemano el valor de verdad de una substancial proporción de variables proposicionales, que pueden ser reemplazadas por este valor, dando como resultado un problema SAT más simple.

La idea intuitiva detrás de nuestro análisis es simple: por medio de un análisis de dataflow, y utilizando las cotas iniciales provistas por TACO, inferimos el conjunto de valores posibles que pueden ser asignados a todas las variables DynAlloy (que representan tanto variables como campos de Java), en cada punto del programa, siguiendo el grafo de flujo de control de este. Esta información es una aproximación segura de los valores concretos que cada variable puede tener. Este análisis de propagación de valores es aplicado en el contexto de verificación acotada, donde es posible utilizar abstracciones ajustadas sin comprometer la terminación.

Si bien en el capítulo 4 veremos los detalles de la formalización del análisis, y en el capítulo 5 cómo esto es llevado a la práctica, veamos primero un ejemplo que ilustre todo esto. El objetivo es comprender los detalles involucrados en el proceso de traducción y la motivación detrás de nuestro análisis.

3.2. Overview

Como mencionamos en la sección 2.2, TACO es una herramienta para el análisis de código Java basada en SAT-solving. Dado un scope limitando el tamaño del dominio de objetos, y un límite para el número de iteraciones de los loops, TACO traduce un programa con anotaciones JML a una fórmula proposicional. Luego, el problema SAT consiste en, dada una fórmula proposicional φ, encontrar una valuación que la satisfaga si es que existe, o en devolver *unsat* si φ es insatisfacible. Si no se encuentra una valuación que satisfaga φ, se concluye que la especificación fuerza la propiedad dentro del scope de análisis. Si por el contrario se encuentra una valuación, una traza que viola la especificación es reportada.

Sigamos este proceso a través de un ejemplo que nos permita comprender las distintas etapas involucradas. La figura 3.1 muestra la implementación de una lista simplemente encadenada. Una lista contiene un campo `head` que devuelve su primer nodo, y cada nodo apunta a su siguiente a través del campo `next`.

```
class Node {
   Node next;
}
class List {
   Node head;
}
```

Figura 3.1: Declaración de una lista simplemente encadenada.

El contenedor `List` está anotado con el invariante de objeto JML mostrado en la figura 3.2. La construcción `\reach(l, T, f)` denota el conjunto de objetos de tipo `T` alcanzables desde una ubicación `l` usando el campo `f`. Esta anotación restringe las estructuras de lista válidas a aquellas que forman una secuencia finita acíclica de elementos de tipo `Node`.

```
/*@ invariant (\forall Node n ;
  @ \reach(this.header, Node, next).has(n);
  @ !\reach(n.next, Node, next).has(n));
  @*/
```

Figura 3.2: Invariante de la lista simplemente encadenada.

El método `removeLast`, cuya implementación podemos ver en la figura 3.3a, elimina el último elemento de la lista, si es que este existe. JML permite la escritura de especificaciones parciales. En este ejemplo, la cláusula de JML `ensures` especifica sólo que el objeto `Node` devuelto no debe ser aclanzable desde la lista.

Como mencionamos en la sección 2.2, TACO traduce el código Java anotado con el contrato JML a una especificación DynAlloy. Como DynAlloy es un lenguaje de especificación relacional, cada variable en DynAlloy puede ser vista como una relación de aridad fija. En la figura 3.3b podemos ver la representación del método `removeLast`, y en la figura 3.4 las signaturas y variables que son introducidas.

Ya que DynAlloy es una extensión procedural de Alloy, DynAlloy hereda la semántica de Alloy. Las signaturas definen conjuntos disjuntos de átomos. El modificador `one` en la signatura `null` la restringe de tal manera que posea un único elemento. Este átomo distinguido modela el valor de Java *null*. Los campos y variables de Java son representados usando variables DynAlloy. Los campos de Java son modelados en DynAlloy como variables binarias funcionales (es decir, `S ->one T`), y las variables de Java son modeladas como variables unarias no vacías (es decir, `one S`).

Si queremos realizar una verificación acotada del contrato de `removeLast`, debemos limitar el dominio de objetos y la cantidad de iteraciones de los loops. Para este ejemplo, asumamos que utilizamos un scope que permite hasta 5 objetos `Node`, 1 objeto `List`, y 3 loop unrolls.

La especificación DynAlloy es entonces traducida a una especificación Alloy, como se describe en [FGLPA05]. Para poder modelar cambios de estado en Alloy, dada su naturaleza

```
/*@
  @ ensures !\reach(this.header,
      Node, next).has(\result);
  @*/
Node removeLast() {
  if (this.head != null) {
    Node prev = null;
    Node curr = this.head;
    while (curr.next != null) {
      prev = curr;
      curr = curr.next;
    }
    if (prev == null)
      this.head = null;
    else
      prev.next = null;
    return curr;
  } else
    return null;
}
```

(a) Implementación Java.

```
(this.head!=null)?;{
  prev := null;
  curr := this.head;
  {
    (curr.next!=null)?;
    prev = curr;
    curr = curr.next
  }*;
  ((curr.next==null)?;
  (prev==null)?;
    this.head := null
  +
  (prev!=null)?;
    prev.next := null;
  );
  return := curr;
}+
(this.head==null)?;
  return := null
```

(b) Representación en DynAlloy.

Figura 3.3: Algoritmo para eliminar el último elemento.

```
one sig null {}
sig List {}
sig Node {}
head: List -> one (Node+null)
next: Node -> one (Node+null)
this: one List
prev: one Node+null
curr: one Node+null
return: one Node+null
```

Figura 3.4: Signaturas y variables de DynAlloy para el método `removeLast`.

estática, el traductor *DynAlloyToAlloy* puede introducir varias relaciones Alloy para representar distintos valores (o encarnaciones) de la misma variable DynAlloy, en el estilo de SSA (Static Single Assignment) [App98].

Para traducir la especificación Alloy a un problema SAT, el Alloy Analyzer traduce cada variable relacional como un conjunto de variables proposicionales. Cada variable proposicional tiene por objeto modelar que una tupla dada está contenida en la relación Alloy. En nuestro ejemplo, ya que el dominio de `Node` está restringido a 5 elementos, $\{N_1, \ldots, N_5\}$ es el conjunto de los 5 átomos de tipo `Node` existentes. Esto lleva al modelado de la relación binaria $\texttt{next}_0$ (que a su vez, modela el estado inicial[1] de la variable DynAlloy `next`) con las variables proposicionales que se muestran en la figura 3.1.

M_{next_0}	N_1	N_2	N_3	N_4	N_5	$null$
N_1	p_{N_1,N_1}	p_{N_1,N_2}	p_{N_1,N_3}	p_{N_1,N_4}	p_{N_1,N_5}	$p_{N_1,null}$
N_2	p_{N_2,N_1}	p_{N_2,N_2}	p_{N_2,N_3}	p_{N_2,N_4}	p_{N_2,N_5}	$p_{N_2,null}$
N_3	p_{N_3,N_1}	p_{N_3,N_2}	p_{N_3,N_3}	p_{N_3,N_4}	p_{N_3,N_5}	$p_{N_3,null}$
N_4	p_{N_4,N_1}	p_{N_4,N_2}	p_{N_4,N_3}	p_{N_4,N_4}	p_{N_4,N_5}	$p_{N_4,null}$
N_5	p_{N_5,N_1}	p_{N_5,N_2}	p_{N_5,N_3}	p_{N_5,N_4}	p_{N_5,N_5}	$p_{N_5,null}$

Cuadro 3.1: Variables proposicionales que modelan $\texttt{next}_0$.

Siguiendo esta representación, la variable proposicional p_{N_2,N_3} es verdadera si y sólo si la tupla $\langle N_2, N_3 \rangle$ está contenida en la relación Alloy $\texttt{next}_0$, o lo que es lo mismo, $\texttt{N}_2\texttt{.next}_0 = \texttt{N}_3$. Dado el scope asumido de análisis, si no hay ningún preprocesamiento involucrado, la fórmula proposicional resultante contendrá 145[2] variables proposicionales. Sólo 36 variables[3], el 25 %, modelan el estado inicial. Las restantes 109, el 75 %, son introducidas para representar los estados intermedios y finales del programa durante la ejecución del método.

Recordemos que como mencionamos en la sección 2.4, Alloy utiliza Kodkod como lenguaje intermedio, y este es traducido luego a una fórmula proposicional en forma normal conjuntiva (CNF). Kodkod permite definir cotas (instancias parciales) para las relaciones Alloy. Por cada relación f, dos instancias relacionales L_f (la cota inferior) y U_f (la cota superior) son incluidas. En todo modelo Alloy `M`, `f` (la interpretación de la relación f en el modelo `M`), debe cumplir $L_f \subseteq \texttt{f} \subseteq U_f$. Entonces, pares que están en L_f deben necesariamente pertenecer a `f`, y pares que no están en U_f no pueden pertenecer a `f`. Si se remueven tuplas de la cota superior, decimos que la cota superior resultante es más ajustada que antes.

Cotas superiores más ajustadas contribuyen removiendo variables proposicionales. Dada una relación Alloy `f`, las variables proposicionales que corresponden a tuplas que no pertenecen a U_f pueden ser directamente reemplazadas en el proceso de traducción con el valor de verdad *false*. Reemplazar estas variables no altera la validez del problema SAT (ya que tuplas que no pertenecen a U_f no pueden pertenecer a `f`), y nos permite reducir el número de variables proposicionales. Dado que, en el peor caso, la complejidad de SAT es exponencial en el número

[1] Cuando hablamos de estado inicial del programa, nos referimos al conjunto formado por todas las encarnaciones iniciales de las relaciones que lo componen. De manera análoga nos referimos a los estados intermedios y finales del programa.

[2] La relación *head* tiene 2 encarnaciones con 6 tuplas cada una, *next*: 2 encarnaciones de 30 tuplas, *this*: 1 encarnación con 1 tupla, *prev* y *curr*: 5 encarnaciones cada una, con 6 tuplas, *ret*: 1 encarnación con 6 tuplas.

[3] TACO calcula cotas para las encarnaciones iniciales de *head* y *next*.

de variables proposicionales que existen en la fórmula, es esperable que una reducción en su cantidad lleve a una mejora del tiempo de análisis.

