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

Tesis de Licenciatura en Ciencias de la Computación

Construcciones de alto nivel como anotaciones para la verificación automática de *software*

Guido de Caso - gdecaso@dc.uba.ar - L.U. 151/03

Directores: Diego Garbervetsky, Daniel Gorín

Diciembre de 2007

Resumen

La verificación automática de *software* es una disciplina que ha crecido mucho en los últimos años, aportando herramientas que con bajo costo permiten aumentar la confianza que se tiene sobre un sistema. En particular la verificación automática de código ha comenzado a ser una alternativa viable, pero gran parte de las herramientas actuales se han montado sobre lenguajes preexistentes. Este enfoque ha brindado soluciones que pueden lidiar sólo con una parte del poder expresivo de los mismos y suelen ser introducidas mediante modificaciones ad-hoc a la sintaxis. Esta metodología no integral desfavorece la adopción de estas técnicas a gran escala. Se presenta en este trabajo un lenguaje imperativo multiprocedural muy básico cuyo diseño está orientado a la verificabilidad. Se dota a este lenguaje de mecanismos de inferencia de precondiciones y postcondiciones que permiten asistir la tarea de anotar ciertos fragmentos del código. Haciendo uso de estas técnicas se extiende el lenguaje con construcciones de iteración de alto nivel para las cuales no es necesario proveer ningún tipo de anotación para su verificación. Se espera que estas caracterísiticas alivien la carga de trabajo que requiere especificar un programa, logrando de esta forma una adopción mayor de las técnicas de verificación automática. Se presenta un prototipo de herramienta que pone a prueba esta hipótesis, trabajando con un conjunto diverso de demostradores de teoremas en paralelo de forma tal de maximizar la cantidad de programas que pueden ser verificados sin intervención del programador.

A la querida memoria de Mario de Caso y Arturo Bilgray.

Agradecimientos

Esta tesis marca la finalización de una etapa que comencé hace ya casi seis años. Miro hacia atrás y no puedo dejar de pensar en todo lo que aprendí, cuánta gente maravillosa conocí y las cosas que viví.

En primer lugar quiero agradecer a la Universidad de Buenos Aires y a todos los que, desde su fundación hasta hoy, lucharon para que sea y siga siendo pública, gratuita, autónoma y laica.

Agradezco también a todos los grandes docentes que tuve: a Mariano Moscato, Nico Rosner, Santi Figueira, Martín Urtasun, Vero Becher, Isabel Méndez Díaz, Javier Marenco, Sergio Mera, entre otros tantos. Su gran entusiasmo es únicamente comparable con su capacidad de transmitir todo lo que saben.

Quiero mencionar especialmente a Charlie López Pombo y a Fer Schapachnik por confiar en mi y brindarme la oportunidad de dar mis primeros pasos como ayudante. Fueron grandes maestros de quienes aprendí mucho más que el arte de la docencia.

Es innumerable la cantidad de gente con la que tuve el agrado de compartir una cursada, una tarde de estudios, una noche de programar TP, un mate en la vereda de la facu... Pablito H., Tiger, Pablo R., Tebi, Román, Mati, Daniel, Fran, el Pocho, Guille, el Cesaro, Monoter, Arti, Christian, Gaboto, Solcí, Lucio, David, Germán, Manix, el Marine, Tommy, el Sabi, Eze, Lata, Toto, J, Pachi, Luigi, Pablito B., ... A todos ellos les debo gran parte de todo lo que viví estos últimos años.

Con mis compañeros de La Mella vivimos humildes victorias y duras derrotas; nunca bajamos los brazos. Desde hace ya más de tres años que junto con ellos laburamos, desde nuestro pequeño aporte, todos los días para hacer más cercana una visión de universidad y de sociedad. A Piter, Ali, Leo, Charlie, Pablo, Mateo, Laura, Nahuel y Beta: más que gracias les pido perdón por no haber podido dedicarme todo lo que quisiera estos últimos tiempos. Con Nacho tuve también el agrado de compartir aventuras inolvidables en nuestro viaje a lo largo de medio país.

Nuevamente quiero agradecer a Schapa por haber confiado en mi para formar parte del grupo de desarrolladores de VInTiMe. Nunca me voy a olvidar de esos meses que pasamos con Lu y con Andran, junto con toda la tropa de Dependex/LaFHIS: Bacardi, Lito, Dieguito P., Aspect, J, Pablo R., Esteban, Alan, Germán, Dipi, Mati L. Agradezco particularmente a Víctor y a Seba por la confianza que siempre me tuvieron y por la forma en la que siempre fueron abiertos a escuchar mis opiniones y tener en cuenta mi punto de vista.

Agradezco enormemente a Juan Pablo Galeotti y a Eduardo Bonelli por lo rápida, precisa y detallada que fue su devolución de este trabajo. Sus comentarios definitivamente sirvieron para completar, revisar, corregir y ampliar el alcance del mismo.

Conocí a Diego Garbervetsky cuando entré en Dependex. De inmediato existía una relación de confianza mutua, como si nos conociéramos de hace años. Vi a Diego laburar mientras dirigía a Dieguito Piemonte, siempre muy dedicado, muy atento. Cuando su "tesista estrella" se recibió le pedí que me dirija y empezamos a laburar muy de a poco pensando y repensando algunos temas. Diego siempre confió en mi y me bancó, con todas mis inquietudes, para sacar adelante este trabajo. Hoy lo veo más

que como un director como un gran amigo.

Este trabajo nunca hubiera existido si no fuera por la idea original de Dani Gorín. Desde el principio siempre se hizo tiempo para ayudarme con mis inquietudes a medida que iban surgiendo. Perdí la cuenta de las veces que golpeé su puerta feliz por algún avance, triste ante un problema, frustrado, harto, ansioso. Dani siempre estuvo ahí con alguna palabra de ánimo y alguna sugerencia. Agradezco infinitamente todo el tiempo que le dedicó a este laburo, así como sus extensivas revisiones, ya sea desde Buenos Aires o desde Nancy.

Realmente no encuentro palabras para darles las gracias debidamente a mis viejos. Desde que tengo memoria ellos estuvieron ahí enseñándome lo hermoso de descubrir, conocer, aprender, pensar, leer, jugar, repensar. Nunca faltó en mi casa algún buen disco, un libro, una peli, un juego, siempre un disfrute. Cuando decidí emprender una carrera universitaria ellos estuvieron ahí siempre atentos, interesados aún si cuando no entienden mucho. Para mis viejos, Cami, Mary y toda mi familia no tengo más que agradecimiento, amor y admiración divergentes.

Índice general

Capítulo 1

Introducción

"*Si quieres conocer el final, mira el comienzo.*"

Proverbio africano

Los sistemas de *software* son día a día más complejos. Cada vez más aspectos de la vida cotidiana descansan de algún modo en procesos controlados por programas. Teléfonos, juguetes, autos, aviones, control de tráfico, entretenimiento. Su mal funcionamiento en general ocasiona disgustos y pérdidas económicas [FPB75]. En algunos casos críticos, puede incluso ocasionar pérdidas humanas.

Asegurar el buen funcionamiento de un sistema de *software* es una tarea ardua. A lo largo de cuatro decadas se utilizaron distintos enfoques con diversos grados de éxito [DHR+07].

El *testing*, por ejemplo, demostró ser útil para encontrar los errores –o *bugs*– más comunes. Sin embargo, aplicaciones comerciales que atraviesan estrictos procesos de *testing* son lanzadas al mercado con decenas de *bugs*.

La verificación formal garantiza la ausencia de errores de un programa respecto de su especificación. La misma puede ser aplicada manualmente sobre pequeños ejemplos. Por ejemplo:

```
1: swap(x, y)
2: :? true
3: :! x = y@pre ∧ y = x@pre
4: {
5:     tmp ← x
6:     x ← y
7:     y ← tmp
8: }
```

Procedimiento *swap*

Se trata de un sencillo procedimiento que realiza el *swap* –o intercambio– entre dos variables enteras. La notación **:?** indica la precondición y **:!** indica la postcondición.

Una vez definida formalmente la semántica operacional de este lenguaje podría demostrarse que, luego de la línea 5, el valor de *tmp* es x**@pre**. De modo similar se puede ver que, luego de la línea 6, el valor de x es y**@pre**. Finalmente, luego de la línea 7, el valor de y es *tmp*, que a su vez es x**@pre**.

Ciertamente este enfoque de verificación formal manual tiene serios problemas de escalabilidad, por lo cual la automatización es una característica altamente deseable.

La verificación formal automatizada ha avanzado mucho en los últimos años, con casos de renombre como por ejemplo Spec# [BLS04]. Sin embargo diversos problemas hacen que no se haya alcanzado aún la capacidad necesaria para poder lidiar con programas a gran escala.

Por un lado, esta técnica requiere una gran cantidad de trabajo extra, en forma de anotaciones, por parte del programador. Para cada procedimiento, método o función se debe especificar su precondición y postcondición. Cada ciclo debe estar acompañado por su invariante y también su función variante (en caso de querer demostrar automáticamente su terminación).

Por otra parte, el poder de demostración automática de teoremas de las herramientas actuales ha ido en aumento, pero sigue siendo limitado. Si bien se están realizando muchos trabajos prometedores en este área [BB04, BLM05], la capacidad actual para resolver problemas es despareja entre una herramienta y otra; no existe una que se destaque por sobre las demás en todo contexto de uso.

1.1. Objetivo

Nuestra principal hipótesis de trabajo es que las herramientas habituales de verificación se incorporan de manera ad-hoc a lenguajes preexistentes. Estos lenguajes fueron diseñados sin tener en cuenta la verificabilidad de sus programas, motivo por el cual las herramientas de verificación que se construyen sobre los mismos presentan una serie de inconvenientes.

Por un lado las anotaciones sobre comportamiento esperado aparecen como un parche al lenguaje, siendo en varios casos presentadas a través de comentarios. Por otra parte surgen redundancias y superposiciones entre los mecanismos que el lenguaje provee para asegurar su corrección: las verificaciones sintácticas y de tipos se realizan por un lado, la verificación de especificaciones por otro.

Este aspecto asimétrico y poco prolijo que presentan estas herramientas conspiran contra la aceptabilidad final por parte de la industria de este tipo de técnicas. Por otra parte, si se presentaran estos mecanismos de forma nativa e integrada, probablemente serían aceptados debido a su utilidad para evitar errores. Por ejemplo los sistemas de tipos hoy en día se encuentran presentes en difundidos entornos de desarrollo que aprovechan sus características sin generar rechazo por parte de los usuarios.

Partiendo de esta hipótesis, uno de nuestros objetivos es investigar las posibilidades que ofrece un lenguaje de programación cuando se lo diseña en función de su verificabilidad. Para ello, es intereante analizar hasta dónde se puede avanzar mejorando su usabilidad sin sacrificar las capacidades de verificación.

1.2. Contribuciones

Diseñamos un nuevo lenguaje de programación pensado como un *core* minimal verificable. Integramos su sistema de tipos con el manejo de precondiciones y postcondiciones, eliminando de esta forma las asimetrías que se pueden encontrar en las herramientas actuales.

Presentamos una serie de conceptos y técnicas que permiten inferir precondiciones y postcondiciones de fragmentos de programas. Haciendo uso de estos mecanismos dotamos al lenguaje con características que permiten asistir al usuario en la tarea de especificar ciertas partes de su código.

Adicionalmente incorporamos nuevas construcciones de iteración que no requieren anotaciones de ningún tipo. Las mismas aprovechan los mecanismos de inferencia de precondiciones y postcondiciones para obtener candidatos a invariantes.

Se puede entender el aporte hecho en este trabajo como dividido en dos dimensiones, o ejes. Por

un lado presentamos un nuevo lenguaje diseñado para ser verificable, junto con sus extensiones y mecanismos que permiten reducir la carga de especificación. Por otra parte proponemos y discutimos un prototipo de herramienta verificadora que lleva a la práctica una gran parte de las ideas que presentamos en este trabajo.

Para atacar el problema de las capacidades dispares de los demostradores automáticos, el prototipo que presentamos trabaja combinando varios de ellos. De esta forma se pueden aprovechar los puntos fuertes de cada uno de estos demostradores, maximizando la cantidad de problemas que se pueden tratar con éxito.

1.3. Trabajo relacionado

Los primeros trabajos sobre verificación de programas podrían atribuirse a Floyd [Flo67] y a Hoare [Hoa69] a finales de la década del sesenta. A partir de ese entonces se desarrolló una gran cantidad de sistemas cuyo objetivo es asistir mecánicamente la prueba de corrección de programas.

Algunos de estos primeros sistemas estaban basados en reglas de reducción para las condiciones de verificación que generaban [Kin70]. Otros dependían fuertemente de la intervención del usuario, quien debía proveer la reducción y verificación final [GLB75, LGvH+79].

A finales de los años setenta surgen también lenguajes de programación cuyo diseño tiene entre otros objetivos facilitar la verificación de programas, por ejemplo los lenguajes Gypsy [Amb77] y Euclid [LHL+77]. Sin embargo ninguno de estos lenguajes fue diseñado para ser de propósito general, por el contrario, su foco está puesto en aplicaciones vinculadas al manejo de procesos y otras tareas que suelen aparecer al implementar *software* de base.

A pesar de estos primeros años con trabajos prometedores, a comienzos de la década del ochenta surge una gran controversia con respecto a la verificación de programas debido a la publicación de un artículo de De Millo, Lipton y Perlis [MLP79]. En el mismo enuncian que la verificación de programas está destinada a ser un fracaso debido a la ausencia de procesos sociales de validación y aceptación de los resultados. Este trabajo genera un cierto escepticismo sobre la mera idea de obtener una herramienta verificadora de caracter confiable.

A mediados de los años ochenta se abandonan parcialmente los esfuerzos por construir verificadores y surgen lenguajes cuyas anotaciones son convertidas en verificaciones en *runtime*. Quizás el ejemplo más notorio es Eiffel [Mey97], un lenguaje de programación orientado a objetos cuya biblioteca estándar se encuentra bien documentada con contratos. El mismo permite que los programas sean ampliados con anotaciones para precondiciones y postcondiciones, sin embargo las mismas únicamente pueden ser comprobadas en tiempo de ejecución. Esto trae problemas de eficiencia y conlleva la incerteza de que ninguna ejecución vaya a violar algún contrato. Algo similar sucede con el mucho más reciente lenguaje de programación D [Bri02].

La herramienta ESC/Java [FLL+02], descendiente directa de ESC/Modula-3 [DLNS98], utiliza demostradores automáticos de teoremas para tratar de ubicar errores comunes de programación. Utiliza anotaciones por parte del usuario, sin embargo sus conclusiones no son *sound* ni *complete*. De esta forma la presencia de un "error" en un determinado programa no implica que tal problema realmente exista; la ausencia de "errores" tampoco es garantía de su correcta ejecución respecto de la especificación.

El entorno de verificación Jive [MPH00] consiste principalmente de un demostrador de propósito específico, diseñado para razonar sobre lógica de Hoare sobre un fragmento de Java. Se comienza con un programa sin ningún tipo de anotación y a partir de allí el usuario incorpora o refina afirmaciones sobre el comportamiento. En esta herramienta las verificaciones tienen por objetivo demostrar la corrección

parcial del programa en cuestión.

Krakatoa [MPMU04] es una herramienta que toma programas Java y los traduce a un lenguaje interno llamado Why. Otras herramientas convierten esta representación intermedia en condiciones que deben ser verificadas para garantizar la corrección del programa. Finalmente se utiliza el asistente de demostración Coq para llevar a cabo las pruebas.

El proyecto Loop [JP03] ataca el problema desde otro ángulo. Se traduce un programa Java a elementos semánticos del demostrador semiautomático PVS. En este caso, las reglas de la lógica de Hoare se presentan mediante una teoría en dicho demostrador. Las mismas no predican sobre la sintáxis del lenguaje, sino sobre elementos de PVS.

En [JMR04] Jacobs realiza una comparación de ESC/Java, Jive, Krakatoa y Loop sobre un caso de estudio completo. Este trabajo sirve como análisis para conocer el estado del arte de diversos enfoques a la hora de atacar un problema real.

El lenguaje de especificación JML [LBR99] permite agregar contratos y otro tipo de información de diseño a programas Java. Desde su creación se lo ha usado para diversos fines que van desde documentación hasta verificación automatizada de ciertos fragmentos del mismo [BCC+05].

En la tesis doctoral de Necula [Nec98] se presenta, entre otras cosas, el concepto de generación de condiciones de verificación. Se trata de obtener un predicado cuya validez indica una propiedad de un programa determinado. Esta estrategia permite atacar el problema de la verificación como un problema de demostración de teoremas.

La herramienta Spec# [BLS04] es una extensión al lenguaje C# con especificación de precondiciones, postcondiciones, invariantes de clase e invariantes de ciclos. La misma permite determinar si un programa satisface su especificación. Esto se realiza a través de una traducción a un lenguaje intermedio de caracter no estructurado llamado Boogie-PL. Actualmente se lo considera el entorno de demostración automática de referencia.

Enumeramos a continuación una serie de puntos en los cuales Spec# se diferencia con el trabajo que se presenta en esta tesis:

- A nivel teórico, Spec# busca probar corrección en caso de que el programa termine; no se prueba terminación. La herramienta presentada en nuestro trabajo busca probar corrección y terminación.
- Desde el punto de vista del diseño del lenguaje, Spec# es montado sobre el lenguaje de propósito general C#, que posee características que dificultan seriamente su verifcabilidad. Algunas de ellas son: concurrencia, *aliasing*, *dynamic binding*, *reflection*, etc. En cambio, la herramienta que se presenta en nuestro trabajo tiene como base un lenguaje *ad-hoc* cuyo diseño persigue metas de verificabilidad por sobre otros aspectos.
- Por último, con respecto al poder de prueba de su implementación actual, Spec# tiene como base el demostrador Z3 [BLM05], con lo cual su poder de verificación está atado a las capacidades y limitaciones de esta herramienta. El prototipo desarrollado como parte de nuestro trabajo utiliza Z3 combinado con otros demostradores, por lo cual la capacidad de verificación es potencialmente más amplia.

Nuestro trabajo involucra un sistema de tipos fuerte que permite especificar el comportamiento de un programa de forma precisa. Los tipos dependientes [NPS90, McK06] permiten razonar con este tipo de especificaciones ricas. En general, al trabajar con tipos dependientes, el diseñador del lenguaje provee reglas que habiliten al *type checker* a razonar sintácticamente sobre los juicios de tipado. Esto incluye introducir reglas que permitan modelar propiedades aritméticas, de arreglos, etc. En cambio, nuestro

enfoque es utilizar la relación de fuerza entre fórmulas como un elemento semántico dentro de las reglas de tipado.

1.4. Estructura de la tesis

En el Capítulo 2 se introduce el lenguaje que es usado a lo largo de toda la tesis. Se presentan formalmente su sintaxis y su semántica operacional. También se muestran algunos ejemplos.

Para poder realizar verificaciones de programas escritos en este lenguaje, en el Capítulo 3 se lo dota de una noción alternativa de semántica. La misma puede ser entendida como un sistema de deducción similar a un sistema de tipos donde un programa bien tipado es correcto con respecto a su especificación: progresa, termina y cumple con su contrato. También se presentan mecanismos que permiten inferir precondiciones y postcondiciones para fragmentos de programas. Aprovechando estos mecanismos se presentan una serie de técnicas que permiten asistir la escritura de anotaciones.

Basándose en los conceptos y métodos presentados, en el Capítulo 4 se presenta la técnica que permite ampliar el lenguaje con nuevas construcciones de iteración que no requieren especificación. Se presentan tres casos concretos de extensiones y algunos ejemplos de su utilización donde puede verse su capacidad de reducir la cantidad de anotaciones que deben escribirse.

En el Capítulo 5 se aplican los conceptos introducidos desde un punto de vista implementativo y se discuten las consideraciones a tener en cuenta para llevarlos a la práctica. Asimismo, se comentan las características principales del prototipo de herramienta que fue implementado y se muestran algunos ejemplos de programas que pueden ser verificados utilizándolo.

Finalmente, en el Capítulo 6 se presentan las conclusiones de esta tesis y se discuten algunas alternativas de trabajo a futuro.

Capítulo 2

Sintaxis y semántica operacional

"*The limits of my language means the limits of my world.*"

Ludwig Josef Johann Wittgenstein

En este capítulo se presenta la versión básica del lenguaje de programación sobre el que se trabaja en el resto de esta tesis.

Se trata de un lenguaje imperativo con dos tipos de datos básicos: los enteros y los arreglos de enteros. Cuenta con llamados a procedimientos, pero no admite recursión.

A diferencia de otros lenguajes de tipo *while* introducidos en la bibliografía [Mit96, Win93], aquí las anotaciones que permiten realizar verificaciones aparecen como un elemento básico, inseparable del lenguaje en sí mismo.

2.1. Sintaxis

Las variables son el elemento sintáctico básico. Se utilizará a lo largo de la tesis un conjunto infinito de símbolos de variable denominado Var. Una clasificación determina si una variable conocida tiene tipo entero o arreglo.

Definición 2.1 (Clasificación)**.**
Una *clasificación* es cualquier función parcial $\Gamma : \text{Var} \to \{\textsc{Int}, \textsc{Array}\}$. Sobre las constantes simbólicas Int y Array sólo se asumirá que son distintas.

Definición 2.2 (Extension a una clasificación)**.**
Sea una clasificación $\Gamma : V \to \Gamma$, una variable $v \notin V$ y una constante simbólica $t \in \{\textsc{Int}, \textsc{Array}\}$. Se define la *extensión a la clasificación* Γ *con la variable* v *de tipo* t notada:

$$\Gamma\{v \mapsto t\} : V \cup \{v\} \to \{\textsc{Int}, \textsc{Array}\}$$

Y dada por:

$$\Gamma\{v \mapsto t\}(v') \;=\; \begin{cases} t & \text{si } v' = v \\ \Gamma(v') & \text{sino} \end{cases}$$

2.1.1. Expresiones

Se cuenta con tres tipos distintos de expresiones:

- Las expresiones enteras se utilizan para realizar asignaciones sobre variables de tipo INT, para indexar arreglos, especificar funciones variantes, etc.
- Las expresiones arreglo pueden usarse para realizar asignaciones sobre variables de tipo ARRAY.
- Las expresiones booleanas sirven como guarda para condicionales y ciclos. Se utilizan también para especificar anotaciones tales como invariantes, precondiciones y postcondiciones.

Definición 2.3 (Expresiones enteras)**.**
A partir de una clasificación Γ se puede construir inductivamente IntExp_Γ, el conjunto de *expresiones enteras* con variables en Γ, tomando al más chico de todos los conjuntos que cumplen las siguientes reglas:

$$\frac{\Gamma(v) = \text{INT}}{v \in \text{IntExp}_\Gamma}\ \text{IE-VAR} \qquad \frac{\Gamma(v) = \text{INT}}{v\mathbf{@pre} \in \text{IntExp}_\Gamma}\ \text{IE-VAR@PRE}$$

$$\frac{A \in \text{ArrayExp}_\Gamma \quad i \in \text{IntExp}_\Gamma}{A[i] \in \text{IntExp}_\Gamma}\ \text{IE-GET}$$

$$\frac{n \in \mathbb{Z}}{\underline{n} \in \text{IntExp}_\Gamma}\ \text{IE-NUM} \qquad \frac{A \in \text{ArrayExp}_\Gamma}{|A| \in \text{IntExp}_\Gamma}\ \text{IE-SIZE}$$

$$\frac{i_1, i_2 \in \text{IntExp}_\Gamma}{i_1 + i_2 \in \text{IntExp}_\Gamma}\ \text{IE-SUM} \qquad \frac{i_1, i_2 \in \text{IntExp}_\Gamma}{i_1 - i_2 \in \text{IntExp}_\Gamma}\ \text{IE-SUBS}$$

$$\frac{i_1, i_2 \in \text{IntExp}_\Gamma}{i_1 * i_2 \in \text{IntExp}_\Gamma}\ \text{IE-MUL} \qquad \frac{i_1, i_2 \in \text{IntExp}_\Gamma}{i_1 / i_2 \in \text{IntExp}_\Gamma}\ \text{IE-DIV}$$

Definición 2.4 (Expresiones arreglo)**.**
A partir de una clasificación Γ se puede construir inductivamente ArrayExp_Γ, el conjunto de *expresiones arreglo* con variables en Γ, tomando al más chico de todos los conjuntos que cumplen las siguientes reglas:

$$\frac{\Gamma(v) = \text{ARRAY}}{v \in \text{ArrayExp}_\Gamma}\ \text{AE-VAR} \qquad \frac{\Gamma(v) = \text{ARRAY}}{v\mathbf{@pre} \in \text{ArrayExp}_\Gamma}\ \text{AE-VAR@PRE}$$

$$\frac{A \in \text{ArrayExp}_\Gamma \quad i_1, i_2 \in \text{IntExp}_\Gamma}{\mathbf{update}\ A\ \mathbf{on}\ i_1\ \mathbf{with}\ i_2 \in \text{ArrayExp}_\Gamma}\ \text{AE-UPDATE}$$

$$\frac{i_1, i_2 \in \text{IntExp}_\Gamma}{\mathbf{array}[i_1]\ \mathbf{of}\ i_2 \in \text{ArrayExp}_\Gamma}\ \text{AE-CONST}$$

Por ejemplo, si $\Gamma(B) = \text{ARRAY}$ y $\Gamma(\mathit{index}) = \text{INT}$, luego:

$$\mathbf{update}\ B\ \mathbf{on}\ \mathit{index}\ \mathbf{with}\ \underline{0} \in \text{ArrayExp}_\Gamma$$

$$(\mathbf{update}\ B\ \mathbf{on}\ \mathit{index}\ \mathbf{with}\ \underline{0})[\mathit{index} + \underline{1}] \in \text{IntExp}_\Gamma$$

Por otra parte, la expresión arreglo **array**$[i_1]$ **of** i_2 se utiliza para referirse a constantes de arreglo. Por ejemplo **array**$[\underline{2}]$ **of** $\underline{15}$ representa un arreglo de tamaño 2, cuyos elementos valen todos 15.

Definición 2.5 (Expresiones booleanas)**.**
A partir de una clasificación Γ se puede construir inductivamente $\mathrm{BoolExp}_\Gamma$, el conjunto de *expresiones booleanas* con variables en Γ, tomando al más chico de todos los conjuntos que cumplen las siguientes reglas:

$$\frac{i_1, i_2 \in \mathrm{IntExp}_\Gamma}{i_1 = i_2 \in \mathrm{BoolExp}_\Gamma}\ \text{BE-IEQ} \qquad \frac{A_1, A_2 \in \mathrm{ArrayExp}_\Gamma}{A_1 = A_2 \in \mathrm{BoolExp}_\Gamma}\ \text{BE-AEQ}$$

$$\frac{i_1, i_2 \in \mathrm{IntExp}_\Gamma}{i_1 < i_2 \in \mathrm{BoolExp}_\Gamma}\ \text{BE-ILT}$$

$$\frac{b_1, b_2 \in \mathrm{BoolExp}_\Gamma}{b_1 \wedge b_2 \in \mathrm{BoolExp}_\Gamma}\ \text{BE-AND} \qquad \frac{b \in \mathrm{BoolExp}_\Gamma}{\neg b \in \mathrm{BoolExp}_\Gamma}\ \text{BE-NOT}$$

$$\frac{v \notin \mathrm{dom}(\Gamma) \quad b \in \mathrm{BoolExp}_{\Gamma\{v \mapsto \textsc{Int}\}}}{\forall\ v : \textsc{Int}\ (b) \in \mathrm{BoolExp}_\Gamma}\ \text{BE-IFORALL}$$

$$\frac{v \notin \mathrm{dom}(\Gamma) \quad b \in \mathrm{BoolExp}_{\Gamma\{v \mapsto \textsc{Array}\}}}{\forall\ v : \textsc{Array}\ (b) \in \mathrm{BoolExp}_\Gamma}\ \text{BE-AFORALL}$$

Adicionalmente, se cuenta con las siguientes abreviaturas:

$$\begin{aligned}
\textbf{true} &\stackrel{\text{def}}{=} \underline{0} = \underline{0} \\
\textbf{false} &\stackrel{\text{def}}{=} \underline{0} < \underline{0} \\
b_1 \vee b_2 &\stackrel{\text{def}}{=} \neg(\neg b_1 \wedge \neg b_2) \\
b_1 \Rightarrow b_2 &\stackrel{\text{def}}{=} \neg b_1 \vee b_2 \\
b_1 \Leftrightarrow b_2 &\stackrel{\text{def}}{=} (b_1 \Rightarrow b_2) \wedge (b_2 \Rightarrow b_1) \\
\exists\ v : \textsc{Int}\ (b) &\stackrel{\text{def}}{=} \neg(\forall\ v : \textsc{Int}\ (\neg b)) \\
\exists\ v : \textsc{Array}\ (b) &\stackrel{\text{def}}{=} \neg(\forall\ v : \textsc{Array}\ (\neg b)) \\
i_1 \neq i_2 &\stackrel{\text{def}}{=} \neg(i_1 = i_2) \\
A_1 \neq A_2 &\stackrel{\text{def}}{=} \neg(A_1 = A_2) \\
i_1 \leq i_2 &\stackrel{\text{def}}{=} i_1 < i_2 \ \vee\ i_1 = i_2 \\
i_1 > i_2 &\stackrel{\text{def}}{=} i_2 < i_1 \\
i_1 \geq i_2 &\stackrel{\text{def}}{=} i_2 \leq i_1
\end{aligned}$$

Por último se utilizarán las siguientes abreviaturas, donde $\prec_a, \prec_b \in \{<, \leq\}$.:

- Ubicar una expresión en rango.

$$i_1 \prec_a i_2 \prec_b i_3 \stackrel{\text{def}}{=} i_1 \prec_a i_2 \ \wedge\ i_2 \prec_b i_3$$

- Cuantificación acotada de enteros.

$$\forall v \ / \ i_1 \prec_a v \prec_b i_2 : \ b \stackrel{\text{def}}{=} \forall \ v : \textsc{Int} \ (i_1 \prec_a v \prec_b i_2 \ \Rightarrow \ b)$$
$$\exists v \ / \ i_1 \prec_a v \prec_b i_2 : \ b \stackrel{\text{def}}{=} \exists \ v : \textsc{Int} \ (i_1 \prec_a v \prec_b i_2 \ \wedge \ b)$$

Por ejemplo, si $\Gamma(B) = \textsc{Array}$ y $k \notin \text{dom}(\Gamma)$, luego:

$$|B| < \underline{15} \ \wedge \ \forall k \ / \ \underline{0} \leq k < \underline{10} : \ B[k] = \underline{-1} \in \text{BoolExp}_\Gamma$$

2.1.2. Programas

En esta sección se introduce el concepto de programa. Un programa está compuesto por una cantidad finita de procedimientos. Cada uno de ellos tiene un nombre único, una lista de parámetros, una precondición, una postcondición y una sentencia que representa su cuerpo.

La semántica de pasaje de parámetros será siempre por referencia, tanto para enteros como para arreglos. Se fuerza que los llamados a procedimientos sean siempre con parámetros concretos distintos para evitar la aparición de *aliasing* entre distintos parámetros formales.

Definición 2.6 (Programas)**.**
Se define el conjunto de *programas* Program como el más chico que cumple las siguientes reglas:

- Programa vacío:

$$\frac{}{\emptyset \in \text{Program}} \ \text{PROG-EMPTY}$$

  Un programa que no contiene ningún procedimiento.

- Extensión de un programa:

$$\frac{\begin{array}{c}\pi \in \text{Program} \quad proc \notin \pi \\ \text{dom}(\Gamma_{proc}) = \{p_1, \ldots, p_k\} \quad i \neq j \Rightarrow p_i \neq p_j \\ pre, post \in \text{BoolExp}_{\Gamma_{proc}} \\ s \in \text{Sentence}_{\Gamma_{proc},\ \pi,\ \Gamma'}\end{array}}{\pi, \ proc(p_1, \ldots, p_k) \ \textbf{:?} \ pre \ \textbf{:!} \ post \ \{ \ s \ \} \in \text{Program}} \ \text{PROG-EXTEND}$$

  Se parte de un programa π y se lo extiende con un nuevo procedimiento *proc*.

  Se requiere una clasificación inicial Γ_{proc} para sus parámetros formales $p_1, \ldots, p_k$, los cuales deben ser todos distintos.

  Además, se requieren dos expresiones booleanas que predican sobre los parámetros formales, una como precondición y otra como postcondición.

  Por último se necesita una sentencia para el cuerpo del procedimiento, las sentencias son definidas a continuación en la sección 2.1.3.

Si $proc \in \pi$, se usa:

- Γ_{proc} para referirse a su clasificación inicial.

- pars($proc$) para referirse a la lista de sus parámetros formales.
- P_{proc} para referirse a la expresión booleana que representa su precondición.
- Q_{proc} para referirse a la expresión booleana que representa su postcondición.
- body($proc$) para referirse a la sentencia que representa su cuerpo.

2.1.3. Sentencias

En esta sección se introducen las sentencias que conforman los posibles cuerpos para un procedimiento. Los llamados a procedimientos son controlados de forma estática asegurándose de que existan y la cantidad y tipo de parámetros sea adecuada. Para ello es necesario conocer qué procedimientos ya existen, cuántos parámetros tienen, de qué tipos. El conjunto de sentencias es paramétrico en un programa y únicamente puede llamar a procedimientos que figuren en él.

Por otra parte una sentencia puede eventualmente ampliar una clasificación para reflejar información sobre las variables que crea. Esto se refleja en el hecho de que el conjunto de sentencias es paramétrico en dos clasificaciones: una antes y otra después de una eventual ejecución de la sentencia.

Definición 2.7 (Sentencias)**.**
Sea una clasificación Γ, un programa π y otra clasificación Γ'.

Se puede construir inductivamente $\mathrm{Sentence}_{\Gamma,\pi,\Gamma'}$, el conjunto de *sentencias* que empieza con variables en Γ, llama potencialmente a procedimientos en π y termina con variables en Γ'.

Cada construcción es acompañada de una breve descripción informal, para ver su semántica operacional, referirse a la Definición 2.17.

