DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA

ESPECIALIDAD EN PROGRAMACIÓN Y TECNOLOGÍA SOFTWARE

CURSO 2009/2010

Avances recientes en análisis estáticos de terminación

Autor: Agustín Daniel Delgado Muñoz

Director: Ricardo Peña Marí

Agradecimientos

Quisiera expresar mi agradecimiento en primer lugar a mi director de proyecto, Ricardo Peña, por todo el tiempo que ha dedicado en ayudarme y guiarme en el mundo de la investigación, por el apoyo recibido por su parte y por soportar estoicamente algunos plantones que le he dado en un año algo complicado para mi. Dentro del plano académico me gustaría también dar las gracias a la profesora Susana Nieva por creer en mi y animarme a continuar con mis estudios durante mi último año de carrera.

Aunque se suele recomendar que en este apartado de agradecimientos se evite mencionar a personas fuera del ámbito académico, no sería de justicia cumplir con este consejo puesto que tanto amigos, como familia, siempre me han animado y apoyado para seguir adelante tanto en los años durante la carrera como en este año de Máster. Por esta razón quiero mencionar aquí a mis amigos tanto de la facultad como los de San Lorenzo de El Escorial, sin hacer ninguna lista puesto que ellos saben perfectamente quienes son y sabrán darse por aludidos.

Por último tengo que darle las gracias a mi madre. Ella ha sabido estar siempre tanto en lo bueno, como sobre todo, en lo malo, dándome el apoyo y el impulso que necesitaba en todo momento. Ha sido mi principal soporte durante mi etapa universitaria y por esa razón, este trabajo también es en parte muy suyo. Va por ti también.

Resumen

En los últimos años han aparecido un número bastante considerable de trabajos sobre análisis de terminación de programas que han supuesto la base teórica de una serie de herramientas prácticas que han dado lugar a resultados bastante prometedores. Nos proponemos presentar algunas de las técnicas más recientes de detección de terminación de programas entre las que hemos destacado la técnica *Size-Change Termination* (*SCT*) propuesta por Ben-Amram, Lee y Jones [POPL' 01] con algunas de sus variantes, y otras técnicas basadas en metodos lineales como la propuesta por Podelski y Rybalchenko [VMCAI' 04] que tienen por objeto inferir funciones de rango para probar la terminación de los programas. Como resultado de nuestro trabajo hemos implementado algunas de estas técnicas con el objetivo de llevarlas a la práctica en el lenguaje funcional de primer orden *Safe*.

Abstract

In recent years, they have published a fairly large number of works about program termination analysis that have provided the theoretical basis of some practical tools leading to some promising results. We present some of the latest techniques for detecting program termination. We have highlighted the *Size-Change Termination* (*SCT*) framework proposed by Ben-Amram, Lee and Jones [POPL' 01] with some of its variants, and other techniques based on linear methods as proposed by Podelski and Rybalchenko [VMCAI' 04] whose aim is to infer ranking functions to prove program termination. As a result of our work, we have implemented some of these methods in order to apply them in the first-order functional language *Safe*.

Palabras clave

Análisis de terminación de programas, *SCT*, funciones de rango, métodos lineales, *Safe*, interpretación abstracta, invariantes.

Key words

Program termination analysis, *SCT*, ranking functions, linear methods, *Safe*, abstract interpretation, invariants.

ÍNDICE GENERAL

1. Capítulo 1: Introducción……9

2. Capítulo 2: Preliminares Matemáticos……..11

2.1. Órdenes y Puntos Fijos……12

2.2. Interpretación Abstracta……15

2.3. Conceptos básicos sobre Grafos……19

2.4. Poliedros……21

3. Capítulo 3: Size-Change Termination (*SCT*)……24

3.1. *SCT*……25

3.2. *SCP*……32

3.3. δ*SCT*……..39

3.4. *SCNP*……44

4. Capítulo 4: Métodos Lineales……..52

4.1. Inferencia de Invariantes……53

4.2. Inferencia de Funciones de Rango……..57

5. Capítulo 5: Una aplicación a la Programación Funcional……64

5.1. El lenguaje funcional *Safe*……65

5.2. Síntesis de Invariantes……..66

5.3. Síntesis de Funciones de Rango……69

6. Capítulo 6: Conclusiones y Trabajo Futuro……72

7. Bibliografía……74

8. Autorización……76

9. Apéndice A: Código fuente de la aplicación……77

10. Apéndice B: Artículo presentado en [IFL' 10]……95

Capítulo 1

Introducción

La terminación es una de las propiedades más deseables que pedimos a los programas informáticos. Por desgracia su detección es un problema indecidible por lo que no existe un algoritmo genérico que sea capaz de resolver la cuestión para cualquier programa. Por este motivo, la forma de abordar esta cuestión es detectar clases de programas que cumplan una serie de restricciones que hagan decidible el problema y diseñar análisis de terminación sobre dichas clases de programas.