Dado un invariante de clase y el scope de verificación deseado, TACO es capaz de sintetizar automáticamente una cota superior más ajustada para el estado inicial. En nuestro ejemplo, sólo listas acíclicas son permitidas debido a la anotación de invariante. Esto hace que, para cualquier modelo Alloy, ninguna tupla de la forma $\langle N_i, N_i \rangle$ pertenezca a la relación $\texttt{next}_0$. Para cada tupla en el scope de verificación, TACO crea una fórmula Alloy y evalúa su validez. Si la fórmula es válida, entonces la tupla es inviable dentro del actual invariante de clase, y puede ser eliminada de la cota superior.

Más tuplas pueden ser inviables debido a restricciones de simetría. Una simetría es una permutación de los átomos de la signatura que no altera el valor de verdad de la fórmula Alloy. Una forma de evitar considerar todas las valuaciones isomorfas de un modelo Alloy es introduciendo predicados de ruptura de simetrías. Como esto puede contribuir a una reducción significativa de los tiempos de SAT-solving [CGLR96], Kodkod incluye sus propios predicados de ruptura de simetría de propósito general [TJ07]. TACO agrega más restricciones de simetría excluyendo todas las valuaciones isomorfas [GRLPF10]. Esto es posible debido al hecho de que estamos tratando con una representación Alloy del heap de memoria inicial de Java. Siguiendo el ejemplo, podemos tener las siguientes instancias de Alloy isomorfas:

$$\text{this} = L_0 \overset{\text{header}_0}{\rightarrow} N_0 \overset{\text{next}_0}{\rightarrow} N_1 \overset{\text{next}_0}{\rightarrow} N_2$$

$$\text{this} = L_0 \overset{\text{header}_0}{\rightarrow} N_1 \overset{\text{next}_0}{\rightarrow} N_2 \overset{\text{next}_0}{\rightarrow} N_0$$

$$\text{this} = L_0 \overset{\text{header}_0}{\rightarrow} N_2 \overset{\text{next}_0}{\rightarrow} N_0 \overset{\text{next}_0}{\rightarrow} N_1$$

$$\vdots$$

Los predicados de ruptura de simetría de TACO van a prevenir todas las valuaciones, salvo la primera instancia. Por lo tanto, tuplas como $\langle N_1, N_0 \rangle$ y $\langle N_2, N_1 \rangle$ no van a pertenecer a ninguna valuación Alloy válida de la relación $\texttt{next}_0$. A su vez, el análisis de viabilidad de estas tuplas las removerá de la cota superior. Este preprocesamiento le permite a TACO remover en este ejemplo el 70 % de las variables proposicionales que representan el estado inicial. En la figura 3.5 podemos ver las cotas iniciales calculadas por TACO, expresadas como restricciones en la sintaxis de Alloy. Para cada relación, se listan las tuplas que pueden pertenecer a ella (es decir, la cota superior).

Estas cotas son almacenadas por TACO en un repositorio (ver figura 2.1), y ya que son frecuentemente reusadas durante el análisis de diferentes métodos en una clase, el costo de computar estas cotas es amortizado. Por otra parte, la viabilidad de una tupla no depende de la viabilidad de otras, lo cual permite el cálculo en paralelo de cotas, que puede producir ganancias de rendimiento significativas.

3.3. Planteo del problema

La técnica utilizada por TACO [GRLPF10] se limita a acotar las variables proposicionales que representan el estado inicial del programa analizado. Ya que el SAT-solver no es capaz de

```
fact {
  next_0 in N0->null
  +N1->null
  +N2->null
  +N3->null
  +N4->null
  +N0->N1
  +N1->N2
  +N2->N3
  +N3->N4
}
```

```
fact {
  head_0 in L0->null
  +L0->N0
}
```

Figura 3.5: Cotas iniciales calculadas por TACO.

reconocer el orden del flujo del programa, no hay garantías de que el proceso de SAT-solving evite valuaciones parciales de resultados intermedios que no llevan a trazas de ejecución válidas.

Usando análisis de dataflow y las cotas iniciales, podemos calcular los conjuntos de valores posibles en los distintos puntos del programa. Para ilustrar esto, y siguiendo con nuestro ejemplo, si sabemos que el único valor posible para `this` es $\{L_0\}$, y que los valores posibles para `head` son $\{\langle L_0, null\rangle, \langle L_0, N_1\rangle\}$, entonces, a partir del statement `Node curr = this.head;` podemos concluir que el conjunto de valores posibles de `curr` es $\{null, N_1\}$. Las reglas precisas del flujo de información las veremos en el capítulo 4.

El análisis de dataflow propuesto en este trabajo sobre aproxima (análisis *may*) el conjunto de los posibles valores que cada variable y campo de Java puede almacenar, dentro del scope de análisis especificado. Creemos que las cotas superiores más ajustadas que resultan de este análisis deberían contribuir al proceso de verificación al permitirle a Kodkod eliminar variables proposicionales. Para ilustrar esto, en el contexto de nuestro ejemplo, de las 109 variables proposicionales restantes que no representan el estado inicial, podemos eliminar 78, lo que representa el 71 %. En la figura 3.6 podemos ver el detalle de las cotas calculadas por el análisis de dataflow. En el capítulo 6 veremos el impacto que esto produce en la performance de la verificación.

```
fact {
  head_1 in L0->null
  +L0->N0
}
fact {
  prev_1 in null
}
fact {
  curr_1 in null
  +N0
}
fact {
  prev_2 in null
  +N0
}
fact {
  curr_2 in null
  +N1
}
fact {
  prev_3 in null
  +N1
}
fact {
  curr_3 in null
  +N2
}
```

```
fact {
  next_1 in N0->null
  +N1->null
  +N2->null
  +N3->null
  +N4->null
  +N0->N1
  +N1->N2
  +N2->N3
  +N3->N4
}
fact {
  prev_4 in null
  +N0
  +N1
  +N2
}
fact {
  curr_4 in null
  +N0
  +N1
  +N2
  +N3
}
```

Figura 3.6: Cotas calculadas por nuestro análisis de dataflow.

Capítulo 4

Análisis de dataflow para el cómputo de cotas

Dado un programa DynAlloy, nuestro análisis calcula una sobre aproximación de todas las posibles asignaciones a variables para cada punto del programa. Como anticipamos en la sección 2.5, DynAlloy es el lenguaje en el que se basa nuestro análisis de dataflow. Esta decisión responde a dos motivos. Por un lado, DynAlloy es mucho más cercano al problema SAT que la representación de Java. Por el otro, es la última representación imperativa del código analizado, donde las nociones de flujo de control y cambio de estado todavía existen.

4.1. Semántica concreta

Comenzamos definiendo la semántica concreta para la ejecución del fragmento de DynAlloy que simula la ejecución de programas Java, presentado en la figura 2.3. Como la semántica relacional de DynAlloy es interpretada en términos de átomos, *Atom* representa el conjunto de todos los átomos en esta interpretación.

Notamos con $JVar \uplus JField$ el conjunto de variables de DynAlloy. Una variable DynAlloy que pertenece a $JVar$ corresponde a la representación de una variable de Java, y su *valor concreto* es un solo átomo. De manera similar, una variable que pertenece a $JField$ modela un campo de Java, cuyo *valor concreto* es un mapping (relación funcional) de átomos en átomos.

La semántica relacional de DynAlloy está definida en [FGLPA05]. Aquí presentamos una definición alternativa basada en *collecting semantics*, que es útil para probar la corrección de la técnica propuesta de dataflow.

Una collecting semantics define cómo la información fluye a través del grafo de flujo de control (CFG) del programa. El CFG describe la estructura del programa. En la collective semantics, cada vez que un nuevo valor atraviesa un nodo, es almacenado. Entonces, cada nodo mantiene la información de los valores que pasaron a través de él.

Definición 1. *Sea* $E = JVar \uplus JField \to Atom \cup \mathcal{P}(Atom \times Atom)$. *Un estado concreto* $c \in E$ *relaciona cada variable DynAlloy con un valor concreto.* E *es el conjunto de todos los estados concretos, y llamamos* C *a* $\mathcal{P}(E)$.

Definición 2. *$X[expr]_c$ es la evaluación de la expresión expr en el estado concreto c. El valor de $X[expr]_c$ para las expresiones de DynAlloy que vamos a considerar se define como:*

$$\begin{array}{lcl} X[null]_c & = & \{\langle null \rangle\} \\ X[v]_c & = & \{c(v)\} \\ X[v.f]_c & = & \{c(v)\}; c(f) \end{array}$$

Definición 3. *$M[\phi]_c$ es el valor de verdad de la fórmula ϕ en el estado concreto c.*

Definición 4. *$R; S$ denota la composición de relaciones, definida como:*

$$R; S = \{\langle a_1, \ldots, a_{i-1}, b_2, \ldots, b_j \rangle : \langle a_1, \ldots, a_i \rangle \in R \wedge \langle b_1, \ldots, b_j \rangle \in S \wedge a_i = b_1\}$$

Definición 5. *$R \to S$ es el producto cartesiano de relaciones, definido como:*

$$R \to S = \{\langle a_1, \ldots, a_i, b_1, \ldots, b_j \rangle : \langle a_1, \ldots, a_i \rangle \in R \wedge \langle b_1, \ldots, b_j \rangle \in S\}$$

Definición 6. *$R + +S$ denota el override relacional, definido como:*

$$R + +S = \{\langle a_1, \ldots, a_n \rangle : \langle a_1, \ldots, a_n \rangle \in R \wedge a_1 \notin dom(S)\} \cup S$$

donde, dada una relación T, $dom(T)$ denota el conjunto $\{a_1 : \exists a_2, \ldots, a_n$ tal que $\langle a_1, a_2, \ldots a_n \rangle \in T\}$

Definición 7. *$[\cdot]_0$ es el operador unario que devuelve el único elemento presente en un conjunto singleton: $[\{a\}]_0 = a$.*

Definición 8. *Dado un programa DynAlloy P, el grafo de flujo de control de P (control flow graph, o $CFG(P)$) es una estructura $\langle N, X, entrada, salida \rangle$, donde N es un conjunto de acciones atómicas anotadas con ubicaciones de programa, $X \subseteq N \times N$ es el conjunto de ejes, y $entrada, salida \subseteq N$ son los statements de entrada y salida, respectivamente.*

En la sección 5.1 veremos los detalles de la construcción del control flow graph.