Se toma al más chico de todos los conjuntos que cumplen las siguientes reglas:

- Asignación a una variable entera:

$$\frac{\Gamma(v) = \textsc{Int} \quad i \in \mathrm{IntExp}_{\Gamma}}{v \leftarrow i \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma}}\ \textsc{sent-iassign}$$

  Si se sabe que el tipo de v es entero y se tiene una expresión entera, se puede asignar el valor de tal expresión a la variable v. Observar que, la definición de expresiones admite referirse a los valores que una variable tenía al comienzo, por ejemplo:

$$x \leftarrow (A\mathbf{@pre})[\underline{20}] + \underline{1}$$

- Asignación a una variable arreglo:

$$\frac{\Gamma(v) = \textsc{Array} \quad A \in \mathrm{ArrayExp}_{\Gamma}}{v \leftarrow A \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma}}\ \textsc{sent-aassign}$$

  Similar al caso anterior, pero asignando una variable A de tipo arreglo con una expresión de tipo arreglo.

- Creación de una variable entera:

$$\frac{v \notin \mathrm{dom}(\Gamma) \quad i \in \mathrm{IntExp}_\Gamma}{\textbf{int}\ v \leftarrow i \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma\{v \mapsto \mathrm{INT}\}}}\ \text{SENT-IDEF}$$

Si v no está definida se puede crear un nuevo entero con tal nombre. Se le asigna un valor inicial determinado por la expresión entera i.

Observar que la clasificación luego de esta sentencia incluye a una nueva variable v de tipo entero.

- Creación de una variable arreglo:

$$\frac{v \notin \mathrm{dom}(\Gamma) \quad A \in \mathrm{ArrayExp}_\Gamma}{\textbf{array}\ v \leftarrow A \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma\{A \mapsto \mathrm{ARRAY}\}}}\ \text{SENT-ADEF}$$

Similar al caso anterior, pero se crea un arreglo inicializado por la expresión arreglo A.

- Sentencia vacía:

$$\frac{}{\textbf{skip} \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma}}\ \text{SENT-SKIP}$$

Una sentencia que no afecta la ejecución.

- Secuenciamiento de sentencias:

$$\frac{s_1 \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma'} \quad s_2 \in \mathrm{Sentence}_{\Gamma',\pi,\Gamma''}}{s_1\ s_2 \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma''}}\ \text{SENT-SEQ}$$

Se secuencian las sentencias s_1 y s_2.

- Condicional:

$$\frac{\begin{array}{c} g \in \mathrm{BoolExp}_\Gamma \quad g \text{ no tiene cuantificadores} \\ s_1 \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma'} \quad s_2 \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma''} \end{array}}{\textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi} \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma}}\ \text{SENT-IF}$$

La construcción condicional usual que, dada una expresión booleana g que sirve como guarda, continúa por la rama que corresponda según su valor. Luego olvida cualquier variable creada por la misma.

Observar que la expresión booleana que se utiliza como guarda no puede contener ningún cuantificador. Más adelante se verá la importancia de esta restricción.

- Ciclo:

$$\frac{\begin{array}{c} g \in \mathrm{BoolExp}_\Gamma \quad g \text{ no tiene cuantificadores} \\ inv \in \mathrm{BoolExp}_\Gamma \quad var \in \mathrm{IntExp}_\Gamma \quad s \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma'} \end{array}}{\textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \textbf{od} \in \mathrm{Sentence}_{\Gamma,\pi,\Gamma}}\ \text{SENT-WHILE}$$

Se trata de una construcción de control de tipo iterativa, pero incluye especificación para garantizar su terminación y postcondición. Además de la guarda g y el cuerpo s, se requiere una expresión booleana inv que determina el invariante, y una expresión entera var que determina el variante.

Tal como en la construcción condicional, la expresión booleana no puede tener cuantificadores.

- Llamado a un procedimiento:

$$\frac{\begin{array}{c} proc \in \pi \\ i \neq j \Rightarrow cp_i \neq cp_j \quad \text{pars}(proc) = p_1, \ldots, p_k \\ \Gamma(cp_i) = \Gamma_{proc}(p_i) \end{array}}{\textbf{call } proc(cp_1, \ldots, cp_k) \in \text{Sentence}_{\Gamma,\pi,\Gamma}} \text{ SENT-CALL}$$

  Para realizar un llamado a un procedimiento, el mismo debe estar definido en π. Por otra parte, los parámetros concretos cp_i deben ser provistos sin repetidos, en cantidad y tipos adecuados con respecto a los parámetros formales p_i.

  Observar que si se permitieran repetidos entre los parámetros formales se estaría produciendo *aliasing*, lo cual dificulta la verificación de programas ampliamente.

Utilizaremos además las siguientes abreviaturas:

- Asignación de un elemento en un arreglo:

$$v[i_1] \leftarrow i_2 \stackrel{\text{def}}{=} v \leftarrow (\textbf{update } v \textbf{ on } i_1 \textbf{ with } i_2)$$

  Siempre y cuando $\Gamma(v) = \text{ARRAY}$.

- Condicionales sin rama "else":

$$\textbf{if } g \textbf{ then } s \textbf{ fi} \stackrel{\text{def}}{=} \textbf{if } g \textbf{ then } s \textbf{ else skip fi}$$

Observación 2.1. La forma en la que se construyen los programas, a partir de extensiones con nuevos procedimientos cuyo cuerpo llama a procedimientos que ya existen, asegura la ausencia de recursión.

A lo largo del resto de la tesis se usa el concepto de variables locales de una sentencia, definido a continuación.

Definición 2.8 (Variables locales)**.**
Se llama localVars a la función que, dada una sentencia $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$, devuelve el conjunto de *variables locales* a la misma. O sea, el conjunto de variables creadas por s, definido como:

$$\text{localVars}(s) = \text{dom}(\Gamma') \setminus \text{dom}(\Gamma)$$

2.2. Semántica operacional

Se introduce en esta sección el concepto de ejecución de una sentencia. Con tal fin, es necesario primero introducir el concepto de valuación, que sirve para conocer el valor de cada una de las variables de un programa. Se puede pensar una valuación como la abstracción de la memoria en una computadora convencional.

2.2.1. Valuaciones

Antes de presentar la definición de una valuación, es necesario contar con una estructura matemática para representar arreglos finitos de enteros.

Definición 2.9 (Arreglos de enteros)**.**
Se define el conjunto de *arreglos de enteros* $\mathbb{A}$ como:

$$\mathbb{A} = \{[]\} \cup \{[a_0, \ldots, a_n] \mid n \geq 0 \wedge \forall\ 0 \leq i \leq n :\ a_i \in \mathbb{Z}\}$$

Dado un arreglo $\mathcal{A} \in \mathbb{A}$:

- Su tamaño está dado por $\|\mathcal{A}\|$.

$$\begin{aligned} \|[]\| &= 0 \\ \|[a_0, \ldots, a_n]\| &= n + 1 \end{aligned}$$

- Con $0 \leq i < \|\mathcal{A}\|$, el elemento en la posición i de $\mathcal{A}$ es notado $\mathcal{A}[i]$.

$$[a_0, \ldots, a_n][i] = a_i$$

- Con $0 \leq j < \|\mathcal{A}\|$, un nuevo arreglo entendido como la actualización de la posición j con un nuevo elemento $m \in \mathbb{Z}$ es notado $\mathcal{A}\{[j] \mapsto m\}$.

$$\mathcal{A}\{[j] \mapsto m\}[i] = \begin{cases} m & \text{si } i = j \\ A[i] & \text{sino} \end{cases}$$

Dados dos arreglos $\mathcal{A}, \mathcal{B} \in \mathbb{A}$, se dirá que son iguales y se notará $\mathcal{A} = \mathcal{B}$ si y sólo si:

$$\|\mathcal{A}\| = \|\mathcal{B}\| \ \wedge\ \forall\ 0 \leq i < \|\mathcal{A}\| :\ \mathcal{A}[i] = \mathcal{B}[i]$$

Una valuación asigna un valor –en $\mathbb{Z}$ ó $\mathbb{A}$, según corresponda– a cada una de las variables en una clasificación Γ. Como las expresiones pueden referirse a valores "**@pre**" luego es necesario asignar también valores iniciales para cada variable.

Definición 2.10 (Valuaciones)**.**
Dada una clasificación Γ se define el conjunto de *valuaciones* M_Γ al compuesto por elementos:

$$\mu = \langle \mu_{\text{int}},\ \mu_{\text{int@pre}},\ \mu_{\text{array}},\ \mu_{\text{array@pre}} \rangle$$

Donde:

- $\mu_{\text{int}} : \{v \mid \Gamma(v) = \textsc{Int}\} \to \mathbb{Z}$
  Determina el valor actual de cada variable entera en Γ.
- $\mu_{\text{int@pre}} : \{v \mid \Gamma(v) = \textsc{Int}\} \to \mathbb{Z}$
  Determina el valor al comienzo de cada variable entera en Γ.

- $\mu_{\text{array}} : \{A \mid \Gamma(A) = \textsc{Array}\} \to \mathbb{A}$
  Determina el valor actual de cada variable arreglo en Γ.
- $\mu_{\text{array@pre}} : \{A \mid \Gamma(A) = \textsc{Array}\} \to \mathbb{A}$
  Determina el valor al comienzo de cada variable entera en Γ.

Las asignaciones y otras operaciones alteran el valor actual para algunas variables. Es necesario que las valuaciones puedan ser extendidas para reflejar estos efectos sobre las variables.

Definición 2.11 (Extensiones a una valuación)**.**
Se definen las siguientes *extensiones a una valuación* $\mu \in \mathrm{M}_\Gamma$:

- Si $\Gamma(v) = \textsc{Int}$ y $n \in \mathbb{Z}$, luego:
  $$\mu' = \mu\{v \mapsto n\}$$
  es una valuación en M_Γ donde el único cambio es que ${\mu_{\text{int}}}'(v) = n$.
- Si $v \notin \mathrm{dom}(\Gamma)$ y $n \in \mathbb{Z}$, luego:
  $$\mu' = \mu\{\text{int } v \mapsto n\}$$
  es una valuación en $\mathrm{M}_{\Gamma\{v \mapsto \textsc{Int}\}}$ donde los cambios son que:
  $${\mu_{\text{int}}}'(v) = {\mu_{\text{int@pre}}}'(v) = n$$
- Si $\Gamma(v) = \textsc{Array}$ y $\mathcal{A} \in \mathbb{A}$, luego:
  $$\mu' = \mu\{v \mapsto \mathcal{A}\}$$
  es una valuación en M_Γ donde el único cambio es que ${\mu_{\text{array}}}'(v) = \mathcal{A}$.
- Si $v \notin \mathrm{dom}(\Gamma)$ y $\mathcal{A} \in \mathbb{A}$, luego:
  $$\mu' = \mu\{\text{array } v \mapsto \mathcal{A}\}$$
  es una valuación en $\mathrm{M}_{\Gamma\{v \mapsto \textsc{Array}\}}$ donde los únicos cambios son que:
  $${\mu_{\text{array}}}'(v) = {\mu_{\text{array@pre}}}'(v) = \mathcal{A}$$

Otras veces es necesario que una valuación olvide los valores que asigna a algunas variables.

Definición 2.12 (Olvido en una clasificación)**.**
Dada una clasificación Γ y un conjunto de variables $V \subseteq \mathrm{dom}(\Gamma)$ se define el *olvido en una clasificación*, notado $\Gamma \ominus V$ que resulta de olvidar la clasificación para las variables en V.

$$\Gamma \ominus V(v) = \begin{cases} \Gamma(v) & \text{si } v \notin V \\ \text{indefinido} & \text{sino} \end{cases}$$

Definición 2.13 (Olvido en una valuación)**.**
Dada una valuación $\mu \in \mathrm{M}_\Gamma$ y un conjunto de variables $V \subseteq \mathrm{dom}(\Gamma)$ se define el *olvido en una valuación*, notado $\mu \ominus V \in \mathrm{M}_{\Gamma \ominus V}$ que resulta de olvidar los valores para las variables en V.

$$\mu \ominus V(v) = \begin{cases} \mu(v) & \text{si } v \notin V \\ \text{indefinido} & \text{sino} \end{cases}$$

2.2.2. Semántica para expresiones

Una valuación asigna valores a cada una de las variables. En la definición siguiente se extiende tal concepto para poder asignar valor a expresiones enteras y booleanas.

Observar que algunas expresiones pueden tener su valor indefinido bajo una cierta valuación μ, por ejemplo:

- $\underline{4}/(\underline{1}-\underline{1})$
- $\underline{4}/i$ si $\mu_{\text{int}}(i) = 0$
- $a[\underline{-20}]$

En esos casos la valuación μ no puede asignarles ningún valor.

Definición 2.14 (Semántica para expresiones enteras)**.**
Dadas una valuación $\mu \in \mathrm{M}_\Gamma$ y una expresión entera $i \in \mathrm{IntExp}_\Gamma$, se usa $[\![i]\!]^{\text{Int}}_\mu = n$ para denotar que la semántica de i bajo la valuación μ está bien definida y su valor es $n \in \mathbb{Z}$.

$$\frac{}{[\![v]\!]^{\text{Int}}_\mu = \mu_{\text{int}}(v)}\ \text{S-IE-VAR} \qquad \frac{}{[\![v\mathbf{@pre}]\!]^{\text{Int}}_\mu = \mu_{\text{int@pre}}(v)}\ \text{S-IE-VAR@PRE}$$

$$\frac{[\![A]\!]^{\text{Array}}_\mu = \mathcal{A} \quad [\![i]\!]^{\text{Int}}_\mu = n \quad 0 \le n < \|\mathcal{A}\|}{[\![A[i]]\!]^{\text{Int}}_\mu = \mathcal{A}[n]}\ \text{S-IE-GET}$$

$$\frac{[\![A]\!]^{\text{Array}}_\mu = \mathcal{A}}{[\![|A|]\!]^{\text{Int}}_\mu = \|\mathcal{A}\|}\ \text{S-IE-SIZE} \qquad \frac{}{[\![\underline{n}]\!]^{\text{Int}}_\mu = n}\ \text{S-IE-NUM}$$

$$\frac{[\![i_1]\!]^{\text{Int}}_\mu = n \quad [\![i_2]\!]^{\text{Int}}_\mu = m}{[\![i_1 + i_2]\!]^{\text{Int}}_\mu = n + m}\ \text{S-IE-SUM} \qquad \frac{[\![i_1]\!]^{\text{Int}}_\mu = n \quad [\![i_2]\!]^{\text{Int}}_\mu = m}{[\![i_1 - i_2]\!]^{\text{Int}}_\mu = n - m}\ \text{S-IE-SUBS}$$

$$\frac{[\![i_1]\!]^{\text{Int}}_\mu = n \quad [\![i_2]\!]^{\text{Int}}_\mu = m}{[\![i_1 * i_2]\!]^{\text{Int}}_\mu = n * m}\ \text{S-IE-MUL} \qquad \frac{[\![i_1]\!]^{\text{Int}}_\mu = n \quad [\![i_2]\!]^{\text{Int}}_\mu = m \quad m \neq 0}{[\![i_1/i_2]\!]^{\text{Int}}_\mu = n/m}\ \text{S-IE-DIV}$$

Observar que no es necesario requerir que los tipos de las variables sean correctos. Esto es garantizado por la forma en la que las expresiones se construyen a partir de una clasificación Γ.

Definición 2.15 (Semántica para expresiones arreglo)**.**
Dadas una valuación $\mu \in \mathrm{M}_\Gamma$ y una expresión arreglo $A \in \mathrm{ArrayExp}_\Gamma$, se usa $[\![A]\!]^{\text{Array}}_\mu = \mathcal{A}$ para denotar que la semántica de A bajo la valuación μ está bien definida y su valor es $\mathcal{A} \in \mathbb{A}$.

$$\frac{}{[\![A]\!]^{\text{Array}}_\mu = \mu_{\text{array}}(A)}\ \text{S-AE-VAR} \qquad \frac{}{[\![A\mathbf{@pre}]\!]^{\text{Array}}_\mu = \mu_{\text{array@pre}}(A)}\ \text{S-AE-VAR@PRE}$$

$$\frac{[\![A]\!]^{\text{Array}}_\mu = \mathcal{A} \quad [\![i_1]\!]^{\text{Int}}_\mu = n \quad [\![i_2]\!]^{\text{Int}}_\mu = m \quad 0 \le n < \|\mathcal{A}\|}{[\![\mathbf{update}\ A\ \mathbf{on}\ i_1\ \mathbf{with}\ i_2]\!]^{\text{Array}}_\mu = \mathcal{A}\{[n] \mapsto m\}}\ \text{S-AE-UPDATE}$$

$$\frac{[\![i_1]\!]^{\text{Int}}_\mu = n \quad [\![i_2]\!]^{\text{Int}}_\mu = m \quad n \ge 0}{[\![\mathbf{array}[i_1]\ \mathbf{of}\ i_2]\!]^{\text{Array}}_\mu = \underbrace{[m, \ldots, m]}_{n \text{ veces}}}\ \text{S-AE-CONST}$$

Definición 2.16 (Semántica para expresiones booleanas).
Dadas una valuación $\mu \in \mathrm{M}_\Gamma$ y una expresión booleana $b \in \mathrm{BoolExp}_\Gamma$, se usa $[\![b]\!]^{\textsc{Bool}}_\mu = x$ para denotar que la semántica de b bajo la valuación μ está bien definida y su valor es $x \in \{\mathrm{true}, \mathrm{false}\}$.

$$\frac{[\![i_1]\!]^{\textsc{Int}}_\mu = n \quad [\![i_2]\!]^{\textsc{Int}}_\mu = n}{[\![i_1 = i_2]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}\ \text{S-BE-IEQ-T} \qquad \frac{[\![i_1]\!]^{\textsc{Int}}_\mu = n \quad [\![i_2]\!]^{\textsc{Int}}_\mu = m \quad n \neq m}{[\![i_1 = i_2]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}\ \text{S-BE-IEQ-F}$$

$$\frac{[\![A_1]\!]^{\textsc{Array}}_\mu = \mathcal{A} \quad [\![A_2]\!]^{\textsc{Array}}_\mu = \mathcal{A}}{[\![A_1 = A_2]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}\ \text{S-BE-AEQ-T} \qquad \frac{[\![A_1]\!]^{\textsc{Array}}_\mu = \mathcal{A} \quad [\![A_2]\!]^{\textsc{Array}}_\mu = \mathcal{B} \quad \mathcal{A} \neq \mathcal{B}}{[\![A_1 = A_2]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}\ \text{S-BE-AEQ-F}$$

$$\frac{[\![i_1]\!]^{\textsc{Int}}_\mu = n \quad [\![i_2]\!]^{\textsc{Int}}_\mu = m \quad n < m}{[\![i_1 < i_2]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}\ \text{S-BE-ILT-T} \qquad \frac{[\![i_1]\!]^{\textsc{Int}}_\mu = n \quad [\![i_2]\!]^{\textsc{Int}}_\mu = m \quad n \geq m}{[\![i_1 < i_2]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}\ \text{S-BE-ILT-F}$$

$$\frac{[\![b_1]\!]^{\textsc{Bool}}_\mu = \mathrm{true} \quad [\![b_2]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}{[\![b_1 \wedge b_2]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}\ \text{S-BE-AND-T}$$

$$\frac{[\![b_1]\!]^{\textsc{Bool}}_\mu = \mathrm{true} \quad [\![b_2]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}{[\![b_1 \wedge b_2]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}\ \text{S-BE-AND-F1} \qquad \frac{[\![b_1]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}{[\![b_1 \wedge b_2]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}\ \text{S-BE-AND-F2}$$

$$\frac{[\![b]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}{[\![\neg b]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}\ \text{S-BE-NOT-T} \qquad \frac{[\![b]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}{[\![\neg b]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}\ \text{S-BE-NOT-F}$$

$$\frac{\text{todo } n \in \mathbb{Z} \text{ cumple que } [\![b]\!]^{\textsc{Bool}}_{\mu\{\text{int } v \mapsto n\}} = \mathrm{true}}{[\![\forall\ v : \textsc{Int}\ (b)]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}\ \text{S-BE-IFORALL-T}$$

$$\frac{\text{algún } n \in \mathbb{Z} \text{ cumple que } [\![b]\!]^{\textsc{Bool}}_{\mu\{\text{int } v \mapsto n\}} = \mathrm{false}}{[\![\forall\ v : \textsc{Int}\ (b)]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}\ \text{S-BE-IFORALL-F}$$

$$\frac{\text{todo } \mathcal{A} \in \mathbb{A} \text{ cumple que } [\![b]\!]^{\textsc{Bool}}_{\mu\{\text{array } v \mapsto \mathcal{A}\}} = \mathrm{true}}{[\![\forall\ v : \textsc{Array}\ (b)]\!]^{\textsc{Bool}}_\mu = \mathrm{true}}\ \text{S-BE-AFORALL-T}$$

$$\frac{\text{algún } \mathcal{A} \in \mathbb{A} \text{ cumple que } [\![b]\!]^{\textsc{Bool}}_{\mu\{\text{array } v \mapsto \mathcal{A}\}} = \mathrm{false}}{[\![\forall\ v : \textsc{Array}\ (b)]\!]^{\textsc{Bool}}_\mu = \mathrm{false}}\ \text{S-BE-AFORALL-F}$$

Observación 2.2. Observar que dada una valuación μ y una expresión entera i, $[\![i]\!]^{\textsc{Int}}_\mu$ es computable.

Del mismo modo, siendo A una expresión arreglo, $[\![A]\!]^{\textsc{Array}}_\mu$ también lo es.

Sin embargo, si se toma b una expresión booleana, $[\![b]\!]^{\textsc{Bool}}_\mu$ no es computable debido a la combinación de cuantificación con expresiones aritméticas. Si b es libre de cuantificadores, su semántica es computable.

2.2.3. Semántica para sentencias

En este apartado se introduce el concepto de ejecución de una sentencia. El mismo se establece como una función que transforma una valuación de origen según de qué sentencia se trata.

Definición 2.17 (Semántica de sentencias)**.**
Se usa $\mu \triangleright s \triangleright \mu'$ para denotar la *semántica de una sentencia válida* $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ partiendo de la valuación $\mu \in \text{M}_\Gamma$, finalizando normalmente su ejecución y obteniendo la valuación $\mu' \in \text{M}_{\Gamma'}$. Su definición es dada por estas reglas:

- Asignación a una variable entera:

$$\frac{[\![i]\!]_\mu^{\text{INT}} = n}{\mu \triangleright v \leftarrow i \triangleright \mu\{v \mapsto n\}} \text{ S-SENT-IASSIGN}$$

  Suponiendo que se le puede dar semántica a la expresión entera i, luego se puede extender la valuación para reflejar la asignación.

- Asignación a una variable arreglo:

$$\frac{[\![A]\!]_\mu^{\text{ARRAY}} = \mathcal{A}}{\mu \triangleright v \leftarrow A \triangleright \mu\{v \mapsto \mathcal{A}\}} \text{ S-SENT-AASSIGN}$$

  De forma análoga, si se le puede dar semántica a la expresión arreglo A, luego se puede extender la valuación para reflejar la asignación.

- Creación de una variable entera:

$$\frac{[\![i]\!]_\mu^{\text{INT}} = n}{\mu \triangleright \textbf{int } v \leftarrow i \triangleright \mu\{\text{int } v \mapsto n\}} \text{ S-SENT-IDEF}$$

  Muy similar a la asignación a una variable entera, pero en este caso la extensión a la valuación es sobre una variable nueva.

- Creación de una variable arreglo:

$$\frac{[\![A]\!]_\mu^{\text{ARRAY}} = \mathcal{A}}{\mu \triangleright \textbf{array } v \leftarrow A \triangleright \mu\{\text{array } v \mapsto \mathcal{A}\}} \text{ S-SENT-ADEF}$$

  Análogamente, si se puede dar semántica a la expresión arreglo A, luego se puede extender la valuación creando una nueva variable de tipo arreglo.

- Sentencia vacía:

$$\frac{}{\mu \triangleright \textbf{skip} \triangleright \mu} \text{ S-SENT-SKIP}$$

  La sentencia vacía siempre se ejecuta correctamente y sin afectar la valuación.

- Secuenciamiento de sentencias:

$$\frac{\mu \rhd s_1 \rhd \mu' \quad \mu' \rhd s_2 \rhd \mu''}{\mu \rhd s_1\ s_2 \rhd \mu''}\ \text{S-SENT-SEQ}$$

Si la primer sentencia se puede ejecutar correctamente, y la segunda se puede ejecutar correctamente desde el punto donde la primera terminó, luego se pueden secuenciar.

- Condicional:

$$\frac{[\![g]\!]^{\text{Bool}}_{\mu} = \text{true} \quad \mu \rhd s_1 \rhd \mu'}{\mu \rhd \textbf{if } g \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \rhd \mu' \ominus \text{localVars}(s_1)}\ \text{S-SENT-IF-T}$$

Suponiendo que la guarda tiene semántica y la misma es verdadera, y suponiendo que la rama "then" del condicional s_1 puede ejecutarse sin problemas, luego se ejecuta tal rama y se olvidan las variables creadas en ella.

$$\frac{[\![g]\!]^{\text{Bool}}_{\mu} = \text{false} \quad \mu \rhd s_2 \rhd \mu'}{\mu \rhd \textbf{if } g \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \rhd \mu' \ominus \text{localVars}(s_2)}\ \text{S-SENT-IF-F}$$

Equivalente a la anterior, pero con la rama "else" cuando la guarda evalúa a false.

- Ciclo:

$$\frac{[\![g]\!]^{\text{Bool}}_{\mu} = \text{false} \quad [\![inv]\!]^{\text{Bool}}_{\mu} = \text{true}}{\mu \rhd \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \rhd \mu}\ \text{S-SENT-WHILE-F}$$

En un ciclo donde la guarda es falsa, se termina la ejecución sin modificar la valuación, siempre y cuando se pueda satisfacer el invariante.

$$\frac{\begin{array}{c}[\![g]\!]^{\text{Bool}}_{\mu} = \text{true} \quad [\![inv]\!]^{\text{Bool}}_{\mu} = \text{true} \quad [\![var]\!]^{\text{Int}}_{\mu} > 0 \\ \mu \rhd s \rhd \mu' \\ [\![inv]\!]^{\text{Bool}}_{\mu' \ominus \text{localVars}(s)} = \text{true} \quad [\![var]\!]^{\text{Int}}_{\mu' \ominus \text{localVars}(s)} < [\![var]\!]^{\text{Int}}_{\mu} \\ \mu' \ominus \text{localVars}(s) \rhd \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \rhd \mu''\end{array}}{\mu \rhd \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \rhd \mu''}\ \text{S-SENT-WHILE-T}$$

Se parte de una valuación μ tal que debe poder hacer verdadera la guarda. Además debe poder satisfacer el invariante y hacer positivo al variante.

Desde tal valuación se debe poder ejecutar el cuerpo del ciclo para llegar a otra valuación μ'.

Olvidando las variables creadas durante el bloque se debe poder volver a satisfacer el invariante y hacer decrecer el variante.

Si finalmente desde esa valuación μ', olvidando las variables locales de s, se puede ejecutar el resto del ciclo para terminar en una valuación μ'' luego la ejecución completa del ciclo culmina en esta última.

- Llamado a un procedimiento:

$$\frac{[\![\text{P}_{proc}]\!]^{\text{Bool}}_{\rho} = \text{true} \quad \rho \rhd \text{body}(proc) \rhd \rho' \quad [\![\text{Q}_{proc}]\!]^{\text{Bool}}_{\rho'} = \text{true}}{\mu \rhd \textbf{call } proc(cp_1, \ldots, cp_k) \rhd \mu'}\ \text{S-SENT-CALL}$$

Para realizar un llamado, es necesario primero construir una valuación auxiliar ρ con el *binding* de los parámetros concretos $cp_1, \ldots, cp_k$ con los formales $\text{pars}(proc) = p_1, \ldots, p_k$. La misma se obtiene usando:

$$\rho(p_i) \stackrel{\text{def}}{=} \mu(cp_i)$$
$$\rho(p_i\mathbf{@pre}) \stackrel{\text{def}}{=} \mu(cp_i)$$

Suponer que la valuación auxiliar ρ logra satisfacer la precondición del procedimiento llamado, y el cuerpo del mismo ejecuta correctamente y termina en una valuación ρ' tal que satisface la postcondición.

Los parámetros concretos son reasignados según el valor en ρ', por lo tanto μ' es igual a μ a excepción de los valores para los parámetros concretos, los cuales se obtienen de ρ' de la siguiente manera:

$$\mu'(cp_i) \stackrel{\text{def}}{=} \rho'(p_i)$$

2.2.4. Semántica para programas

Se introduce el concepto de ejecución de un programa. Puede verse como un llamado a un procedimiento inicial cualquiera.

Definición 2.18 (Semántica para programas)**.**
Se usa $\mu \triangleright \pi \ : \ proc \triangleright \mu'$ para denotar la *semántica de un programa* π iniciando en un procedimiento $proc$, partiendo de la valuación $\mu \in \text{M}_{\Gamma_{proc}}$, finalizando su ejecución y obteniendo la valuación $\mu' \in \text{M}_{\Gamma_{proc}}$. Su definición es dada por la siguiente regla:

$$\frac{proc \in \pi \quad [\![\text{P}_{proc}]\!]^{\text{Bool}}_{\mu} = \text{true} \quad \mu \ \triangleright \ \text{body}(proc) \ \triangleright \ \mu' \quad [\![\text{Q}_{proc}]\!]^{\text{Bool}}_{\mu'} = \text{true}}{\mu \ \triangleright \pi \ : \ proc \ \triangleright \ \mu' \ominus \text{localVars}(\text{body}(proc))} \ \text{SEM-PROG}$$

Observar que se toma como valuación final μ', sin embargo se olvida el valor de las variables locales al cuerpo del procedimiento.

2.3. Ejemplos

Se muestran a continuación algunos ejemplos de pequeños programas escritos en el lenguaje de programación presentado.

El primer ejemplo calcula el cociente y resto de dos naturales a y b.

```
 1: divide(a, b, q, r)
 2: :? a ≥ 0 ∧ b > 0
 3: :! a = q * b + r ∧ 0 ≤ r < b ∧ a = a@pre ∧ b = b@pre
 4: {
 5:     q ← 0
 6:     r ← a
 7:     while r ≥ b
 8:         :?! 0 ≤ r ≤ a ∧ a = q * b + r ∧ a = a@pre ∧ b = b@pre
 9:         :# r + 1
10:     do
11:         r ← r − b
12:         q ← q + 1
13:     od
14: }
```

Cociente y resto

El segundo ejemplo calcula el máximo elemento de un arreglo A.

```
 1: max(A, m)
 2: :? |A| > 0
 3: :! ∀k / 0 ≤ k < |A| : m ≥ A[k] ∧
 4:     ∃k / 0 ≤ k < |A| : m = A[k] ∧ A = A@pre
 5: {
 6:     m ← A[0]
 7:     int i ← 1
 8:     while i < |A|
 9:         :?! 1 ≤ i ≤ |A| ∧
10:             ∀k / 0 ≤ k < i : m ≥ A[k] ∧
11:             ∃k / 0 ≤ k < i : m = A[k] ∧ A = A@pre
12:         :# |A| − i
13:     do
14:         if A[i] > m then
15:             m ← A[i]
16:         fi
17:         i ← i + 1
18:     od
19: }
```

El máximo de un arreglo

2.4. Conclusiones

Se ha presentado un lenguaje de programación imperativo multiprocedural. El mismo cuenta con soporte nativo para anotaciones de precondiciones, postcondiciones, invariantes y funciones variantes.

Dada una sentencia cualquiera $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ y una valuación inicial $\mu \in \text{M}_\Gamma$ se puede dar una de las siguientes dos situaciones:

1. La semántica de la sentencia s se encuentra bien definida a partir de la valuación μ, o sea:

$$\exists\, \mu' \in \mathrm{M}_{\Gamma'} :\ \mu \triangleright s \triangleright \mu'$$

   Por como están definidas las reglas para semántica esto significa que la sentencia s se ejecuta de forma correcta y finaliza. O sea, realiza accesos a arreglos siempre dentro del rango, nunca divide por 0, satisface todas las precondiciones de los métodos que llama, no viola ningún invariante, etc.

2. La semántica no se encuentra definida a partir de la valuación μ, o sea:

$$\neg\big(\exists\, \mu' \in \mathrm{M}_{\Gamma'} :\ \mu \triangleright s \triangleright \mu'\big)$$

   Esto significa que en algún punto de su ejecución, la sentencia s "intentó" realizar un acceso fuera de rango en un arreglo, o violó un invariante o una postcondición, o dividió por 0.

Observar que una sentencia no puede ejecutarse indefinidamente, ya que no se cuenta con recursión y las reglas para el ciclo fuerzan que el variante decrezca y sea positivo en cada iteración. Si se quisiera modelar programas que no terminan habría que incorporar algún tipo de construcción adicional. Se trata de una extensión sencilla pero escapa al interés del trabajo pensado para esta tesis.

A partir de las reglas para la semántica operacional, podría construirse un intérprete que siga paso a paso la ejecución de un programa. Sin embargo, tanto las precondiciones como las postcondiciones e invariantes son expresiones booleanas que pueden contener cuantificación no acotada sobre un universo infinito. Esto impide que un intérprete pueda realizar verificaciones en tiempo de ejecución.

En el capítulo siguiente se presenta un formalismo que permitirá, bajo ciertas condiciones, asegurar que un programa ejecutará correctamente. Esto permitiría poder realizar ejecuciones del mismo sin necesidad de verificar accesos a arreglos, ni divisiones por cero, ni verificaciones de precondiciones, invariantes y postcondiciones. Al garantizar que el programa siempre es correcto, se podrían "apagar" las verificaciones en el intérprete, de esta forma se evita el problema de tener que revisar anotaciones cuyo valor de verdad es indecidible.

Capítulo 3

Anotaciones como un sistema de tipos

"*Logic is the art of going wrong with confidence.*"

Joseph Wood Krutch

En el capítulo anterior se presentó un lenguaje de programación imperativo con soporte para procedimientos cuya sintáxis y semántica operacional se encuentran definidas formalmente. Tal lenguaje cuenta con soporte nativo para anotaciones que indican precondiciones, postcondiciones, e invariantes y variantes en el caso de ciclos.