Para afrontar la terminación de programas se tienen que tener en cuenta distintos factores. Uno de ellos, es la relación de órden entre los datos que intervienen. En los análisis de terminación suelen suponerse órdenes bien fundados puesto que, como veremos durante este trabajo, esta propiedad de buena fundamentación facilita la detección de cómputos infinitos que nos conducirían a probar la no terminación de los programas. Otro factor a tener en cuenta es la representación abstracta que podemos hacer de los programas. En este trabajo veremos diferentes técnicas en las que se tomaran clases de programas que pueden representarse de diferentes formas.

Uno de los métodos más populares de detección de terminación es el de encontrar una *función de rango* (*ranking function*) de un programa. La existencia de estas funciones nos aseguran la propiedad de terminación por lo que muchos de los estudios que se han hecho en relación con la terminación de programas tratan de la inferencia de este tipo de funciones ([SAS' 10], [TOPLAS' 09], [LMCS' 09], [TACAS' 08]).

En el Capítulo 2 de este trabajo presentaremos la base matemática en la que se basan todas las técnicas que presentaremos posteriormente. El propósito de este apartado es el de conseguir que este documento sea lo más autocontenido posible.

Un artículo clásico en este campo es el de Ben-Amram, Lee y Jones [POPL '01] en el que se nos presenta la técnica *SCT (Size-Change Termination)* de terminación de programas, basada en la representación de los mismos como conjuntos de grafos donde cada grafo representa una llamada dentro del programa. En este marco, la forma de abordar los ciclos será tratar las componentes conexas de estos grafos. A partir de [POPL' 01] han aparecido en los últimos años distintos trabajos que han dado la base teórica a diferentes herramientas de terminación de costes computacionales aceptables. Esta familia de técnicas de detección de terminación de programas la abordaremos en el Capítulo 3 de este trabajo.

El principal escollo al que nos enfrentamos al analizar la terminación de los programas reside en los bucles y en las llamadas recursivas. Un enfoque para tratarlos consiste en tomar las relaciones de tamaño de los parámetros de los programas. Las relaciones de tamaño son una abstracción consistente en identificar cada uno de los parámetros por sus tamaños y nos informa de la forma en la que crecen o decrecen cada uno de ellos y de las relaciones existentes entre ellos. Estas relaciones de tamaño podemos obtenerlas a partir de la inferencia de los invariantes de los bucles, por lo que explicaremos como tema lateral a este trabajo un método de síntesis de invariantes basado en interpretación abstracta sobre poliedros, o visto de otro modo, sobre conjuntos de restricciones lineales. En el Capítulo 4 veremos técnicas de terminación que utilizan este modelo cuando dichas relaciones de tamaño son lineales. Entre ellas destacamos la dada por Podelski y Rybalchenko [VMCAI' 04] por tratarse de un método completo para este tipo de programas.

En el Capítulo 5, explicaremos cómo hemos implementado en el lenguaje de programación lógico Prolog el método antes mencionado de síntesis de invariantes y la técnica de inferencia de funciones de rango propuesta por Podelski y Rybalchanko en [VMCAI '04] obteniendo algunos resultados interesantes. Nuestra intención es la aplicación práctica de estos métodos al lenguaje funcional de primer orden *Safe* y en particular sobre su versión *Core-Safe* sobre la cual se realizan otro tipo de análisis. Este trabajo lo hemos plasmado en el artículo [IFL' 10] presentado a principios de septiembre de 2010 y que adjuntamos en el Apéndice B de esta memoria. El código fuente del programa Prolog viene adjuntado en el Apéndice A de este trabajo.

Por último, hemos de destacar que aunque la terminación de programas siempre se ha visto como un tema muy ligado a la teoría, tiene también aplicaciones prácticas. Por ejemplo, muchos comprobadores de modelos incluyen un detector de terminación, se trata de una propiedad relevante en software crítico, o tiene aplicación también en el campo del código con certificado o código con demostración asociada (*Proof Carrying Code*, o *PCC*) que ha experimentado un gran avance en los últimos años.