Definición 9. *Llamamos DS al conjunto de sentencias DynAlloy sobre el que definimos nuestro análisis. $DS = \{skip, \phi?, v := expr, v.f := expr\}$, donde ϕ es una fórmula, v y f variables DynAlloy que pertenecen a $JVar$ y $JField$ respectivamente, y expr una expresión de DynAlloy.*

Definición 10. *Dado un estado concreto $c \in E$, variables $x, y \in JVar \uplus JField$ y un valor $v \in Atom \cup \mathcal{P}(Atom \times Atom)$, definimos la actualización del estado c como*

$$c[x \mapsto v](y) \quad = \quad \begin{cases} v & si\ y = x \\ c(y) & en\ caso\ contrario \end{cases}$$

La definicón de actualización de un estado concreto puede ser extendida naturalmente a un conjunto de estados concretos.

Definición 11. *Dado un conjunto de estados concretos $cs \in C$, una variable $x \in JVar \uplus JField$ y un valor $v \in Atom \cup \mathcal{P}(Atom \times Atom)$, definimos la actualización del conjunto de estados como*

$$cs[x \mapsto v] = \{c[x \mapsto v] \mid c \in cs\}$$

Definición 12. *La función de transferencia concreta* $\mathcal{F}\colon DS \times C \to C$ *se define como:*

$$\begin{array}{lcl} \mathcal{F}(skip, cs) & = & cs \\ \mathcal{F}(\phi?, cs) & = & \{c \mid c \in cs \wedge M[\phi]_c\} \\ \mathcal{F}(v := expr, cs) & = & \{c[v \mapsto [X[expr]_c]_0] \mid c \in cs\} \\ \mathcal{F}(v.f := expr, cs) & = & \{c[f \mapsto c(f) + +(\{c(v)\} \to X[expr]_c)] \mid c \in cs\} \end{array}$$

La función de transferencia $\mathcal{F}$ modela cómo cambia el estado concreto cuando una sentencia DynAlloy es ejecutada.

Definición 13. *Dado un programa DynAlloy* P *y una fórmula* ϕ *representando los posibles estados de entrada, la collecting semantics de* P *comenzando con estado* ϕ *es el menor punto fijo de las siguientes ecuaciones:*

Por cada n *en el grafo de flujo de control de* P *(*$CFG(P)$*)*

$$\begin{array}{rcl} in(n) & = & \begin{cases} \{c_0 \mid M[\phi]_{c_0}\} & n \text{ punto de entrada de } CFG(P) \\ \bigcup\limits_{p \in pred(n)} out(p) & \text{caso contrario} \end{cases} \\ out(n) & = & \mathcal{F}(n, in(n)) \end{array}$$

donde $in(n)$ *y* $out(n)$ *denotan los valores de input y output del nodo* n *respectivamente, y* $pred(n)$ *el conjunto de predecesores de* n *en el CFG.*

4.2. Semántica abstracta

En nuestro enfoque, un *valor abstracto* representa el conjunto de todos los valores concretos que una variable DynAlloy puede tener (es decir, una sobre aproximación) en un punto del programa dado. El valor abstracto de una variable DynAlloy que representa una variable Java es un conjunto de átomos, y el valor abstracto de una variable DynAlloy que modela un campo de Java es una relación (probablemente no funcional) binaria de átomos en átomos.

Definición 14. *Sea* $A = JVar \uplus JField \to \mathcal{P}(Atom) \cup \mathcal{P}(Atom \times Atom)$*. Un estado abstracto* $a \in A$ *relaciona cada variable DynAlloy con su correspondiente valor abstracto.* A *es el conjunto de todos los estados abstractos.*

Para poder operar con esta abstracción, necesitamos que A sea un semi-reticulado con top: un conjunto parcialmente ordenado en donde todo subconjunto finito no vacío tiene un supremo. Definimos entonces la relación de orden $\sqsubseteq$ y el operador de join $\sqcup$:

Definición 15. *Sean* $a, a' \in A$*:*

- $a \sqsubseteq a' \iff \forall x \in JVar \uplus JField\ (a(x) \subseteq a'(x))$
- $a \sqcup a' = a''$ *tal que* $a'' \in A \wedge \forall x \in JVar \uplus JField\ (a''(x) = a(x) \cup a'(x))$

La función de abstracción $\alpha\colon C \to A$ formaliza la noción de aproximación de un estado concreto por un estado abstracto.

Definición 16. *Dado un conjunto de estados concretos $cs \in C$:*

$\alpha(cs) = a$ *tal que* $\forall v \in JVar(a(v) = \{c(v) \mid c \in cs\}) \wedge \forall f \in JField(a(f) = \bigcup_{c \in cs} c(f))$

Definición 17. *La función de concretización $\gamma \colon A \to C$ se define como:*

$\gamma(a) = \bigcup\{cs \mid \alpha(cs) \sqsubseteq a\}$

Definición 18. *$\hat{X}[expr]_a$ es la evaluación de la expresión expr en el estado abstracto a. $\hat{X}[expr]_a$ se define, de manera análoga a $X[expr]_c$, como:*

$$\begin{array}{lcl} \hat{X}[null]_a & = & \{\langle null \rangle\} \\ \hat{X}[v]_a & = & a(v) \\ \hat{X}[v.f]_a & = & a(v); a(f) \end{array}$$

Definición 19. *Análogamente a la definición 10, dado un estado abstracto $a \in A$, variables $x, y \in JVar \uplus JField$ y un valor $v \in \mathcal{P}(Atom) \cup \mathcal{P}(Atom \times Atom)$, definimos la actualización del estado a como*

$$a[x \mapsto v](y) \quad = \quad \begin{cases} v & \text{si } y = x \\ a(y) & \text{en caso contrario} \end{cases}$$

Definición 20. *La función de transferencia abstracta $\hat{\mathcal{F}} \colon DS \times A \to A$ se define como:*

- $\hat{\mathcal{F}}(skip, a) = a$
- $\hat{\mathcal{F}}(\alpha?, a) = a$
- $\hat{\mathcal{F}}(v := expr, a) = a[v \mapsto \hat{X}[expr]_a]$
- $\hat{\mathcal{F}}(v.f := expr, a) =$ ***let*** $from = a(v)$, $to = \hat{X}[expr]_a$ ***in***

  if $|from| = 1$

  then $a[f \mapsto a(f) + +(from \to to)]$ *(strong update)*

  else $a[f \mapsto a(f) \cup (from \to to)]$ *(weak update)*

Es importante notar que la semántica de la operación de store distingue dos casos: cuando la abstracción es suficientemente precisa para actualizar una única fuente (strong update), y cuando una sobre aproximación debe ser efectuada (weak update).

Definición 21. *Dado un programa DynAlloy P y una fórmula ϕ representando los posibles estados de entrada, la collecting semantics abstracta de P comenzando con estado ϕ es el menor punto fijo de las siguientes ecuaciones:*

Por cada n en el grafo de flujo de control de P (CFG(P))

$$\begin{array}{lcl} \hat{in}(n) & = & \begin{cases} \bigsqcup\{\alpha(c_0) \mid M[\phi]_{c_0}\} & n \text{ punto de entrada de } CFG(P) \\ \bigsqcup_{p \in pred(n)} \hat{out}(p) & \text{caso contrario} \end{cases} \\ \hat{out}(n) & = & \hat{\mathcal{F}}(n, in(n)) \end{array}$$

donde $\hat{in}(n)$ y $\hat{out}(n)$ denotan los valores de input y output del nodo n respectivamente, y $pred(n)$ el conjunto de predecesores de n en el CFG.

4.3. Correctitud y terminación

Veamos primero que α y γ forman una conexión de Galois [NNH99, p. 234] entre los conjuntos parcialmente ordenados $(C, \subseteq)$ y $(A, \sqsubseteq)$. Para esto, debemos probar que α y γ son funciones monótonas que satisfacen:

$$\forall cs \in C, \quad \gamma \circ \alpha(cs) \supseteq cs \tag{4.1}$$

$$\forall a \in A, \quad \alpha \circ \gamma(a) \sqsubseteq a \tag{4.2}$$

Proposición 1. *La función de abstracción $\alpha \colon C \to A$ es monótona, i.e. $\forall cs, cs' \in C$:*

$cs' \subseteq cs \implies \alpha(cs') \sqsubseteq \alpha(cs)$

Demostración. Por el absurdo. Supongamos $cs' \subseteq cs \wedge \alpha(cs') \not\sqsubseteq \alpha(cs)$:

$\exists x \in JVar \uplus JField,\ \alpha(cs')(x) \not\sqsubseteq \alpha(cs)(x)$ *entonces*
$\exists v \in Atom \cup \mathcal{P}(Atom \times Atom)$ (valor concreto), $v \in \alpha(cs')(x) \wedge v \notin \alpha(cs)(x)$
por def de α, $\exists c \in E$ (estado concreto), $c(x) = v$, $c \in cs' \wedge c \notin cs$
que es absurdo, pues $cs' \subseteq cs$. Podemos concluir entonces que
$cs' \subseteq cs \implies \alpha(cs') \sqsubseteq \alpha(cs)$ □

Proposición 2. *La función de concretización $\gamma \colon A \to C$ es monótona, i.e. $\forall a, a' \in A$:*

$a' \sqsubseteq a \implies \gamma(a') \subseteq \gamma(a)$

Demostración. Por el absurdo. Supongamos $a' \sqsubseteq a \wedge \gamma(a') \not\subseteq \gamma(a)$:

$\exists c \in E,\ c \in \gamma(a') \wedge c \notin \gamma(a)$ *entonces*
$\exists c \in E,\ c \in \bigcup\{cs \mid \alpha(cs) \sqsubseteq a'\} \wedge c \notin \bigcup\{cs \mid \alpha(cs) \sqsubseteq a\}$ *entonces*
$\exists c \in E,\ (\exists cs \in C,\ \alpha(cs) \sqsubseteq a' \wedge c \in cs) \wedge (\nexists cs' \in C,\ \alpha(cs') \sqsubseteq a \wedge c \in cs')$ *entonces*
$\exists c \in E,\ (\exists cs \in C,\ \alpha(cs) \sqsubseteq a' \wedge c \in cs) \wedge (\forall cs' \in C,\ \alpha(cs') \not\sqsubseteq a \vee c \notin cs')$ *entonces*
$\exists c \in E,\ \exists cs \in C,\ c \in cs \wedge \alpha(cs) \sqsubseteq a' \wedge \alpha(cs) \not\sqsubseteq a$ *entonces*
$a' \not\sqsubseteq a$, lo que es absurdo, pues supusimos $a' \sqsubseteq a$. Podemos concluir entonces que
$a' \sqsubseteq a \implies \gamma(a') \subseteq \gamma(a)$ □

Proposición 3. $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *es una conexión de Galois.*

Demostración. Probemos primero la condición (4.1), i.e. que para todo cs que pertenece a C, $\gamma \circ \alpha(cs) \sqsupseteq cs$.