La escritura de este tipo de anotaciones supone una carga de trabajo extra para quien se encuentra escribiendo programas. Sin embargo, en este capítulo se presentarán una serie de técnicas que aprovechan dichas anotaciones para corroborar que un programa se ejecute de forma adecuada con respecto a su especificación, la cual no necesariamente es exhaustiva. Al garantizar que un programa no viola sus anotaciones se logra el propósito buscado que es verificar un programa respecto de su comportamiento esperado.

En la Sección 3.1 se define el concepto de expresiones seguras, que permite caracterizar las condiciones que debe satisfacer una valuación para poder asignar valor a una expresión determinada.

La Sección 3.2 presenta una semántica al estilo Floyd-Hoare [Flo67, Hoa69] para el lenguaje de programación propuesto; la misma se basa en transformaciones de expresiones booleanas que caracterizan estados genéricos. Por ejemplo:

$$\{a = b \ \wedge \ b = \underline{10}\}$$

$$b \leftarrow b + \underline{1}$$

$$\{\exists\, b' : \textsc{Int}\ (a = b' \ \wedge \ b' = \underline{10} \ \wedge \ b = b' + \underline{1})\}$$

En la Sección 3.3 se utiliza esta nueva semántica para caracterizar los programas cuyas ejecuciones están bien definidas. Se demuestra que esta caracterización coincide con la semántica operacional presentada en el capítulo anterior.

En la Sección 3.4 se definen dos cálculos que permiten acercarse hacia una implementación de una herramienta verificadora. Uno de ellos permite obtener una postcondición de una sentencia; el otro permite obtener una precondición.

Sin embargo la utilización de estas técnicas de verificación automatizada requiere la escritura de muchas anotaciones engorrosas y potencialmente derivables a partir del código. En la Sección 3.5 atacamos este problema proponiendo una técnica basada en los métodos presentados en la sección anterior, que permite reforzar las anotaciones y de esta manera aliviarle la tarea al programador.

3.1. Expresiones seguras

Tal como se mencionó anteriormente, no todas las expresiones tienen semántica bajo cualquier valuación. Sin embargo, dada una expresión, pueden establecerse las condiciones que una valuación debe cumplir para poder darle semántica.

El objetivo de este apartado es, a partir de una expresión e de cualquier tipo, obtener una expresión booleana cuya validez garantice que la semántica de e está bien definida.

Se llama *condición para expresión segura* a esta expresión booleana y a continuación se define para los distintos tipos de expresiones introducidos en el capítulo anterior.

Definición 3.1 (Condiciones para expresiones enteras seguras)**.**
Se define $\text{safe}^{\text{I}} : \text{IntExp}_{\Gamma} \to \text{BoolExp}_{\Gamma}$, la *condición para expresión entera segura*.

$$\begin{aligned}
\text{safe}^{\text{I}}(v) &= \textbf{true} \\
\text{safe}^{\text{I}}(v\textbf{@pre}) &= \textbf{true} \\
\text{safe}^{\text{I}}(\underline{n}) &= \textbf{true} \\
\text{safe}^{\text{I}}(|A|) &= \text{safe}^{\text{A}}(A) \\
\text{safe}^{\text{I}}(A[i]) &= \text{safe}^{\text{A}}(A) \wedge \text{safe}^{\text{I}}(i) \wedge 0 \leq i < |A| \\
\text{safe}^{\text{I}}(i_1 + i_2) &= \text{safe}^{\text{I}}(i_1) \wedge \text{safe}^{\text{I}}(i_2) \\
\text{safe}^{\text{I}}(i_1 - i_2) &= \text{safe}^{\text{I}}(i_1) \wedge \text{safe}^{\text{I}}(i_2) \\
\text{safe}^{\text{I}}(i_1 * i_2) &= \text{safe}^{\text{I}}(i_1) \wedge \text{safe}^{\text{I}}(i_2) \\
\text{safe}^{\text{I}}(i_1 / i_2) &= \text{safe}^{\text{I}}(i_1) \wedge \text{safe}^{\text{I}}(i_2) \wedge i_2 \neq 0
\end{aligned}$$

Definición 3.2 (Condiciones para expresiones arreglo seguras)**.**
Se define $\text{safe}^{\text{A}} : \text{ArrayExp}_{\Gamma} \to \text{BoolExp}_{\Gamma}$, la *condición para expresión arreglo segura*.

$$\begin{aligned}
\text{safe}^{\text{A}}(A) &= \textbf{true} \\
\text{safe}^{\text{A}}(A\textbf{@pre}) &= \textbf{true} \\
\text{safe}^{\text{A}}(\textbf{update } A \textbf{ on } i_1 \textbf{ with } i_2) &= \text{safe}^{\text{A}}(A) \wedge \text{safe}^{\text{I}}(i_1) \wedge \text{safe}^{\text{I}}(i_2) \wedge 0 \leq i_1 < |A| \\
\text{safe}^{\text{A}}(\textbf{array}[i_1] \textbf{ of } i_2) &= \text{safe}^{\text{I}}(i_1) \wedge \text{safe}^{\text{I}}(i_2) \wedge i_1 \geq 0
\end{aligned}$$

Definición 3.3 (Condiciones para expresiones booleanas seguras)**.**
Se define la $\text{safe}^{\text{B}} : \text{BoolExp}_{\Gamma} \to \text{BoolExp}_{\Gamma}$, la *condición para expresión booleana segura*.

$$\begin{aligned}
\text{safe}^{\text{B}}(i_1 = i_2) &= \text{safe}^{\text{I}}(i_1) \wedge \text{safe}^{\text{I}}(i_2) \\
\text{safe}^{\text{B}}(i_1 < i_2) &= \text{safe}^{\text{I}}(i_1) \wedge \text{safe}^{\text{I}}(i_2) \\
\text{safe}^{\text{B}}(A_1 = A_2) &= \text{safe}^{\text{A}}(A_1) \wedge \text{safe}^{\text{A}}(A_2) \\
\text{safe}^{\text{B}}(b_1 \wedge b_2) &= \text{safe}^{\text{B}}(b_1) \wedge \text{safe}^{\text{B}}(b_2) \\
\text{safe}^{\text{B}}(\neg b) &= \text{safe}^{\text{B}}(b) \\
\text{safe}^{\text{B}}(\forall\ v : \textsc{Int}\ (b)) &= \forall\ v : \textsc{Int}\ (\text{safe}^{\text{B}}(b)) \\
\text{safe}^{\text{B}}(\forall\ v : \textsc{Array}\ (b)) &= \forall\ v : \textsc{Array}\ (\text{safe}^{\text{B}}(b))
\end{aligned}$$

Por ejemplo:

$$\text{safe}^{\text{I}}(A[x] + |\textbf{update } A \textbf{ on } \underline{8} \textbf{ with } \underline{-15}|) \;\equiv\; 0 \leq x < |A| \wedge 0 \leq 8 < |A|$$

Se utiliza el símbolo $\equiv$ ya que en realidad se ha simplificado la expresión eliminando los **true** de las conjunciones.

Teorema 3.1 (Las condiciones para expresiones seguras son correctas)**.**
Sean $i \in \text{IntExp}_\Gamma, A \in \text{ArrayExp}_\Gamma, b \in \text{BoolExp}_\Gamma$. *Sea* $\mu \in \text{M}_\Gamma$*, luego:*

$$\begin{aligned}
&\textit{si } [\![\text{safe}^{\text{I}}(i)]\!]_\mu^{\text{Bool}} = \text{true } \textit{luego existe algún } n \in \mathbb{Z} : [\![i]\!]_\mu^{\text{Int}} = n \\
&\textit{si } [\![\text{safe}^{\text{A}}(A)]\!]_\mu^{\text{Bool}} = \text{true } \textit{luego existe algún } \mathcal{A} \in \mathbb{A} : [\![A]\!]_\mu^{\text{Array}} = \mathcal{A} \\
&\textit{si } [\![\text{safe}^{\text{B}}(b)]\!]_\mu^{\text{Bool}} = \text{true } \textit{luego existe algún } x \in \{\text{true}, \text{false}\} : [\![b]\!]_\mu^{\text{Bool}} = x
\end{aligned}$$

Dem:

Se demuestra por inducción estructural. Los detalles concretos pueden verse en la Sección A.1.

□

3.2. Semántica abstracta para sentencias

Antes de definir la semántica basada en transformación de estados genéricos es necesario introducir dos nociones que serán usadas en el resto de este trabajo.

La primera de ellas es la noción de fuerza entre expresiones booleanas.

Definición 3.4 (Relacion de fuerza entre expresiones booleanas)**.**
Dadas dos expresiones booleanas $b_1, b_2 \in \text{BoolExp}_\Gamma$, se dice que b_1 *fuerza* a b_2 si toda valuación $\mu \in \text{M}_\Gamma$ tal que $[\![b_1]\!]_\mu^{\text{Bool}} = \text{true}$ también cumple que $[\![b_2]\!]_\mu^{\text{Bool}} = \text{true}$.
Se nota $b_1 \models b_2$.

Por ejemplo:

$$x \geq 10 \;\wedge\; z < 15 \;\models\; x \geq 8$$

Observación 3.1. Observar que dadas dos expresiones booleanas $b_1, b_2 \in \text{BoolExp}_\Gamma$, determinar si $b_1 \models b_2$ es, en general, no decidible. Esto se debe a que una expresión booleana puede codificar la aritmética de Peano, para la cual no hay un método de decisión.

La segunda de estas nociones es la sustitución en expresiones.

Definición 3.5 (Sustitución en expresiones)**.**
Sea $i \in \text{IntExp}_\Gamma$ y sean e, e' dos expresiones del mismo tipo. Se define la sustitución en i de todas las apariciones de e por e', notada $i\lfloor e \mapsto e' \rfloor$.

$$i\lfloor e \mapsto e' \rfloor \stackrel{\text{def}}{=} e' \quad \text{si } i = e$$

Si $i \neq e$ se define por reglas a continuación:

$$\begin{aligned}
v\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} v \\
v\mathbf{@pre}\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} v\mathbf{@pre} \\
(i_1[i_2])\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} (i_1\lfloor e \mapsto e' \rfloor)[i_2\lfloor e \mapsto e' \rfloor] \\
\underline{n}\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} \underline{n} \\
(|i|)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} |i\lfloor e \mapsto e' \rfloor| \\
(i_1 + i_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} i_1\lfloor e \mapsto e' \rfloor + i_2\lfloor e \mapsto e' \rfloor \\
(i_1 - i_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} i_1\lfloor e \mapsto e' \rfloor - i_2\lfloor e \mapsto e' \rfloor \\
(i_1 * i_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} i_1\lfloor e \mapsto e' \rfloor * i_2\lfloor e \mapsto e' \rfloor \\
(i_1/i_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} i_1\lfloor e \mapsto e' \rfloor / i_2\lfloor e \mapsto e' \rfloor
\end{aligned}$$

De forma similar, si $A \in \text{ArrayExp}_\Gamma$, luego $A\lfloor e \mapsto e' \rfloor$ es la sustitución de e por e' en A.

$$A\lfloor e \mapsto e' \rfloor \stackrel{\text{def}}{=} e' \quad \text{si } A = e$$

Si $A \neq e$ se define por reglas a continuación:

$$\begin{aligned}
v\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} v \\
v\mathbf{@pre}\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} v\mathbf{@pre} \\
(\mathbf{update}\ A\ \mathbf{on}\ i_1\ \mathbf{with}\ i_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} \mathbf{update}\ A\lfloor e \mapsto e' \rfloor\ \mathbf{on}\ i_1\lfloor e \mapsto e' \rfloor\ \mathbf{with}\ i_2\lfloor e \mapsto e' \rfloor \\
(\mathbf{array}[i_1]\ \mathbf{of}\ i_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} \mathbf{array}[i_1\lfloor e \mapsto e' \rfloor]\ \mathbf{of}\ i_2\lfloor e \mapsto e' \rfloor
\end{aligned}$$

O si $b \in \text{BoolExp}_\Gamma$, luego $b\lfloor e \mapsto e' \rfloor$ es la sustitución de e por e' en b.

$$b\lfloor e \mapsto e' \rfloor \stackrel{\text{def}}{=} e' \quad \text{si } b = e$$

Si $b \neq e$ se define por reglas a continuación:

$$\begin{aligned}
(i_1 = i_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} i_1\lfloor e \mapsto e' \rfloor = i_2\lfloor e \mapsto e' \rfloor \\
(A_1 = A_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} A_1\lfloor e \mapsto e' \rfloor = A_2\lfloor e \mapsto e' \rfloor \\
(i_1 < i_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} i_1\lfloor e \mapsto e' \rfloor < i_2\lfloor e \mapsto e' \rfloor \\
(b_1 \wedge b_2)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} b_1\lfloor e \mapsto e' \rfloor \wedge b_2\lfloor e \mapsto e' \rfloor \\
(\neg b)\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} \neg(b\lfloor e \mapsto e' \rfloor) \\
(\forall\ v : \textsc{Int}\ (b))\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} \forall\ v : \textsc{Int}\ (b\lfloor e \mapsto e' \rfloor) \quad \text{si } v \notin e' \\
(\forall\ v : \textsc{Array}\ (b))\lfloor e \mapsto e' \rfloor &\stackrel{\text{def}}{=} \forall\ v : \textsc{Array}\ (b\lfloor e \mapsto e' \rfloor) \quad \text{si } v \notin e'
\end{aligned}$$

En el caso de las cuantificaciones la hipótesis de que $v \notin e'$ no representa un obstáculo ya que en caso de que suceda puede reemplazarse la cuantificación sobre v por otra sobre una variable fresca v'.

Se define a continuación la semántica abstracta para una sentencia. Esta noción trabaja con valuaciones simbólicas representadas mediante expresiones booleanas.

La semántica operacional definida en el capítulo anterior permite establecer la transformación que realiza una sentencia sobre una valuación concreta, o sea una ejecución. En el caso de la semántica abstracta se presenta una transformación entre la expresión booleana que caracteriza una valuación simbólica antes y después de la ejecución de una sentencia; se trata de caracterizar un conjunto potencialmente infinito de ejecuciones.

Definición 3.6 (Semántica abstracta para sentencias)**.**
Dadas $p \in \text{BoolExp}_{\Gamma}$, $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ y $q \in \text{BoolExp}_{\Gamma'}$ se usa $\{p\}\ s\ \{q\}$ para notar la semántica abstracta de una sentencia s partiendo de una expresión booleana p y llegando a otra q.

- Asignación a una variable entera:

$$\frac{p \models \text{safe}^{\text{I}}(i) \qquad \exists\, v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = i\lfloor v \mapsto v' \rfloor) \models q}{\{p\}\ v \leftarrow i\ \{q\}}\ \text{A-SENT-IASSIGN}$$

  Se le puede dar semántica abstracta a una asignación desde p a q si sucede que:

  - p es suficiente para garantizar que la expresión entera i está bien definida.
  - Se utiliza v' para representar el valor que tenía la variable v previo a la asignación. Se reemplazan las apariciones de v por v' en la expresión p de forma tal de preservar todo lo que se conocía. Por otra parte, luego de la asignación se cumple que v tiene el valor de la expresión i, tomando el recaudo de reemplazar allí también la variable v por su valor anterior v'.

  Observar que esta regla admite como consecuencia una expresión q siempre y cuando esta sea igual o más débil que el efecto de la asignación sobre la expresión p.

- Asignación a una variable arreglo:

$$\frac{p \models \text{safe}^{\text{A}}(A) \qquad \exists\, v' : \textsc{Array}\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = A\lfloor v \mapsto v' \rfloor) \models q}{\{p\}\ v \leftarrow A\ \{q\}}\ \text{A-SENT-AASSIGN}$$

  Similar al caso anterior, pero para arreglos.

- Creación de una variable entera:

$$\frac{p \models \text{safe}^{\text{I}}(i) \qquad p\ \wedge\ v = i \models q}{\{p\}\ \textbf{int}\ v \leftarrow i\ \{q\}}\ \text{A-SENT-IDEF}$$

  El caso de la creación de una nueva variable es similar a la asignación. La única diferencia es que ya no es necesario introducir un existencial para recordar el viejo valor de la variable v, ya que ésta no existe previo a la definición.

- Creación de una variable arreglo:

$$\frac{p \models \text{safe}^{\text{A}}(A) \qquad p \ \wedge \ v = A \models q}{\{p\}\ \textbf{array}\ v \leftarrow A\ \{q\}}\ \text{A-SENT-ADEF}$$

  Similar al caso anterior, pero se crea una variable arreglo.

- Sentencia vacía:

$$\frac{p \models q}{\{p\}\ \textbf{skip}\ \{q\}}\ \text{A-SENT-SKIP}$$

  La sentencia vacía permite probar cualquier expresión q que sea igual o más débil que p.

- Secuenciamiento:

$$\frac{\{p\}\ s_1\ \{r\} \quad \{r\}\ s_2\ \{q\}}{\{p\}\ s_1\ s_2\ \{q\}}\ \text{A-SENT-SEQ}$$

  Se le puede dar semántica abstracta a dos sentencias secuenciadas si existe una expresión booleana r que representa el estado intermedio entre ambas.

- Condicional:

$$\frac{\begin{array}{c} p \models \text{safe}^{\text{B}}(g) \\ p \wedge g \models p_1 \qquad \{p_1\}\ s_1\ \{q_1\} \qquad q_1 \models q \\ p \wedge \neg g \models p_2 \qquad \{p_2\}\ s_2\ \{q_2\} \qquad q_2 \models q \end{array}}{\{p\}\ \textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ \{q\}}\ \text{A-SENT-IF}$$

  Se le puede dar semántica abstracta a un condicional entre p y q si:

  - p es suficiente para garantizar que g es una expresión booleana segura.
  - La semántica abstracta de la rama "then" está definida partiendo de alguna expresión p_1 que sea más débil que p sumado al hecho de que la guarda vale, y llegando a algún q_1 suficiente para garantizar q.
  - Lo mismo sucede con la rama "else", pero partiendo de alguna expresión p_2 que sea más débil que p sumado al hecho de que la guarda no vale.

- Ciclo:

$$\frac{\begin{array}{c} \textbf{true} \models \text{safe}^{\text{B}}(inv) \quad inv \models \text{safe}^{\text{B}}(g) \quad inv \models \text{safe}^{\text{I}}(var) \\ p \models inv \qquad inv \wedge g \models p' \qquad p' \models var > \underline{0} \\ \{p'\}\ var_0 \leftarrow var \ \ s\ \{q'\} \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q \end{array}}{\{p\}\ \textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \textbf{od}\ \{q\}}\ \text{A-SENT-WHILE}$$

  Se asume que las anotaciones del ciclo son seguras, lo cual se pide en la primera línea. En particular, observar que el invariante debe ser seguro en todo contexto. Esto permite independizar la condición de seguridad del ciclo del contexto en el cual el mismo aparece.

  Un ciclo tiene semántica abstracta partiendo de p y llegando a q si puede cumplirse con el teorema del invariante [Dij97], que en este caso equivale a que:

- El invariante vale al comienzo, o sea que p fuerza el invariante.
- El invariante y el hecho de que la guarda vale satisfacen una cierta expresión p' que representa el estado al comenzar una iteración genérica. Dicha expresión p' es tal que garantiza que el variante es positivo.
- Al cuerpo del ciclo se le incorpora una asignación para recordar el valor del variante inicial. Tal cuerpo aumentado tiene semántica abstracta entre p' y otra expresión booleana q'. Esto representa la transformación de estados de una ejecución genérica.
- Dicha expresión q' preserva el invariante y hace que el variante decrezca.
- Por último, el invariante y la negación de la guarda fuerzan q.

- Llamado a un procedimiento:

$$\frac{1 \leq i \leq k \quad p\lfloor cp_i \mapsto p_i \rfloor \models \mathrm{P}_{proc} \qquad \exists o_1, \ldots, o_k \ (p\lfloor cp_i \mapsto o_i \rfloor \ \wedge \ \mathrm{Q}_{proc}\lfloor p_i \mapsto cp_i,\ p_i\mathbf{@pre} \mapsto o_i \rfloor) \models q}{\{p\}\ \mathbf{call}\ proc(cp_1, \ldots, cp_k)\ \{q\}}\ \text{A-SENT-CALL}$$

  Observar que puede asumirse sin perder generalidad que los parámetros formales $\mathrm{pars}(proc) = p_1, \ldots, p_k$ son disjuntos con respecto a todas las otras variables que puedan aparecer en el programa.

  Un llamado a procedimiento tiene semántica abstracta entre p y q si:

  - Reemplazando en p los parámetros concretos cp_i por los formales p_i se puede garantizar la precondición del procedimiento *proc*.
  - Tal como en el caso de una asignación, sigue valiendo p, pero sobre ciertos valores viejos para los parámetros concretos, denominados o_i en la regla. Por otra parte vale la postcondición del procedimiento, donde se sustituyen los parámetros formales por los concretos, y el valor inicial de los parámetros formales por o_i, o sea los valores viejos de los parámetros concretos.

  En esta regla se comete un abuso de notación al realizar k sustituciones simultáneas. Esto no es problema ya que se trata de todas expresiones distintas.

Observar que todos los consecuentes de las reglas para la semántica abstracta son de la forma $\{p\}\ s\ \{q\}$ sin instanciar p ni q. De esta forma las reglas pueden entenderse de forma reversible como una transformación "hacia adelante" o "hacia atrás", según como se las mire. Aprovecharemos esta característica en la Sección 3.4.

Los siguientes casos sirven a modo de ejemplo para apreciar el funcionamiento de estas reglas:

$$\{x = \underline{10}\}\ x \leftarrow x + \underline{1}\ \{\exists\ x' : \textsc{Int}\ (x' = \underline{10}\ \wedge\ x = x' + \underline{1})\}$$

En este primer caso se aplica la regla de la asignación de enteros, dejando como consecuencia (q en la regla) la expresión más fuerte posible. Sería lógicamente equivalente decir que la consecuencia es $x = \underline{11}$.

$$\{\mathbf{true}\}\ \mathbf{if\ true\ then}\ a \leftarrow \underline{1}\ \mathbf{else}\ a \leftarrow \underline{0}\ \mathbf{fi}\ \{a = \underline{1}\}$$

El condicional siempre tomará la rama "then", por ende se obtiene el resultado de la misma como consecuencia.

$$\{i = \underline{14}\}\ \textbf{while}\ i \neq \underline{0}\ \textbf{:?!}\ i \geq \underline{0}\ \textbf{:\#}\ i\ \textbf{do}\ i \leftarrow i - \underline{1}\ \textbf{od}\ \{i = \underline{0}\}$$

Este es el caso de un ciclo para el cual las premisas de su regla pueden satisfacerse (el invariante vale al comienzo, el variante decrece, etc.). La salida del mismo está dada por la conjunción del invariante con la negación de la guarda, que en este caso es lógicamente equivalente a $i = \underline{0}$.

3.3. Programas seguros

En el capítulo anterior se introdujo la noción de semántica operacional y se vio que en algunos casos una ejecución podía concluir en una valuación final y en otros casos no. Sin embargo, debido a la aparición de verificaciones sobre invariantes, precondiciones y postcondiciones que pueden contener cuantificadores no es posible construir un intérprete para partir de una valuación inicial y ver si se puede llegar o no a alguna valuación final.

Es por ello que surge la necesidad de caracterizar de alguna manera a aquellos programas cuyo comportamiento será siempre correcto. Este subconjunto de programas debe ser tal que asegure que ninguna ejecución viola ninguno de los contratos. En esta sección utilizaremos la semántica abstracta para caracterizar a los programas en dicho subconjunto, a los cuales se llamará *programas seguros*.

Se dirá que un programa es seguro si todos sus procedimientos están definidos de forma tal que cualquier ejecución que parta de una valuación que cumple la precondición se ejecuta correctamente y culmina en otra valuación que satisface la postcondición.

La siguiente definición es acompañada de una idea intuitiva acerca del cumplimiento de este postulado, el mismo será demostrado en el Teorema 3.2.

Definición 3.7 (Programas seguros)**.**
Un programa $\pi \in \text{Program}$ se denominará *seguro* según las siguientes reglas:

$$\frac{}{\emptyset \text{ es seguro}}\ \text{SAFE-PROG-EMPTY}$$

Un programa vacío no admite ninguna ejecución, con lo cual todas sus ejecuciones están bien definidas.

$$\frac{\pi \text{ es seguro} \quad \{pre\}\ s\ \{post\}}{\pi,\ proc(p_1, \ldots, p_k)\ \textbf{:?}\ pre\ \textbf{:!}\ post\ \{\ s\ \}\ \text{es seguro}}\ \text{SAFE-PROG-EXTEND}$$

Las ejecuciones de la extensión pueden ser a partir de procedimientos en π, en cuyo caso ya se sabe que es seguro; o a partir de *proc*.

En este caso, como se sabe que $\{pre\}\ s\ \{post\}$ luego se puede probar que si se parte de una valuación que satisface *pre*, se puede ejecutar s y llegar a otra valuación que satisface *post*.

Se presenta a continuación el resultado que establece que un programa seguro tiene siempre su semántica operacional bien definida si se parte de valuaciones que satisfacen la precondición del procedimiento inicial que se use. Este teorema puede entenderse también como un resultado de adecuación de la semántica abstracta con respecto a la semántica operacional.

Teorema 3.2 (Los programas seguros tienen semántica operacional bien definida)**.**
Sea un programa seguro π y sea p un procedimiento en π. Luego:

$$\textit{para toda valuación } \mu \in \mathrm{M}_{\Gamma_p} \textit{ sucede que si } [\![\mathrm{P}_p]\!]_{\mu}^{\mathrm{BOOL}} = \mathrm{true}$$

$$\textit{luego existe una valuación } \mu' \in \mathrm{M}_{\Gamma_p} \textit{ tal que } \mu \rhd \pi \ : \ p \rhd \mu' \textit{ y } [\![\mathrm{Q}_p]\!]_{\mu'}^{\mathrm{BOOL}} = \mathrm{true}$$

Dem:

Se demuestra por inducción estructural, los detalles se pueden encontrar en la Sección A.3.

□

Recordar que la definición de semántica operacional que presentamos en el capítulo anterior determina que una ejecución es correcta sólo si no se viola ninguna anotación y nunca una expresión se indefine. Por ende, un programa seguro **siempre** será correcto respecto de su especificación.

3.4. Cálculo de postcondiciones y precondiciones

La semántica abstracta puede entenderse como una ejecución a través de transformaciones de expresiones booleanas que predican sobre el valor de las variables. Este concepto es utilizado para definir programas seguros, que son aquellos para los cuales sus ejecuciones siempre están bien definidas.

Otra forma de entender la semántica abstracta es como un mecanismo de prueba similar a un sistema de tipos. Así, podríamos decir que si se llega al juicio $\{pre\}\ s\ \{post\}$ para cada uno de los procedimientos, luego se trata de un programa seguro –o tipable– y por ende se ejecutará correctamente.

Al tener un sistema de tipos siempre es deseable automatizar la inferencia de juicios utilizando sus reglas. En esta sección introducimos dos cálculos: dada una sentencia s uno obtendrá una postcondición de la misma a partir de una expresión de comienzo; el otro obtendrá una precondición a partir de una expresión de llegada. Estos cálculos pueden ser vistos como mecanismos de inferencia para el sistema de tipos representado por la semántica abstracta.

Sin embargo aún contando con estos cálculos, estos dependerían de la relación de fuerza $\models$, la cual es no decidible. En el Capítulo 1 mencionamos que existen herramientas que permiten resolver automáticamente este problema de manera parcial en una cantidad de casos interesantes. O sea, dadas dos expresiones booleanas b_1 y b_2, hay 3 resultados posibles:

- *Valid* significa que se pudo probar que $b_1 \models b_2$.
- *Not valid* significa que se pudo probar que no es cierto que $b_1 \models b_2$.
- *Unknown* significa que no se sabe si $b_1 \models b_2$ o no.

Observar que, permitiendo la tercera respuesta, este es un problema decidible[1], y existen herramientas que permiten resolver –sin devolver *Unknown*– cada vez más instancias de este problema. Por ahora se asumirá que estas herramientas constituyen un oráculo y que en caso de que retorne *Unknown* no se podrá continuar con la verificación. En el Capítulo 5 se comentará en mayor detalle cómo se utilizaron estas herramientas, intentando minimizar las respuestas de tipo *Unknown*.

Asumiendo por ahora que el oráculo se comporta de forma adecuada, cerramos formalmente el círculo del proceso de verificación. Los cálculos de postcondiciones y precondiciones permiten capturar juicios

[1] En particular se puede realizar una implementación que siempre devuelva *Unknown*.

acerca de la semántica abstracta; la semántica abstracta permite establecer si un programa es seguro; un programa seguro sólo tiene ejecuciones que terminan; las ejecuciones terminan sólamente si no violan ninguna anotación.

Antes de definir los cálculos para precondiciones y postcondiciones es necesario introducir la clausura existencial de una fórmula con respecto a un conjunto de variables.

Definición 3.8 (Clausura existencial)**.**
Dada una expresión booleana $b \in \text{BoolExp}_\Gamma$ y un conjunto de variables $V \subseteq \text{dom}(\Gamma)$, se define la *clausura existencial* $Cl_{\exists V}(b) \in \text{BoolExp}_{\Gamma \ominus V}$ de la siguiente forma:

$$Cl_{\exists V}(b) = \exists\ i_1, \ldots, i_n : \textsc{Int}\ (\exists\ A_1, \ldots, A_m : \textsc{Array}\ (b))$$

Donde $\{i_1, \ldots, i_n\}$ son las variables enteras en V y $\{A_1, \ldots, A_m\}$ son las variables arreglo en V.

3.4.1. Cálculo de postcondiciones

Definimos a continuación la *postcondición calculada*. La misma se obtiene utilizando una versión especializada de las reglas de semántica abstracta, para las cuales la expresión booleana de partida –o precondición– está fijada.

Primero se introducen las reglas, luego se comentan algunos ejemplos y finalmente presentamos un resultado que vincula este cálculo con la semántica abstracta.

Definición 3.9 (Postcondición calculada)**.**
Sea π un programa seguro, $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ y $p \in \text{BoolExp}_\Gamma$ se define la postcondición de la sentencia s con respecto a la expresión booleana p, notada $\text{post}(s,\ p) \in \text{BoolExp}_{\Gamma'}$.

- Asignaciones:

$$\frac{p \models \text{safe}^{\text{I}}(i)}{\text{post}(v \leftarrow i,\ p) = \exists\ v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = i\lfloor v \mapsto v' \rfloor)}\ \textsc{post-iassign}$$

$$\frac{p \models \text{safe}^{\text{A}}(A)}{\text{post}(v \leftarrow A,\ p) = \exists\ v' : \textsc{Array}\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = A\lfloor v \mapsto v' \rfloor)}\ \textsc{post-aassign}$$

  En estos casos las reglas son muy similares a las de la semántica abstracta; entre todas las expresiones booleanas de llegada q que permitían, se utiliza la más fuerte posible.

- Creación de variables:

$$\frac{p \models \text{safe}^{\text{I}}(i)}{\text{post}(\textbf{int}\ v \leftarrow i,\ p) = p\ \wedge\ v = i}\ \textsc{post-idef}$$

$$\frac{p \models \text{safe}^{\text{A}}(A)}{\text{post}(\textbf{array}\ v \leftarrow A,\ p) = p\ \wedge\ v = A}\ \textsc{post-adef}$$

  Tal como en el caso de las asignaciones, se utiliza el q más fuerte posible.

- Sentencia vacía:

$$\frac{}{\text{post}(\textbf{skip},\ p) = p}\ \text{POST-SKIP}$$

- Secuenciamiento:

$$\frac{\text{post}(s_1,\ p) = r \quad \text{post}(s_2,\ r) = q}{\text{post}(s_1\ s_2,\ p) = q}\ \text{POST-SEQ}$$

Observar que el hecho de que las distintas reglas tengan antecedentes que podrían no llegar a cumplirse obliga a requerir como hipótesis que se encuentre bien definida la postcondición de s_1 y también la de s_2.

- Condicional:

$$\frac{p \models \text{safe}^{\text{B}}(g) \quad \text{post}(s_1,\ p \wedge g) = q_1 \quad \text{post}(s_2,\ p \wedge \neg g) = q_2}{\text{post}(\textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi},\ p) = Cl_{\exists\,\text{localVars}(s_1)}(q_1) \vee Cl_{\exists\,\text{localVars}(s_2)}(q_2)}\ \text{POST-IF}$$

Se utiliza como q la clausura existencial de las variables locales sobre la postcondición para cada una de las ramas. De esta forma se olvidan los valores para aquellas variables que quedan fuera de *scope*.

- Ciclo:

$$\frac{\begin{array}{c}\textbf{true} \models \text{safe}^{\text{B}}(inv) \quad inv \models \text{safe}^{\text{B}}(g) \quad inv \models \text{safe}^{\text{I}}(var) \\ p \models inv \quad inv \wedge g \models var > \underline{0} \\ \text{post}(var_0 \leftarrow var\ \ s,\ inv \wedge g) = q' \\ q' \models inv \quad q' \models var < var_0\end{array}}{\text{post}(\textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \textbf{od},\ p) = inv\ \wedge\ \neg g}\ \text{POST-WHILE}$$

Esta regla tiene un gran parecido con la regla de semántica abstracta para ciclos. Se usa $inv \wedge g$ como p', precondición de una iteración genérica. Se devuelve el q más fuerte posible, o sea $inv \wedge \neg g$.

- Llamado a un procedimiento:

$$\frac{1 \leq i \leq k \quad p\lfloor cp_i \mapsto p_i \rfloor \models \text{P}_{pr}}{\text{post}(\textbf{call}\ pr(cp_1, \ldots, cp_k),\ p) = \exists_{o_1,\ldots,o_k}\ (p\lfloor cp_i \mapsto o_i \rfloor\ \wedge\ \text{Q}_{pr}\lfloor p_i \mapsto cp_i,\ p_i\textbf{@pre} \mapsto o_i \rfloor)}\ \text{POST-CALL}$$

Donde $p_1, \ldots, p_k$ son los parámetros formales del procedimiento pr. Se retorna el q más fuerte que la regla para la semántica abstracta permite.