Capítulo 2

Preliminares Matemáticos

El objetivo de este capítulo es conseguir que este trabajo sea lo más autocontenido posible por lo que presentaremos la notación y los conceptos matemáticos fundamentales que se utilizarán en los capítulos posteriores.

Pasaremos a presentar conceptos sobre órdenes, puntos fijos, interpretación abstracta y algunos conceptos básicos sobre grafos y poliedros.

La parte referida a órdenes y puntos fijos, tratada en el apartado 2.1, constituye el soporte matemático básico de los análisis de terminación que estudiaremos. La interpretación abstracta, tratada en el apartado 2.2, constituye de por sí una técnica de análisis estático de programas que hemos utilizado para poder desarrollar un algoritmo que extrae invariantes de programas escritos en un lenguaje funcional de primer orden, que detallaremos en el Capítulo 5. Los grafos son estructuras bien conocidas en el ámbito informático y existe mucha bibliografía, muy extensa, sobre ellos, siendo tratados en cualquier libro de matemática discreta o de estructuras de datos. En el apartado 2.3 nos limitaremos a dar algunos de los conceptos más elementales sobre ellos y que nos serviran para poder entender mejor la técnica de análisis de terminación *SCT* que trataremos en el Capítulo 3. Los poliedros son utilizados en una técnica de síntesis de invariantes de programas que explicaremos en el Capítulo 4. En el apartado 2.4 explicaremos de qué manera puede verse un programa puede interpretarse en un dominio abstracto que es un retículo de poliedros, definiremos el concepto de poliedro de manera rigurosa y justificaremos por qué la técnica de interpretación abstracta puede hacer uso de ellos.

2.1. Órdenes y Puntos Fijos

Una relación n-ária R sobre los conjuntos $A_1, A_2, \dots, A_n$ es un subconjunto del producto cartesiano $A_1 \times A_2 \times \dots \times A_n$. Cuando todos estos conjuntos son iguales, sea $A = A_1 = A_2 = \cdots = A_n$, diremos que R es una relación n-ária sobre el conjunto A. El caso más común de relación, es el de relación binaria, definida sobre dos conjuntos, para el cuál se denotará xRy para indicar que los elementos x e y están relacionados mediante la relación R.

Un orden, como pasamos a ver en la siguiente definición, es una relación que cumple una serie de propiedades:

Definición 2.1: Sea A un conjunto no vacío y $\sqsubseteq$ una relación binaria sobre A. El par $(A, \sqsubseteq)$ es un **orden parcial** (o **poset**) si cumple las siguientes propiedades:

1. Reflexividad: $\forall x \in A: x \sqsubseteq x$
2. Antisimetría: $\forall x, y \in A: x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
3. Transitividad: $\forall x, y, z \in A: x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$

Dado un poset $(A, \sqsubseteq)$, definimos su **parte estricta** $(A, \sqsubset)$ de la siguiente manera: $x \sqsubset y \overset{def}{\iff} x \sqsubseteq y \wedge x \neq y$.

Se dice que un **orden** $(A, \sqsubseteq)$ es **total** cuando todos sus elementos pueden relacionarse entre si, i.e. $\forall x, y \in A: x \sqsubseteq y \vee y \sqsubseteq x$. Diremos que $(A, \sqsubseteq)$ es un **orden bien fundado o bien fundamentado** si todo subconjunto no vacío de A posee un **elemento minimal**, esto es, un elemento $x \in A$ tal que $\nexists y \in A: y \sqsubset x$.

Dado un *poset* $(A, \sqsubseteq)$, diremos que una **cadena** es un subconjunto $X \subseteq A$ tal que $(X, \sqsubseteq)$ es un orden total. Se dice que un elemento $u \in A$ es una **cota superior** de $X \subseteq A$ si verifica que $\forall x \in X: x \sqsubseteq u$ y se define el **supremo** del conjunto X, si este existe, como su mínima cota superior, denotado $\bigsqcup X = min\{u \in X | \forall x \in X: x \sqsubseteq u\}$. Analogamente, diremos que un elemento $u \in A$ es una **cota inferior** de $X \subseteq A$ si verifica que $\forall x \in X: u \sqsubseteq x$ y se define el **ínfimo** del conjunto X, si este existe, como su máxima cota inferior, denotado $\bigsqcap X = max\,\{u \in A | \forall x \in X: u \sqsubseteq x\}$.