$\gamma \circ \alpha(cs) = \gamma(\alpha(cs)) = \bigcup\{cs' \mid \alpha(cs') \sqsubseteq \alpha(cs)\}$
En particular, cs es uno de los cs' que cumplen $\alpha(cs') \sqsubseteq \alpha(cs)$, por lo que
cs está contenido en el supremo. Entonces:
$cs \subseteq \bigcup\{cs' \mid \alpha(cs') \sqsubseteq \alpha(cs)\}$, de lo que concluimos que
$\gamma \circ \alpha(cs) \supseteq cs$

Probemos ahora la condición (4.2), i.e., que para todo a que pertenece a A, $\alpha \circ \gamma(a) \sqsubseteq a$, por el absurdo. Supongamos:

$\neg\forall a \in A,\ \alpha \circ \gamma(a) \sqsubseteq a \quad entonces$

$\exists a \in A,\ \alpha(\gamma(a)) \not\sqsubseteq a \quad entonces,$ por def de $\sqsubseteq$

$\exists a \in A,\ \exists x \in JVar \uplus JField,\ \alpha(\gamma(a))(x) \not\subseteq a(x) \quad entonces,$ por def de α

$\exists a \in A,\ (\exists x \in JVar,\ \{c(x) \mid c \in \gamma(a)\} \not\subseteq a(x) \ \vee\ \exists x \in JField,\ \bigcup_{c \in \gamma(a)} c(x) \not\subseteq a(x))$

De esto último podemos inferir que

$\exists a \in A,\ \exists c \in E,\ c \in \gamma(a) \wedge$ (1)

$\quad \exists x \in JVar,\ \{c(x)\} \not\subseteq a(x) \ \vee\ \exists x \in JField,\ c(x) \not\subseteq a(x)$ (2)

De (1) podemos deducir:

$\exists a \in A,\ \exists c \in E,\ c \in \bigcup\{cs \mid \alpha(cs) \sqsubseteq a\} \quad entonces$

$\exists a \in A,\ \exists c \in E,\ c \in \bigcup\{cs \mid \forall v \in JVar \uplus JField,\ \alpha(cs)(v) \subseteq a(v)\} \quad entonces$

$\exists a \in A,\ \exists c \in E,\ c \in \bigcup\{cs \mid \forall v \in JVar,\ \{c'(v) \mid c' \in cs\} \subseteq a(v) \wedge$

$\qquad \forall v \in JField,\ \bigcup_{c' \in cs} c'(v) \subseteq a(v)\}$

Notemos ahora que c pertenece al supremo de un conjunto donde cada elemento cs cumple que si c' pertenece a cs, entonces para todo v que pertenece a $JVar \uplus JField$ se cumple que $\{c'(v)\} \subseteq a(v)$ ó $c'(v) \subseteq a(v)$, dependiendo de si v pertenece a $JVar$ ó $JField$ respectivamente. Entonces:

$\forall v \in JVar,\ \{c(v)\} \subseteq a(v) \ \wedge\ \forall v \in JField,\ c(v) \subseteq a(v)$

Pero esto es absurdo pues contradice (2), por lo que podemos concluir que

$\forall a \in A,\ \alpha \circ \gamma(a) \sqsubseteq a$, como queríamos probar.

□

Para poder demostrar la correctitud de nuestra técnica, introducimos primero los siguientes lemas auxiliares.

Lema 1. *Sean* $M_1, M_2 \in JVar \uplus JField \to \mathcal{P}(Atom) \cup \mathcal{P}(Atom \times Atom)$, $v \in JVar \uplus JField$ *y* $A, B \in \mathcal{P}(Atom) \cup \mathcal{P}(Atom \times Atom)$,

$$M_1 \sqsubseteq M_2 \wedge A \subseteq B \implies M_1[v \mapsto A] \sqsubseteq M_2[v \mapsto B]$$

Demostración.

Para $x \in JVar \uplus JField$, $x \neq v$,

$M_1[v \mapsto A](x) = M_1(x) \wedge M_2[v \mapsto B](x) = M_2(x)$, y como $M_1 \sqsubseteq M_2$, entonces

$M_1[v \mapsto A](x) \subseteq M_2[v \mapsto B](x)$

Por otro lado:

$M_1[v \mapsto A](v) = A \wedge M_2[v \mapsto B](v) = B$, y como $A \subseteq B$, entonces

$M_1[v \mapsto A](v) \subseteq M_2[v \mapsto B](v)$

Entonces:

$\forall x \in JVar \uplus JField$, $M_1[v \mapsto A](x) \subseteq M_2[v \mapsto B](x)$, entonces

$M_1[v \mapsto A] \sqsubseteq M_2[v \mapsto B]$

□

Lema 2. *Sean* $cs \in C$, $a \in A$ *y* expr *una expresión de DynAlloy,*

$\alpha(cs) \sqsubseteq a \implies \forall c \in cs,\ X[expr]_c \subseteq \hat{X}[expr]_a$

Demostración. Tratemos por casos las posibles formas de *expr*: *null*, v y $v.f$:

- $expr = null$:

  $\forall c \in cs,\ X[null]_c = \{\langle null \rangle\} = \hat{X}[null]_a \quad entonces$

  $\forall c \in cs,\ X[null]_c \subseteq \hat{X}[null]_a$

- $expr = v$:

  $\alpha(cs) \sqsubseteq a \quad entonces$

  $\alpha(cs)(v) \subseteq a(v)$

  Por def de α, $\alpha(cs)(v) = \{c(v) \mid c \in cs\} \quad entonces$

  $\{c(v) \mid c \in cs\} \subseteq a(v) \quad entonces$

  $\forall c \in cs,\ \{c(v)\} \subseteq a(v)$

  Y como $X[v]_c = \{c(v)\}$ y $\hat{X}[v]_a = a(v) \quad entonces$

  $\forall c \in cs,\ X[v]_c \subseteq \hat{X}[v]_a$

- $expr = v.f$:

  $\alpha(cs) \sqsubseteq a \quad entonces$

  $\alpha(cs)(v) \subseteq a(v)$ y $\alpha(cs)(f) \subseteq a(f)$

  Por def de α, $\alpha(cs)(v) = \{c(v) \mid c \in cs\}$ y $\alpha(cs)(f) = \bigcup_{c \in cs} c(f) \quad entonces$

  $\{c(v) \mid c \in cs\} \subseteq a(v)$ y $\bigcup_{c \in cs} c(f) \subseteq a(f) \quad entonces$

  $\forall c \in cs,\ \{c(v)\} \subseteq a(v)$ y $c(f) \subseteq a(f)$

  Por def de ;, $\{c(v)\}; c(f) \subseteq a(v); a(f)$

  Y como $X[v.f]_c = \{c(v)\}; c(f)$ y $\hat{X}[v.f]_a = a(v); a(f) \quad entonces$

  $\forall c \in cs,\ X[v.f]_c \subseteq \hat{X}[v.f]_a$

□

Ahora estamos en condiciones de demostrar que la abstracción es una sobre aproximación segura de la collecting semantics.

Teorema 1 (Correctitud)**.** *Sean* $cs \in C$, $a \in A$, $n \in CFG(P)$*:*

$\alpha(cs) \sqsubseteq a \implies \alpha(\mathcal{F}(n, cs)) \sqsubseteq \hat{\mathcal{F}}(n, a)$

Demostración. Tratamos por casos los distintos nodos del CFG:

- $n =$ *skip*

  $\alpha(\mathcal{F}(skip, cs)) = \alpha(cs) \overset{Hip.}{\sqsubseteq} a = \hat{\mathcal{F}}(skip, a)$ *entonces*
  $\alpha(\mathcal{F}(skip, cs)) \sqsubseteq \hat{\mathcal{F}}(skip, a)$

- $n = \phi?$

  $\mathcal{F}(\phi?, cs) = \{c \mid c \in cs \wedge M[\phi]_c\} \subseteq cs$ *entonces*, por monotonía de α
  $\alpha(\mathcal{F}(\phi?, cs)) \sqsubseteq \alpha(cs) \overset{Hip.}{\sqsubseteq} a = \hat{\mathcal{F}}(\phi?, a)$ *entonces*
  $\alpha(\mathcal{F}(\phi?, cs)) \sqsubseteq \hat{\mathcal{F}}(\phi?, a)$

- $n = v := expr$

  $\alpha(cs) \sqsubseteq a$ *entonces*, por lemas 1 y 2
  $\forall c \in cs,\ \alpha(cs)[v \mapsto X[expr]_c] \sqsubseteq a[v \mapsto \hat{X}[expr]_a]$ *entonces*, por def de α
  $\forall c \in cs,\ \alpha(cs[v \mapsto [X[expr]_c]_0]) \sqsubseteq \hat{\mathcal{F}}(v := expr, a)$ *entonces*, por def. 11
  $\forall c \in cs,\ \alpha(\{c'[v \mapsto [X[expr]_c]_0] \mid c' \in cs\}) \sqsubseteq \hat{\mathcal{F}}(v := expr, a)$ *entonces*
  $\alpha(\{c'[v \mapsto [X[expr]_{c'}]_0] \mid c' \in cs\}) \sqsubseteq \hat{\mathcal{F}}(v := expr, a)$ *entonces*
  $\alpha(\mathcal{F}(v := expr, cs)) \sqsubseteq \hat{\mathcal{F}}(v := expr, a)$

- $n = v.f := expr$

  Si $|X[v]_a| = 1$ (strong update):
  $\forall c \in cs,\ c(f) + +(X[v]_c \to X[expr]_c) \subseteq a(f) + +(\hat{X}[v]_a \to \hat{X}[expr]_a)$
  Entonces, por $\alpha(cs) \sqsubseteq a$ y por lema 1
  $\forall c \in cs,\ \alpha(cs)[f \mapsto c(f) + +(X[v]_c \to X[expr]_c)] \sqsubseteq$
  $\quad a[f \mapsto a(f) + +(\hat{X}[v]_a \to \hat{X}[expr]_a)]$ *entonces*
  $\forall c \in cs,\ \alpha(cs[f \mapsto c(f) + +(X[v]_c \to X[expr]_c)]) \sqsubseteq \hat{\mathcal{F}}(v.f := expr, a)$