Por ejemplo:

$$\text{post}(\textbf{if}\ x = \underline{0}\ \textbf{then}\ x \leftarrow \underline{1}\ \textbf{else}\ x \leftarrow \underline{0}\ \textbf{fi},\quad \textbf{true}) \ =$$

$$= \exists\ x' : \text{Int}\ (\textbf{true} \wedge x' = 0 \wedge x = 1) \vee \exists\ x' : \text{Int}\ (\textbf{true} \wedge x' \neq 0 \wedge x = 0) \ \models$$

$$\models\ x = 1 \vee x = 0$$

En cambio, si se parte de una expresión booleana más precisa:

$$\text{post}(\textbf{if } x = \underline{0} \textbf{ then } x \leftarrow \underline{1} \textbf{ else } x \leftarrow \underline{0} \textbf{ fi}, \quad x > \underline{0}) \;=$$

$$= \exists\, x' : \textsc{Int}\ (x' > 0 \land x' = 0 \land x = 1) \lor \exists\, x' : \textsc{Int}\ (x' > 0 \land x' \neq 0 \land x = 0) \;\models$$

$$\models\; x = 0$$

Otro ejemplo interesante es ver qué sucede en el caso de toparse con un ciclo. Por ejemplo:

$$\text{post}(\textbf{while } \underbrace{i > \underline{0}}_{g} \textbf{ :?! } \underbrace{i \geq \underline{0}}_{inv} \textbf{ :\# } \underbrace{i}_{var} \textbf{ do } \underbrace{i \leftarrow i - \underline{1}}_{s} \textbf{ od}, \underbrace{i = \underline{10}}_{p})$$

Para resolver este caso es necesario asegurarse primero de satisfacer todas las hipótesis de la regla POST-WHILE.

$$\begin{aligned}
\textbf{true} \models \text{safe}^{\text{B}}(inv) \;&\rightsquigarrow\; \textbf{true} \models \textbf{true} \\
inv \models \text{safe}^{\text{B}}(g) \;&\rightsquigarrow\; i \geq \underline{0} \models \textbf{true} \\
inv \models \text{safe}^{\text{I}}(var) \;&\rightsquigarrow\; i \geq \underline{0} \models \textbf{true} \\
p \models inv \;&\rightsquigarrow\; i = \underline{10} \models i \geq \underline{0} \\
inv \land g \models var > \underline{0} \;&\rightsquigarrow\; i \geq \underline{0} \;\land\; i > \underline{0} \models i > \underline{0} \\
\text{post}(var_0 \leftarrow i \quad i \leftarrow i - \underline{1},\; i \geq \underline{0} \;\land\; i > \underline{0}) \;&\equiv\; \underbrace{\exists\, i' : \textsc{Int}\ (i' > \underline{0} \;\land\; var_0 = i' \;\land\; i = i' - \underline{1})}_{q'} \\
q' \models inv \;&\rightsquigarrow\; \exists\, i' : \textsc{Int}\ (\ldots) \models i \geq \underline{0} \\
q' \models var < var_0 \;&\rightsquigarrow\; \exists\, i' : \textsc{Int}\ (\ldots) \models i < var_0
\end{aligned}$$

Puede comprobarse que efectivamente todas estas relaciones de fuerza se satisfacen. De esta manera la postcondición calculada sería:

$$\text{post}(\textbf{while } i > \underline{0} \textbf{ :?! } i \geq \underline{0} \textbf{ :\# } i \textbf{ do } i \leftarrow i - \underline{1} \textbf{ od},\; i = \underline{10}) \;=\; i \geq \underline{0} \;\land\; i \leq \underline{0} \;\equiv\; i = \underline{0}$$

Estas reglas que definen el cálculo de postcondiciones son compatibles con la noción de semántica abstracta. En otras palabras, la inferencia que realizan es correcta con respecto al sistema de tipos introducido. Presentamos este resultado a continuación.

Teorema 3.3 (El cálculo de postcondiciones respeta la semántica abstracta)**.**
Sea $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ *y sea* $p \in \text{BoolExp}_{\Gamma}$. *Si* $\text{post}(s,\; p) = q$ *luego* $\{p\}\ s\ \{q\}$.

Dem:

Un esbozo de su demostración por inducción estructural puede hallarse en la Sección A.4.

□

Tal como se mencionó al abrir esta sección, con este mecanismo se puede capturar el concepto de programa seguro como aquel para cual todos sus procedimientos $proc(p_1, \ldots, p_k)$ **:?** *pre* **:!** *post* { *s* } cumplen que:

$$\text{post}(s,\; pre) \models post$$

3.4.2. Cálculo de precondiciones

De forma análoga, introducimos a continuación el concepto de *precondición calculada*. En este caso las reglas parten de una expresión booleana de llegada –o postcondición– fijada como parámetro.

Las ideas básicas a partir de las cuales se construyen estas reglas pueden atribuirse directamente a la operación de *weakest precondition* definida en [Dij75] aunque en nuestro caso no se pretende obtener la precondición más débil. En particular, ante un ciclo utilizamos el invariante para caracterizar algún estado seguro (no necesariamente el más débil) de entrada.

Tal como en el caso del cálculo de postcondiciones, presentamos primero las reglas, luego algunos ejemplos y finalmente el resultado que vincula este cálculo con la semántica abstracta.

Definición 3.10 (Precondición calculada)**.**
Sea π un programa seguro, $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ y $q \in \text{BoolExp}_{\Gamma'}$ se define la precondición de la sentencia s con respecto a la expresión booleana q, notada $\text{pre}(s,\ q) \in \text{BoolExp}_{\Gamma}$.

- Asignaciones y creación de variables:

$$\frac{}{\text{pre}(v \leftarrow i,\ q) = \text{safe}^{\text{I}}(i)\ \wedge\ q\lfloor v \mapsto i \rfloor}\ \text{PRE-IASSIGN}$$

$$\frac{}{\text{pre}(v \leftarrow A,\ q) = \text{safe}^{\text{A}}(A)\ \wedge\ q\lfloor v \mapsto A \rfloor}\ \text{PRE-AASSIGN}$$

$$\frac{}{\text{pre}(\textbf{int}\ v \leftarrow i,\ q) = \text{safe}^{\text{I}}(i)\ \wedge\ q\lfloor v \mapsto i \rfloor}\ \text{PRE-IDEF}$$

$$\frac{}{\text{pre}(\textbf{array}\ v \leftarrow A,\ q) = \text{safe}^{\text{A}}(A)\ \wedge\ q\lfloor v \mapsto A \rfloor}\ \text{PRE-ADEF}$$

  En estos primeros cuatro casos es necesario garantizar que la expresión es segura. Por otra parte, para llegar a q, se reemplazan en tal expresión todas las apariciones de la variable por la expresión.

- Sentencia vacía:

$$\frac{}{\text{pre}(\textbf{skip},\ q) = q}\ \text{PRE-SKIP}$$

- Secuenciamiento:

$$\frac{\text{pre}(s_1,\ r) = p \quad \text{pre}(s_2,\ q) = r}{\text{pre}(s_1\ s_2,\ q) = p}\ \text{PRE-SEQ}$$

- Condicional:

$$\frac{\text{pre}(s_1,\ q) = p_1 \quad \text{pre}(s_2,\ q) = p_2}{\text{pre}(\textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi},\ q) = \text{safe}^{\text{B}}(g)\ \wedge\ (g \wedge p_1\ \vee\ \neg g \wedge p_2)}\ \text{PRE-IF}$$

  Se pide que la guarda sea una expresión segura y también que se cumplan una de las dos situaciones:

- La guarda es verdadera y se satisface la precondición de la rama "then" para llegar a q.
- La guarda es falsa y se satisface la precondición de la rama "else" para llegar a q.

- Ciclo:

$$\frac{\begin{array}{c}\mathbf{true} \models \mathrm{safe}^{\mathrm{B}}(inv) \quad inv \models \mathrm{safe}^{\mathrm{B}}(g) \quad inv \models \mathrm{safe}^{\mathrm{I}}(var) \\ inv \wedge g \models p' \qquad p' \models var > \underline{0} \\ \mathrm{post}(var_0 \leftarrow var \;\; s,\; inv \wedge g) = q' \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q\end{array}}{\mathrm{pre}(\mathbf{while}\ g\ \mathbf{:?!}\ inv\ \mathbf{:\#}\ var\ \mathbf{do}\ s\ \mathbf{od},\; q) = inv}\ \text{PRE-WHILE}$$

  Para obtener la precondición de un ciclo se pide, de forma muy similar al caso de la postcondición de un ciclo, que se cumpla el teorema del invariante[2].

  Por otra parte se pide que la salida del ciclo, dada por $inv \wedge \neg g$ sea suficiente para probar la postcondición deseada q. En ese caso la precondición del ciclo será su invariante.

  Esta regla no garantiza que la precondición sea la más débil posible. Sólo se trata de una forma de proseguir "hacia arriba" obteniendo una posible precondición que permita llegar a q.

- Llamado a un procedimiento:

$$\frac{1 \leq i \leq k}{\begin{array}{c}\mathrm{pre}(\mathbf{call}\ proc(cp_1, \ldots, cp_k),\; q) = \mathrm{P}_{proc}\lfloor p_i \mapsto cp_i \rfloor \wedge \\ (\exists a_1, \ldots, a_k (\mathrm{Q}_{proc}\lfloor p_i \mapsto a_i,\; p_i\mathbf{@pre} \mapsto cp_i \rfloor \;\wedge\; q\lfloor cp_i \mapsto a_i \rfloor))\end{array}}\ \text{PRE-CALL}$$

  Es necesario que se cumpla la precondición del procedimiento llamado con las sustituciones de los parámetros formales por los concretos.

  Por otra parte es necesario que la postcondición del procedimiento pueda forzar q, realizando las sustituciones de parámetros formales por los concretos. Tener en cuenta el reemplazo de parámetros anteriores por sus valores cuantificados, tal como en la regla para la semántica abstracta.

Por ejemplo:

$$\begin{array}{c}\mathrm{pre}(j \leftarrow j + \underline{1}, \quad i = j \wedge j = \underline{20}) \;= \\ = \;\mathbf{true} \;\wedge\; i = j + \underline{1} \;\wedge\; j + \underline{1} = \underline{20} \;\models \\ \models \;i = \underline{20} \;\wedge\; j = \underline{19}\end{array}$$

Otro ejemplo interesante es el caso de un ciclo. Observar el mismo ejemplo que se usó para el cálculo de postcondiciones:

$$\mathrm{pre}(\mathbf{while}\ i > \underline{0}\ \mathbf{:?!}\ i \geq \underline{0}\ \mathbf{:\#}\ i\ \mathbf{do}\ i \leftarrow i - \underline{1}\ \mathbf{od},\; i = \underline{0}) \;=\; i \geq \underline{0}$$

[2]Observar que para verificar esta condición se utiliza el cálculo de postcondiciones. Podría haberse utilizado el cálculo de precondiciones, pero en ese caso la regla sería más difícil de seguir y muy distinta con respecto a la regla de semántica abstracta para este caso. Esta diferencia hubiera dificultado la prueba del resultado que vincula este cálculo con la semántica abstracta.

Tal como en el caso del cálculo de su postcondición, habría que chequear todas las mismas condiciones que garantizan que el ciclo es correcto respecto del teorema del invariante.

También puede analizarse el caso de un condicional. Por ejemplo:

$$\text{pre}(\textbf{if } j = \underline{0} \textbf{ then } i \leftarrow i + \underline{1} \textbf{ else } i \leftarrow i + \underline{2} \textbf{ fi},\ i = \underline{10})$$

En ese caso hay que calcular la precondición de cada rama:

$$\text{pre}(i \leftarrow i + \underline{1},\ i = \underline{10}) \ = \ \textbf{true} \ \wedge \ i + \underline{1} = \underline{10} \ \equiv \ i = \underline{9}$$

$$\text{pre}(i \leftarrow i + \underline{2},\ i = \underline{10}) \ = \ \textbf{true} \ \wedge \ i + \underline{2} = \underline{10} \ \equiv \ i = \underline{8}$$

Teniendo estos dos resultados auxiliares, se puede calcular una precondición para el condicional:

$$\text{pre}(\textbf{if } j = \underline{0} \textbf{ then } i \leftarrow i + \underline{1} \textbf{ else } i \leftarrow i + \underline{2} \textbf{ fi},\ i = \underline{10}) \ = \ \textbf{true} \ \wedge \ (j = \underline{0} \wedge i = \underline{9} \ \vee \ j \neq \underline{0} \wedge i = \underline{8})$$

Por último suponer que se cuenta con un procedimiento:

$$inc(x) \textbf{ :? true :! } x = x\textbf{@pre} + \underline{1} \ \{ \ \ldots \ \}$$

Y queremos obtener una precondición para un llamado a dicho procedimiento:

$$\text{pre}(\textbf{call } inc(y),\ y = \underline{15})$$

El resultado es la conjunción entre la precondición y un existencial para cada parámetro. La precondición es **true** con lo cual no afecta nada. Para el existencial, aparece una única variable a, la cual reemplaza la aparición del parámetro formal x en la postcondición. Por otra parte, debe reemplazarse x**@pre** por el parámetro concreto y. Por último se reemplaza y por su valor anterior a en la expresión booleana de llegada $y = \underline{15}$. Se obtiene:

$$\textbf{true} \ \wedge \ \exists \ a : \textsc{Int} \ (a = y + \underline{1} \ \wedge \ a = \underline{15}) \ \equiv \ y = \underline{14}$$

Por lo tanto podría decirse que:

$$\text{pre}(\textbf{call } inc(y),\ y = \underline{15}) \ \equiv \ y = \underline{14}$$

También con el cálculo de precondiciones es necesario probar que la inferencia que se realiza es consistente con el sistema de tipos, o semántica abstracta.

Teorema 3.4 (El cálculo de precondiciones respeta la semántica abstracta)**.**
Sea $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ *y sea* $q \in \text{BoolExp}_{\Gamma'}$. *Si* $\text{pre}(s,\ q) = p$ *luego* $\{p\}\ s\ \{q\}$.

Dem:

Un esbozo de su demostración por inducción estructural puede hallarse en la Sección A.5.

□

Este mecanismo no es necesario para garantizar que un programa es seguro; alcanza con contar con un cálculo de postcondiciones. Sin embargo en el capítulo siguiente haremos fuerte uso del cálculo de precondiciones para poder inferir invariantes de forma automática.

3.5. Reforzando anotaciones

Hasta aquí en este capítulo presentamos formalismos que capturan el comportamiento de las ejecuciones mediante transformaciones de estados, representados con expresiones booleanas. Los cálculos de postcondiciones y precondiciones permiten obtener formas efectivas de recorrer una sentencia "hacia adelante" o "hacia atrás" respectivamente. Con algoritmos adecuados y usando el oráculo tal como se verá en el Capítulo 5 se puede llegar a determinar si un programa es seguro.

Ya se mencionó que para realizar este tipo de verificaciones de corrección y terminación es indispensable que las anotaciones introducidas por el programador sean suficientemente precisas. Esto genera una carga de trabajo extra que en muchos casos podría llegar a ser considerada como una molestia.

Haciendo uso de los cálculos que presentamos en la sección anterior, a continuación introducimos dos técnicas que permiten aliviar esta carga de trabajo. La primera de ellas es la inferencia de precondiciones y postcondiciones de procedimientos. La segunda es el fortalecimiento de invariantes de ciclos.

3.5.1. Inferencia de especificaciones de procedimientos

La primera de las técnicas que tienen como objetivo aliviar la carga de especificación es la inferencia de precondiciones y postcondiciones.

Observar el siguiente procedimiento:

```
1: inc(x)
2: :? true
3: :! x = x@pre + 1
4: {
5:     x ← x + 1
6: }
```

Incrementar en uno

Dado que contamos con una forma de inferir postcondiciones de una sentencia el programador podría ahorrarse la tarea de escribir la postcondición del procedimiento. En particular podría inferirse:

$$\text{post}(x \leftarrow x + \underline{1},\ x = x\text{@pre}) \ = \ \exists\, x' : \textsc{Int}\ (x' = x\text{@pre} \wedge x = x' + \underline{1})$$

Y esta postcondición inferida es equivalente a la postcondición del procedimiento del ejemplo, si bien es más larga y engorrosa de leer. Observar que en cambio de partir desde **true** se partió desde $x = x\text{@pre}$, lo cual siempre es válido para todos los parámetros al comenzar la ejecución.

Observar este otro procedimiento:

```
1: div(a, b, r)
2: :? b ≠ 0
3: :! r = a/b ∧ a = a@pre ∧ b = b@pre
4: {
5:     r ← a/b
6: }
```

División entera

Suponer por un momento que no se cuenta con especificación alguna para este procedimiento y se quiere inferir la misma a partir de la sentencia $r \leftarrow a/b$. Se puede comenzar por obtener una precondición

que garantice que se alcance a completar la ejecución, o sea la precondición calculada con respecto a alcanzar **true**.

$$\text{pre}(r \leftarrow a/b,\ \textbf{true}) \ = \ \text{safe}^{\text{I}}(a/b) \ = \ b \neq \underline{0}$$

Una vez obtenida esta precondición, se puede partir desde allí para obtener la postcondición del procedimiento.

$$\text{post}(r \leftarrow a/b,\ b \neq \underline{0} \ \wedge \ a = a\textbf{@pre} \ \wedge \ b = b\textbf{@pre} \ \wedge \ r = r\textbf{@pre}) \ =$$

$$\exists\, r' : \textsc{Int}\ (b \neq \underline{0} \ \wedge \ a = a\textbf{@pre} \ \wedge \ b = b\textbf{@pre} \ \wedge \ r' = r\textbf{@pre} \ \wedge \ r = a/b)$$

Nuevamente la postcondición obtenida por este método es equivalente a la que el programador hubiera escrito en primer término.

En general podremos inferir la especificación de un procedimiento aunque la calidad resultante dependerá de la precisión de los invariantes de los ciclos.

Por ejemplo tomar el caso de un programa que incrementa en uno todos los elementos de un arreglo. Si el invariante de ese ciclo únicamente dice que la variable de iteración utilizada está en rango, luego la postcondición no podrá capturar el hecho de que los elementos del arreglo fueron incrementados.

La definición siguiente permite inferir especificaciones en el caso en que tanto la precondición como la postcondición sean desconocidas. Realizarlo cuando alguna de estas dos anotaciones se encuentra presente son casos más sencillos que no serán desarrollados.

Definición 3.11 (Inferencia de especificación)**.**
Sea $proc(p_1, \ldots, p_k)$ **:? _ :! _** $\{\ s\ \}$ un procedimiento con precondición y postcondición desconocidas.
Sean:

$$P \ = \ \text{pre}(s,\ \textbf{true})$$

$$Q \ = \ \text{post}(s,\ P \ \wedge \ p_1 = p_1\textbf{@pre} \ \wedge \ \ldots \ \wedge \ p_k = p_k\textbf{@pre})$$

Se dice que P es la *precondición inferida* de *proc* y Q es su *postcondición inferida.*

La inferencia de especificaciones presentada realiza una única pasada hacia arriba y luego hacia abajo. Podría realizarse algún tipo de cálculo de punto fijo realizando pasadas sucesivas hasta que la precondición y la postcondición inferida no sufran cambios. Esta alternativa no ha sido desarrollada, sin embargo intuimos que sería necesario algún mecanismo que asegure la convergencia, probablemente mediante la utilización de aproximaciones (conocidas como *widenings*) utilizando la relación de fuerza $\models$.

3.5.2. Fortalecimiento de invariantes

En esta sección introducimos una traducción que aumentará la precisión de los invariantes que acompañan los ciclos. La idea intuitiva es incorporar en los invariantes información relativa al punto de entrada a los mismos.

Por ejemplo, si se sabe que antes de llegar a un ciclo una variable j siempre vale 10 y se sabe que el ciclo no toca tal variable, puede agregarse $j = \underline{10}$ al invariante.

Antes de presentar esta técnica es primero necesario tener una forma de obtener las variables que una sentencia modifica. En el caso general esto es no decidible ya que por ejemplo habría que poder

resolver qué rama toma un condicional. Sin embargo se puede sobreaproximar de manera sintáctica el conjunto de variables modificadas, obteniendo el *conjunto de variables potencialmente modificadas por una sentencia*.

Definición 3.12 (Variables potencialmente modificadas)**.**
Se llama modVars a la función que, dada una sentencia $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$, devuelve el conjunto de *variables potencialmente modificadas* por la misma, definido como:

$$\begin{aligned}
\text{modVars}(v \leftarrow i) &= \{v\} \\
\text{modVars}(\textbf{int } v \leftarrow i) &= \{v\} \\
\text{modVars}(v \leftarrow A) &= \{v\} \\
\text{modVars}(\textbf{array } v \leftarrow A) &= \{v\} \\
\text{modVars}(\textbf{skip}) &= \emptyset \\
\text{modVars}(s_1\ s_2) &= \text{modVars}(s_1) \cup \text{modVars}(s_2) \\
\text{modVars}(\textbf{if } g \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) &= \text{modVars}(s_1) \cup \text{modVars}(s_2) \\
\text{modVars}(\textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od}) &= \text{modVars}(s) \\
\text{modVars}(\textbf{call } proc(cp_1, \ldots, cp_k)) &= \{cp_i \mid p_i \in \text{modVars}(\text{body}(proc))\}
\end{aligned}$$

Por ejemplo:

$$\text{modVars}(\textbf{if true then } a \leftarrow \underline{4} \textbf{ else } b \leftarrow \underline{0} \textbf{ fi}) = \{a, b\}$$

Definición 3.13 (Traducción de sentencias)**.**
Dada una sentencia $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ la misma puede ser traducida para obtener otra cuyos invariantes sean más precisos. Se llamará $\text{tr}^{\text{S}}(s,\ p)$ a la traducción de la sentencia s asumiendo que antes de su ejecución siempre es válida la expresión booleana $p \in \text{BoolExp}_{\Gamma}$.

La traducción no afecta ninguna de las sentencias básicas tales como asignaciones, sentencia vacía o llamado a procedimientos.

$$\begin{aligned}
\text{tr}^{\text{S}}(v \leftarrow i,\ p) &\stackrel{\text{def}}{=} v \leftarrow i \\
\text{tr}^{\text{S}}(v \leftarrow A,\ p) &\stackrel{\text{def}}{=} v \leftarrow A \\
\text{tr}^{\text{S}}(\textbf{int } v \leftarrow i,\ p) &\stackrel{\text{def}}{=} \textbf{int } v \leftarrow i \\
\text{tr}^{\text{S}}(\textbf{array } v \leftarrow A,\ p) &\stackrel{\text{def}}{=} \textbf{array } v \leftarrow A \\
\text{tr}^{\text{S}}(\textbf{skip},\ p) &\stackrel{\text{def}}{=} \textbf{skip} \\
\text{tr}^{\text{S}}(\textbf{call } proc(cp_1, \ldots, cp_k),\ p) &\stackrel{\text{def}}{=} \textbf{call } proc(cp_1, \ldots, cp_k)
\end{aligned}$$

Al secuenciar sentencias se traduce la primera en base a p. Luego se traduce la segunda en base a la postcondición de la traducción de la primera.

$$\text{tr}^{\text{S}}(s_1\ s_2,\ p) \stackrel{\text{def}}{=} S_1\ \text{tr}^{\text{S}}\Big(s_2,\ \text{post}(S_1,\ p)\Big) \quad \text{donde } S_1 = \text{tr}^{\text{S}}(s_1,\ p)$$

Ante un condicional se traducen ambas ramas a partir de p y el valor correspondiente de la guarda.

$$\text{tr}^{\text{S}}(\textbf{if } g \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi},\ p) \stackrel{\text{def}}{=} \textbf{if } g \textbf{ then } \text{tr}^{\text{S}}(s_1,\ p \wedge g) \textbf{ else } \text{tr}^{\text{S}}(s_2,\ p \wedge \neg g) \textbf{ fi}$$

Finalmente, en el caso de un ciclo se fortalece su invariante utilizando la información en p. Dicha expresión debe ser clausurada existencialmente sobre las variables modificadas para de esa forma preservar toda la información conocida sobre las variables que el ciclo no modifica. El cuerpo del ciclo es traducido usando el invariante fortalecido.

$$\mathrm{tr}^{\mathrm{S}}(\textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od},\ p) \stackrel{\text{def}}{=} \textbf{while } g \textbf{ :?! } I \textbf{ :\# } var \textbf{ do } \mathrm{tr}^{\mathrm{S}}(s,\ I \wedge g) \textbf{ od}$$

Donde $I = inv \wedge Cl_{\exists \,\mathrm{modVars}(s)}(p)$.

Por ejemplo:

$$\mathrm{tr}^{\mathrm{S}}(\textbf{while } i > 0 \textbf{ :?! } i \geq 0 \textbf{ :\# } i \textbf{ do } i \leftarrow i - \underline{1} \textbf{ od},\ j = i + \underline{1}) =$$

$$\textbf{while } i > 0 \textbf{ :?! } i \geq 0 \wedge \exists\, i' : \textsc{Int}\ (j = i' + \underline{1}) \textbf{ :\# } i \textbf{ do } i \leftarrow i - \underline{1} \textbf{ od}$$

La definición anterior podría llegar a fortalecer invariantes de formas distintas dependiendo de cómo se genera el árbol de sintaxis abstracta de una sentencia. En particular la forma en la que se asocie una cadena de sentencias secuenciadas. Para evitar ambigüedades al respecto, asumiremos que una secuencia de sentencias asocia siempre a derecha.

3.6. Conclusiones

En el capítulo anterior definimos una noción de semántica basada en transformaciones de valuaciones para las variables de un programa. Dado un programa, una valuación para los parámetros y un procedimiento inicial, la ejecución podía llegar a una valuación final o podía indefinirse en algún punto debido a la violación de alguna anotación o la aparición de una división por cero o un acceso fuera de rango. Determinar a priori qué sucedería es en general no decidible.

En este capítulo presentamos una noción alternativa de semántica. A diferencia de la semántica operacional, se trabaja con transformaciones de expresiones booleanas que representan valuaciones genéricas. Otra forma de entender esta forma alternativa de semántica es como un mecanismo de prueba análogo a un sistema de tipos.

Utilizando esta versión abstracta de la semántica caracterizamos un subconjunto de programas a los que se llamó programas seguros. El hecho de que un programa sea seguro garantiza que su semántica operacional estará bien definida, siempre que se parta de valuaciones iniciales que satisfagan la precondición.

Vimos que se puede entender el conjunto de reglas que hacen a un programa seguro como un sistema de tipos. En todo sistema de tipos siempre es muy útil contar con mecanismos de inferencia y por eso presentamos los cálculos de postcondiciones y precondiciones. Estos cumplen el rol de motor de inferencia, sin embargo hacen uso de la noción de fuerza, la cual es no decidible. Se comentó que existen herramientas que resuelven de forma parcial este problema, aunque los detalles de su uso serán presentados en el Capítulo 5.

Sin embargo, aún contando con buenas herramientas que permitan resolver el problema de $\models$, la verificación sigue siendo una tarea engorrosa. Incluso un programa de tamaño moderado requeriría por parte del programador la escritura de decenas de invariantes, precondiciones y postcondiciones. Es por eso que se presentaron una serie de refuerzos de anotaciones. Los mismos permiten obviar ciertos elementos de la especificación, dejando que los complete el motor de inferencia nutrido por los cálculos de precondiciones y postcondiciones.

Una de las técnicas presentadas permite aumentar la precisión de los invariantes, sin embargo lo hace aumentándolos con información sobre las variables que no modifica el ciclo. Sigue siendo necesario aportar "a mano" un invariante que predique sobre las variables que el ciclo modifica.

Hay una serie de tareas relativamente frecuentes para las cuales se suelen utilizar ciclos que obedecen a ciertos patrones. Inicializar todos los elementos de un arreglo o sumarle 3 a cada uno de ellos son dos tareas muy similares. El invariante en estos dos casos sería casi idéntico. Copiar un arreglo tampoco diferiría mucho.

En el capítulo siguiente presentaremos nuevas construcciones para el lenguaje. Las mismas permiten escribir de forma concisa este tipo de tareas para las cuales el invariante es inferible utilizando *templates*. En caso de que el programador opte por usar estas construcciones, puede obviar la tarea de proveer invariante y variante. De esta forma se reduce al mínimo la necesidad de anotar el código con el objetivo de verificarlo automáticamente.

Capítulo 4

Construcciones de alto nivel como anotaciones

"*Language is not an abstract construction of the learned, or of dictionary makers, but is something arising out of the work, needs, ties, joys, affections, tastes, of long generations of humanity, and has its bases broad and low, close to the ground.*"

Noah Webster

En los dos capítulos anteriores presentamos un lenguaje que cuenta con anotaciones que permiten realizar verificaciones de corrección y terminación. Para que estas verificaciones puedan llevarse a cabo con éxito, las anotaciones provistas deben ser suficientemente precisas y esto supone una carga de trabajo extra.

En el Capítulo 3 introdujimos mecanismos que permiten calcular una precondición (o una postcondición) a partir de una sentencia dada. El objetivo de este capítulo es combinar estos mecanismos de forma tal de obtener nuevas construcciones de iteración que no requieran que se provea un invariante ni una función variante.

El lenguaje presentado, como sucede con casi todos los lenguajes imperativos, se encuentra muy ligado a la arquitectura de las computadoras. Las estructuras de control condicionales y de iteración son un reflejo refinado de las instrucciones básicas que tienen los microprocesadores.

En otros paradigmas, tales como la programación funcional y crecientemente la programación orientada a objetos, las estructuras de recursión y control respectivamente, han evolucionado.

En Haskell [HPF99], por ejemplo, se trabaja con patrones de recursión tales como *map*, que permite aplicar una función a cada elemento en una lista; o *filter*, que preserva de una lista únicamente aquellos elementos que cumplen un cierto criterio.

En Smalltalk [GR83] existen diversos patrones de iteración tales como *select*, *collect*, *inject*. Los mismos son implementados como mensajes a ciertos objetos, que permiten realizar tareas tales como filtrar una colección según un criterio, aplicar una función a cada elemento, etc.

Incluso Java, un lenguaje masivamente usado, ha sido extendido en su versión 5.0 [Aus04] con una construcción de tipo *for each* que evita la utilización de un índice o iterador al recorrer cualquier colección.

Todos estos son ejemplos de tareas que podrían ser realizadas con un ciclo tradicional (o recursión explícita en el caso de Haskell). Sin embargo las versiones más específicas son preferibles ya que:

1. Ahorran tiempo de programación.
2. El código fuente se hace notablemente más corto y legible, ya que en general los nombres de estas construcciones están muy ligados a la tarea que realizan.
3. Evitan errores comunes, como por ejemplo olvidarse de incrementar un índice.

Además de estas ventajas, tanto el invariante como el variante para estas estructuras de control obedecen a ciertos patrones. Por ejemplo, al realizar un *map* se sabe que todos los elementos ya visitados son el resultado de aplicar la función. Por otra parte, los elementos aún no visitados siguen inalterados. Por último, si se garantiza que la función que se aplica en cada elemento no toca el índice, luego se puede asegurar la terminación.

Sin embargo estos patrones permiten conocer la forma general que tendría el invariante en esos casos, faltaría completar qué quiere decir que un elemento "ya fue visitado", o qué quiere decir que un elemento "cumple la propiedad buscada". Para ello utilizamos los mecanismos de inferencia de postcondiciones y precondiciones vistos en el capítulo anterior.

A pesar de muchos esfuerzos en esta línea, obtener invariantes para un ciclo sin anotaciones es en general una tarea difícil. La herramienta DAIKON [EPG+07], por ejemplo, utiliza trazas de código instrumentado para dinámicamente obtener candidatos a invariantes. En cambio, la herramienta HOUDINI [FL01] genera y luego verifica los invariantes con técnicas análogas a las presentadas en este trabajo.

Nuestro enfoque combina técnicas de inferencia basadas en lógica de Floyd-Hoare con patrones de iteración. De esta forma podemos contar con las ventajas de cada uno de estos mundos. Las técnicas de inferencia nos otorgan versatilidad y los patrones de iteración nos brindan un marco que permite caracterizar de manera muy precisa los invariantes inferidos.

A continuación presentamos algunas de las posibles construcciones con las que el lenguaje podría ser ampliado. Observar que la enumeración no pretende ser exhaustiva, sino que intenta cubrir de manera eficaz una serie suficientemente variada de tareas realizables con ciclos.

4.1. Metasentencia *map*

Observar los siguientes programas:

```
 1: arrayInc(A)
 2: :? true
 3: :! ∀k / 0 ≤ k < |A| :  A[k] = A@pre[k] + 1
 4: {
 5:     int i ← 0
 6:     while i < |A|
 7:         :?! 0 ≤ i ≤ |A| ∧
 8:             ∀k / 0 ≤ k < i :  A[k] = A@pre[k] + 1 ∧
 9:             ∀k / i ≤ k < |A| :  A[k] = A@pre[k]
10:         :# |A| − i
11:     do
12:         A[i] ← A[i] + 1
13:         i ← i + 1
14:     od
15: }
```

Incrementar un arreglo en 1

```
 1: firstN(A)
 2: :? |A| > 0
 3: :! ∀k / 0 ≤ k < |A| :  A[k] = k
 4: {
 5:     int i ← 0
 6:     while i < |A|
 7:         :?! 0 ≤ i ≤ |A| ∧
 8:             ∀k / 0 ≤ k < i :  A[k] = k ∧
 9:             ∀k / i ≤ k < |A| :  A[k] = A@pre[k]
10:         :# |A| − i
11:     do
12:         A[i] ← i
13:         i ← i + 1
14:     od
15: }
```

Lista de los primeros naturales

El primero de ellos incrementa en 1 todos los elementos de un arreglo. El segundo retorna la lista de los primeros n naturales, donde n está dado por el tamaño del arreglo que se recibe como parámetro.

En principio estos dos programas calculan un resultado muy diferente. Sin embargo, la forma en que llegan a tal resultado es en gran medida similar. Se recorre todo el arreglo, realizando una tarea sobre cada uno de los elementos del mismo.

Los programas coinciden en que:

- El cuerpo del ciclo realiza una transformación sobre un elemento del arreglo. La misma no depende de otros elementos del arreglo.
- Se cuenta con una variable de iteración i que cumple que:
  1. Comienza en 0.
  2. Se itera mientras valga menos que $|A|$.
  3. Se incrementa al final de cada iteración.
  4. No es alterada por la transformación que se hace a cada elemento del arreglo A.
- El invariante especifica que:
  1. La variable de iteración i está en rango.
  2. Todos los elementos ya visitados han sufrido la transformación.
  3. Todos los elementos aún no visitados permanecen intactos.
- El variante indica que se está recorriendo el arreglo hacia adelante.
- La postcondición indica que todos los elementos del arreglo sufrieron la transformación.