Para nuestros propósitos nos interesará que los órdenes con los que trabajamos cumplan algunas otras propiedades adicionales. En particular estamos interesados en los *órdenes parciales completos* (abreviadamente *cpos*) y en los *retículos completos* que pasamos a definir seguidamente:

Definición 2.2: Un **orden parcial completo (*cpo*)** es un orden parcial $(A, \sqsubseteq)$ en el que para toda cadena $X \subseteq A$ existe su supremo $\bigsqcup X$. Si además posee un elemento mínimo que denotaremos $\bot \in A$, diremos que es un **cpo apuntado**.

Definición 2.3: Un **retículo completo** es una estructura $(A, \sqsubseteq, \bot, \top, \bigsqcup, \bigsqcap)$ tal que $(A, \sqsubseteq)$ es un *poset* en el que todo subconjunto $X \subseteq A$ tiene supremo $\bigsqcup X$ e ínfimo $\bigsqcap X$ y que en particular cumple $\bot = \bigsqcup \emptyset = \bigsqcap A$ y $\top = \bigsqcup A = \bigsqcap \emptyset$.

En el caso de los retículos, solamente se pide que exista supremo e ínfimo para cada par de elementos del dominio.

De las definiciones anteriores se pueden extraer fácilmente las siguientes propiedades de los *posets*, los *cpos* y los retículos:

1. Todo *poset* finito es un *cpo*.
2. Todo retículo finito es un retículo completo
3. Todo retículo completo es un *cpo*.

Las siguientes definiciones tratan sobre puntos fijos y nos servirán como apoyo para entender los conceptos que presentaremos sobre interpretación abstracta posteriormente:

Definición 2.4: Dados dos *posets* $(A_1, \sqsubseteq_1)$, $(A_2, \sqsubseteq_2)$, diremos que una **función** $f: A_1 \to A_2$ es **monótona** si preserva el orden, i.e. $\forall x, y \in A_1 : x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y)$.

Definición 2.5: Dados dos *cpos* $(A_1, \sqsubseteq_1)$, $(A_2, \sqsubseteq_2)$, diremos que una **función** $f: A_1 \to A_2$ es **contínua** si para toda cadena $X \subseteq A_1$ se cumple $f(\bigsqcup_1 X) = \bigsqcup_2 f(X)$.

Se cumplen las siguientes propiedades:

1. Toda función contínua es monótona.
2. $f: A_1 \to A_2$ monótona y A_1 finito $\Rightarrow f$ contínua.

Definición 2.6: Dada una función $f: A \to A$, diremos que un elemento $x \in A$ es un **punto fijo** de f si $f(x) = x$. Diremos que es el **mínimo punto fijo** de f, denotado $lfp(f)$ si $x \sqsubseteq y$ para cualquier otro punto fijo y de f. Diremos que es el **máximo punto fijo** de f, denotado $gfp(f)$ si $y \sqsubseteq x$ para cualquier otro punto fijo y de f.

El siguiente teorema nos dice cómo calcular el menor punto fijo cuando trabajamos con *cpos* y funciones contínuas:

Teorema 2.7: Dados un *cpo* apuntado $(A, \sqsubseteq)$ y una función contínua $f: A \to A$, entonces $lfp(f) = \bigsqcup \{f^n(\bot)\}_{n \geq 0}$.

La cadena $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots \sqsubseteq f^n(\bot) \sqsubseteq \cdots$ a la que hace referencia el resultado anterior se denomina *cadena ascendente de Kleene*.

Uno de los resultados más importantes en la teoría de puntos fijos es el *Teorema de Tarski*. Este teorema nos establece un resultado más general puesto que supone retículos completos y funciones monótonas.

Teorema 2.8 (Teorema de Tarski): Dados un retículo completo $(A, \sqsubseteq, \bot, \top, \bigsqcup, \bigsqcap)$ y una función monótona $f: A \to A$, se cumple lo siguiente:

1. El conjunto $f^{=} = \{x \in A | f(x) = x\}$ de los puntos fijos de f, forman un retículo completo.
2. $lfp(f) = \bigsqcap f^{=} = \bigsqcap f^{\sqsubseteq}$, donde $f^{\sqsubseteq} = \{x \in A | f(x) \sqsubseteq x\}$ es el conjunto de post-puntos fijos de f o *zona reductiva* del retículo completo $f^{=}$.
3. $gfp(f) = \bigsqcup f^{=} = \bigsqcup f^{\sqsupseteq}$ donde $f^{\sqsupseteq} = \{x \in A | f(x) \sqsupseteq x\}$ es el conjunto de pre-puntos fijos de f o *zona extensiva* del retículo completo $f^{=}$.