  Si $|X[v]_a| \neq 1$ (weak update):
  $\forall c \in cs,\ c(f) + +(X[v]_c \to X[expr]_c) \subseteq a(f) \cup (\hat{X}[v]_a \to \hat{X}[expr]_a)$
  Entonces, por $\alpha(cs) \sqsubseteq a$ y por lema 1
  $\forall c \in cs,\ \alpha(cs)[f \mapsto c(f) + +(X[v]_c \to X[expr]_c)] \sqsubseteq$
  $\quad a[f \mapsto a(f) \cup (\hat{X}[v]_a \to \hat{X}[expr]_a)]$ *entonces*
  $\forall c \in cs,\ \alpha(cs[f \mapsto c(f) + +(X[v]_c \to X[expr]_c)]) \sqsubseteq \hat{\mathcal{F}}(v.f := expr, a)$

En ambos casos, tanto si $|X[v]_a| = 1$ ó $|X[v]_a| \neq 1$ vale que

$\forall c \in cs,\ \alpha(cs[f \mapsto c(f) + +(X[v]_c \to X[expr]_c)]) \sqsubseteq \hat{\mathcal{F}}(v.f := expr, a)$

Y entonces, por def. 11

$\forall c \in cs,\ \alpha(\{c'[f \mapsto c(f) + +(X[v]_c \to X[expr]_c)] \mid c' \in cs\}) \sqsubseteq \hat{\mathcal{F}}(v.f := expr, a)$

Por lo que en particular, con $c = c'$ vale

$\alpha(\{c'[f \mapsto c'(f) + +(X[v]_{c'} \to X[expr]_{c'})] \mid c' \in cs\}) \sqsubseteq \hat{\mathcal{F}}(v.f := expr, a)$

Y por def. de $\mathcal{F}$ podemos concluir que

$\alpha(\mathcal{F}(v.f := expr, cs)) \sqsubseteq \hat{\mathcal{F}}(v.f := expr, a)$

□

Corolario 1. *Para cada nodo $n \in CFG(P)$, el menor punto fijo (least fixed point, o LFP) de las ecuaciones de dataflow del análisis es una sobre aproximación segura del menor punto fijo de las ecuaciones correspondientes de la collecting semantics concreta. Es decir, para cada nodo $n \in CFG(P)$, $\alpha(LFP(in(n))) \subseteq LFP(\hat{in}(n))$ y $\alpha(LFP(out(n))) \subseteq LFP(\hat{out}(n))$.*

Demostración. Del teorema 1 y de [CC79] sabemos que $\alpha(LFP(\mathcal{F}(n)) \subseteq LFP(\hat{\mathcal{F}}(n))$. Entonces, como las ecuaciones de dataflow concretas y abstractas son funciones monótonas compuestas de $\mathcal{F}$ y $\hat{\mathcal{F}}$ respectivamente, la propiedad es preservada. □

Teorema 2 (Terminación)**.** *Las ecuaciones de dataflow convergen (i.e., el análisis termina) para todo programa P.*

Demostración. El semi-reticulado con top de abstracciones es finito, pues es el conjunto de partes del conjunto de átomos, que es finito. Como cada subconjunto de este conjunto tiene un supremo, nuestro semi-reticulado con top es completo, y ya que las ecuaciones de dataflow son monótonas, el teorema de Knaster-Tarski [Tar55] garantiza la existencia del menor punto fijo. □

Capítulo 5

Diseño

Teniendo el marco teórico definido, veamos cómo todo esto es llevado a la práctica. Para poder abordar el problema y describir los detalles de su solución, descompongámoslo en sus componentes principales: la construcción del control flow graph, el framework de dataflow, y nuestro análisis en sí.

Idealmente, para poder implementar directamente los resultados del capítulo 4, deberíamos trabajar sobre un programa DynAlloy que sea el resultado de una transformación que preserva la semántica, tal que cumple:

1. Unrolling: Todos los loops del programa son desenrollados la cantidad de veces especificada en el scope de la verificación.

2. Inlining: Las invocaciones a diferentes programas son reemplazadas por el cuerpo del método correspondiente.

3. SSA: El programa DynAlloy está transformado a static single assignment, de manera tal que cada variable DynAlloy tenga su contrapartida en una relación Alloy.

4. No hay variables locales: Las variables locales son promovidas a parámetros. Esto permite, reemplazando adecuadamente parámetros formales por su instanciación, acceder globalmente a todas las variables y no tener problemas de colisión de nombres.

Si tuviéramos todas estas propiedades, las variables DynAlloy serían una correspondencia de las relaciones Alloy, y obtendríamos una sobre aproximación de todos los valores posibles que una relación Alloy podría almacenar al realizar nuestro análisis de dataflow en la representación DynAlloy.

En la práctica, utilizar el resultado DynAlloy intermedio que obtenemos de TACO sólo nos garantiza las propiedades 1 y 4, ya que las restantes transformaciones son pasos ejecutados en la traducción a Alloy, y no podemos aprovecharlos. Es por esto que la construcción de nuestro análisis debe encargarse de garantizar las propiedades 2 y 3. Inlining, como veremos en la siguiente sección, se cumple por la construcción del CFG. La propiedad 3 es garantizada *on the fly* por nuestro análisis (sección 5.3), aprovechando el mapeo DynAlloy-Alloy de la traducción de TACO.

5.1. Construcción del CFG

Para construir el CFG del programa DynAlloy a verificar, nos basamos en su *abstract syntax tree* (AST). Las construcciones que permite la gramática de DynAlloy son: `Assignment` (asignación), `Choice` (decisión no determinística), `Closure` (iteración no acotada), `Composition` (composición secuencial), `InvokeProgram` (invocación de programa), `Skip` y `TestPredicate` (assume). Para crear el CFG, hacemos un recorrido *depth first search* (DFS) del AST, creando un nodo del CFG por cada una de las construcciones atómicas `Assignment`, `Skip` y `TestPredicate`. Cabe destacar que, debido a que los loops han sido desenrollados, no hay `Closure` dentro del AST, lo que lleva a la construcción de un CFG sin ciclos.

El caso del `InvokeProgram` es más delicado y amerita una descripción más detallada, ya que en esencia, implica resolver el problema del análisis interprocedural. Lo que hacemos en este caso es construir el CFG del programa invocado, y conectarlo al CFG del programa invocador en el punto de la llamada. De esta forma, obtenemos como resultado un CFG único sobre el cual construir nuestro análisis. Tenemos garantizada la ausencia de recursión, puesto que no es soportada por el *DynAlloyToAlloy Translator*, así que podemos realizar esto en todos los casos. Ya que, como parte del proceso de traducción, todas las variables locales son promovidas a parámetros, y en la especificación Alloy resultante son eliminados los cuantificadores y reemplazados por una signatura global (lo que en definitiva nos permite acotar todas las variables de estados intermedios), este es un sencillo método para resolver el análisis interprocedural.

Para poder hacer esto correctamente, queda un punto por resolver: la instanciación de parámetros. En el programa invocado los parámetros son definidos con un nombre formal, referenciado a lo largo de todo el programa. Cuando el programa es invocado, los parámetros formales son instanciados con los parámetros actuales. Lo que hacemos para resolver esto es crear un mapeo entre definición de parámetro e instanciación, que es utilizado para saber de qué variable (o expresión) se trata realmente cuando estamos manipulando una variable.

5.2. Framework de dataflow

Nuestro motor de dataflow permite implementar cualquier clase de análisis forward. En particular, nuestro análisis de dataflow de cotas está construido a partir de él. El acercamiento genérico suele ser utilizar un algoritmo iterativo o de work list [App97] para encontrar un punto fijo de los conjuntos de entrada y salida de todos los nodos del CFG. Ya que en nuestro caso tenemos la particularidad de un CFG acíclico, aprovechamos esto para construir un algoritmo más sencillo y eficiente, cuyo código podemos ver en la figura 5.1.

Esencialmente, el algoritmo procede de la siguiente manera: mantiene una cola de nodos a procesar (*nodesToProcessQueue*), que contiene inicialmente los puntos de entrada del CFG. Existen dos Map que, para todos los nodos, devuelven el estado abstracto (de tipo *FlowSet*) a la entrada del nodo (*nodeToBeforeFlow*) y el estado abstracto a la salida (*nodeToAfterFlow*). Los nodos de la cola son procesados sólo si todos sus predecesores ya fueron procesados. Para procesar un nodo se llama al procedimiento que implementa la función de transferencia (*flowThrough*), y el estado abstracto resultante se une (*mergeInto*) al de todos los sucesores del nodo, que son agregados luego a la lista de nodos a procesar. Cuando no quedan nodos

```
//Asignamos el flujo inicial a todos los nodos de entrada del CFG.
for (CfgNode entry : cfg.getHeads()) {
   nodeToBeforeFlow.put(entry, entryInitialFlow());
   nodesToProcessQueue.add(entry);
}

while (!nodesToProcessQueue.isEmpty()) {
   CfgNode n = nodesToProcessQueue.remove();

   //Chequeamos que todos los predecesores de este nodo hayan sido procesados.
   //Si no, lo eliminamos de la lista, ya que será procesado después de que
   //uno de esos predecesores no procesados lo agregue nuevamente.
   boolean ready = true;
   for (CfgNode pred : n.getPredecessors()) {
      if (!processedNodes.contains(pred)) {
         ready = false;
         break;
      }
   }
   if (!ready)
      continue;

   //Computamos el análisis en el nodo y almacenamos su output.
   FlowSet<T> afterFlow = newInitialFlow();
   flowThrough(nodeToBeforeFlow.get(n), n, afterFlow);
   nodeToAfterFlow.put(n, afterFlow);
   processedNodes.add(n);

   //Unimos el flow de salida del nodo al de entrada de todos sus sucesores
   for (CfgNode succ : n.getSuccessors()) {
      FlowSet<T> succBeforeFlow;
      if (nodeToBeforeFlow.containsKey(succ))
         succBeforeFlow = nodeToBeforeFlow.get(succ);
      else {
         succBeforeFlow = newInitialFlow();
         nodeToBeforeFlow.put(succ, succBeforeFlow);
      }
      mergeInto(n, succBeforeFlow, afterFlow);

      if (!nodesToProcessQueue.contains(succ))
         nodesToProcessQueue.add(succ);
   }
}
```

Figura 5.1: Fragmento de código que muestra el algoritmo de forward flow.

en esta lista, ya se terminó de calcular el análisis de dataflow para el programa, y todos los nodos fueron procesados sólo una vez.