Este tipo de iteración puede ser codificada con una metasentencia *map*, la cual se presenta a continuación.

Definición 4.1 (Metasentencia *map*).
Sea Γ una clasificación tal que $\Gamma(A) = \textsc{Array}$ e $i \notin \text{dom}(\Gamma)$.

Sea $s \in \text{Sentence}_{\Gamma\{i \mapsto \textsc{Int}\},\pi,\Gamma'}$ una sentencia tal que su conjunto de variables potencialmente modificadas $\text{modVars}(s)$ contiene únicamente variables locales y a A. Más específicamente, se debe garantizar sintácticamente que A es modificado únicamente en la posición i. Adicionalmente, s sólo puede acceder a A para obtener dicha posición o para obtener su tamaño.

Luego, se define la *metasentencia* map:

$$\textbf{map } s \textbf{ in } A[..\ i\ ..]$$

Se trata de una metasentencia. La misma puede entederse como una *macroexpansión* que se reemplaza por:

$$\begin{array}{l}
\textbf{int } i \leftarrow 0 \\
\textbf{while } i < |A| \\
\quad \textbf{:?!} \ 0 \leq i \leq |A| \ \wedge \\
\qquad \forall k \ / \ 0 \leq k < i : \ post_s \lfloor i \mapsto k \rfloor \ \wedge \\
\qquad \forall k \ / \ i \leq k < |A| : \ A[k] = A\textbf{@pre}[k] \\
\quad \textbf{:\#} \ |A| - i \\
\textbf{do} \\
\quad s \\
\quad i \leftarrow i + 1 \\
\textbf{od}
\end{array}$$

Donde $post_s = \text{post}(s,\ 0 \leq i < |A| \ \wedge \ Cl_{\exists\{A\}}(p))$. Se le realiza una sustitución de i por k porque se la utiliza para expresar que todos los elementos ya visitados (indexados con k) satisfacen que s ha sido aplicado sobre ellos.

Una vez definida la metasentencia *map*, se puede utilizar para escribir de forma más compacta y legible los programas anteriores.

```
1: easyArrayInc(A)
2: :? |A| > 0
3: :! ∀k / 0 ≤ k < |A| : A[k] = A@pre[k] + 1
4: {
5:     map
6:         A[i] ← A[i] + 1
7:     in A[.. i ..]
8: }
```

Incrementar un arreglo en 1 usando *map*

```
1: easyFirstN(A)
2: :? |A| > 0
3: :! ∀k / 0 ≤ k < |A| : A[k] = k
4: {
5:     map
6:         A[i] ← i
7:     in A[.. i ..]
8: }
```

Lista de los primeros naturales usando *map*

Con un esquema como este ya no es necesario proveer invariante para un número importante de casos. Observar cómo desaparece el engorroso invariante que acompañaba de forma necesaria a estos programas.

4.2. Metasentencia *find*

Otra tarea que aparece con cierta frecuencia al escribir programas es encontrar el primer elemento de un arreglo que satisface una determinada propiedad. Observar los siguientes ejemplos:

```
1: linearSearch(A, e, i)
2: :? ∀k / 0 ≤ k < |A| : A[k] = e
3: :! A[i] = e ∧ A = A@pre ∧ e = e@pre
4: {
5:     i ← −1
6:     int j ← 0
7:     while j < |A| ∧ i = −1
8:         :?! 0 ≤ j ≤ |A| ∧
9:             (i = −1 ⇒ ∀k / 0 ≤ k < j : (A[k] ≠ e)) ∧
10:            (i ≠ −1 ⇒ ∀k / 0 ≤ k < i : (A[k] ≠ e) ∧ a[i] = e) ∧
11:            A = A@pre ∧ e = e@pre
12:        :# |A| − j
13:    do
14:        if A[j] = e then
15:            i ← j
16:        fi
17:        j ← j + 1
18:    od
19: }
```

Busqueda lineal

```
1:  fixedPoint(A, i)
2:  :? true
3:  :! (i = −1 ⇒ ∀k / 0 ≤ k < |A| : (A[k] ≠ k)) ∧
4:      (i ≠ −1 ⇒ ∀k / 0 ≤ k < i : (A[k] ≠ k) ∧ a[i] = i) ∧ A = A@pre
5:  {
6:      i ← −1
7:      int j ← 0
8:      while j < |A| ∧ i = −1
9:          :?! 0 ≤ j ≤ |A| ∧
10:             (i = −1 ⇒ ∀k / 0 ≤ k < j : (A[k] ≠ k)) ∧
11:             (i ≠ −1 ⇒ ∀k / 0 ≤ k < i : (A[k] ≠ k) ∧ a[i] = i) ∧
12:             A = A@pre
13:         :# |A| − j
14:     do
15:         if A[j] = j then
16:             i ← j
17:         fi
18:         j ← j + 1
19:     od
20: }
```

Busqueda del primer punto fijo en un arreglo

El primero de ellos retorna en i la posición de la primera aparición de e en el arreglo A. El segundo retorna en i la posición del primer elemento del arreglo A que coincide con su índice, ó -1 en caso de que ninguno coincida.

Tal como los ejemplos presentados en el apartado anterior, estos dos programas calculan resultados distintos, sin embargo su estructura es muy similar.

Ambos programas coinciden en que:

- El cuerpo del ciclo se fija si un elemento del arreglo cumple una determinada propiedad.
- El ciclo itera mientras no se haya llegado al final del arreglo y no se haya encontrado ningún elemento que cumpla la propiedad.
- Se cuenta con una variable de resultado i que vale -1 al comienzo, representando que aún no se encontró ningún elemento que cumpla la propiedad.
- Se cuenta con una variable de iteración j que cumple que:
  1. Comienza en $\underline{0}$.
  2. Se itera mientras valga menos que $|A|$.
  3. Se incrementa al final de cada iteración.
- El invariante especifica que:
  1. La variable de iteración j está en rango.
  2. En caso de que la variable de resultado sea -1, todos los elementos ya visitados no cumplen la propiedad.

3. En caso de que la variable de resultado sea distinta de -1, luego todos los elementos anteriores a i no cumplen la propiedad y el elemento ubicado en i la cumple.

- El variante indica que se está recorriendo el arreglo hacia adelante.
- La postcondición indica que la variable de resultado i indica el primer elemento que cumple la propiedad del arreglo, o -1 en caso de no haber ninguno.
- El arreglo no es modificado durante el ciclo.

Este tipo de iteración será codificada con una sentencia *find*, la cual se presenta a continuación.

Definición 4.2 (Metasentencia *find*).
Sea Γ una clasificación tal que $\Gamma(i) = \text{INT}$, $\Gamma(A) = \text{ARRAY}$ y $x, j \notin \text{dom}(\Gamma)$.

Sea $s \in \text{Sentence}_{\Gamma\{x,j\mapsto\text{INT}\},\pi,\Gamma'}$ una sentencia que únicamente modifica variables locales y a x. Esto se garantiza utilizando $\text{modVars}(s)$ para obtener una sobreaproximación sintáctica del conjunto de variables modificadas por s.

La sentencia s representa el test que determina si el j-ésimo elemento de A cumple una cierta propiedad, alterando x para dejarla en 0 si el elemento j del arreglo A no la cumple, o un número distinto de 0 si la cumple. Finalmente, en la variable de resultado i se dejará la posición del primer elemento del arreglo A que cumple la propiedad, o -1 si ninguno la cumple.

Para ello se define la *metasentencia* find:

$$i \leftarrow \textbf{find } x \textbf{ as } s \textbf{ in } A[..\ j\ ..]$$

Tal como la otra metasentencia presentada, un *find* es una *macro* que se reemplaza por:

$i \leftarrow -1$
int $j \leftarrow 0$
while $j < |A| \wedge v = -1$
  :?! $0 \leq j \leq |A| \wedge$
    $(i = -1 \ \Rightarrow\ \forall k \ /\ 0 \leq k < j :\ (post_s\lfloor j \mapsto k\rfloor \Rightarrow x = 0)) \wedge$
    $(i \neq -1 \ \Rightarrow\ \forall k \ /\ 0 \leq k < i :\ (post_s\lfloor j \mapsto k\rfloor \Rightarrow x = 0) \wedge (post_s\lfloor j \mapsto i\rfloor \Rightarrow x \neq 0))$
  :# $|A| - j$
do
  s
  if $x \neq 0$ **then**
    $i \leftarrow j$
  fi
  $j \leftarrow j + 1$
od

Donde $post_s = \text{post}(s,\ 0 \leq j < |A| \ \wedge\ Cl_{\exists\{i\}}(p))$. Se la utiliza sustituyendo j por k cuando se quiere predicar sobre elementos anteriores. También se la utiliza sustituyendo j por i para referirse al elemento resultado.

Utilizando la nueva metasentencia, podemos reescribir los programas presentados anteriormente:

```
1:  easyLinearSearch(A, e, i)
2:  :? true
3:  :! A[i] = e ∧ A = A@pre ∧ e = e@pre
4:  {
5:      i ← find x as
6:          if A[j] = e then
7:              x ← 1
8:          else
9:              x ← 0
10:         fi
11:     in A[.. j ..]
12: }
```

Busqueda lineal utilizando *find*

```
1:  easyFixedPoint(A, i)
2:  :? true
3:  :! (i = −1 ⇒ ∀k / 0 ≤ k < |A| : (A[k] ≠ k)) ∧
4:      (i ≠ −1 ⇒ ∀k / 0 ≤ k < i : (A[k] ≠ k)∧a[i] = i) ∧ A = A@pre
5:  {
6:      i ← find x as
7:          if A[j] = j then
8:              x ← 1
9:          else
10:             x ← 0
11:         fi
12:     in A[.. j ..]
13: }
```

Punto fijo utilizando *find*

Tal como en el caso anterior, los invariantes largos y difíciles de leer desaparecen. De todas formas la postcondición de "easyFixedPoint" sigue siendo engorrosa, pero eso es únicamente porque queremos indicar con precisión que el punto fijo hallado es el primero. Caso contrario bastaría poner:

$$i \neq -1 \ \Rightarrow \ A[i] = i$$

La expresividad de la metasentencia *find* puede ser ampliada agregando un parámetro que indique el sentido de iteración. Adicionalmente, se podrían incorporar cotas de iteración paramétricas que permitirían encontrar el primer (o último) elemento dentro de un subarreglo determinado.

4.3. Metasentencia *for*

Hasta aquí se presentaron dos construcciones que permiten evitar la escritura del invariante y el variante cuando la tarea a realizar se trata de una asignación simultánea o una búsqueda. Si bien suelen aparecer con cierta frecuencia en diversos programas, su alto grado de especificidad hace que su uso esté acotado a las tareas para las cuales fueron diseñadas.

En esta sección presentamos la metasentencia *for*. La misma se trata de una construcción de iteración que utiliza una variable entera i acotada en un cierto intervalo $[l, h)$. Esta sentencia suele aparecer en una gran parte de los lenguajes imperativos o con orientación a objetos.

Sin embargo, en este trabajo presentamos un mecaniso que permite verificar esta construcción sin la necesidad de que se provea un invariante. A diferencia de las otras dos construcciones introducidas en este capítulo, el *for* no está diseñado para realizar una tarea específica. Por el contrario, la inferencia de invariante en este caso no se restringe a ningún patrón prefijado.

Observar el programa que calcula el máximo de un arreglo en la página 30. La postcondición del mismo establece que todo elemento del arreglo es menor o igual al máximo y que algún elemento del arreglo es igual al máximo.

$$\forall k \ / \ \underline{0} \leq k < |A| : \ m \geq A[k] \quad \wedge \quad \exists k \ / \ \underline{0} \leq k < |A| : \ m = A[k]$$

El invariante del ciclo que calcula dicho máximo establece que, una vez recorridos los primeros i elementos, todos son menores o iguales al candidato a máximo y alguno es igual al candidato a máximo.

$$\forall k \ / \ \underline{0} \leq k < i : \ m \geq A[k] \quad \wedge \quad \exists k \ / \ \underline{0} \leq k < i : \ m = A[k]$$

Observar la similitud entre la postcondición del ciclo y su invariante. La única diferencia, la cota superior de la cuantificación, coincide justamente con la condición de salida del ciclo, o sea $i = |A|$.

En general, si un procedimiento para ser seguro requiere que un *for* alcance a probar una determinada propiedad Q, luego debe valer que:

$$I \ \wedge \ i = h \ \Rightarrow \ Q$$

Donde I es el invariante del *for*, i su variable de iteración y h su cota superior.

Podría esperarse que en algunos casos alcance con utilizar como invariante la postcondición del *for*, reemplazando en ella las apariciones de la cota superior por la variable de iteración. O sea:

$$I \ \cong \ Q\lfloor h \mapsto i \rfloor$$

Basándose en esta idea intuitiva, presentamos a continuación la metasentencia *for*.

Definición 4.3 (Metasentencia *for*)**.**
Sea Γ una clasificación tal que $i \notin \mathrm{dom}(\Gamma)$. Sean l, h dos expresiones enteras en IntExp_Γ. Sea s una sentencia en $\mathrm{Sentence}_{\Gamma,\pi,\Gamma}$ tal que no modifica ninguna variable que aparezca en l ni en h ni a la variable i.

Luego se define la metasentencia *for* notada:

for i **from** l **to** h **do** s **od**

La misma es una *macro* que se reemplaza por:

int $i \leftarrow l$
while $i < h$
  :?! I
  :# $h - i$
do
  s
  $i \leftarrow i + \underline{1}$
od

El invariante I se obtiene a partir de la postcondición del ciclo Q_c de la siguiente forma:

$$I \stackrel{\text{def}}{=} l \leq i \leq h \ \wedge \ Q_c\lfloor h \mapsto i \rfloor$$

Se incorporan las cotas de la variable i; tal información nunca podría venir de Q_c ya que i es una variable local al *for*.

Resta indicar la forma en la que se calcula la postcondición Q_c para el *for*. Si s es una sentencia en donde se encuentra el *for* que quiere reemplazarse y q es la postcondición a la que s debe llegar, se utiliza forQ(s, q) para obtener la postcondición Q_c del *for* dentro de s para llegar a q. Se define de la siguiente manera:

$$\text{forQ}(\textbf{for i from l to h do s od}, \text{ q}) \stackrel{\text{def}}{=} q$$

$$\text{forQ}(\text{s}_1 \ \text{s}_2, \text{ q}) \stackrel{\text{def}}{=} \begin{cases} \text{forQ}(\text{s}_2, \text{ q}) & \text{si } \textbf{for } i \textbf{ from } l \textbf{ to } h \textbf{ do } s \textbf{ od} \in s_2 \\ \text{forQ}(\text{s}_1, \text{ pre}(\text{s}_2, \text{ q})) & \text{sino} \end{cases}$$

$$\text{forQ}(\textbf{if g then } \text{s}_1 \textbf{ else } \text{s}_2 \textbf{ fi}, \text{ q}) \stackrel{\text{def}}{=} \begin{cases} \text{forQ}(\text{s}_1, \text{ q}) & \text{si } \textbf{for } i \textbf{ from } l \textbf{ to } h \textbf{ do } s \textbf{ od} \in s_1 \\ \text{forQ}(\text{s}_2, \text{ q}) & \text{sino} \end{cases}$$

$$\text{forQ}(\textbf{while g :?! inv :\# var do s od}, \text{ q}) \stackrel{\text{def}}{=} \text{forQ}(\text{s}, \text{ q})$$

Con la nueva metasentencia *for*, podemos reescribir el programa que calcula el máximo de un arreglo de la siguiente manera:

```
 1: easyMax(A, m)
 2: :? |A| > 0
 3: :! ∀k / 0 ≤ k < |A| : m ≥ A[k] ∧
 4:     ∃k / 0 ≤ k < |A| : m = A[k] ∧ A = A@pre
 5: {
 6:     m ← A[0]
 7:     for i from 1 to |A| do
 8:         if A[i] > m then
 9:             m ← A[i]
10:         fi
11:     od
12: }
```

El máximo de un arreglo usando *for*

Recordar que en el caso de una metasentencia *for* la inferencia de invariante se realiza sin obedecer a ningún patrón de iteración. Sin embargo, el candidato a invariante que se obtiene puede ser incorrecto o insuficiente para poder probar la corrección del ciclo.

Esto no supone problema alguno para el proceso de verificación. El candidato a invariante obtenido mediante la técnica de inferencia presentada es utilizado para ver si alcanza a probar la corrección del ciclo. Se espera que esto sea así la mayoría de las veces, pero si se fallara para probar esta propiedad –ya sea porque el invariante es muy débil, o porque las herramientas de prueba no alcanzan a probarlo– hay algunas opciones a tener en cuenta.

La primera de ellas es realizar el reemplazo de la metasentencia en el código, junto con su candidato a invariante y a partir de allí se podría mejorar "a mano" la precisión del mismo. Esto ahorraría la escritura inicial de un invariante potencialmente grande, dejando para el programador la única tarea de refinarlo.

Otra opción es ampliar la metasentencia *for* para que pueda especificarse una parte del invariante y dejar que el resto corra por cuenta del mecanismo de inferencia. De esta forma se combina el mecanismo de inferencia con el invariante parcial provisto "a mano".

4.4. Conclusiones

En los capítulos anteriores presentamos un lenguaje junto con una serie de formalismos que permiten constatar si un programa es seguro o no. Se comentó la necesidad de que el programador escriba anotaciones suficientemente precisas para que el proceso de verificación pueda ser llevado a cabo exitosamente. Por último introdujimos mecanismos de inferencia que permiten obtener una precondición o una postcondición a partir de una sentencia.

En este capítulo utilizamos los mecanismos de inferencia para lograr estructuras de iteración que no requieren la escritura de invariantes, preservando la verificabilidad. Para ilustrar esto presentamos tres metasentencias concretas: *map*, *find* y *for*. Las mismas no forman parte del lenguaje; son traducidas para obtener sentencias, evitando de esta manera la escritura de fragmentos de código –en particular anotaciones de invariantes y variantes– potencialmente mucho más largos y engorrosos.

Las dos primeras metasentencias introducidas tienen como objetivo realizar tareas específicas. La tercera, en cambio, es de propósito general, aunque su expresividad se encuentra limitada debido a la forma en la que obtiene su invariante. Más allá de las limitaciones de estas tres metasentencias, ilustramos su utilidad mediante pequeños casos en los que la utilización de estas abreviaturas permiten ahorrar una cantidad relevante de trabajo. Asimismo, estas estructuras ofrecen las mismas ventajas que cualquier otro elemento de alto nivel, tales como facilidad de lectura y ahorro en tiempo de programación.

Ante nuevas necesidades otras metasentencias distintas podrían incorporarse, por ejemplo filtrar un arreglo según un bloque de código, ver si todos los elementos de un arreglo cumplen alguna propiedad, etc. La incorporación de nuevas metasentencias no supone riesgo alguno para los resultados presentados en este trabajo, esto se debe a que las mismas no alteran el lenguaje de base, sobre el que todas las demostraciones son realizadas.

En general, la técnica de utilizar construcciones de alto nivel permite reemplazar la tarea de encontrar un invariante por la de buscar una metasentencia adecuada para la tarea a realizar. De esta forma, se podría afirmar que la metasentencia *es* en cierta forma el invariante. La especificación de ciclos se realiza a través de metasentencias y fragmentos de código, los cuales se espera sean más familiares para el programador que la lógica de primer orden.

A lo largo de estos primeros 4 capítulos introdujimos diversos formalismos, mecanismos y procedimientos cuya composición permite llevar a cabo de manera relativamente poco intrusiva la prueba de corrección y terminación de programas. En el capítulo siguiente ahondaremos en los detalles y consideraciones a tener en cuenta al implementar estas técnicas. Presentaremos además un prototipo de herramienta que lleva a la práctica el grueso de estos conceptos.

Capítulo 5

Implementación

"*In seeking absolute truth we aim at the unattainable and must be content with broken portions.*"

William Osler

En los capítulos anteriores propusimos un lenguaje de programación muy simple, cuyo diseño obedece principalmente a la meta de facilitar la verificabilidad de sus programas. Presentamos técnicas y conceptos que permiten demostrar formalmente la corrección y terminación de programas escritos en este lenguaje. Para poder determinar de forma automática si un programa es seguro (es correcto y termina) es necesario contar con una herramienta que implemente estas técnicas.

Recordar que para que un programa sea seguro debe suceder que todos sus procedimientos $proc(p_1, \ldots, p_k)$ **:?** pre **:!** $post$ { s } cumplan:

$$\{pre\}\ s\ \{post\}$$

Una forma muy simple de corroborar este hecho es:

- Implementar un algoritmo que use algún tipo de oráculo para calcular $Q = \text{post}(s,\ pre)$.
- Preguntarle a dicho oráculo si $Q \models post$.

Si no se toman ciertas precauciones dicho enfoque tiene un serio inconveniente: el tamaño de las preguntas que se le hacen al oráculo puede crecer exponencialmente respecto a la entrada. Como ejemplo suponer que se cuenta con la siguiente sentencia s:

if $x > \underline{0}$ **then**
    $x \leftarrow x + \underline{1}$
else
    $x \leftarrow x + \underline{2}$
fi

En ese caso:

$$\text{post}(s,\ x = \underline{7}) \ =\ \exists\, x' : \textsc{Int}\ (x' = \underline{7} \wedge x' > \underline{0} \wedge x = x' + \underline{1}) \vee \exists\, x' : \textsc{Int}\ (x' = \underline{7} \wedge x' \leq \underline{0} \wedge x = x' + \underline{2})$$

Si en cambio se secuencia s dos veces consecutivas, luego:

$$\text{post}(s\ s,\ x = \underline{7}) \ =\ \exists\, x'' : \textsc{Int}\ ((\text{post}(s,\ x = \underline{7}))\lfloor x \mapsto x'' \rfloor \ \wedge\ x'' > \underline{0} \wedge x = x'' + \underline{1}) \vee$$

$$\exists\, x'' : \text{INT}\ ((\text{post}(s,\ x = \underline{7}))\lfloor x \mapsto x''\rfloor\ \wedge\ x'' \leq \underline{0} \wedge x = x'' + \underline{2})$$

Observar que post($s\ s,\ x = \underline{7}$) contiene a post($s,\ x = \underline{7}$) dos veces. Si se hiciera post($s\ s\ s,\ x = \underline{7}$) tal subexpresión aparecería 4 veces. Se puede apreciar el crecimiento exponencial con respecto a la cantidad de condicionales que aparecen en el código (aun cuando estos no están anidados). Este crecimiento exponencial indica que un enfoque "ingenuo" (calcular la postcondición del cuerpo y ver que implica el contrato) no es eficiente. Se trata de un hecho conocido en el ámbito de la verificación.

A continuación analizamos el funcionamiento de una de las herramientas que representa el estado del arte, SPEC# [BLS04]. Algunos aspectos de nuestro enfoque están inspirados en esta herramienta, aunque con una serie de diferencias importantes. Estos puntos de contacto, así como las cuestiones en las cuales nuestro enfoque se aparta, serán detallados en la Subsección 5.1.1.

La metodología usada por la herramienta SPEC# es traducir programas en un dialecto de C# a un lenguaje intermedio denominado BOOGIE-PL [DL05]. Esta traducción es realizada de tal forma que si el programa traducido es correcto luego el programa original también lo es.

El lenguaje intermedio BOOGIE-PL tiene una semántica no determinística y no está pensado para que sus programas sean ejecutados sino para especificar, mediante una abstracción, las características que cumplen todas las trazas de un programa en el lenguaje original.

Al contar con un lenguaje intermedio, la verificación puede desdoblarse en dos problemas:

1. Traducir programas escritos en cualquier lenguaje imperativo a "programas" del lenguaje intermedio que describen todas las trazas posibles. La traducción tiene que ser tal que preserve la corrección de la verificación que se hará sobre ella respecto de la corrección del programa original.

   Esto presenta inconvenientes tales como describir en el lenguaje intermedio el efecto de las asignaciones, de los ciclos, los condicionales y las características de objetos como *dynamic binding*. En el caso particular de BOOGIE-PL existen traducciones desde SPEC# y desde C [CLQR07].

2. Verificar que todas las trazas descriptas en la traducción hecha en el punto anterior satisfacen todos los contratos. Para ello se utilizan herramientas de demostración automática o asistida.

   El lenguaje BOOGIE-PL fue diseñado teniendo en cuenta este segundo punto. Un programa BOOGIE-PL se puede traducir de forma lineal para obtener un problema que una herramienta de verificación puede tratar [BL05].

Los principales comandos[1] de BOOGIE-PL son:

- **assume** f

  Este comando indica que todas las trazas del programa satisfacen f.

- **assert** f

  Este comando indica que todas las trazas del programa *deberían* satisfacer f. Es utilizado para verificar que el programa original cumple con todos sus contratos.

- $c_1 \,\square\, c_2$

  Se trata de la decisión no determinística entre los comandos c_1 y c_2. Una traza puede seguir cualquiera de los dos caminos.

- $c_1\ c_2$

  El secuenciamiento de los comandos c_1 y c_2.

[1] Llamaremos *comando* a cualquier instrucción de un lenguaje intermedio cuyo fin es la verificación de programas.

- havoc(v)

  Este comando asigna de forma no determinística un valor cualquiera a la variable v.

La herramienta Spec# traduce las asignaciones siguiendo un esquema *dynamic single assignment* o *DSA*. El mismo utiliza encarnaciones, las cuales pueden entenderse como fotografías del valor que tiene una variable en un determinado punto del programa. Cada vez que se asigna un valor a una variable se crea una nueva encarnación para la misma.

Por ejemplo, el siguiente fragmento de un programa:

$$a = 15;$$
$$a = a + 10;$$
$$a = a * 25;$$

Se traduce de la siguiente forma:

$$\textbf{assume } a_0 = 15$$
$$\textbf{assume } a_1 = a_0 + 10$$
$$\textbf{assume } a_2 = a_1 * 25$$

Un condicional de la siguiente forma:

$$\textbf{if}(g)\ s_1\ \textbf{else}\ s_2$$

Se traduce del siguiente modo:

$$(\textbf{assume } g\ c_1)\ \square\ (\textbf{assume } \neg g\ c_2)$$

Donde c_1 y c_2 son las traducciones de s_1 y s_2 respectivamente.

Un ciclo:

$$\textbf{while}(i > 0)\ \textbf{invariant}(i >= 0)\ i--;$$

Se traduce de la siguiente forma:

$$\textbf{assert } i >= 0$$
$$\textbf{assume } i_c >= 0$$
$$(\textbf{assume } i_c > 0\ \textbf{assume } i'_c = i_c - 1\ \textbf{assert } i'_c >= 0\ \textbf{assume } false)\ \square\ \textbf{assume } i_c <= 0$$

En general un ciclo se traduce utilizando una encarnación fresca para cada una de las variables que es modificada por el mismo. Por otra parte se utiliza el operador de decisión no determinística para describir que en una traza la ejecución puede realizar una iteración o puede dejar de iterar. En caso de producirse una iteración genérica, la misma debe preservar el invariante sobre las encarnaciones de las variables al finalizar.

Tal como mencionamos en el Capítulo 1, la herramienta Spec# no verifica terminación, por lo tanto no se preocupa por el decrecimiento de una función variante. Sólo comprueba que todas las ejecuciones que terminan sean correctas. Nuestra herramienta, en cambio, debe proveer una verificación más específica al garantizar que todas las ejecuciones terminan **y** son correctas.

Los llamados a métodos se traducen incorporando un *assert* de su precondición y un *assume* de su postcondición. Por el contrario las definiciones de métodos son traducidas incorporando antes del cuerpo un *assume* con la precondición y agregando al final un *assert* con la postcondición.

En general, los comandos havoc(v) que asigna no determinísticamente un valor a la variable v no son necesarios como parte del lenguaje intermedio. El esquema de traducción *DSA* puede simplemente incorporar una encarnación fresca para v sobre la cual no se conoce absoultamente nada.

Una vez traducido todo el programa se obtiene una gran secuencia de comandos de BOOGIE-PL que describe todas las posibles trazas que el mismo puede tomar. Si ninguno de los *asserts* de la traducción es violado, luego ninguna ejecución del programa original violará contrato alguno. Eso significa que todos los contratos explícitos (tales como postcondiciones e invariantes) serán cumplidos y adicionalmente no se harán accesos fuera de rango, divisiones por cero u otros requerimientos implícitos.

La ventaja de utilizar un lenguaje intermedio de las características de BOOGIE-PL, cuyos comandos no tienen efectos secundarios y utiliza un mecanismo de *DSA* es que se evita el problema de la explosión exponencial. Esto se debe a que existe una forma en la que se puede calcular la *weakest precondition* de una secuencia de comandos de forma lineal [BL05]. Se trata básicamente de seguir las siguientes reglas:

$$\begin{aligned} \mathrm{wp}(\textbf{assume } p,\ q) &\stackrel{\text{def}}{=} p \Rightarrow q \\ \mathrm{wp}(\textbf{assert } p,\ q) &\stackrel{\text{def}}{=} p \wedge q \\ \mathrm{wp}(c_1 \,\square\, c_2,\ q) &\stackrel{\text{def}}{=} \mathrm{wp}(c_1,\ q) \ \wedge\ \mathrm{wp}(c_2,\ q) \\ \mathrm{wp}(c_1\ c_2,\ q) &\stackrel{\text{def}}{=} \mathrm{wp}(c_1,\ \mathrm{wp}(c_2,\ q)) \end{aligned}$$

La *weakest precondition* de la traducción del programa es una fórmula cuya validez lógica implica que ningún *assert* es violado. Por ende para ver que un programa no viola ninguno de sus contratos alcanza con demostrar que esta fórmula es teorema bajo las teorías de los arreglos, enteros, reales, etc.

En el caso de SPEC# este problema es resuelto utilizando un demostrador automático llamado Z3, el cual pertenece a una clase de programas denominados *demostradores de satisfactibilidad módulo teorías* o *SMT solvers*. Se trata de herramientas muy difundidas en el mundo de la verificación de *hardware*, aunque crecientemente se las ha encontrado muy útiles para verificar *software* [BdMS05].

En el caso general, una instancia del problema de la satisfactibilidad módulo teorías puede ser vista como una fórmula en lógica de primer orden en la cual ciertos predicados y símbolos de función tienen una interpretación basada en una serie de teorías de primer orden con igualdad como por ejemplo los enteros, los reales, los arreglos, etc.

Un *SMT solver* abstrae los predicados y los reemplaza por variables booleanas. De esta forma el problema se convierte en una instancia que puede ser resuelta por un *SAT solver*. Una vez obtenidos cuáles de los predicados abstraídos deben ser verdaderos para satisfacer la fórmula, el *SMT solver* procede a ver si tal combinación es consistente. Para ello utiliza módulos que interpretan cada una de las teorías que el *solver* incorpora. Combinando los diversos módulos establece si los predicados que deben ser verdaderos suponen una contradicción. De ser así se busca otra combinación de predicados, caso contrario la fórmula es satisfacible.

Las expresiones que este tipo de herramientas atacan involucran teorías que son, muchas veces, indecidibles (como por ejemplo aritmética de Peano) y por ende se debe contemplar "desconocido" como una respuesta posible. En definitiva la salida puede tomar una de tres formas:

1. La herramienta pudo demostrar que la fórmula es satisfacible.
2. La herramienta pudo demostrar que la fórmula es insatisfacible.
3. La herramienta no pudo demostrar ninguno de los dos casos anteriores.

Debido a que se busca probar que la *wp* de la traducción del programa es válida, se le pregunta a Z3 si su negación es insatisfacible. En caso de que lo sea, el programa no viola ninguno de sus contratos. En caso de que no lo sea, Spec# utiliza diversos mecanismos para rastrear qué parte de la fórmula no pudo ser probada y relacionar dicho problema con un punto del programa para notificar al programador.

Un problema de este enfoque "monolítico" es que no permite ser mejorado en cuanto a la posibilidad de incorporar diversos demostradores. Incluso atacando la única fórmula con diversos *SMT solvers* podría suceder que la misma no sea demostrable ya que diversos fragmentos de la misma hacen fracasar a distintos demostradores.

En definitiva, este enfoque fuerza que cada *solver* sepa manejar todas las teorías mencionadas en la fórmula. Mientras que en la práctica, en general, cada demostrador maneja sólo algunas de ellas.

El resto de este capítulo se divide en tres partes. En la primera de ellas presentamos un enfoque alternativo de traducción y posterior verificación. El mismo explota las características de nuestro lenguaje para permitir la combinación modular de diversos *SMT solvers* sobre fragmentos a verificar.

En la segunda parte brindamos una breve explicación acerca del prototipo de herramienta que implementamos como parte de este trabajo. Comentamos sus principales características y marcamos algunas consideraciones sobre su sintaxis concreta.

En la tercera y última parte analizamos el comportamiento del prototipo que implementamos. Se prueba su desempeño sobre algunos ejemplos que permiten apreciar sus capacidades y limitaciones.

5.1. Un enfoque modular para la verificación

La verificación de programas a través de un lenguaje intermedio tiene la ventaja de evitar la explosión combinatoria que se da al calcular las postcondiciones, tal como se vio al comienzo de este capítulo. Sin embargo el enfoque monolítico adoptado por Spec# genera una única *wp* para todo un programa y luego la ataca con un único *SMT solver*.

Como primera mejora pensamos en utilizar diversos demostradores para atacar dicha fórmula. Esto en principio podría ser positivo, ya que existen diversos *SMT solvers* desarrollados por universidades, grupos de investigación o empresas. Cada uno de ellos tiene implementadas distintas reglas que los hacen comportarse mejor o peor que el resto según la fórmula que se esté probando. Existen problemas que son resueltos muy rápidamente por alguna de estas herramientas, mientras que otras no pueden o se les hace muy difícil tratarlos.

A modo de ejemplo, suponer que se cuenta con un demostrador que funciona bien con fórmulas que prediquen sobre arreglos y se tiene otro demostrador que funciona bien con fórmulas aritméticas. En ese caso se puede utilizar el demostrador adecuado según el problema que presente la fórmula a tratar. En caso de que una fórmula contenga problemas de arreglos y de aritmética no habría forma de tratarla con ninguno de los dos demostradores. Sin embargo, con una partición conveniente de la misma, se podría alimentar a cada uno de los demostradores con el fragmento sobre el que trabaja mejor.