Para crear un análisis de dataflow forward a partir de este motor, se deben implementar las siguientes operaciones:

- *flowThrough*: Es la función de transferencia abstracta. Dado el estado abstracto de entrada, se calcula el estado abstracto de salida del nodo del CFG.
- *merge*: Es la combinación de dos estados abstractos (el operador de *join* del reticulado).
- *entryInitialFlow*: Devuelve el estado abstracto que representa el inicio del análisis, y es usado como entrada para la función de transferencia en los puntos de entrada del CFG.

5.3. Análisis de cotas

Nuestro análisis de dataflow de cotas está implementado sobre el motor de dataflow descrito en la sección anterior. Las implementaciones de las funciones necesarias son llevadas a la práctica gracias a las definiciones del capítulo 4: la función *flowThrough* es la implementación de la función de transferencia abstracta (definición 20), y la función *merge* sigue la definición de la operación de *join* $\sqcup$ del reticulado de abstracciones (definición 15).

En la implementación de *entryInitialFlow* es donde usamos las cotas iniciales provistas por TACO. El estado abstracto inicial contiene esta abstracción refinada para las variables correspondientes. A las variables que no son acotadas por TACO y modelan un parámetro de Java, les asignamos todas las tuplas que satisfacen la definición de tipo para el scope de análisis. Para cualquier otra variable DynAlloy x, no asociamos ninguna tupla (es decir, $a(x) = \emptyset$) .

Como mencionamos al comienzo del capítulo, el programa DynAlloy sobre el que trabajamos no está en forma SSA. Sin embargo, su traducción a Alloy sí lo está, con lo cual no podemos relacionar directamente una variable DynAlloy a una relación Alloy. La solución a esta situación consta de dos partes, implementadas en el *flowThrough* y en el *merge*, que nos permiten simular SSA [CFR+91]. Por un lado, en el *flowThrough*, utilizamos información del proceso de traducción de TACO para conocer (en cada nodo del CFG) su traducción a Alloy, y derivar de esto de qué encarnación de las variables DynAlloy se trata en cada caso. De esta forma la información fluye a través del CFG para las variables DynAlloy con sus respectivas encarnaciones, que sí tienen su contrapartida en las relaciones Alloy. Para que esto sea correcto, sin embargo, se requiere de un ajuste en el *merge*: lo que hay que hacer, ya que estamos simulando SSA, es introducir la función ϕ [App98] para sincronizar las encarnaciones que provienen de distintas ramas. De esta forma, las variables DynAlloy de nuestro dominio son una correspondencia de las relaciones Alloy.

Para introducir correctamente las cotas ajustadas (producto de nuestro análisis) para las relaciones Alloy, utilizamos el valor abstracto de cada variable DynAlloy en la salida del CFG. Dada una variable DynAlloy $x \in JVar \uplus JField$, el valor abstracto de x obtenido en la salida puede ser escrito como una cota superior de la relación Alloy x, que luego es introducida como input de Kodkod, llevando a la eliminación de variables proposicionales innecesarias. Para los casos donde el valor abstracto de x tiene como imagen un conjunto vacío, se tiene que tomar una medida especial para asegurar el cumplimiento de las restricciones estructurales de Alloy.

Definición 22. *Sea a_{exit} el estado abstracto computado para la salida del CFG. Definimos entonces la cota superior calculada por dataflow como:*

$$U_x^{DF} = \begin{cases} a_{exit}(x) & si\ a_{exit}(x) \neq \emptyset \\ defVal(x) & en\ caso\ contrario \end{cases}$$

donde $defVal(x)$ devuelve el valor por defecto de Java (e.g, 0 para `Integer`*, false para* `boolean`*, etc) en caso que x modele una variable Java, y una función total cuyo rango contiene valores por defecto en caso que x represente un campo.*

Llamamos U_x^{TACO} a la cota superior provista por TACO a Kodkod cuando no se realiza análisis de dataflow. El siguiente teorema asegura que nuestra técnica es segura (es decir, no se pierden fallas).

Teorema 3. *Sea θ la fórmula Alloy resultante del traductor* DynAlloyToAlloy*. Dado un modelo Alloy I tal que $M[\theta \wedge \bigwedge_x (x \subseteq U_x^{TACO})]_I =$* `true`

Entonces, existe un modelo Alloy I' tal que $M[\theta \wedge \bigwedge_x (x \subseteq U_x^{DF})]_{I'} =$ `true`

Demostración. Sea I un modelo Alloy que satisface la hipótesis. Sabemos por definición que $defVal(x) \subseteq U_x^{DF}$. Sea x una relación Alloy tal que $I(x) \subseteq U_x^{TACO} \backslash U_x^{DF}$. Debido al corolario de correctitud (corolario 1), x debe ser la correspondencia de una variable DynAlloy que cumple simultáneamente que: 1) x representa una variable local, y 2) no ocurre ni asignación ni acceso a x en el conjunto de trazas codificadas por I. Entonces, el valor de x no tiene efecto sobre el conjunto de trazas codificadas por I. Esto significa que este conjunto permanece inalterado si reemplazamos el valor de x con cualquier otro valor (por ejemplo, $defVal(x)$). Debido al hecho de que θ codifica una aserción de correctitud parcial [FGLPA05], la satisfacibilidad de θ no depende del valor de x. Consecuentemente, $M[\theta]_I = M[\theta]_{I[x \mapsto defVal(x)]}$ por sustitución. Entonces, podemos definir I' como:

$$I'(x) = \begin{cases} I(x) & \text{si } I(x) \subseteq U_x^{DF} \\ defVal(x) & \text{en caso contrario} \end{cases}$$

donde $M[\theta]_I = M[\theta]_{I'}$ e $I'(x) \subseteq U_x^{DF}$. □

Por último, un detalle de nuestro análisis que vale la pena destacar es el manejo de enteros: todas las operaciones sobre ellos son interpretadas. Como estamos trabajando dentro del marco de verificación acotada, la cantidad de enteros no es la excepción. Al comienzo del análsis se define la cantidad de bits de su representación, y siendo su valor n, tenemos 2^n enteros distintos, que van desde el -2^{n-1} hasta el $2^{n-1} - 1$. La interpretación de las funciones se da entonces dentro del marco de la aritmética modular, permitiéndonos acotar con precisión las variables DynAlloy enteras.

5.4. Loop unroll

Analizando la sobre aproximación calculada por las cotas, y qué tan ajustados eran los resultados, enfocamos nuestra atención en el proceso de unrolling realizado por TACO.

Como mencionamos al comienzo del capítulo 4, la construcción while de Java `while B do P od` puede ser expresada en DynAlloy como $(B?;P)^*;(\neg B)?$. Dado un límite de loop k, la fase de unroll del proceso de traducción transforma el ciclo en:

$$\underbrace{((B?;P)+skip);\ldots;((B?;P)+skip)}_{k-veces};(\neg B)?$$

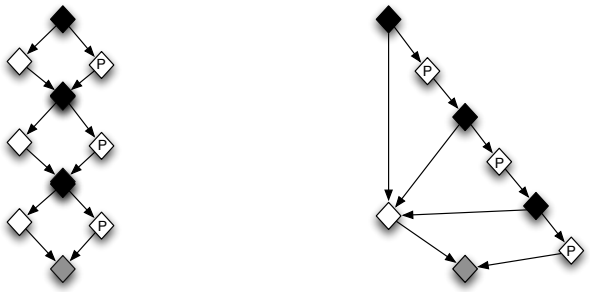
A pesar de ser semánticamente correcta, esta representación del while permite múltiples permutaciones. Por ejemplo, la traza de programa $B?;P;skip$ es equivalente a $skip;B?;P$. Esta aparentemente inofensiva simetría tiene un tremendo impacto, ya que nuestro análsis de dataflow no es *branch sensitive*. Debido a esto, la sobre aproximación calculada no es tan ajustada como podría ser.

Esta observación nos llevó a modificar el proceso de loop unrolling. La nueva representación de unrolling anidada transforma `while B do P od` en $T_k(B,P);(\neg B)?$, donde T se define recursivamente como:

$$\begin{aligned} T_0(B,P) &= skip \\ T_n(B,P) &= (((B?;P);T_{n-1}(B,P))+skip) \end{aligned}$$

En la figura 5.2 podemos observar una representación gráfica del cambio de unroll. Esto lleva a una disminución en la cantidad de caminos resultantes: si realizamos n loop unrolls, la vieja representación lleva a 2^n caminos, mientras que la nueva a sólo $2(n-1)$.

Esta nueva representación, además de mitigar la pérdida de precisión de nuestro análisis, impacta directamente en el proceso de verificación de TACO. Los resultados de esto serán analizados en el capítulo 6.



(a) Codificación original. (b) Nueva codificación.

Figura 5.2: Métodos de loop unrolling.

Capítulo 6

Evaluación experimental

6.1. Introducción

En este capítulo presentamos la experimentación realizada para evaluar nuestro trabajo. Como es de esperarse, el objetivo principal de esta experimentación es medir el impacto de nuestra técnica en la verificación. Queremos ver si la performance del análisis SAT mejora, y si esto se traduce en un aumento de su escalabilidad.

Para esto, y siguiendo la línea de experimentos usados para evaluar TACO [GRLPF10], analizamos 7 clases de colecciones con creciente complejidad de invariantes:

LList: Una implementación de secuencias basada en listas simplemente encadenadas.

AList: La implementación AbstractLinkedList de la interface List, del paquete commons.collections de Apache, basada en listas circulares doblemente encadenadas.

CList: La implementación NodeCachingLinkedList de la interface List de commons.collections de Apache.

BSTree: Una implementación de binary search tree de [VPP06].

TreeSet: La implementación de la clase TreeSet de java.util, basada en red-black trees.

AVLTree: Una implementación de AVL trees obtenida del caso de estudio usado en [BRD09].

BinHeap: Una implementación de binomial heaps usada como parte de un benchmark en [VPP06].

Ya que estas clases son correctas[1], esto permite que las comparaciones de tiempo se realicen en una situación donde el espacio de estados debe ser analizado en su totalidad. Ya que el proceso de detección de bugs finaliza cuando no se encuentran más errores, analizar código correcto no es una situación artificial, sino una prueba de stress que necesariamente surge en un proceso de análisis. Para todos los casos, los loops son desenrollados 10 veces, y seleccionamos el máximo scope que TACO pudo analizar utilizando un timeout de 10 horas. El máximo

[1] Salvo por el método ExtractMin de BinHeap, cuya falla fue reportada por primera vez gracias a TACO en [GRLPF10]

scope está restringido a 20 nodos para cada experimento, ya que este es el tope utilizado en los trabajos previos para evaluar TACO.

En la figura 6.1 mostramos los tiempos utilizados por TACO para calcular las cotas iniciales. Es importante notar que estos tiempos se calculan una sola vez por cada clase, y luego son reutilizados en cada análisis, con lo que su costo es amortizado. Estas son las abstracciones iniciales que utiliza nuestro análisis de dataflow.

Clase	s10	s12	s15	s17	s20
LList	00:15	00:24	00:47	01:04	01:37
AList	00:51	00:33	00:55	03:15	300:20
CList	01:11	01:38	01:46	05:16	575:00
BSTree	00:16	00:38	01:56	04:05	21:19
TreeSet	01:44	02:51	05:19	16:42	40:37
AVL	01:55	03:46	10:36	47:25	168:23
BinHeap	02:46	04:41	10:32	34:50	117:20

Cuadro 6.1: Tiempos de TACO para calcular las cotas iniciales por clase y scope.

En todos los casos se está chequeando que los invariantes se preserven. Para las clases LList, AList y CList analizamos los métodos que implementan las operaciones de secuencia. Análogamente, para TreeSet, AVLTree y BSTree analizamos los métodos de las operaciones de conjunto. Finalmente, para BinHeap, analizamos los métodos que implementan las operaciones de cola de prioridad.

Por último, para completar la información de contexto, cabe mencionar que todos los casos de estudio fueron corridos en un servidor con procesador Intel i5-750 @ 2.67 Ghz, 8GB de RAM, y bus DDR3 de 1.3GHz, corriendo sobre Debian GNU/Linux 6.0, kernel 2.6.32-5-amd64.

6.2. Resultados

6.2.1. Costo del análisis de dataflow

En primera instancia, analicemos el overhead introducido por nuestra técnica. En la tabla 6.2 podemos ver, para cada método (con los scopes que utilizamos en la sección siguiente para medir el tiempo de análisis) los tiempos necesarios para computar las cotas con el análisis de dataflow.

Para complementar estos datos, resulta ilustrativo tener una noción de cómo se comparan estos costos con el tiempo requerido para realizar la verificación (que veremos en la siguiente sección). Para visualizar esto, podemos utilizar la siguiente métrica: el promedio, para cada método de cada clase, del tiempo consumido para calcular dataflow, sobre el tiempo de análisis contando el cálculo de dataflow. Esto es, si llamamos T al tiempo de análisis de cada método y D al tiempo de cálculo del dataflow de cotas: $Promedio(D/(T+D))$. El resultado de esto, aplicado a todos los experimentos, es $\mathbf{0,0189}$. Esto nos da la pauta de que el overhead causado por nuestro análisis es despreciable.

Método	Tiempo
SList, Contains, s20	0,130
SList, Insert, s20	0,141
SList, Remove, s20	0,122
AList, Contains, s20	0,190
AList, Insert, s20	0,143
AList, Remove, s20	0,173
CList, Contains, s20	0,167
CList, Add, s20	0,180
CList, Remove, s20	0,334
AvlTree, FindMax, s20	0,146
AvlTree, Find, s20	1,968
AvlTree, Insert, s17	98,789
BinHeap, FindMin, s20	0,854
BinHeap, DecKey, s18	2,632
BinHeap, Insert, s17	8,520
BinHeap, extMin, s20	21,714
TreeSet, Contains, s20	1,749
TreeSet, Add, s13	35,123
BSTree, Find, s13	0,193
BSTree, Remove, s12	0,441
BSTree, Add, s09	1,792

Cuadro 6.2: Tiempos, en segundos, demorados en computar el análisis de dataflow.

6.2.2. Impacto en el análisis

Analicemos ahora el punto más interesante: el impacto de nuestra técnica en el tiempo de SAT-solving. Para evaluar esto, observemos los tiempos de análisis. A partir de este punto, cuando nos referimos a *TACO* estamos hablando de la herramienta original (con predicado de ruptura de simetría, cotas iniciales, y el método original de loop unrolling), y cuando hablamos de *TACOFlow*, es el resultado de aplicar nuestro análisis de dataflow a TACO, incluyendo el nuevo unroll.

Para analizar esto presentamos las tablas 6.3 y 6.4. En la primera podemos ver, tanto para TACO como para TACOFlow, el máximo scope de análisis (con un tope de 20) antes de obtener timeout, para cada método de cada clase experimental. En la segunda, comparamos para cada método de cada clase (usando el máximo scope que no dio timeout ni con TACO ni con TACOFlow), el tiempo de análisis para cada herramienta, desglosando en el caso de TACOFlow el tiempo utilizado en calcular dataflow y el de realizar el SAT-solving. Además comparamos los tiempos de TACO y TACOFlow realizando el cociente entre ellos, en la columna *Speed-up*.

Para visualizar sintéticamente cuál es la mejora medida en aumento de scope permitida por el uso de nuestra técnica, veamos el promedio de la diferencia entre el scope permitido por TACOFlow y TACO, es decir: $promedio(TF - T)$, donde TF es el máximo scope permitido por TACOFlow y T el de TACO, por cada método antes de obtener timeout. El resultado de esto es $\mathbf{0,52}$, lo que representa nuestro aumento promedio de scope.

Por otro lado, podemos realizar algo parecido para ver qué tan significativa fue la mejora del tiempo de análisis, viendo el promedio del speed-up. Lo calculamos como $promedio(T/TF)$, donde esta vez T y TF representan los tiempos de análisis de TACO y TACOFlow, respectivamente. Este promedio es de $\mathbf{18,69}$, lo que muestra un significativo aumento de performance, donde TACOFlow es casi 19 veces más rápido que TACO.

Método	TACO	TACOFlow
SList, Contains	20	20
SList, Insert	20	20
SList, Remove	20	20
AList, Contains	20	20
AList, Insert	20	20
AList, Remove	20	20
CList, Contains	20	20
CList, Add	20	20
CList, Remove	20	20
AvlTree, FindMax	20	20
AvlTree, Find	20	20
AvlTree, Insert	17	17
BinHeap, FindMin	20	20
BinHeap, DecKey	18	20
BinHeap, Insert	17	20
BinHeap, extMin	20	20
TreeSet, Contains	20	20
TreeSet, Add	13	17
BSTree, Find	13	14
BSTree, Remove	13	12
BSTree, Add	9	11

Cuadro 6.3: Máximo scope alcanzado antes de obtener timeout para TACO y TACOFlow.

Método	T. TACO	T. TACOFlow	T. dataflow	T. SAT-Solving	Speed-up
SList, Contains, s20	8,25	5,342	0,132	5,21	1,54
SList, Insert, s20	9,91	11,301	0,141	11,16	0,88
SList, Remove, s20	18,11	8,202	0,122	8,08	2,21
AList, Contains, s20	29,2	8,43	0,19	8,24	3,46
AList, Insert, s20	10,66	9,043	0,143	8,9	1,18
AList, Remove, s20	144,35	11,943	0,173	11,77	12,09
CList, Contains, s20	109,7	47,537	0,167	47,37	2,31
CList, Add, s20	59,9	45,82	0,18	45,64	1,31
CList, Remove, s20	1649,47	353,664	0,334	353,33	4,66
AvlTree, FindMax, s20	25,82	25,286	0,146	25,14	1,02
AvlTree, Find, s20	1439,32	119,038	1,968	117,07	12,09
AvlTree, Insert, s17	32018,14	20744,999	98,789	20646,21	1,54
BinHeap, FindMin, s20	109,86	10,454	0,854	9,6	10,51
BinHeap, DecKey, s18	18216,79	335,422	2,632	332,79	54,31
BinHeap, Insert, s17	25254,66	122,22	8,52	113,7	206,63
BinHeap, extMin, s20	1149,71	451,194	21,714	429,48	2,55
TreeSet, Contains, s20	14475,49	769,649	1,749	767,9	18,81
TreeSet, Add, s13	26447,76	719,783	35,123	684,66	36,74
BSTree, Find, s13	28602,33	9190,953	0,193	9190,76	3,11
BSTree, Remove, s12	7251,39	14322,001	0,441	14321,56	0,51
BSTree, Add, s09	18344,38	1214,492	1,792	1212,7	15,10

Cuadro 6.4: Tiempos y speed-up para TACO y TACOFlow.

6.2.3. Análisis de las mejoras

En base a los resultados presentados anteriormente, en esta sección nos interesa analizar de dónde provienen las mejoras obtenidas. Ya que TACOFlow difiere de TACO en la nueva codificación de loop unrolling y en las cotas calculadas por dataflow para eliminar variables proposicionales, queremos medir cada contribución por separado. A partir de este momento, llamaremos *TACO*$^+$ a TACO, pero con la nueva versión de unroll (o equivalentemente, a TACOFlow pero sin los resultados del análisis de cotas).

Veamos entonces la comparación entre TACO$^+$ y TACOFlow. En la tabla 6.5 podemos ver el máximo scope analizado sin timeout por cada uno, y en la tabla 6.6 los tiempos comparativos, junto con el speed-up (calculado, al igual que antes, como el cociente entre el tiempo de TACO$^+$ y TACOFlow), la cantidad de variables proposicionales de TACO$^+$ y el porcentaje de estas eliminado por TACOFlow.

Método	TACO$^+$	TACOFlow
SList, Contains	20	20
SList, Insert	20	20
SList, Remove	20	20
AList, Contains	20	20
AList, Insert	20	20
AList, Remove	20	20
CList, Contains	20	20
CList, Add	20	20
CList, Remove	20	20
AvlTree, FindMax	20	20
AvlTree, Find	20	20
AvlTree, Insert	17	17
BinHeap, FindMin	20	20
BinHeap, DecKey	20	20
BinHeap, Insert	20	20
BinHeap, extMin	20	20
TreeSet, Contains	20	20
TreeSet, Add	17	17
BSTree, Find	13	14
BSTree, Remove	12	12
BSTree, Add	11	11

Cuadro 6.5: Máximo scope alcanzado antes de obtener timeout para TACO$^+$ y TACOFlow.