Con esta intuición en mente, decidimos que valía la pena buscar un enfoque que permita encarar la verificación de forma gradual. Al atacar la fórmula parte por parte se puede combinar la potencia de diversos demostradores aprovechando los puntos fuertes de cada uno de ellos.

Suponiendo que se pudiera partir el problema de la verificación en pequeñas instancias de verificación más pequeñas, luego se puede utilizar un oráculo compuesto por diversos *SMT solvers* para atacar cada una de ellas. El oráculo puede ser visto como una interfaz abstracta hacia un orquestador que le realiza una pregunta a cada uno de los *solvers*. Ante la primer respuesta definitoria –o sea, distinta de

"desconocido"– se detiene. En caso de exceder cierto límite de tiempo o si ninguna herramienta da una respuesta definitoria, el oráculo responde "desconocido". Esta estrategia puede observarse gráficamente en la Figura 5.1.

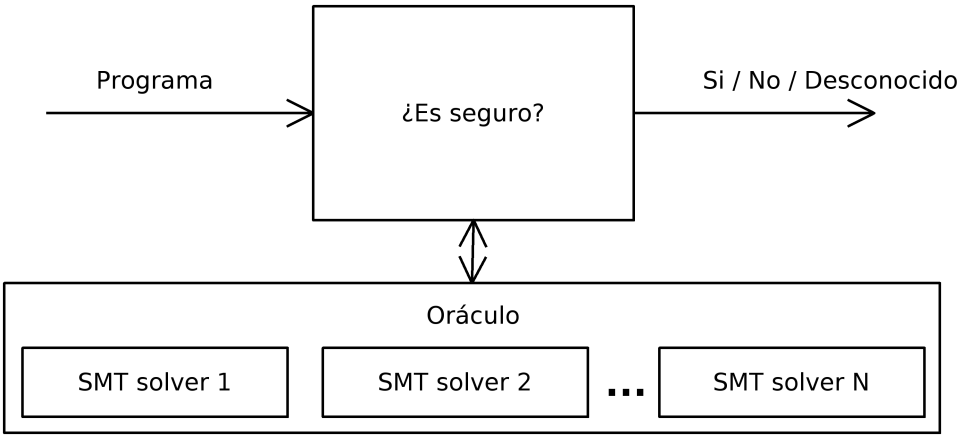

Figura 5.1: Una combinación de diversos *SMT solvers* como oráculo.

En los siguientes apartados proponemos un lenguaje intermedio que permite atacar de forma modular el problema de la verificación. Asimismo, detallamos cómo se traducen las metasentencias directamente al lenguaje intermedio. Finalmente presentamos una serie de reducciones que permiten incrementar aún más la modularidad.

5.1.1. Lenguaje intermedio modular

Para poder partir el problema de la verificación en varios problemas más pequeños es necesario replantearse la forma que tiene el lenguaje intermedio. Nuestra intuición es que gran parte de la dificultad de la demostración puede ser resuelta mediante lemas auxiliares, tales como los que se plantean para probar la corrección y terminación de un ciclo.

Para eso presentamos un lenguaje intermedio que en vez de describir las posibles trazas de un programa, describe la estructura que debería tener una demostración de que el mismo termina y es correcto respecto de su especificación. Este lenguaje intermedio cuenta con los siguientes comandos:

- **assume** p

  A partir de este punto se asume en la demostración que la expresión booleana p es válida.

- **verify** q

  Con toda la información con la que se cuenta hasta este punto debe poder verificarse que la expresión booleana q es válida.

- c_1 c_2

  Un esquema de demostración compuesto por dos subesquemas secuenciados c_1 y c_2.

- **case** g **in** c_1 **and** c_2

  Un esquema de demostración que asume g para llevar a cabo el esquema de demostración c_1. Luego asume $\neg g$ para llevar a cabo c_2. Observar que no se trata de seguir un plan de demostración

ú otro de acuerdo al valor de g. Por el contrario, se siguen ambos caminos, pero con asunciones distintas.

- **lemma** c

  Un lema dado por el esquema de demostración c. Todo lo asumido dentro del lema es olvidado luego.

- **skip**

  Un esquema de demostración vacío.

A continuación presentamos una traducción que toma una sentencia y retorna un plan de demostración de corrección y terminación. Se trata de proveer un plan de ejecución del chequeo de su semántica abstracta, siguiendo las reglas presentadas en el Capítulo 3.

Tal traducción se realiza siguiendo un esquema *DSA*, por lo tanto hacemos uso de un mapeo de variables a encarnaciones de variables.

Definición 5.1 (Traducción a comandos del lenguaje intermedio)**.**
Sea $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ y sea D un mapeo de variables a encarnaciones de variables. Si $\text{cmd}(s,\ D) = (c, D')$ luego se dirá que c es el comando que se corresponde a la sentencia s y D' es el mapeo de variables a encarnaciones luego de la sentencia.

$$\text{cmd}(v \leftarrow i,\ D) \stackrel{\text{def}}{=} \Big(\textbf{verify}\ \text{safe}^{\text{I}}(D(i))\ \textbf{assume}\ v' = D(i), \quad D\{v \mapsto v'\}\Big)$$
$$\text{cmd}(v \leftarrow A,\ D) \stackrel{\text{def}}{=} \Big(\textbf{verify}\ \text{safe}^{\text{A}}(D(A))\ \textbf{assume}\ v' = D(A), \quad D\{v \mapsto v'\}\Big)$$

Se traduce la parte derecha según el mapeo y se verifica que sea una expresión segura[2]. Luego se asume el valor para una nueva encarnación fresca v' de la variable v y se actualiza el mapeo.

$$\text{cmd}(\textbf{int}\ v \leftarrow i,\ D) \stackrel{\text{def}}{=} \Big(\textbf{verify}\ \text{safe}^{\text{I}}(D(i))\ \textbf{assume}\ v_0 = D(i), \quad D\{v \mapsto v_0\}\Big)$$
$$\text{cmd}(\textbf{array}\ v \leftarrow A,\ D) \stackrel{\text{def}}{=} \Big(\textbf{verify}\ \text{safe}^{\text{A}}(D(A))\ \textbf{assume}\ v_0 = D(A), \quad D\{v \mapsto v_0\}\Big)$$

Similar a los casos de asignación pero se parte desde la encarnación v_0, la cual es agregada al mapeo.

$$\text{cmd}(\textbf{skip},\ D) \stackrel{\text{def}}{=} \Big(\textbf{skip},\ D\Big)$$

La sentencia vacía y el mapeo permanecen inalterados.

$$\text{cmd}(s_1\ s_2,\ D) \stackrel{\text{def}}{=} \Big(c_1\ c_2, \quad D''\Big) \quad \text{donde } \text{cmd}(s_1,\ D) = (c, D'),\ \text{cmd}(s_2,\ D') = D''$$

$$\begin{aligned}\text{cmd}(\textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi},\ D) \stackrel{\text{def}}{=} \Big(&\textbf{verify}\ \text{safe}^{\text{B}}(D(g))\\ &\textbf{case}\ D(g)\ \textbf{in}\\ &\quad c_1\ \textbf{assume}\ V' = D_1(V)\\ &\textbf{and}\\ &\quad c_2\ \textbf{assume}\ V' = D_2(V)\\ &,\ D\{V \mapsto V'\}\Big)\end{aligned}$$

[2]Observar que se comete un abuso de notación al aplicar un mapeo D directamente sobre la expresión i (ó A, según la regla). El concepto de mapeo es extendido naturalmente a expresiones.

Donde $\text{cmd}(s_1,\ D) = (c_1,\ D_1)$ y $\text{cmd}(s_2,\ D) = (c_2,\ D_2)$. Se utiliza V' para dar encarnaciones nuevas a todas las variables modificadas por alguna de las dos ramas[3], identificado por el conjunto V, definido de la siguiente forma:

$$V \stackrel{\text{def}}{=} \{v \mid D_1(v) \neq D_2(v)\}$$

Ante un condicional aparece el requisito de que la guarda sea una expresión booleana segura. Por otra parte se genera un comando condicional que al final de cada rama tiene un *binding* entre el último valor de las variables en la misma con respecto al valor de las variables luego del condicional según el mapeo resultante.

$$\begin{aligned}\text{cmd}(\textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od},\ D) \stackrel{\text{def}}{=} \Big(&\textbf{verify } \text{safe}^{\text{B}}(D(inv) \wedge D(g)) \ \wedge \ \text{safe}^{\text{I}}(D(var))\\ &\textbf{verify } D(inv)\\ &\textbf{lemma}\\ &\quad \textbf{assume } M(inv) \wedge M(g)\\ &\quad \textbf{verify } \text{safe}^{\text{B}}(M(inv) \wedge M(g)) \ \wedge \ \text{safe}^{\text{I}}(M(var))\\ &\quad \textbf{verify } M(var) > \underline{0}\\ &\quad c\\ &\quad \textbf{verify } M'(inv)\\ &\quad \textbf{verify } M'(var) < M(var)\\ &\textbf{assume } D'(inv) \wedge \neg D'(g)\\ &, D'\Big)\end{aligned}$$

Donde $M = D\{V \mapsto V'\}$ con $V = \text{modVars}(s)$. Por otra parte $\text{cmd}(s,\ M) = (c, M')$. Y finalmente $D' = D\{V \mapsto V''\}$.

Se asume como conocido el invariante sobre un mapeo de variables genérico y se incorpora un *verify* por cada una de las premisas de la regla para la semántica abstracta.

El mapeo de variables resultante de la traducción de un ciclo únicamente tiene encarnaciones nuevas para aquellas variables que fueron potencialmente modificadas por una iteración genérica. El resto de las variables conserva sus encarnaciones previas al ciclo por ende toda la información que se conocía sobre ellas es preservada. De esta manera se está trabajando como si se hubiera fortalecido el invariante del ciclo ampliándolo para preservar la información conocida sobre las variables que el ciclo no modifica, tal como se presentó en la Subsección 3.5.2.

$$\begin{aligned}\text{cmd}(\textbf{call } proc(cp_1, \ldots, cp_k),\ D) \stackrel{\text{def}}{=} \Big(&\textbf{verify } D(\text{P}_{proc}\lfloor p_i \mapsto cp_i \rfloor)\\ &\textbf{assume } D'(\text{Q}_{proc}\lfloor p_i \mapsto cp_i \rfloor)\\ &, D'\Big)\end{aligned}$$

Donde $D' = D\{p_i \mapsto p_i'\}$

[3]Se comete un abuso de notación al realizar un *assume* sobre muchas variables a la vez. Esto puede verse como *syntactic sugar* que se reemplaza por un *assume* para cada una de las variables en V. De forma similar se extiende naturalmente el concepto de mapeo de variables aplicado a un conjunto V.

En el caso de un llamado a procedimiento se verifica que valga la precondición del mismo. Para eso se la instancia en los parámetros concretos y con las encarnaciones de las variables determinadas por D.

Para la salida del procedimiento se cuenta con nuevas encarnaciones para cada parámetro concreto, tal como lo determina la definición de D'. Sobre estas nuevas encarnaciones se asume que vale la postcondición del procedimiento llamado, a la cual se le sustituyen los parámetros formales por los concretos.

Una vez obtenido el plan de demostración brindado por la traducción de un programa se puede proceder a "ejecutar" el mismo. Esto se realiza siguiendo el siguiente algoritmo FOLLOW:

FOLLOW(*in* c : comando, *inout* A : conjunto de hechos conocidos)

según c sea:

- **assume** p

  $A \leftarrow A \cup \{p\}$

- **verify** q

  Se le pregunta al oráculo si $A \models q$.

  Si no puede resolver la relación de fuerza se **detiene** el proceso de verificación.

  Si puede resolver la relación de fuerza se **prosigue** y además:

  $A \leftarrow A \cup \{q\}$

- **skip**

  No se realiza acción alguna.

- c_1 c_2

  FOLLOW(c_1, A)

  FOLLOW(c_2, A)

- **case** g **in** c_1 **and** c_2

  $A \leftarrow A \cup \{g\}$

  FOLLOW(c_1, A)

  $A \leftarrow (A \setminus \{g\}) \cup \{\neg g\}$

  FOLLOW(c_2, A)

  $A \leftarrow A \setminus \{\neg g\}$

- **lemma** c

  $A_{copy} \leftarrow A$

  FOLLOW(c, A_{copy})

De esta forma se cuenta con un procedimiento que utiliza el oráculo de manera modular. Cada una de las hipótesis de las reglas es preguntada de forma independiente y en caso de ser garantizada por algún *SMT solver* es agregada como hipótesis para que los demás demostradores puedan asumir que es válida al intentar resolver futuras preguntas.

En caso de que alguna falle se puede reportar el error de manera precisa, para que de esta forma el programador pueda tener conocimiento de qué es lo que impide demostrar que su programa es seguro. Sin modificar sustancialmente el lenguaje intermedio puede incorporarse información adicional en el comando *verify* que permita informar en qué línea se produjo el problema y de qué error se trata ("no se pudo probar el invariante", "acceso fuera de rango", etc.)

5.1.2. Reemplazo de metasentencias

El objetivo de este apartado es indicar de qué manera son tratadas cada una de las metasentencias introducidas en el capítulo anterior. Se presenta la forma en la que las mismas son traducidas al lenguaje intermedio. Dicho de otra forma, se presenta la forma en la que se obtiene un plan de demostración para las mismas.

Traducción de la metasentencia *map*

La metasentencia **map** s **in** $A[..\ i\ ..]$ se convierte en un ciclo que aplica s a cada uno de los elementos de A. Tanto el invariante como el variante de la traducción presentada en el capítulo anterior reflejan este hecho.

Una posibilidad es realizar primero una traducción a sentencias y luego aplicarles la traducción a las mismas para obtener comandos del lenguaje intermedio. Este enfoque, si bien es correcto, generaría una serie de comandos *verify* que serían innecesarios.

Por ejemplo, la condición que exige que el variante $|A| - i$ decrezca en cada iteración es obligatoriamente cumplida. Esto se debe a que s no modifica el valor de i y en cada paso se realiza $i \leftarrow i + \underline{1}$.

De manera similar, verificar que el invariante se preserva es innecesario ya que el mismo establece que luego de una iteración hay un elemento más al que se ha visitado; la ejecución de s garantiza este hecho.

Obviando las verificaciones innecesarias, se puede traducir la metasentencia directamente sobre el lenguaje intermedio de comandos de la siguiente manera:

$$\begin{aligned}\text{cmd}(\textbf{map } s \textbf{ in } A[..\ i\ ..],\ D) \stackrel{\text{def}}{=} \Big(&\textbf{lemma}\\ &\quad \textbf{assume } 0 \leq i < |A|\\ &\quad c\\ &\textbf{assume } \forall k\ /\ 0 \leq k < |A| :\ post_s\lfloor i \mapsto k,\ A \mapsto A'\rfloor\\ &,\ D\{A \mapsto A'\}\Big)\end{aligned}$$

Donde $\text{cmd}(s,\ D\{i \mapsto i_0\}) = (c, M)$

Notar que el mapeo de variables M correspondiente a una iteración genérica del cuerpo s no es relevante ya que sólo serviría para ver que el invariante se preserva. Por construcción sabemos que esto siempre ocurre y por ende dicha verificación es obviada. De todos modos la demostración debe incluir los comandos correspondientes al cuerpo s (representados por c) para poder verificar que una iteración genérica no viola ningún contrato.

Traducción de la metasentencia *find*

De manera similar, traducimos la metasentencia *find* directamente sobre el lenguaje intermedio. De esta forma evitamos agregar las verificaciones de que el invariante vale al comienzo y se preserva, el variante es positivo y decrece; todas estas verificaciones son innecesarias ya que la forma en que se construyen hace que siempre sean correctas.

La traducción para un *find*, evitando realizar las verificaciones que siempre serían correctas, es:

$$
\begin{aligned}
\text{cmd}(i \leftarrow \textbf{find } x \textbf{ as } s \textbf{ in } A[..\ j\ ..],\ D) \stackrel{\text{def}}{=} \Big(&\textbf{lemma} \\
&\quad \textbf{assume } 0 \leq j < |A| \\
&\quad c \\
&\textbf{assume} \\
&\quad (i' \neq -1 \Rightarrow \forall k\ /\ 0 \leq k < i' :\ \neg post_s\lfloor j \mapsto k \rfloor \wedge post_s\lfloor j \mapsto i' \rfloor) \\
&\quad \wedge\ (i' = -1 \Rightarrow \forall k\ /\ 0 \leq k < |A| :\ \neg post_s\lfloor j \mapsto k \rfloor) \\
&,\ D\{i \mapsto i'\}\Big)
\end{aligned}
$$

Donde $\text{cmd}(s,\ D\{j \mapsto j_0\}) = (c, M)$

De forma similar a la traducción de la metasentencia *map*, en este caso no se utiliza el mapeo M ya que se conoce por construcción que el invariante inferido es preservado.

Traducción de la metasentencia *for*

El caso de la metasentencia *for* es ligeramente distinto a las dos traducciones mencionadas. Tal como se presentó en el capítulo anterior, para traducir una metasentencia *for* es necesario contar con un candidato a invariante. El mismo no obedece a ningún patrón particular, por el contrario es obtenido a partir de una transformación de la postcondición del *for*, la cual llamamos Q_c.

Tal como mencionamos en la Sección 4.3, para obtener una postcondición Q_c nuestro enfoque es comenzar desde el final del procedimiento tomando la postcondición del contrato. A partir de allí se "sube" aplicando el cálculo de precondiciones hasta llegar al punto final de la metasentencia *for* que se desea traducir.

Sin embargo, el cálculo de precondiciones presenta el mismo problema de explosión exponencial que el cálculo de postcondiciones. Si se siguiera directamente la definición dada en el capítulo anterior se obtendría un invariante exponencialmente grande respecto de la distancia que separa la metasentencia *for* del final de procedimiento.

El lenguaje intermedio se introdujo como un medio para poder obtener precondiciones y postcondiciones de forma eficiente. Por lo tanto si se pudiera trabajar con la traducción al lenguaje intermedio de la cola del procedimiento (es decir, el fragmento que va desde el final del *for* hasta el final del procedimiento) luego se podría calcular su precondición de forma eficiente.

Se produce una situación de dependencia circular. Para poder traducir un *for* es necesario su invariante. Su invariante depende de la postcondición deseada Q_c. Esta postcondición depende de la traducción de la cola del procedimiento. La traducción de la cola del procedimiento no puede ser obtenida hasta no traducir el *for*.

Presentamos a continuación un ejemplo que muestra la forma en la que resolvimos este problema. Contamos con el siguiente procedimiento que calcula el mínimo de un arreglo:

```
1:  easyMin(A, m)
2:  :? |A| > 0
3:  :! ∀k / 0 ≤ k < |A| :  m ≤ A[k]
4:  {
5:      int min ← A[0]
6:      for i from 1 to |A| do
7:          if A[i] < min then
8:              min ← A[i]
9:          fi
10:     od
11:     m ← min
12: }
```

El mínimo de un arreglo usando *for*

Para poder traducir la metasentencia *for* que aparece en el procedimiento es necesario obtener la postcondición deseada. Sin embargo, en una primera pasada no se traduce el *for*. Lo que obtenemos no es un plan de demostración, sino un *protoplan de demostración* en el cual todavía permanecen elementos ajenos al lenguaje intermedio.

En el caso del ejemplo se obtendría:

assume $|A| > 0$
assume $min_0 = A[0]$
for i **in** $[1, |A|)$
    case $A[i] < min_f$ **in assume** $min'_f = A[i]$ **and skip**
assume $m_0 = min_l$
verify $\forall k \ / \ 0 \leq k < |A| : \ m_0 \leq A[k]$

Observar la tercera línea, en la cual aparece un elemento *for*, ajeno al lenguaje intermedio de planes de demostración. Se trata de un *protoplan de demostración*.

A partir de este punto se puede realizar una segunda pasada, en la cual se parte desde el final del protoplan y se recorre "hacia arriba" obteniendo precondiciones hasta el punto en el que se encuentra algún elemento ajeno al lenguaje intermedio.

En el caso del ejemplo se llega hasta el *for* habiendo acumulado:

$$\forall k \ / \ 0 \leq k < |A| : \ min_l \leq A[k]$$

Observar que con respecto al último *verify*, la encarnación m_0 es reemplazada por min_l gracias al comando **assume** $m_0 = min_l$.

Utilizando esta postcondición deseada para el *for* Q_c se puede obtener el siguiente invariante:

$$\forall k \ / \ 0 \leq k < i : \ min \leq A[k]$$

Una vez que se tiene el invariante se puede convertir el elemento *for* ajeno al lenguaje intermedio. Obtenidos los comandos correspondientes a tal traducción se puede calcular una precondición de los mismos y proseguir "hacia arriba". Se repite este procedimiento hasta quitar todos los elementos ajenos al lenguaje intermedio para obtener un plan de demostración que pueda ser verificado utilizando el algoritmo FOLLOW.

En el ejemplo, el elemento *for* ajeno al plan de demostración se traduce para obtener:

assume $|A| > \underline{0}$
assume $min_0 = A[\underline{0}]$
verify $\forall k \ / \ \underline{0} \leq k < \underline{1} : \ min_0 \leq A[k]$
lemma
    assume $\forall k \ / \ \underline{0} \leq k < i : \ min_f \leq A[k]$
    assume $0 \leq i < |A|$
    case $A[i] < min_f$ **in assume** $min'_f = A[i]$ **and skip**
    verify $\forall k \ / \ \underline{0} \leq k < i + \underline{1} : \ min'_f \leq A[k]$
assume $\forall k \ / \ \underline{0} \leq k < |A| : \ min_l \leq A[k]$
assume $m_0 = min_l$
verify $\forall k \ / \ \underline{0} \leq k < |A| : \ m_0 \leq A[k]$

En el caso general, una primera pasada trataría a una metasentencia *for* de la siguiente forma para obtener un protoplan de demostración:

$$\mathrm{tr}^{\mathrm{S}}(\textbf{for } i \textbf{ from } l \textbf{ to } h \textbf{ do } s \textbf{ od}, \ D) \stackrel{\text{def}}{=} \Big(\textbf{for } i \textbf{ in } [l, h) \ c, \ D'\Big)$$

Donde $\mathrm{tr}^{\mathrm{S}}(s, \ D\{i \mapsto i_0\}) = (c, D')$.

Para la segunda pasada, tal como mencionamos, se requiere extender el concepto de cálculo de precondición para un comando del lenguaje intermedio. Esto se realiza de forma muy similar al cálculo de *wp* presentado al comienzo de este capítulo:

$$\begin{aligned}
\mathrm{pre}(\textbf{assume } p, \ q) &\stackrel{\text{def}}{=} p \Rightarrow q \\
\mathrm{pre}(\textbf{verify } q', \ q) &\stackrel{\text{def}}{=} q' \wedge q \\
\mathrm{pre}(c_1 \ c_2, \ q) &\stackrel{\text{def}}{=} \mathrm{pre}(c_1, \ \mathrm{pre}(c_2, \ q)) \\
\mathrm{pre}(\textbf{case } g \textbf{ in } c_1 \textbf{ and } c_2, \ q) &\stackrel{\text{def}}{=} (g \Rightarrow \mathrm{pre}(c_1, \ q)) \ \wedge \ (\neg g \Rightarrow \mathrm{pre}(c_2, \ q)) \\
\mathrm{pre}(\textbf{lemma } c, \ q) &\stackrel{\text{def}}{=} \mathrm{pre}(c, \ q) \\
\mathrm{pre}(\textbf{skip}, \ q) &\stackrel{\text{def}}{=} q
\end{aligned}$$

Teniendo en cuenta que el lenguaje intermedio no es más que un plan de demostración, el cálculo de precondiciones puede entenderse como obtener una hipótesis que sirva para seguir el plan correctamente, probando todos sus ítems de verificación.

Observar lo simple que resulta este cálculo con respecto al presentado en el Capítulo 3. Esto se debe a que el programa a esta altura ya sufrió una transformación y ahora las condiciones de expresiones seguras se encuentran codificadas como comandos *verify*. Por otra parte, ninguno de los comandos del lenguaje tiene efectos secundarios; únicamente predican sobre hechos conocidos sobre alguna encarnación de las variables.

Con el cálculo de precondiciones para los comandos definido, puede realizarse la segunda pasada de la traducción de las metasentencias *for*. En este caso se parte desde la postcondición del procedimiento y se recorren los comandos "de abajo hacia arriba" acumulando una precondición para los mismos. Si

en algún punto aparece un elemento ajeno al lenguaje intermedio se lo traduce de la siguiente forma:

$$\begin{array}{rcl}
\textbf{for } i \textbf{ in } [l,h)\ c & \leadsto & \textbf{verify } I\lfloor i \mapsto l \rfloor \\
& & \textbf{lemma} \\
& & \quad \textbf{assume } I \\
& & \quad \textbf{assume } 0 \leq i < h \\
& & \quad c \\
& & \quad \textbf{verify } I\lfloor i \mapsto i + \underline{1} \rfloor \\
& & \textbf{assume } Q_c
\end{array}$$

Donde Q_c es la precondición que se vino acumulando e I se obtiene de Q_c según:

$$I \stackrel{\text{def}}{=} Q_c\lfloor h \mapsto i \rfloor$$

Observar que en la traducción anterior aparece I en reiteradas ocasiones. En realidad, las encarnaciones de las variables que podrían aparecer en I son distintas en cada caso. Estos mapeos de variables a encarnaciones que se utilizan pueden ser almacenados como parte del comando *for* al hacer la primera pasada. Los detalles técnicos de cómo se manejan estos mapeos fueron obviados porque no son esenciales para el funcionamiento de la técnica presentada.

Tal como se mencionó anteriormente, al finalizar la segunda pasada se obtiene un plan de demostración con comandos del lenguaje intermedio libres de comandos *for*. A partir de ese punto se realiza la verificación tal como se venía haciendo, utilizando el algoritmo FOLLOW.

5.1.3. Simplificar y partir fórmulas

Aún utilizando una serie de *SMT solvers* para atacar los problemas de forma modular, puede suceder que la complejidad de las expresiones que se les arroja a los mismos sea excesiva. En [DFS06] se investigan simplificaciones de fórmulas, de baja complejidad, con las que obtienen buenos resultados empíricos. Algunos ejemplos para reducciones de booleanos son:

$$\begin{array}{rcl}
\neg\textbf{true} & \leadsto & \textbf{false} \\
\neg\textbf{false} & \leadsto & \textbf{true} \\
\textbf{true} \wedge b & \leadsto & b \\
\textbf{false} \wedge b & \leadsto & \textbf{false} \\
\textbf{true} \Rightarrow b & \leadsto & b \\
\textbf{false} \Rightarrow b & \leadsto & \textbf{true}
\end{array}$$

O reducciones de enteros tales como:

$$\begin{array}{rcll}
\underline{n} < \underline{m} & \leadsto & \textbf{true} & \text{si } n < m \\
\underline{n} = \underline{n} & \leadsto & \textbf{true} & \\
\underline{n} + \underline{m} & \leadsto & \underline{n+m} &
\end{array}$$

En dicho artículo se presentan ejemplos en donde se produce una mejora en los resultados, en cuanto a tiempo y cantidad de problemas atacables, al aplicar estas simplificaciones.

Como parte de este trabajo ideamos otro preproceso que se le puede realizar a las expresiones antes de enviarlas a un *solver*. La intuición es que una fórmula más corta es más sencilla de demostrar, por ende siempre que se pueda se intenta partirlas. Una expresión $b_1 \wedge b_2$ puede ser preguntada en etapas. Primero se pregunta b_1 y sólo si esta tiene éxito se pregunta b_2. En caso de que b_1 falle no es necesario preguntar por b_2 y el reporte de errores puede ser mucho más preciso y útil para el usuario a la hora de corregir el problema.

Las reglas que establecimos para partir las fórmulas son:

$$\begin{aligned}
\text{parts}(b_1 \wedge b_2) &\stackrel{\text{def}}{=} \text{parts}(b_1) \cup \text{parts}(b_2) \\
\forall\, x : \textsc{Int}\ (b) &\stackrel{\text{def}}{=} \{\forall\, x : \textsc{Int}\ (c) \mid c \in \text{parts}(b)\} \\
\forall\, x : \textsc{Array}\ (b) &\stackrel{\text{def}}{=} \{\forall\, x : \textsc{Array}\ (c) \mid c \in \text{parts}(b)\} \\
\text{parts}(b) &\stackrel{\text{def}}{=} \{b\} \qquad \text{en cualquier otro caso}
\end{aligned}$$

Al partir las fórmulas usando las reglas anteriores se pueden atacar problemas que antes eran irresolubles. Suponer que se tiene una fórmula de la forma:

$$F \stackrel{\text{def}}{=} A \wedge B$$

Donde A utiliza elementos de una teoría T_1 y B utiliza elementos de una teoría T_2. Si contamos con un demostrador D_1 que sólo entiende la teoría T_1 entonces no podrá atacar la fórmula F. Lo mismo pasaría con un demostrador D_2 que únicamente maneje la teoría T_2. En definitiva, ni siquiera la combinación de D_1 y D_2 podría atacar la fórmula F tal como está escrita.

Sin embargo:

$$\text{parts}(A \wedge B) = \{A, B\}$$

Por lo tanto la fórmula F puede ser resuelta usando el demostador D_1 para atacar A y el demostrador D_2 sobre B.

5.2. Prototipo de herramienta

Presentamos en esta sección un prototipo de herramienta que implementamos para poner a prueba las ideas desarrolladas en esta tesis. El mismo fue implementado utilizando el lenguaje de programación Java y consta de aproximadamente 12000 líneas de código.

La arquitectura básica del prototipo de herramienta puede apreciarse en la Figura 5.2. Se parsea el código del archivo de entrada y se genera un árbol de sintaxis abstracta para el programa en cuestión. Se transforma dicho árbol en la representación intermedia del lenguaje de comandos *assume/assert*. Luego se recorre tal representación intermedia de forma tal de eliminar todas las apariciones de comandos *for*.

En ese punto se cuenta con una serie de comandos del lenguaje intermedio y se procede a verificarlos utilizando el algoritmo follow. Para ello primero se los simplifica y particiona utilizando decenas de reglas de reducción sintácticas, tales como las mencionadas en la sección anterior. Se utiliza el oráculo, el cual combina los demostradores CVC3 [BB04], Z3 [BLM05] y Yices [DdM06].

El resultado de la verificación consta de una descripción de aquellas verificaciones que fallaron en caso de haber alguna. Se reporta la línea en donde se encuentra cada problema y una descripción detallada que indica de qué problema se trata. El hecho de haber fragmentado las fórmulas permite que el reporte de errores sea muy conciso, indicando qué parte de un invariante no pudo preservarse, o qué parte de una precondición de un procedimiento llamado no se cumple.

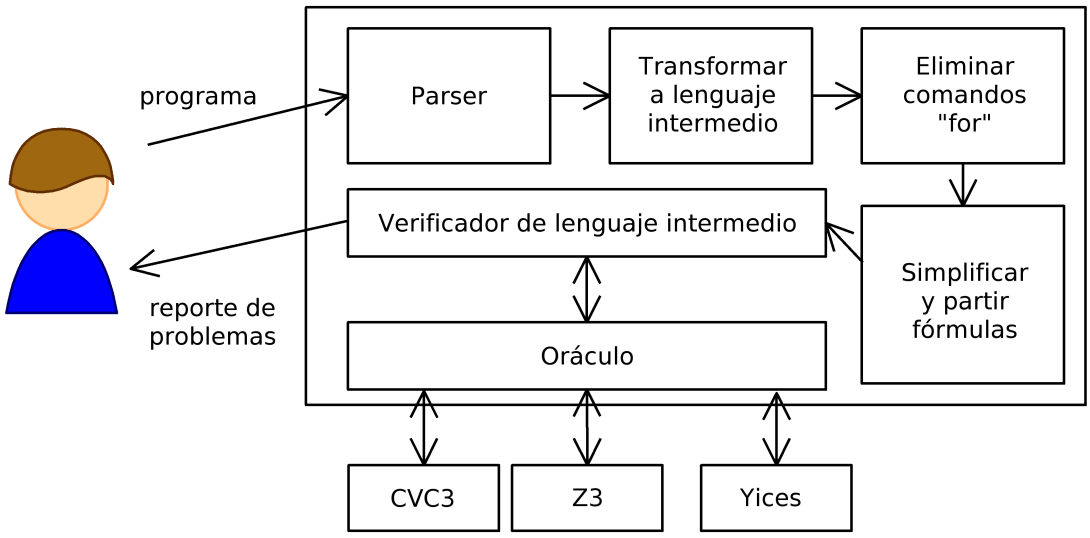

Figura 5.2: Arquitectura básica del prototipo de herramienta.

El prototipo de herramienta tiene una sintaxis concreta que presenta algunas sutiles diferencias con respecto a las definiciones formales presentadas en esta tesis. Mencionamos a continuación las diferencias más notorias:

- La sintaxis del lenguaje de programación es ligeramente distinta. Se utilizan llaves para abrir y cerrar bloques en cambio de los **fi**, **od**, etc.

  Por ejemplo, un condicional sería:

  $$\textbf{if } g \textbf{ then } \{ \ s_1 \ \} \textbf{ else } \{ \ s_2 \ \}$$

- Se permite introducir bloques en cualquier punto del programa. De esa forma se pueden crear múltiples niveles abitrarios de *scoping* de variables.

  Por ejemplo, el siguiente programa **no** sería válido:

  $$\{ \ \textbf{int } x \leftarrow \underline{10} \ \} \ \textbf{int } y \leftarrow x$$

  En este caso, luego de la segunda sentencia el valor de x es 20. La vieja variable x desaparece al cerrar el bloque.

- Se puede llamar a un procedmiento pasándole cualquier expresión como parámetro. Esto en realidad es *syntactic sugar* para crear una variable fresca, asignarle dicha expresión y llamar al procedimiento pasándole la variable.

  Por ejemplo:

  $$\textbf{call } inc(\underline{10})$$

  Es *syntactic sugar* para:

  $$\begin{array}{l} \{ \\ \quad \textbf{int } p_0 \leftarrow \underline{10} \\ \quad \textbf{call } inc(p_0) \\ \} \end{array}$$

- Las metasentencias *map* y *find* tienen parámetros adicionales que indican los límites inferior y superior de iteración. Esto permite recorrer un subarreglo, aumentando la expresividad de estos comandos. Adicionalmente, la metasentencia *find* permite recorrer desde atrás hacia adelante.

  La sintaxis concreta para *map* es:

  $$\textbf{map}\ e\ \textbf{in}\ A[l..i..h]\ \{s\}$$

  La variable e sirve como *binding* para referirse a $A[i]$ dentro de s. La variable i itera entre l y h inclusive y estas cotas deben estar en rango.

  La sintaxis concreta para *find* es:

  $$i\ \leftarrow\ \textbf{find first}\ v\ \textbf{in}\ A[l..j..h]\ \{s\}$$

  Se itera usando la variable j empezando en l y avanzando hasta llegar a h. Se deja en i el primer valor de j que haga que s asigne un valor distinto de 0 a la variable v. Reemplazando **first** por **last** se itera hacia atrás.

- Los tipos de los parámetros formales de un procedimiento deben ser especificados de manera manual sin embargo podría implementarse un algoritmo que los infiera.

  Mostramos el siguiente ejemplo para presentar la sintaxis concreta para definir procedimientos:

  $$name(p_1, p_2[], \ldots, p_k)\ :?\ pre\ :!\ post\ :*\ p_{i_1}, \ldots, p_{i_m}\ \{s\}$$

  Se sufija el nombre de un parámetro formal con [] en su definición para indicar que se trata de un parámetro de tipo arreglo. Por otra parte observar que la definición incluye una cláusula $:*$ que indica cuáles de los parámetros formales son modificables. Si un parámetro formal p no es declarado como modificable luego se verifica que nunca sea asignado en el cuerpo del procedimiento y se agrega $p = p$**@pre** a la postcondición.

Aparte de la sintaxis, la otra diferencia entre las definiciones formales presentadas hasta aquí y el prototipo implementado son los chequeos de expresiones seguras. En la versión con la que contamos al momento de la confección de este documento aún no se encuentra implementada la generación de planes de prueba que incluyan chequeos de expresiones seguras. La ausencia de esta característica no nos pareció crítica a la hora de desarrollar una herramienta que sirva como prueba de concepto para llevar a la práctica nuestras ideas.

Una característica del prototipo que es digna de ser remarcada es la utilización del patrón de diseño *visitor* [PJ98]. El mismo permite la definición de operaciones sobre objetos en una estructura de clases sin necesidad de alterar el código de las mismas. Tiene una forma de funcionamiento que podría compararse con la definición de funciones por *pattern matching* en el paradigma funcional.

Haciendo uso de la herencia y el patrón *visitor*, el prototipo permite ser fácilmente extendido con nuevas metasentencias, nuevos demostradores automáticos, etc.

5.3. Casos de prueba

Antes de introducir los casos de prueba, es preciso mencionar que por una cuestión de consistencia, en su presentación se optó por preservar la sintaxis con la que se trabajó toda la tesis.

El primer caso se trata de un programa que no hace uso de ninguna metasentencia, sin embargo es de interés por tratarse de un algoritmo de ordenamiento clásico. Los invariantes son complejos, con

cuantificaciones doblemente anidadas, por lo tanto se espera que se trate de un caso interesante de verificar.

```
1: bubbleSort(A)
2: :? |A| > 0 ∧                                                          ▷ arreglo no vacío
3:     ∀i1 / 0 ≤ i1 < |A| : ∀i2 / 0 ≤ i2 < |A| : i1 ≠ i2 ⇒ A[i1] ≠ A[i2]   ▷ sin repetidos
4: :! ∀k / 0 ≤ k < |A| − 1 : A[k] < A[k + 1] ∧                              ▷ ordenado
5:     ∀i1 / 0 ≤ i1 < |A| : ∃i2 / 0 ≤ i2 < |A| : A[i1] = A@pre[i2]         ▷ permutación
6: {
7:     int i ← |A| − 1
8:     while i > 0
9:         :?! 0 ≤ i ≤ |A| − 1 ∧
10:            ∀k / i ≤ k < |A| − 1 : A[k] ≤ A[k + 1] ∧                    ▷ parcialmente ordenado
11:            ∀n / 0 ≤ n ≤ i : ∀k / i < k < |A| : A[n] ≤ A[k] ∧           ▷ i separa mayores y menores
12:            ∀i1 / 0 ≤ i1 < |A| : ∀i2 / 0 ≤ i2 < |A| : i1 ≠ i2 ⇒ A[i1] ≠ A[i2] ∧   ▷ sin repetidos
13:            ∀i1 / 0 ≤ i1 < |A| : ∃i2 / 0 ≤ i2 < |A| : A[i1] = A@pre[i2]   ▷ permutación
14:        :# i
15:    do
16:        int j ← 0
17:        while j < i
18:            :?! 0 ≤ i ≤ |A| − 1 ∧
19:                ∀k / i ≤ k < |A| − 1 : A[k] ≤ A[k + 1] ∧
20:                ∀n / 0 ≤ n ≤ i : ∀k / i < k < |A| : A[n] ≤ A[k] ∧
21:                ∀i1 / 0 ≤ i1 < |A| : ∀i2 / 0 ≤ i2 < |A| : i1 ≠ i2 ⇒ A[i1] ≠ A[i2] ∧
22:                ∀i1 / 0 ≤ i1 < |A| : ∃i2 / 0 ≤ i2 < |A| : A[i1] = A@pre[i2] ∧   ▷ I de afuera
23:                0 ≤ j ≤ i ∧
24:                ∀k / 0 ≤ k < j : A[k] ≤ A[j]                          ▷ j-ésimo mayor a su izquierda
25:            :# i − j
26:        do
27:            if A[j] > A[j + 1] then
28:                int j1 ← j + 1
29:                call swap(A, j, j1)
30:            fi
31:            j ← j + 1
32:        od
33:        i ← i − 1
34:    od
35: }

36: swap(A, i, j)
37: :? 0 ≤ i < |A| ∧ 0 ≤ j < |A|
38: :! ∀k / 0 ≤ k < |A| : k ≠ i ∧ k ≠ j ⇒ A[k] = A@pre[k] ∧ A[i] = A@pre[j] ∧ A[j] = A@pre[i]
39: {
40:     int tmp ← A[i]
41:     A[i] ← j
42:     A[j] ← tmp
43: }
```

Algoritmo de ordenamiento *bubble sort*

Observar que se requiere que todos los elementos del arreglo sea distintos y se garantiza que el resultado es una permutación ordenada del arreglo original. Idealmente nos gustaría poder verificar un programa que contemple la aparición de repetidos, sin embargo se necesitarían predicados de especificación que puedan expresar la cantidad de apariciones para poder establecer adecuadamente que el resultado es una permutación.

El contrato del procedimiento principal no refleja detalles de implementación, por el contrario, especifica el comportamiento esperado. Por otra parte, los invariantes responden a la forma particular en la que implementamos el algoritmo de ordenamiento. Esta diferencia conceptual entre "qué" y "cómo" es resuelta gracias a la capacidad deductiva de los demostradores.

Este programa genera 45 preguntas que se le realizan al oráculo. La gran mayoría de ellas es respondida por el demostrador YICES, sin embargo una es resuelta únicamente por Z3. De no haber utilizado los demostradores en paralelo este sería un caso que no se podría atacar.

El tiempo de ejecución es de aproximadamente 4 segundos en una computadora PENTIUM IV de 2800 MHz con 1 GB de memoria RAM.

Presentamos a continuación un ejemplo que involucra la metasentencia *for* y anotaciones con aritmética no lineal. Esto supone un reto más que interesante ya que la gran mayoría de los demostradores automáticos no soportan este tipo de aritmética. Sin embargo, CVC3 tiene cierto soporte muy limitado de aritmética no lineal que permite resolver algunas cuestiones.

```
1: sumFirstN(n, s)
2: :? n > 0
3: :! s = n * (n − 1)/2 ∧ n = n@pre
4: {
5:     s ← 0
6:     for i from 1 to n do
7:         s ← s + i
8:     od
9: }
```

Suma de los primeros $n-1$ naturales

El invariante inferido en este caso es:

$$1 \leq i \leq n \ \wedge \ s = i * (i - 1) \ \wedge \ n = n@\mathbf{pre}$$

Este programa genera un total de 3 preguntas al oráculo, de las cuales 2 son respondidas por YICES y una de ellas, que representa la preservación del invariante y por ende contiene no linearidad, es respondida por CVC3.

El tiempo de ejecución en este caso es despreciable.

Por último, mostramos un ejemplo donde se pueden apreciar algunas de las limitaciones del enfoque de verifiación automática usando demostradores de teoremas. En este caso se trata de un procedimiento para determinar si un arreglo se encuentra ordenado.

```
1:  isSorted(A, b)
2:  :? |A| > 0
3:  :! A = A@pre ∧ (b ≠ 0 ⇔ ∀i / 0 ≤ i < |A| − 1 : A[i] ≤ A[i + 1])
4:  {
5:      b ← 1
6:      int k ← 0
7:      while k < |A| − 1
8:          :?! 0 ≤ k ≤ |A| − 1 ∧
9:              (b ≠ 0 ⇔ ∀i / 0 ≤ i ≤ k : ∀j / 0 ≤ j ≤ k : i ≤ j ⇒ A[i] ≤ A[j])
10:         :# |A| − k
11:     do
12:         if A[k] > A[k + 1] then
13:             b ← 0
14:         fi
15:         k ← k + 1
16:     od
17: }
```

Verificar si un arreglo se encuentra ordenado

Este ejemplo genera 9 preguntas al oráculo de las cuales 8 son respondidas afirmativamente. La pregunta restante corresponde a verificar que la segunda parte de la postcondición se cumple y esto es efectivamente cierto. Sin embargo ninguno de los *SMT solvers* que utilizamos en este trabajo fue suficientemente poderoso como para resolver que esta expresión:

$$\forall i \ / \ 0 \leq i \leq k : \ \forall j \ / \ 0 \leq j \leq k : \ i \leq j \ \Rightarrow \ A[i] \leq A[j] \ \wedge \ k = |A| - 1$$

Fuerza a esta otra:

$$\forall i \ / \ 0 \leq i < |A| - 1 : \ A[i] \leq A[i + 1]$$

Esta debilidad descansa en el hecho de que, por más que se haya avanzado mucho, los demostradores actuales dependen fuertemente de la estructura sintáctica de las expresiones con las que se trabaja. Si se hubiera escrito el invariante siguiendo el "estilo" de la postcondición, entonces no hubiera habido ningún problema.

En el ejemplo del *bubbleSort* las diferencias entre los invariantes y la postcondición eran resueltas con éxito y en este caso no. Esto pone al descubierto la dificultad de poder predecir en general si una herramienta de demostración podrá atacar con éxito un determinado problema.

Capítulo 6

Conclusiones y trabajo a futuro

"La música empieza donde se acaba el lenguaje."

E.T.A. Hoffmann

6.1. Conclusiones

Partimos en este trabajo de la hipótesis de que las herramientas de verificación automática más difundidas en la actualidad tienen ciertos inconvenientes propios de la forma en la que son concebidas como adiciones a un lenguaje preexistente y complejo.

Inspirados en este sentido nos planteamos el objetivo de explorar el potencial de trabajar con verificadores y lenguajes de programación concebidos a la par. Para ello propusimos un lenguaje de programación muy básico, pero cuyas metas de diseño y sus características fueron establecidas teniendo en cuenta la verificabilidad.

Dotamos a este lenguaje de un sistema de tipos extendido que captura la noción de corrección y terminación de programas mediante el uso de un oráculo. Propusimos técnicas que, basadas en este sistema de tipos, infieren una precondición o una postcondición para un fragmento de un programa.

Con estas herramientas nos dedicamos a ampliar el lenguaje con mecanismos que permiten ahorrar la tarea de escribir ciertas anotaciones para la verificación. En este sentido propusimos la inferencia de anotaciones de procedimientos, el refuerzo de invariantes y las metasentencias. Estas últimas combinan una estructura de iteración conocida y frecuente con las técnicas de inferencia para obtener buenos candidatos a invariantes.

Para complementar el trabajo teórico realizado implementamos un prototipo de herramienta verificadora que lleva a la práctica una gran parte de las ideas propuestas. Esta herramienta aplica una serie de reducciones sintácticas a las fórmulas y combina diversos demostradores automáticos de forma tal de obtener mejores resultados.

A partir del trabajo realizado podemos concluir que efectivamente el hecho de diseñar un lenguaje al mismo tiempo que su *framework* de verificación permite avanzar en ciertas líneas que de otra manera no serían posibles. Al concebir una herramienta verificadora para un lenguaje preexistente se busca realizar la menor cantidad de modificaciones posibles al mismo. Un enfoque de ese tipo jamás podría aprovechar las ventajas de ampliar el lenguaje con construcciones de alto nivel como las presentadas en nuestro trabajo.

Por otra parte, al trabajar con un lenguaje *core* muy simple, el desarrollo de técnicas de inferencia

se convirtió en una tarea relativamente sencilla. Proponer y demostrar motores de inferencia de precondiciones y postcondiciones sería una tarea excesivamente compleja en el caso de un lenguaje de uso masivo, con características tales como *aliasing*, *reflection*, carga dinámica y concurrencia.

Sin embargo, a pesar de trabajar con un lenguaje pequeño, simple y ad-hoc hay una serie de obstáculos para la verificación que siguen apareciendo. Incluso un programa relativamente simple como es el caso de un *bubble sort* requiere la escritura de invariantes no triviales. Invariantes que definitivamente pueden ser tomados como una carga de trabajo extra, haciendo que las herramientas de verificación sean abandonadas.

Por último, consideramos que los aportes hechos durante este trabajo representan un paso adelante al presentar algunos puntos de ataque hacia los problemas más urgentes que tienen las técnicas de verificación automática: alta necesidad de especificación y capacidad limitada de los demostradores. Creemos que aumentando la capacidad de inferencia para atacar más problemas y combinando nuevos y mejores demostradores se podrán obtener herramientas de verificación automática cada vez mas usables y versátiles.

6.2. Trabajo a futuro

El primer punto a desarrollar en un futuro cercano será la finalización del prototipo para dotarlo de aquellas características que quedaron fuera del mismo como ser la verificación de expresiones seguras y la inferencia de los tipos de las variables. Con esta tarea realizada se podría avanzar en la integración de esta herramienta en algún entorno de desarrollo que permita a potenciales usuarios experimentar y ampliar este *core* verificable.

Otra línea de trabajo interesante es la experimentación con otros demostradores. A lo largo del presente trabajo hemos probado con demostradores automáticos para lógica de primer orden sin obtener buenos resultados. Una investigación más exhaustiva sobre otros tipos de herramientas de demostración –como por ejemplo demostradores para lógica de alto orden– podría ampliar las capacidades de nuestra herramienta. Observar que su arquitectura permite la incorporación de nuevas herramientas de demostración de forma muy sencilla, sean estas automáticas o asistidas.

La incorporación de nuevas construcciones de iteración de alto nivel al lenguaje podría ser otro punto interesante. Las técnicas de inferencia presentadas en este trabajo pueden ser explotadas de muchas otras formas distintas, permitiendo crear nuevas instrucciones para desarrollar tareas que aparecen con frecuencia.

Nos interesaría aumentar la expresividad de las expresiones booleanas con las que se especifican los programas incorporando predicados que permitan referirse al máximo de un arreglo, a la cantidad de elementos que cumplen una propiedad, etc. Incluso podría ofrecerse un mecanismo mediante el cual el usuario pueda definir sus propios predicados. Para ello hay que tener muy en cuenta la vinculación de los mismos con su contraparte teórica para que los demostradores puedan razonar sobre su validez.

El enfoque presentado en este trabajo no explota la interacción con el usuario. En general, la persona que escribe un programa tiene una intuición de por qué cree que el mismo es correcto y termina. En la herramienta presentada la única forma que tiene el usuario de establecer esa intuición es mediante las anotaciones que provee. Se puede explorar la utilización de otras formas más innovadoras de interacción con el usuario, tales como preguntas por parte del sistema que el usuario debe responder o mecanismos de anotación incremental.

Una línea prometedora puede surgir al combinar estas técnicas presentadas con otro tipo de mecanismos, en particular en lo que respecta a inferencia de invariantes. Se pueden utilizar herramientas

dinámicas tales como DAIKON o mecanismos estáticos como por ejemplo la interpretación abstracta.

Por último, el lenguaje presentado en este trabajo no es más que un *core* minimal que deja de lado muchísimas funcionalidades indispensables que un lenguaje moderno debe proveer. Entre las características que deberían considerarse se encuentra la definición de tipos por parte del usuario, el soporte de recursión con anotaciones que permitan verificar terminación, el manejo de memoria dinámica y algún mecanismo de control de *aliasing*, clases con herencia e invariantes de representación.

Creemos que trabajando estos puntos se podría obtener un lenguaje prolijo, sólido y versátil dotado con un motor verificador de amplias capacidades.

Apéndice A

Demostraciones

A.1. Demostración del Teorema 3.1

Sean $i \in \text{IntExp}_\Gamma, A \in \text{ArrayExp}_\Gamma, b \in \text{BoolExp}_\Gamma$. Sea $\mu \in \text{M}_\Gamma$, luego:

$$\begin{aligned}
\text{si } [\![\text{safe}^{\text{I}}(i)]\!]_\mu^{\text{Bool}} = \text{true luego existe algún } n \in \mathbb{Z} : [\![i]\!]_\mu^{\text{Int}} = n \\
\text{si } [\![\text{safe}^{\text{A}}(A)]\!]_\mu^{\text{Bool}} = \text{true luego existe algún } \mathcal{A} \in \mathbb{A} : [\![A]\!]_\mu^{\text{Array}} = \mathcal{A} \\
\text{si } [\![\text{safe}^{\text{B}}(b)]\!]_\mu^{\text{Bool}} = \text{true luego existe algún } x \in \{\text{true}, \text{false}\} : [\![b]\!]_\mu^{\text{Bool}} = x
\end{aligned}$$

Dem:

Inducción estructural en $i \in \text{IntExp}_\Gamma, A \in \text{ArrayExp}_\Gamma, b \in \text{BoolExp}_\Gamma$.

- Caso expresión entera atómica $i = v,\ i = v\textbf{@pre},\ i = \underline{n}$.

  Sus reglas para semántica no tienen hipótesis, con lo cual siempre existe un $n \in \mathbb{Z}$ tal que $[\![i]\!]_\mu^{\text{Int}} = n$.

- Caso tamaño de un arrreglo $i = |A|$.

  Observar que $\text{safe}^{\text{I}}(|A|)$ pide $\text{safe}^{\text{A}}(A)$, por lo tanto si la primera es verdad también lo es la segunda. Si $\text{safe}^{\text{A}}(A)$ es verdad luego por hipótesis inductiva vale que A tiene su valor definido. Por ende se le puede dar valor a $|A|$.

- Caso acceso a un arreglo $i = A[i]$.

  Las reglas para semántica en estos casos exigen que:

  1. El arreglo esté definido: $[\![A]\!]_\mu^{\text{Array}} = \mathcal{A}$.
  2. El índice esté definido: $[\![i]\!]_\mu^{\text{Int}} = n$.
  3. El índice esté en rango: $0 \leq n < \|\mathcal{A}\|$.

  Observar que $\text{safe}^{\text{I}}(A[i])$ pide que $\text{safe}^{\text{A}}(A)$ y que $\text{safe}^{\text{I}}(i)$. Luego se puede aplicar la hipótesis inductiva para obtener las primeras dos condiciones.

  La tercera condición es satisfecha por la última parte de $\text{safe}^{\text{I}}(A[i])$, la cual pide que $0 \leq i < |A|$. Recordar que $[\![|A|]\!]_\mu^{\text{Int}} = \|\mathcal{A}\|$ si $[\![A]\!]_\mu^{\text{Array}} = \mathcal{A}$.

- Caso expresión entera binaria excepto división $i = i_1 + i_2,\ i = i_1 - i_2,\ i = i_1 * i_2$.

  Observar que safe^{I} de una expresión binaria exige las condiciones de expresión segura para ambas partes. Aplicando la hipótesis inductiva se sabe que cada una de las subexpresiones tiene su valor definido. En esas condiciones se le puede dar valor a la expresión binaria.

- Caso división $i = i_1/i_2$.

  Similar a la regla anterior, pero adicionalmente la regla para semántica pide que si $[\![i_2]\!]_\mu^{\text{Int}} = m$ luego $m \neq 0$. Esto es satisfecho gracias a que $\text{safe}^{\text{I}}(i_1/i_2)$ pide que $i_2 \neq 0$.

- Caso expresión arreglo variable $A = v, A = v\textbf{@pre}$.

  Similar a las reglas para variables enteras. La semántica en estos casos no exige ninguna hipótesis. Siempre existe un arreglo $\mathcal{A} \in \mathbb{A}$ tal que $[\![A]\!]_\mu^{\text{Array}} = \mathcal{A}$.

- Caso actualización de un arreglo en una posición $A = \textbf{update}\ A'\ \textbf{on}\ i_1\ \textbf{with}\ i_2$.

  En este caso, la semántica exige que:

  1. El arreglo esté definido: $[\![A']\!]_\mu^{\text{Array}} = \mathcal{A}'$.
  2. El índice esté definido: $[\![i_1]\!]_\mu^{\text{Int}} = n$.
  3. El nuevo elemento esté definido: $[\![i_2]\!]_\mu^{\text{Int}} = m$.
  4. El índice esté en rango: $0 \leq n < \left|\mathcal{A}'\right|$.

  Se trata de un caso muy similar al acceso a un arreglo. La única diferencia es el tercer punto, el cual es garantizado por hipótesis inductiva, ya que safe^{A} en este caso exige también $\text{safe}^{\text{I}}(i_2)$.

- Caso constante de arreglo $A = \textbf{array}[i_1]\ \textbf{of}\ i_2$.

  La regla para semántica en este caso requiere que:

  1. El tamaño esté definido: $[\![i_1]\!]_\mu^{\text{Int}} = n$.
  2. El elemento inicializador esté definido: $[\![i_2]\!]_\mu^{\text{Int}} = m$.
  3. El tamaño sea no negativo: $n \geq 0$.

  Las primeras dos condiciones son satisfechas por la hipótesis inductiva, ya que safe^{A} en este caso exige tanto $\text{safe}^{\text{I}}(i_1)$ como $\text{safe}^{\text{I}}(i_2)$.

  La última condición es satisfecha porque safe^{A} pide $i_1 \geq 0$.

- Caso comparación entre enteros $b = (i_1 = i_2),\ b = (i_1 < i_2)$.

  Las dos partes están definidas por hipótesis inductiva ya que safe^{B} exige $\text{safe}^{\text{I}}(i_1)$ y $\text{safe}^{\text{I}}(i_2)$.

  La igualdad tiene dos reglas, una para cuando la semántica es igual de ambos lados, la otra para cuando es distinta. Con lo cual siempre se puede aplicar alguna de las dos reglas.

  De forma similar, la comparación por menor tiene dos reglas. Una se puede aplicar cuando la semántica de una parte es menor que la de la otra. La otra regla se puede aplicar cuando la semántica de una parte es mayor o igual a la de la otra.

- Caso comparación entre arreglos $b = (A_1 = A_2)$.

  Similar al caso anterior. Por hipótesis inductiva ambas partes tienen semántica. Y siempre alguna de las dos reglas se puede aplicar.

- Caso conjunción $b = b_1 \wedge b_2$.

  Por hipótesis inductiva ambas partes tienen semántica ya que safe^{B} exige que $\text{safe}^{\text{B}}(b_1)$ y $\text{safe}^{\text{B}}(b_2)$.

  Si $[\![b_1]\!]_\mu^{\text{Bool}} = \text{false}$ luego se aplica la regla s-be-and-f2.

  Caso contrario, según el valor de $[\![b_2]\!]_\mu^{\text{Bool}}$ se puede aplicar s-be-and-f1 o s-be-and-t.

- Caso negación $b = \neg b'$.

  Por hipótesis inductiva se sabe que la semántica de b' está bien definida. Dependiendo de su valor siempre se puede aplicar alguna de las dos reglas para la semántica de la negación.

- Caso cuantificación universal $b = \forall\ v : \text{Int}\ (b'), b = \forall\ v : \text{Array}\ (b')$.

  Por hipótesis inductiva se sabe que la semántica está definida para b' con cualquier valor de v. Se pueden dar dos casos:

  1. La semántica, para cualquier valor de v, es siempre verdadera. En ese caso se puede aplicar la regla para true (ya sea para enteros o arreglos).
  2. En algún v_0, la semántica es falsa. En ese otro caso se puede aplicar la regla para false.

□

A.2. Lema sobre extensiones a valuaciones

Antes de demostrar el Teorema 3.2, se introducen un resultado auxiliar.

Lema A.1. *Las extensiones a una valuación no afectan el valor de una expresión booleana si se alteran variables que no aparecen en ella.*

Sean $\mu \in \mathrm{M}_\Gamma, b \in \mathrm{BoolExp}_\Gamma$ *tales que* $[\![b]\!]^{\mathrm{BOOL}}_\mu = \mathrm{true}$.

Sea v *una variable que no aparece libre en* b. *Luego:*

- $[\![b]\!]^{\mathrm{BOOL}}_{\mu\{v \mapsto n\}} = \mathrm{true}$.
- $[\![b]\!]^{\mathrm{BOOL}}_{\mu\{int\ v \mapsto n\}} = \mathrm{true}$.
- $[\![b]\!]^{\mathrm{BOOL}}_{\mu\{v \mapsto \mathcal{A}\}} = \mathrm{true}$.
- $[\![b]\!]^{\mathrm{BOOL}}_{\mu\{array\ v \mapsto \mathcal{A}\}} = \mathrm{true}$.
- $[\![b]\!]^{\mathrm{BOOL}}_{\mu \ominus \{v\}} = \mathrm{true}$.

Este resultado también se aplica a expresiones enteras o arreglo.

Dem:

Tanto las extensiones como la poda únicamente alteran la valuación μ en lo que respecta al valor para v.

El valor de la expresión booleana b bajo una determinada valuación, tal como está definido, únicamente depende de los valores que tal valuación asigne a sus variables libres.

Como v no aparece libre en la expresión booleana b, luego ni las extensiones ni la poda afectan su valor.

□

A.3. Demostración del Teorema 3.2

Sea un programa seguro π y sea p un procedimiento en π. Luego:

$$\text{para toda valuación } \mu \in \mathrm{M}_{\Gamma_p} \text{ sucede que si } [\![\mathrm{P}_(\, p)]\!]_{\mu}^{\textsc{Bool}} = \text{true} \text{ luego existe una valuación } \mu' \in \mathrm{M}_{\Gamma_p}$$

$$\text{tal que } \mu \rhd \pi \ : \ p \rhd \mu' \text{ y } [\![\mathrm{Q}_(\, p)]\!]_{\mu'}^{\textsc{Bool}} = \text{true}$$

Dem:

El caso en que π sea $\emptyset$ es trivial, ya que no existe un procedimiento p tal que $p \in \emptyset$.

Suponiendo que π es seguro, entonces se le agrega un nuevo procedimiento:

$$proc(p_1, \ldots, p_k) \textbf{ :? } pre \textbf{ :! } post \ \{ \ m_1, \ldots, m_n \ \} s$$

Tal que $\{pre\}\ s\ \{post\}$.

Una ejecución de π extendido con *proc* que comienza desde el procedimiento p cumple uno de los siguientes dos escenarios:

1. $p \in \pi$ en cuyo caso aplicando la hipótesis inductiva se sabe que la propiedad vale.
2. p es el procedimiento recién agregado, *proc*. Hay que demostrar que cualquier valuación que satisface *pre*, puede ejecutar el cuerpo s y llegar a otra valuación que satisface *post*. Se analizará este caso en detalle.

Se sabe que $\{pre\}\ s\ \{post\}$, se hará inducción en las demostraciones que permiten llegar a tal juicio:

- Caso asignación a una variable entera:

$$\frac{p \models \mathrm{safe}^{\mathrm{I}}(i) \qquad \exists\, v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor \ \wedge \ v = i\lfloor v \mapsto v' \rfloor) \models q}{\{p\}\ v \leftarrow i\ \{q\}}\ \textsc{a-sent-iassign}$$

  Suponiendo $p \models \mathrm{safe}^{\mathrm{I}}(i)$, a partir del Teorema 3.1, se sabe que existe un $n \in \mathbb{Z} : \ [\![i]\!]_{\mu}^{\textsc{Int}} = n$.

  Con lo cual, se cumple la hipótesis de la regla s-sent-iassign, la cual permite afirmar que:

$$\mu \rhd v \leftarrow i \rhd \mu\{v \mapsto n\}$$

  Falta ver que $[\![q]\!]_{\mu\{v \mapsto n\}}^{\textsc{Bool}}$, pero como $\exists\, v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor \ \wedge \ v = i\lfloor v \mapsto v' \rfloor) \models q$ basta ver que:

$$[\![\exists\, v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor \ \wedge \ v = i\lfloor v \mapsto v' \rfloor)]\!]_{\mu\{v \mapsto n\}}^{\textsc{Bool}} = \text{true}$$

  Lo cual es válido si hay algún valor entero para v' que haga válido a:

$$p\lfloor v \mapsto v' \rfloor \ \wedge \ v = i\lfloor v \mapsto v' \rfloor$$

  Tal valor es $v' = \mu(v)$, o sea, el valor de v antes de la asignación. Con lo cual la fórmula que debe ser válida bajo $\mu\{v \mapsto n\}$ es:

$$p\lfloor v \mapsto \mu(v) \rfloor \ \wedge \ v = i\lfloor v \mapsto \mu(v) \rfloor$$

La validez de la primer parte se desprende del hecho de que $[\![p]\!]^{\text{Bool}}_{\mu} = \text{true}$ por hipótesis, con lo cual reemplazar v por su valor concreto en μ es inocuo.

Adicionalmente se puede afirmar usando el Lema A.1, que $p\lfloor v \mapsto \mu(v)\rfloor$ es válido bajo la valuación extendida $\mu\{v \mapsto n\}$, ya que luego de la sustitución no aparece la variable v.

Para la segunda parte, observar primero que $[\![i]\!]^{\text{Int}}_{\mu} = n$, luego $[\![i\lfloor v \mapsto \mu(v)\rfloor]\!]^{\text{Int}}_{\mu} = n$ ya que reemplazar v por su valor concreto antes de la asignación no altera la valuación de la expresión entera i.

Por otra parte, usando el Lema A.1, extender μ asignándole un nuevo valor para v, no altera el valor de $i\lfloor v \mapsto \mu(v)\rfloor$, ya que luego de la sustitución v no aparece en dicha expresión.

Con lo cual $[\![v = i\lfloor v \mapsto \mu(v)\rfloor]\!]^{\text{Bool}}_{\mu\{v\mapsto n\}} = [\![v = n]\!]^{\text{Bool}}_{\mu\{v\mapsto n\}}$ lo cual es trivialmente cierto.

- Caso asignación de una variable arreglo:

$$\frac{p \models \text{safe}^{\text{A}}(A) \qquad \exists\, v' : \text{Array}\ (p\lfloor v \mapsto v'\rfloor \ \wedge\ v = A\lfloor v \mapsto v'\rfloor) \models q}{\{p\}\ v \leftarrow A\ \{q\}}\ \text{A-SENT-AASSIGN}$$

Análogo al caso anterior.

- Caso creación de una variable entera:

$$\frac{p \models \text{safe}^{\text{I}}(i) \qquad p \ \wedge\ v = i \models q}{\{p\}\ \textbf{int}\ v \leftarrow i\ \{q\}}\ \text{A-SENT-IDEF}$$

Suponiendo la hipótesis, gracias al Teorema 3.1, se sabe que existe un $n \in \mathbb{Z} :\ [\![i]\!]^{\text{Int}}_{\mu} = n$.

Esa es precisamente la hipótesis para la regla SEM-SENT-INTDEF, con lo cual se puede afirmar que:

$$\mu \ \triangleright\ \textbf{int}\ v \leftarrow i \ \triangleright\ \mu\{\text{int}\ v \mapsto n\}$$

Falta ver que $[\![q]\!]^{\text{Bool}}_{\mu\{\text{int}\ v\mapsto n\}}$, pero como $p \ \wedge\ v = i \models q$ basta ver que:

$$[\![p \ \wedge\ v = i]\!]^{\text{Bool}}_{\mu\{\text{int}\ v\mapsto n\}} = \text{true}$$

La primer parte se desprende del hecho de que, por hipótesis, $[\![p]\!]^{\text{Bool}}_{\mu} = \text{true}$. Por construcción de $p \in \text{BoolExp}_{\Gamma}$ se sabe que $v \notin p$, ya que por construcción de s sucede que $v \notin \text{dom}(\Gamma)$. Luego la extensión a la valuación no afecta la validez de p, usando el Lema A.1.

La segunda parte se desprende de la definición de extensión a una valuación y el hecho de que $[\![i]\!]^{\text{Int}}_{\mu} = n$.

- Caso definición de arreglo:

$$\frac{p \models \text{safe}^{\text{A}}(A) \qquad p \ \wedge\ v = A \models q}{\{p\}\ \textbf{array}\ v \leftarrow A\ \{q\}}\ \text{A-SENT-ADEF}$$

Análogo al caso anterior.

- Caso sentencia vacía:

$$\frac{p \models q}{\{p\}\ \mathbf{skip}\ \{q\}}\ \text{A-SENT-SKIP}$$

Sin necesidad de hipótesis, se puede aplicar la regla S-SENT-SKIP, la cual permite afirmar que:

$$\mu \triangleright \mathbf{skip} \triangleright \mu$$

Resta verificar que toda valuación $\mu \in \mathrm{M}_\Gamma$ tal que cumple que $[\![p]\!]^{\text{BOOL}}_\mu = \text{true}$ cumple también que $[\![q]\!]^{\text{BOOL}}_\mu = \text{true}$, lo cual es precisamente la definición de $p \models q$, lo cual se sabe que se cumple ya que es una hipótesis de la regla.

- Caso secuenciamiento:

$$\frac{\{p\}\ s_1\ \{r\} \quad \{r\}\ s_2\ \{q\}}{\{p\}\ s_1\ s_2\ \{q\}}\ \text{A-SENT-SEQ}$$

Suponiendo las hipótesis de la regla, por hipótesis inductiva, para toda valuación $\mu \in \mathrm{M}_\Gamma$ tal que $[\![p]\!]^{\text{BOOL}}_\mu = \text{true}$, vale que $\mu \triangleright s_1 \triangleright \mu'$ y $[\![r]\!]^{\text{BOOL}}_{\mu'} = \text{true}$.