Estos resultados fueron algo sorprendentes. Nuestra hipótesis era que las mejoras se debían a la reducción de variables proposicionales. Pero hay que tener en cuenta que la nueva representación del while no altera la cantidad de variables, sino que reduce trazas isomorfas en el código. Y como se puede apreciar a partir de la comparación del speed-up de las tablas 6.4 y 6.6, la mayor parte de las mejoras proviene del cambio de unroll, con lo cual no podemos establecer una clara correlación entre porcentaje de variables removidas y speed-up.

Con esto, podemos concluir que el aporte al speed-up por parte del nuevo mecanismo de unroll es muy importante. La nueva representación evita muchos caminos en el CFG que llevan a valuaciones isomorfas en la fórmula proposicional. Como describimos en la sección 5.4, la aplicación de n loop unrolls en TACO lleva a 2^n caminos, mientras que en TACO$^+$ lleva a $2(n-1)$. Aunque este resultado no era esperado inicialmente, es una consecuencia de la implementación de nuestro análisis en TACOFlow, y una contribución en sí misma.

Por otro lado, también podemos concluir que el aporte de la propagación de cotas es menos

Método	T. TACO$^+$	T. TACOFlow	Speed-up	Vars TACO$^+$	Reducción
SList, Contains, s20	5,02	5,34	0,94	1743	2,70 %
SList, Insert, s20	15,56	11,30	1,38	3892	11,00 %
SList, Remove, s20	10,08	8,20	1,23	3749	15,92 %
AList, Contains, s20	10,21	8,43	1,21	3573	34,73 %
AList, Insert, s20	9,3	9,04	1,03	4732	0,97 %
AList, Remove, s20	10,24	11,94	0,86	4580	11,55 %
CList, Contains, s20	74,72	47,54	1,57	2530	28,74 %
CList, Add, s20	86,54	45,82	1,89	4512	30,78 %
CList, Remove, s20	423,55	353,66	1,20	5365	11,39 %
AvlTree, FindMax, s20	26,68	25,29	1,06	1412	5,95 %
AvlTree, Find, s20	113,14	119,04	0,95	2505	2,95 %
AvlTree, Insert, s17	29405,43	20.745,00	1,42	204799	30,24 %
BinHeap, FindMin, s20	7,85	10,45	0,75	2224	9,80 %
BinHeap, DecKey, s20	745,27	684,00	1,09	11398	1,06 %
BinHeap, Insert, s20	327,74	283,76	1,15	43077	26,61 %
BinHeap, extMin, s20	488,96	451,19	1,08	55188	27,55 %
TreeSet, Contains, s20	1108,86	769,65	1,44	3032	2,51 %
TreeSet, Add, s17	16383,02	18229,89	0,90	59917	0,82 %
BSTree, Find, s13	6948,85	9190,953	0,76	648	0,15 %
BSTree, Remove, s12	12032,55	14322,00	0,84	2165	14,18 %
BSTree, Add, s11	21919,93	16506,71	1,33	17634	5,58 %

Cuadro 6.6: Tiempos (en segundos) para TACO$^+$ y TACOFlow, junto con el speed-up y la cantidad de variables obtenida en la fórmula proposicional.

importante de lo que se estimaba al principio, pero igualmente relevante. También podemos concluir que es fundamental tener un unroll eficiente para lograr cotas ajustadas.

La cantidad de casos en los que el tiempo de análisis con cotas de dataflow empeora respecto de la versión sin cotas (pero con el nuevo unroll), nos lleva a conjeturar que el predicado de ruptura de simetría de Kodkod [Tor09] es sensible a las cotas intermedias. Sería interesante como trabajo a futuro experimentar aplicando la reducción de variables directamente al nivel de la forma normal conjuntiva de la fórmula proposicional, para ver si podemos obtener las ganancias de las cotas ajustadas sin alterar el funcionamiento del predicado de ruptura de simetría.

6.3. Validez de los resultados

Primero, analicemos la representatividad de los casos de estudio seleccionados. Las estructuras aquí utilizadas se han convertido en benchmarks aceptados para la comparación de herramientas de análisis en la comunidad de análisis de programas (ver por ejemplo [BKM02], [DCJ06], [JV00], [VPP06]). Como se discute en [VPP06], las clases contenedoras están presentes en todos lados, con lo que proporcionar confianza en su correctitud es una tarea importante en sí misma. Además, como se argumenta en [SK09], estas estructuras (que combinan listas y árboles), son representativas de una clase más amplia de estructuras que incluye, por ejemplo, documentos XML, árboles de parseo, etc. Por lo tanto, a pesar de que no es posible realizar afirmaciones generales sobre todas las aplicaciones, pueden ser usadas como una medida relativa de qué tan bien funciona el enfoque propuesto comparado con otras herramientas apuntadas a verificar algoritmos de manipulación del heap.

Por otro lado, en [GRLPF10] se comparó la eficiencia de TACO contra otras herramientas basadas en SAT, model checking y SMT-solving. Esto nos permite por lo tanto concentrarnos

en la comparación entre TACO y los resultados de nuestra técnica.

Finalmente, como ya mencionamos en la sección 6.1, TACO (y por lo tanto TACOFlow) requiere tener pre-computado el conjunto de cotas superiores iniciales. El costo computacional distribuido de este cálculo es significativo respecto del tiempo de análisis secuencial. Sin embargo, ya que estas cotas se calculan una sola vez de acuerdo a los invariantes de cada clase, el costo puede ser amortizado a lo largo de distintos análisis.

Capítulo 7

Conclusiones y trabajo a futuro

En este trabajo presentamos un análisis de propagación de valores apuntado a disminuir los costos de la verificación basada en SAT. Aplicar esta técnica requirió la implementación de un framework de dataflow en TACO, así como la formalización y construcción del análisis.

Como un medio para evitar la pérdida de precisión del cómputo de cotas, modificamos el mecanismo de loop unrolling de TACO, de manera de que preserve la semántica, pero que nos permita obtener cotas más ajustadas. Esto tuvo un gran e inesperado impacto en las mejoras de la performance de la verificación, y lo consideramos como una contribución en sí misma.

Conjeturamos que el hecho de no hallar una clara correlación entre la disminución de variables proposicionales y de tiempo de análisis, donde incluso en algunos casos se produce una pérdida de performance, se debe a los predicados de ruptura simetría de Kodkod. Ya que nuestro trabajo reduce las variables al nivel Alloy, suponemos que Kodkod es sensible a esto, y la eliminación de variables que representan estados intermedios degrada la calidad de los predicados que puede generar.

Si bien, en síntesis, toda la técnica presentada llevó a un importante aumento de performance, creemos firmemente que todavía hay lugar para reducir los costos de verificación utilizando análisis de dataflow.

La dirección natural de investigación para continuar el trabajo presentado, sería someter a prueba la hipótesis sobre la sensibilidad de los predicados de ruptura de simetría de Kodkod. Para esto habría que manipular directamente la fórmula CNF resultante del proceso de traducción, para inyectar a ese nivel las cotas calculadas.

La implementación del framework de dataflow para DynAlloy abre las puertas a la creación de diversos análisis. En particular, una posibilidad interesante sería crear un análisis de points-to, con la intención de ajustar automáticamente, en base a las relaciones de aliasing, el scope de la verificación.

Por último, y en esta misma dirección, pruebas iniciales con un incipiente análisis de points-to nos llevan a creer que los resultados pueden ser prometedores. Como trabajo a futuro, sería interesante ver una implementación formal junto al análisis de dataflow. Sus resultados, en vez de utilizarse para reducir variables proposicionales, servirían en este caso para agregar restricciones adicionales que permitan descartar valuaciones no viables en el proceso de SAT-solving.

Bibliografía

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[App97] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge University Press, 1997.

[App98] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33:17–20, 1998.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods*, pages 1–20. Springer Berlin / Heidelberg, 2004.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133. ACM, 2002.

[BKS08] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: a tool for the analysis of systemc models. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 467–470. Springer-Verlag, 2008.

[BRD09] Jason Belt, Robby, and Xianghua Deng. Sireum/topi ldp: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 355–364. ACM, 2009.

[Bro75] Frederick Phillips Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1975.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, 1991.

[CGLR96] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 570–574. Springer-Verlag, 2005.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[DCJ06] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with sat. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 109–120. ACM, 2006.

[DHR+07] Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *2007 Future of Software Engineering*, pages 120–136. IEEE Computer Society, 2007.

[Dij88] Edsger W. Dijkstra. On the cruelty of really teaching computer science. Circulated privately, dec 1988.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., 1990.

[DVT07] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a sat solver. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 195–204. ACM, 2007.

[DYJ08] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 130–145. Springer-Verlag, 2008.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 333–336. Springer Berlin / Heidelberg, 2004.

[FGLPA05] Marcelo Frias, Juan Pablo Galeotti, Carlos López Pombo, and Nazareno Aguirre. Dynalloy: upgrading alloy with actions. In *Proceedings of the 27th international conference on Software engineering*, pages 442–451. ACM, 2005.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM, 2002.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19:19–32, 1967.

[Gal10] Juan Pablo Galeotti. *Verificación de Software Usando Alloy*. PhD thesis, Universidad de Buenos Aires, 2010.

[GRLPF10] Juan Pablo Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 25–36. ACM, 2010.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

[IYG+05] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *CAV'05*, pages 301–306, 2005.

[Jac00] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, pages 130–139. ACM, 2000.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[JD96] Daniel Jackson and Craig A. Damon. Elements of style: analyzing a software design feature with a counterexample detector. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 239–249. ACM, 1996.

[JV00] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 14–25. ACM, 2000.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, 1976.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer-Verlag New York Inc, 1999.

[SGKP10] Danhua Shao, Divya Gopinath, Sarfraz Khurshid, and Dewayne E. Perry. Optimizing incremental scope-bounded checking with data-flow analysis. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 408–417. IEEE Computer Society, 2010.

[SK09] Junaid Haroon Siddiqui and Sarfraz Khurshid. An empirical study of structural constraint solving techniques. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, pages 88–106. Springer-Verlag, 2009.

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., 1989.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 632–647. Springer-Verlag, 2007.

[Tor09] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, 2009.

[TSJ06] Mana Taghdiri, Robert Seater, and Daniel Jackson. Lightweight extraction of syntactic specifications. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 276–286. ACM, 2006.

[Tur37] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42:230–265, 1937.

[VPP06] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for java containers using state matching. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48. ACM, 2006.

[XA07] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29:16–es, 2007.