Por otra parte, y también usando la hipótesis inductiva, como μ' es tal que $[\![r]\!]^{\text{BOOL}}_{\mu'} = \text{true}$, luego vale que $\mu' \triangleright s_2 \triangleright \mu''$ y $[\![q]\!]^{\text{BOOL}}_{\mu''} = \text{true}$.

Por lo tanto, se tienen las hipótesis suficientes para aplicar la regla S-SENT-SEQ, con lo cual:

$$\mu \triangleright s_1\ s_2 \triangleright \mu''$$

Y μ'' es tal que $[\![q]\!]^{\text{BOOL}}_{\mu''} = \text{true}$.

- Caso condicional:

$$\frac{\begin{array}{c} p \models \text{safe}^{\text{B}}(g) \\ p \wedge g \models p_1 \qquad \{p_1\}\ s_1\ \{q_1\} \qquad q_1 \models q \\ p \wedge \neg g \models p_2 \qquad \{p_2\}\ s_2\ \{q_2\} \qquad q_2 \models q \end{array}}{\{p\}\ \mathbf{if}\ g\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}\ \{q\}}\ \text{A-SENT-IF}$$

Sea $\mu \in \mathrm{M}_\Gamma$ una valuación tal que $[\![p]\!]^{\text{BOOL}}_\mu = \text{true}$.

Suponiendo las hipótesis de la regla, como $p \models \text{safe}^{\text{B}}(g)$, a partir del Teorema 3.1, se puede garantizar que existe un $x \in \{\text{true}, \text{false}\} : [\![g]\!]^{\text{BOOL}}_\mu = x$.

Se divide la prueba en dos casos, según el valor de x:

1. Caso $x = \text{true}$.
   En este primer caso $[\![p \wedge g]\!]^{\text{BOOL}}_\mu = \text{true}$, con lo cual también vale que $[\![p_1]\!]^{\text{BOOL}}_\mu$ y por lo tanto, a partir de la hipótesis inductiva, se sabe que:

$$\mu \triangleright s_1 \triangleright \mu'$$

   Con μ' tal que $[\![q_1]\!]^{\text{BOOL}}_{\mu'} = \text{true}$, por lo tanto $[\![q]\!]^{\text{BOOL}}_{\mu'}$.

Se puede entenonces aplicar la regla S-SENT-IF-T, para obtener:

$$\mu \triangleright \textbf{if } g \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \triangleright \mu' \ominus \text{localVars}(s_1)$$

Falta ver que:

$$[\![q]\!]^{\text{BOOL}}_{\mu' \ominus \text{localVars}(s_1)} = \text{true}$$

Esto se desprende del hecho de que $q \in \text{BoolExp}_\Gamma$, por lo tanto ninguna de las variables locales de s_1 puede aparecer en q.

2. Caso $x = \text{false}$.
   En este otro caso $[\![p \wedge \neg g]\!]^{\text{BOOL}}_\mu = \text{true}$, con lo cual también vale que $[\![p_2]\!]^{\text{BOOL}}_\mu$ y por lo tanto, a partir de la hipótesis inductiva, se sabe que:

$$\mu \triangleright s_2 \triangleright \mu''$$

   Con μ'' tal que $[\![q_2]\!]^{\text{BOOL}}_{\mu''} = \text{true}$, por lo tanto $[\![q]\!]^{\text{BOOL}}_{\mu''}$.
   Se puede entonces aplicar la regla S-SENT-IF-F, para obtener:

$$\mu \triangleright \textbf{if } g \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \triangleright \mu'' \ominus \text{localVars}(s_2)$$

   Falta ver que:

$$[\![q]\!]^{\text{BOOL}}_{\mu'' \ominus \text{localVars}(s_2)} = \text{true}$$

   Tal como en la rama "then", esto es cierto ya que ninguna variable en $\text{localVars}(s_2)$ aparece en q.

- Caso ciclo:

$$\frac{\begin{array}{c}\textbf{true} \models \text{safe}^{\text{B}}(inv) \quad inv \models \text{safe}^{\text{B}}(g) \quad inv \models \text{safe}^{\text{I}}(var) \\ p \models inv \qquad inv \wedge g \models p' \qquad p' \models var > \underline{0} \\ \{p'\}\ var_0 \leftarrow var\ \ s\ \{q'\} \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q\end{array}}{\{p\}\ \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od}\ \{q\}}\ \text{A-SENT-WHILE}$$

  Sea $\mu \in \text{M}_\Gamma$ una valuación tal que $[\![p]\!]^{\text{BOOL}}_\mu = \text{true}$.

  Suponiendo las hipótesis de la regla, como $p \models inv$ y también $inv \models \text{safe}^{\text{B}}(g)$, a partir del Teorema 3.1, se puede garantizar que existe un $x \in \{\text{true}, \text{false}\} : [\![g]\!]^{\text{BOOL}}_\mu = x$.

  Dado que $\textbf{true} \models \text{safe}^{\text{B}}(inv)$ es también una hipótesis de la regla, a través del mismo teorema se sabe que el invariante tiene su semántica bien definida. Más concretamente, como $p \models inv$, se sabe que $[\![inv]\!]^{\text{BOOL}}_\mu = \text{true}$.

  Se divide la prueba en dos casos, según x, el valor al que evalúa la guarda.

  1. Caso $x = \text{false}$.
     En este caso, como se sabe que μ hace verdadero al invariante, se puede aplicar la regla S-SENT-WHILE-F, con lo cual:

$$\mu \triangleright \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \triangleright \mu$$

     Falta ver que $[\![q]\!]^{\text{BOOL}}_\mu$, y como $inv \wedge \neg g \models q$, alcanza con ver:

$$[\![inv \wedge \neg g]\!]^{\text{BOOL}}_\mu = \text{true}$$

- $[\![inv]\!]^{\text{BOOL}}_{\mu} = \text{true}$ se vió antes de partir en casos.
- $[\![\neg g]\!]^{\text{BOOL}}_{\mu} = \text{true}$ se desprende del hecho de que $[\![g]\!]^{\text{BOOL}}_{\mu} = x = \text{false}$.

2. Caso $x = \text{true}$.
   Como $[\![inv]\!]^{\text{BOOL}}_{\mu}$ y $inv \models \text{safe}^{\text{I}}(var)$ luego por Teorema 3.1, existe un $m \in \mathbb{Z} : [\![var]\!]^{\text{INT}}_{\mu} = m$.
   Por otra parte, como $inv \wedge g \models p'$ y $p' \models var > \underline{0}$, luego sucede que $[\![var > \underline{0}]\!]^{\text{BOOL}}_{\mu} = \text{true}$.
   Por lo tanto $m > 0$.
   Se quiere probar que se puede aplicar la regla para semántica operacional que permite continuar la ejecución de un ciclo:

$$\frac{\begin{array}{c}[\![g]\!]^{\text{BOOL}}_{\mu} = \text{true} \quad [\![inv]\!]^{\text{BOOL}}_{\mu} = \text{true} \quad [\![var]\!]^{\text{INT}}_{\mu} > 0 \\ \mu \rhd s \rhd \mu' \\ [\![inv]\!]^{\text{BOOL}}_{\mu' \ominus \text{localVars}(s)} = \text{true} \quad [\![var]\!]^{\text{INT}}_{\mu' \ominus \text{localVars}(s)} < [\![var]\!]^{\text{INT}}_{\mu} \\ \mu' \ominus \text{localVars}(s) \rhd \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \rhd \mu''\end{array}}{\mu \rhd \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \rhd \mu''} \text{ S-SENT-WHILE-T}$$

   Para eso es necesario satisfacer todas sus hipótesis:
   - $[\![g]\!]^{\text{BOOL}}_{\mu} = \text{true}$ es cierto porque es la hipótesis para este caso.
   - $[\![inv]\!]^{\text{BOOL}}_{\mu} = \text{true}$ ya se vió antes de partir en casos.
   - $[\![var]\!]^{\text{INT}}_{\mu} > 0$ se desprende del hecho de que $m > 0$.
   - $\mu \rhd s \rhd \mu'$
     Por hipótesis inductiva, como $\{p'\}\ var_0 \leftarrow var\ \ s\ \{q'\}$ y $[\![p']\!]^{\text{BOOL}}_{\mu} = \text{true}$, luego sucede que $\mu \rhd var_0 \leftarrow var\ \ s \rhd \mu'$ con $\mu' \in \text{M}_{\Gamma'}$ tal que $[\![q']\!]^{\text{BOOL}}_{\mu'} = \text{true}$.
     Observar que la asignación $var_0 \leftarrow var$ únicamente afecta la variable var_0, que únicamente se usa para ver que el variante decrece. Con lo cual, puede ignorarse y se obtiene $\mu \rhd s \rhd \mu'$.
   - $[\![inv]\!]^{\text{BOOL}}_{\mu' \ominus \text{localVars}(s)} = \text{true}$
     Como $[\![q']\!]^{\text{BOOL}}_{\mu'} = \text{true}$ y además $\textbf{true} \models \text{safe}^{\text{B}}(inv)$, luego por Teorema 3.1 el invariante tiene semántica bien definida en μ'. Por otra parte, como $q' \models inv$, luego $[\![inv]\!]^{\text{BOOL}}_{\mu'} = \text{true}$.
     Por construcción ninguna variable en localVars(s) puede aparecer en inv, luego por Lema A.1, $[\![inv]\!]^{\text{BOOL}}_{\mu' \ominus \text{localVars}(s)} = \text{true}$.
   - $[\![var]\!]^{\text{INT}}_{\mu' \ominus \text{localVars}(s)} < [\![var]\!]^{\text{INT}}_{\mu}$
     Observar que var_0 almacena el valor m, o sea el valor del variante en μ.
     Por otra parte se sabe que $[\![q']\!]^{\text{BOOL}}_{\mu'}$, y también $q' \models inv$ y $inv \models \text{safe}^{\text{I}}(var)$ con lo cual, usando el Teorema 3.1, existe un $m' \in \mathbb{Z}$ tal que $[\![var]\!]^{\text{INT}}_{\mu'} = m'$.
     Por último se sabe que $q' \models var < var_0$, y como var_0 no es modificado por s vale que $m' < m$, o sea $[\![var]\!]^{\text{INT}}_{\mu' \ominus \text{localVars}(s)} < [\![var]\!]^{\text{INT}}_{\mu}$.
     Observar que por construcción ninguna variable en localVars(s) puede aparecer en var. Por lo tanto el valor de var en $\mu' \ominus \text{localVars}(s)$ es idéntico al valor en μ'.
   - $\mu' \ominus \text{localVars}(s) \rhd \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \rhd \mu''$
     Esto se puede demostrar utilizando la misma regla para semántica abstracta pero a partir de q' y con $\mu' \ominus \text{localVars}(s)$. Se utiliza inducción en m', el valor del variante en $\mu' \ominus \text{localVars}(s)$.

- Caso $m' \leq 0$.
  Observar que si se usa q' en lugar de p', la regla para la semántica abstracta requiere que $q' \models var > \underline{0}$. Sin embargo esto no puede ser ya que $m' \leq 0$.
  Esta contradicción proviene del hecho de que $inv \ \wedge \ g \models q'$ y sin embargo $p \models inv$, con lo cual la guarda es necesariamente falsa.
  Por lo tanto se aplica la regla de semántica operacional que finaliza la ejecución del ciclo, de forma similar al caso $x = \text{false}$ visto anteriormente.
- Tomar un $m' > 0$ y suponer que vale la propiedad para cualquier valor menor a m', ver que la propiedad vale para m'.
  Por razonamientos análogos a todos los hechos hasta aquí pero reemplazando p' por q' y μ por $\mu' \ominus \text{localVars}(s)$, se puede llegar hasta este punto.
  Adicionalmente, como se sabe que el variante decrece, luego se tiene el variante m'' (el análogo a m' en esta demostración) tal que $m'' < m'$. Por lo tanto se aplica la hipótesis inductiva y se sabe que: $\mu' \ominus \text{localVars}(s) \triangleright \textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \triangleright \mu''$

Falta ver que:

$$[\![inv \ \wedge \ \neg g]\!]^{\text{Bool}}_{\mu''} = \text{true}$$

- $[\![inv]\!]^{\text{Bool}}_{\mu''} = \text{true}$
  Se puede ver por inducción en m', el valor del variante en μ'.
  Si $m' \leq 0$, luego $\mu'' = \mu'$. Ya se vió que $[\![inv]\!]^{\text{Bool}}_{\mu'} = \text{true}$.
  Si $m' > 0$ y se supone que la propiedad vale para valores más chicos que m', ver que vale para m'. En este caso se puede aplicar el mismo razonamiento pero con m' en lugar donde antes estaba m. Se llega a un m'' tal que $m'' < m'$, luego allí vale que el invariante se preserva. Y una única aplicación de la regla para semántica operacional que continúa la ejecución del ciclo garantiza que el invariante se preserva.
- $[\![\neg g]\!]^{\text{Bool}}_{\mu''} = \text{true}$
  La única forma de finalizar la ejecución del ciclo es que g sea falso. Esto se puede demostrar por inducción en m'.
  Si $m' \leq 0$, ya se vió que como $inv \wedge g \models p' \models var > \underline{0}$, luego la única posibilidad es que g sea falsa.
  Si en cambio $m' > 0$, se puede aplicar la hipótesis inductiva y el ciclo siempre termina con la aplicación de la regla cuya hipótesis es que la guarda es falsa.

- Caso llamado a un procedimiento:

$$\frac{\begin{array}{c} p\lfloor cp_i \mapsto p_i \rfloor \models \text{P}_(\, proc) \\ \exists o_1, \ldots, o_k \ (p\lfloor cp_i \mapsto o_i \rfloor \ \wedge \ \text{Q}_(\, proc)\lfloor p_i \mapsto cp_i, \ p_i\textbf{@pre} \mapsto o_i \rfloor) \models q \end{array}}{\{p\} \ \textbf{call } proc(cp_1, \ldots, cp_k) \ \{q\}} \text{ A-SENT-CALL}$$

  Donde $\text{pars}(proc) = p_1, \ldots, p_k$

  Suponer que se parte de una valuación $\mu \in \text{M}_\Gamma$ tal que $[\![p]\!]^{\text{Bool}}_{\mu} = \text{true}$.

  Definiendo ρ como en la regla para semántica operacional para llamados a procedimientos, sucede que $[\![p\lfloor cp_i \mapsto p_i \rfloor]\!]^{\text{Bool}}_{\rho} = \text{true}$. Esto se debe a que sustituir los elementos en la fórmula es equivalente a cambiarles su nombre en la valuación.

  Por lo tanto, como $p\lfloor cp_i \mapsto p_i \rfloor \models \text{P}_(\, proc)$ luego $[\![\text{P}_(\, proc)]\!]^{\text{Bool}}_{\rho} = \text{true}$.

Como π es seguro, se sabe que:

$$\{\mathrm{P}_{(}\, proc)\}\ \mathrm{body}(proc)\ \{\mathrm{Q}_{(}\, proc)\}$$

Luego por hipótesis inductiva:

$$\rho \triangleright \mathrm{body}(proc) \triangleright \rho'$$

Con ρ' tal que $[\![\mathrm{Q}_{(}\, proc)]\!]^{\textsc{Bool}}_{\rho'} = \mathrm{true}$.

Con lo cual se cumplen las hipótesis de la regla SEM-SENT-CALL, por lo tanto:

$$\mu \triangleright proc(cp_1, \ldots, cp_k) \triangleright \mu'$$

Con μ' definido en función de ρ' tal como en la regla para semántica operacional de llamados a procedimientos.

Falta ver que $[\![q]\!]^{\textsc{Bool}}_{\mu'}$, y como $\exists o_1, \ldots, o_k\ (p\lfloor cp_i \mapsto o_i \rfloor \wedge \mathrm{Q}_{(}\, proc)\lfloor p_i \mapsto cp_i,\ p_i\mathbf{@pre} \mapsto o_i \rfloor) \models q$, alcanza con ver que:

$$[\![\exists o_1, \ldots, o_k\ (p\lfloor cp_i \mapsto o_i \rfloor \wedge \mathrm{Q}_{(}\, proc)\lfloor p_i \mapsto cp_i,\ p_i\mathbf{@pre} \mapsto o_i \rfloor)]\!]^{\textsc{Bool}}_{\mu'}$$

Antes que nada observar la similitud con la regla de la asignación. Se está trabajando en este caso sobre k variables a la vez, por lo tanto es necesario cuantificar existencialmente sus k valores anteriores.

Tal como en el caso de la asignación, sigue valiendo p pero reemplazando los parámetros concretos por sus viejos valores, denominados o_i. Los valores que harán verdadera la fórmula son $[\![cp_i]\!]^{\textsc{Int}}_{\mu}$ ó $[\![cp_i]\!]^{\textsc{Array}}_{\mu}$, dependiendo del tipo de cp_i.

Con tales valores se puede ver que $p\lfloor cp_i \mapsto o_i \rfloor$ es cierto, ya que $[\![p]\!]^{\textsc{Bool}}_{\mu}$ y justamente se están reemplazando los parámetros concretos por su valor en μ.

La segunda parte se desprende del hecho de que $[\![\mathrm{Q}_{(}\, proc)]\!]^{\textsc{Bool}}_{\rho'} = \mathrm{true}$. A partir de la postcondición, se pueden reemplazar los parámetros formales por los concretos, del mismo modo que μ' se construye a partir de ρ', con lo cual la valuación no es afectada.

□

A.4. Demostración del Teorema 3.3

Sea $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ y sea $p \in \text{BoolExp}_{\Gamma}$. Si $\text{post}(s,\ p) = q$ luego $\{p\}\ s\ \{q\}$.

Dem:

Se demuestra por inducción estructural en s.

- Caso asignaciones y creación de variables. Son todos casos muy similares, observar por ejemplo el caso de la asignación de una variable entera:

$$\frac{p \models \text{safe}^{\text{I}}(i)}{\text{post}(v \leftarrow i,\ p) = \exists\ v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = i\lfloor v \mapsto v' \rfloor)}\ \textsc{post-iassign}$$

  En este caso la postcondición es la más fuerte que permite la regla de semántica abstracta:

$$\frac{p \models \text{safe}^{\text{I}}(i) \qquad \exists\ v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = i\lfloor v \mapsto v' \rfloor) \models q}{\{p\}\ v \leftarrow i\ \{q\}}\ \textsc{a-sent-iassign}$$

  Dicho de otra forma:

$$q = \exists\ v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = i\lfloor v \mapsto v' \rfloor)$$

  Esto es correcto ya que para toda expresión booleana b vale que $b \models b$.

- El caso de la sentencia vacía y el secuenciamiento son triviales.

- Caso condicional. La regla para el cálculo en este caso es:

$$\frac{p \models \text{safe}^{\text{B}}(g) \quad \text{post}(s_1,\ p \wedge g) = q_1 \quad \text{post}(s_2,\ p \wedge \neg g) = q_2}{\text{post}(\textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi},\ p) = Cl_{\exists\,\text{localVars}(s_1)}(q_1) \vee Cl_{\exists\,\text{localVars}(s_2)}(q_2)}\ \textsc{post-if}$$

  La semántica abstracta en este caso es:

$$\frac{\begin{array}{c} p \models \text{safe}^{\text{B}}(g) \\ p \wedge g \models p_1 \qquad \{p_1\}\ s_1\ \{q_1\} \qquad q_1 \models q \\ p \wedge \neg g \models p_2 \qquad \{p_2\}\ s_2\ \{q_2\} \qquad q_2 \models q \end{array}}{\{p\}\ \textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ \{q\}}\ \textsc{a-sent-if}$$

  Se usa en este caso $p \wedge g$ como p_1, lo cual es válido ya que $p \wedge g \models p \wedge g$. Por hipótesis inductiva se sabe que $\text{post}(s_1,\ p \wedge g) = q_1$ luego $\{p \wedge g\}\ s_1\ \{q_1\}$.

  De forma análoga con s_2 se tiene sabe que $\{p \wedge \neg g\}\ s_2\ \{q_2\}$.

  Se utiliza como postcondición:

$$q = Cl_{\exists\,\text{localVars}(s_1)}(q_1) \vee Cl_{\exists\,\text{localVars}(s_2)}(q_2)$$

  Falta ver que $q_1 \models Cl_{\exists\,\text{localVars}(s_1)}(q_1)$ y $q_2 \models Cl_{\exists\,\text{localVars}(s_2)}(q_2)$. Estos dos casos son idénticos y pueden demostrarse diciendo que si una valuación logra satisfacer una expresión booleana luego también puede satisfacer a cualquier clausura existencial de la misma.

- Caso ciclo. En este caso la postcondición se calcula según:

$$\frac{\begin{array}{c}\mathbf{true} \models \mathrm{safe}^{\mathrm{B}}(inv) \quad inv \models \mathrm{safe}^{\mathrm{B}}(g) \quad inv \models \mathrm{safe}^{\mathrm{I}}(var) \\ p \models inv \quad inv \wedge g \models var > \underline{0} \\ \mathrm{post}(var_0 \leftarrow var \;\; s,\; inv \wedge g) = q' \\ q' \models inv \quad q' \models var < var_0\end{array}}{\mathrm{post}(\mathbf{while}\ g\ \mathbf{:?!}\ inv\ \mathbf{:\#}\ var\ \mathbf{do}\ s\ \mathbf{od},\ p) = inv\ \wedge\ \neg g}\ \text{POST-WHILE}$$

  La regla para semántica abstracta es:

$$\frac{\begin{array}{c}\mathbf{true} \models \mathrm{safe}^{\mathrm{B}}(inv) \quad inv \models \mathrm{safe}^{\mathrm{B}}(g) \quad inv \models \mathrm{safe}^{\mathrm{I}}(var) \\ p \models inv \qquad inv \wedge g \models p' \qquad p' \models var > \underline{0} \\ \{p'\}\ var_0 \leftarrow var \;\; s\ \{q'\} \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q\end{array}}{\{p\}\ \mathbf{while}\ g\ \mathbf{:?!}\ inv\ \mathbf{:\#}\ var\ \mathbf{do}\ s\ \mathbf{od}\ \{q\}}\ \text{A-SENT-WHILE}$$

  Tal como se hizo en otros casos, se utiliza como p' la expresión más fuerte posible. En este caso $p' = ing \wedge g$. De forma similar, se usa $q = inv \wedge \neg g$, que es la expresión más fuerte que la regla de la semántica abstracta permite.

- Caso llamado a procedimiento. La postcondición se obtiene con la siguiente regla:

$$\frac{1 \leq i \leq k \quad p\lfloor cp_i \mapsto p_i \rfloor \models P}{\mathrm{post}(\mathbf{call}\ pr(cp_1, \ldots, cp_k),\ p) = \exists_{o_1,\ldots,o_k}\ (p\lfloor cp_i \mapsto o_i \rfloor\ \wedge\ Q\lfloor p_i \mapsto cp_i,\ p_i\mathbf{@pre} \mapsto o_i \rfloor)}\ \text{POST-CALL}$$

  La semántica abstracta en este caso establece que:

$$\frac{1 \leq i \leq k \quad p\lfloor cp_i \mapsto p_i \rfloor \models \mathrm{P}_(\, proc)}{\exists o_1, \ldots, o_k\ (p\lfloor cp_i \mapsto o_i \rfloor\ \wedge\ \mathrm{Q}_(\, proc)\lfloor p_i \mapsto cp_i,\ p_i\mathbf{@pre} \mapsto o_i \rfloor) \models q}\ \text{A-SENT-CALL}$$

  Tal como en casos anteriores se utiliza la postcondición más fuerte que la regla de la semántica abstracta permite, o sea:

$$q = \exists o_1, \ldots, o_k\ (p\lfloor cp_i \mapsto o_i \rfloor\ \wedge\ \mathrm{Q}_(\, proc)\lfloor p_i \mapsto cp_i,\ p_i\mathbf{@pre} \mapsto o_i \rfloor)$$

□

A.5. Demostración del Teorema 3.4

Sea $s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}$ y sea $q \in \text{BoolExp}_{\Gamma'}$. Si $\text{pre}(s,\ q) = p$ luego $\{p\}\ s\ \{q\}$.

Dem:

Se demuestra por inducción estructural en s.

- Caso asignaciones y creación de variables. Todos casos similares, observar por ejemplo la asignación de enteros. Su cálculo de precondición establece que:

$$\frac{}{\text{pre}(v \leftarrow i,\ q) = \text{safe}^{\text{I}}(i)\ \wedge\ q\lfloor v \mapsto i \rfloor}\ \text{PRE-IASSIGN}$$

  La semántica abstracta dice en ese caso:

$$\frac{p \models \text{safe}^{\text{I}}(i) \qquad \exists\ v' : \textsc{Int}\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = i\lfloor v \mapsto v' \rfloor) \models q}{\{p\}\ v \leftarrow i\ \{q\}}\ \text{A-SENT-IASSIGN}$$

  En este caso:

$$p\ =\ \text{safe}^{\text{I}}(i)\ \wedge\ q\lfloor v \mapsto i \rfloor$$

  Puede verse fácilmente que:

$$\text{safe}^{\text{I}}(i)\ \wedge\ q\lfloor v \mapsto i \rfloor \models \text{safe}^{\text{I}}(i)$$

  Faltaría ver que:

$$\exists\ v' : \textsc{Int}\ ((\text{safe}^{\text{I}}(i)\ \wedge\ q\lfloor v \mapsto i \rfloor)\lfloor v \mapsto v' \rfloor\ \wedge\ v = i\lfloor v \mapsto v' \rfloor) \models q$$

  Lo cual es equivalente a:

$$\exists\ v' : \textsc{Int}\ (\text{safe}^{\text{I}}(i)\lfloor v \mapsto v' \rfloor\ \wedge\ (q\lfloor v \mapsto i \rfloor)\lfloor v \mapsto v' \rfloor\ \wedge\ v = i\lfloor v \mapsto v' \rfloor) \models q$$

  Observar que se reemplazan en q todas las apariciones de v por i, lo cual podría hacer que dicha expresión resultante no fuerce q. Sin embargo también se conoce el hecho de que $v = i$. Se obvian en este razonamiento las sustituciones de v por v'.

- Caso sentencia vacía y secuenciamiento. Son triviales.

- Caso condicional. La regla para el cálculo de precondiciones es:

$$\frac{\text{pre}(s_1,\ q) = p_1 \quad \text{pre}(s_2,\ q) = p_2}{\text{pre}(\textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi},\ q) = \text{safe}^{\text{B}}(g)\ \wedge\ (g \wedge p_1\ \vee\ \neg g \wedge p_2)}\ \text{PRE-IF}$$

  La semántica abstracta es determinada en este caso por:

$$\frac{\begin{array}{c} p \models \text{safe}^{\text{B}}(g) \\ p \wedge g \models p_1 \qquad \{p_1\}\ s_1\ \{q_1\} \qquad q_1 \models q \\ p \wedge \neg g \models p_2 \qquad \{p_2\}\ s_2\ \{q_2\} \qquad q_2 \models q \end{array}}{\{p\}\ \textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ \{q\}}\ \text{A-SENT-IF}$$

  Se utiliza en este caso:

$$p\ =\ \text{safe}^{\text{B}}(g)\ \wedge\ (g \wedge p_1\ \vee\ \neg g \wedge p_2)$$

  Por otra parte q es usado como q_1 y también como q_2. Por hipótesis inductiva se obtienen precondiciones p_1 y p_2 tales que satisfacen el teorema.

- Caso ciclo. En este caso la precondición se calcula usando:

$$\frac{\begin{array}{c}\mathbf{true} \models \mathrm{safe}^{\mathrm{B}}(inv) \quad inv \models \mathrm{safe}^{\mathrm{B}}(g) \quad inv \models \mathrm{safe}^{\mathrm{I}}(var) \\ inv \wedge g \models p' \qquad p' \models var > \underline{0} \\ \mathrm{post}(var_0 \leftarrow var \;\; s,\; inv \wedge g) = q' \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q\end{array}}{\mathrm{pre}(\mathbf{while}\ g\ \mathbf{:?!}\ inv\ \mathbf{:\#}\ var\ \mathbf{do}\ s\ \mathbf{od},\; q) = inv}\ \textsc{pre-while}$$

  La semántica abstracta para este caso, obviando las condiciones para expresiones seguras, es determinada por:

$$\frac{\begin{array}{c}\mathbf{true} \models \mathrm{safe}^{\mathrm{B}}(inv) \quad inv \models \mathrm{safe}^{\mathrm{B}}(g) \quad inv \models \mathrm{safe}^{\mathrm{I}}(var) \\ p \models inv \qquad inv \wedge g \models p' \qquad p' \models var > \underline{0} \\ \{p'\}\ var_0 \leftarrow var \;\; s\ \{q'\} \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q\end{array}}{\{p\}\ \mathbf{while}\ g\ \mathbf{:?!}\ inv\ \mathbf{:\#}\ var\ \mathbf{do}\ s\ \mathbf{od}\ \{q\}}\ \textsc{a-sent-while}$$

  Tal como en el caso del cálculo de postcondiciones se usa $inv \wedge g$ como p'. En este caso la respuesta es $p = inv$, por lo tanto siempre se sabe que $p \models inv$.

- Caso llamado a procedimiento. La precondición se calcula usando la siguiente regla:

$$\frac{1 \leq i \leq k}{\begin{array}{c}\mathrm{pre}(\mathbf{call}\ proc(cp_1, \ldots, cp_k),\; q) = \mathrm{P}_(\, proc)\lfloor p_i \mapsto cp_i \rfloor \;\wedge \\ (\exists a_1, \ldots, a_k (\mathrm{Q}_(\, proc)\lfloor p_i \mapsto a_i,\; p_i\mathbf{@pre} \mapsto cp_i \rfloor \Rightarrow q \lfloor cp_i \mapsto a_i \rfloor))\end{array}}\ \textsc{pre-call}$$

  La semántica abstracta en este caso está dada por:

$$\frac{\begin{array}{c}1 \leq i \leq k \quad p\lfloor cp_i \mapsto p_i \rfloor \models \mathrm{P}_(\, proc) \\ \exists o_1, \ldots, o_k\ (p\lfloor cp_i \mapsto o_i \rfloor \;\wedge\; \mathrm{Q}_(\, proc)\lfloor p_i \mapsto cp_i,\; p_i\mathbf{@pre} \mapsto o_i \rfloor) \models q\end{array}}{\{p\}\ \mathbf{call}\ proc(cp_1, \ldots, cp_k)\ \{q\}}\ \textsc{a-sent-call}$$

  En el caso del cálculo de precondiciones se cuantifica existencialmente los valores posteriores de los parámetros concretos usando $a_1, \ldots, a_k$. De modo análogo la semántica operacional utiliza $o_1, \ldots, o_k$ para cuantificar los valores anteriores de los parámetros concretos.

  Las sustituciones se hacen sobre lo que ya se sabía –o sea, p– en el caso de la semántica abstracta, o sobre las anotaciones del procedimiento llamado en el caso del cálculo de precondiciones. En ambos casos el resultado es equivalente.

  □

Bibliografía

[Amb77] Allen L. Ambler. GYPSY: A language for specification and implementation of verifiable programs. In *Language Design for Reliable Software*, pages 1–10, 1977.

[Aus04] Calvin Austin. J2SE 5.0 in a nutshell. May 2004.

[BB04] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[BdMS05] Clark Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability modulo theories competition. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the* 17^{th} *International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23. Springer-Verlag, July 2005. Edinburgh, Scotland.

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, USA, 2005. ACM.

[BLM05] Thomas Ball, Shuvendu K. Lahiri, and Madanlal Musuvathi. Zap: Automated theorem proving for software analysis. In *LPAR*, pages 2–22, 2005.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview, 2004.

[Bri02] W. Bright. The D programming language. *Dr. Dobb's Journal*, 27(2):36–40, 2002.

[CLQR07] S. Chatterjee, S.K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. 2007.

[DdM06] B. Dutertre and L. de Moura. The Yices SMT solver. *Available a t http://yices. csl. sri. com/, August*, 2006.

[DFS06] E. Denney, B. Fischer, and J. Schumann. An Empirical Evaluation of Automated Theorem Provers in Software Certification. *International Journal on Artificial Intelligence Tools*, 15(1):81–107, 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 *FOSE '07: 2007 Future of Software Engineering*, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[Dij97] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

[DL05] R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Technical Report MSR-TR-2005-70, Microsoft Research, 2005, 2005.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report #159, Palo Alto, USA, 1998.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[FL01] C. Flanagan and K. Leino. Houdini, an annotation assistant for ESC/Java, 2001.

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

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[FPB75] Jr. Fred P. Brooks. The mythical man-month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM Press.

[GLB75] Donald I. Good, Ralph L. London, and W. W. Bledsoe. An interactive program verification system. In *Proceedings of the international conference on Reliable software*, pages 482–492, New York, NY, USA, 1975. ACM Press.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HPF99] P. Hudak, J. Peterson, and J. H. Fasel. A gentle introduction to Haskell 98. 1999.

[JMR04] B. Jacobs, C. Marche, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. *Algebraic Methodology and Software Technology (AMAST)*, 2004.

[JP03] B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. *Software Security–Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS*, pages 4–6, 2003.

[Kin70] James Cornelius King. *A program verifier*. PhD thesis, 1970.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[LGvH+79] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford pascal verifier user manual. Technical report, Stanford, CA, USA, 1979.

[LHL+77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Not.*, 12(2):1–79, 1977.

[McK06] J. McKinna. Why dependent types matter. *Proc. ACM Symp. on Principles of Programming Languages (POPL 2006)*, pages 1–1, 2006.

[Mey97] Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Mit96] John C. Mitchell. *Foundations of programming languages*. MIT Press, Cambridge, MA, USA, 1996.

[MLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

[MPH00] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, 1785:63–77, 2000.

[MPMU04] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[Nec98] G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[NPS90] B.Ñordström, K. Petersson, and J.M. Smith. *Programming in Martin-Lof's type theory: an introduction*. Clarendon Press New York, NY, USA, 1990.

[PJ98] J. Palsberg and CB Jay. The essence of the Visitor pattern. *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 9–15, 1998.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.