Vemos, por último, unas pocas propiedades interesantes sobre los *cpos*:

1. El conjunto de funciones contínuas entre dos cpos $(A_1, \sqsubseteq_1)$, $(A_2, \sqsubseteq_2)$, denotado $[A_1 \to A_2]$ es un cpo donde $f \sqsubseteq g \overset{def}{\iff} \forall x \in A_1 : f(x) \sqsubseteq_2 g(x)$.
2. Dado un cpo $(A, \sqsubseteq)$, $f : A \to A$ contínua, $x \in f^{\sqsubseteq}$, $x \sqsupseteq lfp(f)$, la cadena $x \sqsupseteq f(x) \sqsupseteq f^2(x) \sqsupseteq \cdots \sqsupseteq f^n(x) \sqsupseteq \cdots$ cumple $\forall n \in \mathbb{N} : f^n(x) \sqsupseteq lfp(f)$ aunque su ínfimo no coincida necesariamente con $lfp(f)$.
3. Dado un cpo $(A, \sqsubseteq)$, $f : A \to A$ contínua, $x \in f^{\sqsupseteq}$, $x \sqsubseteq lfp(f)$, la cadena $x \sqsubseteq f(x) \sqsubseteq f^2(x) \sqsubseteq \cdots \sqsubseteq f^n(x) \sqsubseteq \cdots$ cumple $\forall n \in \mathbb{N} : f^n(x) \sqsubseteq lfp(f)$ aunque su supremo no coincida necesariamente con $lfp(f)$.

2.2. Interpretación Abstracta

La interpretación abstracta es una teoría basada en semántica denotacional cuyo principal objetivo es realizar análisis estáticos de programas que extraigan información segura sobre el comportamiento dinámico de estos sin tener la necesidad de ejecutarlos.

Su base matemática básica parte de lo expuesto en el anterior apartado. Aquí presentaremos dos conceptos fundamentales para esta técnica: conexiones de Galois y operadores de ensanchamiento.

El marco matemático de la interpretación abstracta puede utilizarse para aproximar puntos fijos como veremos más adelante. En el Capítulo 5, explicaremos cómo hemos utilizado interpretación abstracta sobre poliedros dados como conjuntos de restricciones lineales para extraer invariantes de programas funcionales de primer orden mediante un algoritmo de punto fijo que tras una serie de iteraciones aplica un operador de ensanchamiento para finalizar.

Partiremos de un *cpo* (o en algunas ocasiones, un retículo completo) $(A^b, \sqsubseteq^b)$ que describe la *semántica de recolección de propiedades*, la cual es la semántica más precisa que puede concebirse por el estudio de una cierta propiedad. El orden $\sqsubseteq^b$ representa la precisión relativa entre propiedades.

Se definirá otro *cpo* (o retículo completo) $(A^{\#}, \sqsubseteq^{\#})$ que describirá la *semántica abstracta* que puede entenderse como una aproximación de la semántica de recolección de propiedades. El orden $\sqsubseteq^{\#}$ representa la precisión relativa de las propiedades abstractas.

La semántica de una propiedad abstracta viene definida por una función denominada *función de concreción*, $\gamma: A^{\#} \to A^{b}$ mientras que la aproximación de una propiedad concreta viene dada por una función llamada *función de abstracción* $\alpha: A^{b} \to A^{\#}$. Estas dos funciones deben formar lo que se denomina una *conexión de Galois*:

Definición 2.9: Dados dos posets $(A^{b}, \sqsubseteq^{b})$, $(A^{\#}, \sqsubseteq^{\#})$, una **conexión de Galois** es un par de funciones (α, γ) tales que $\forall a^{b} \in A^{b} \forall a^{\#} \in A^{\#}: \alpha(a^{b}) \sqsubseteq^{\#} a^{\#} \Leftrightarrow a^{b} \sqsubseteq^{b} \gamma(a^{\#})$. Utilizaremos la notación $A^{b}(\sqsubseteq^{b}) \leftrightarrows_{\alpha}^{\gamma} A^{\#}(\sqsubseteq^{\#})$ para denotar la conexión de Galois anterior.

A continuación enunciamos algunas propiedades interesantes de las conexiones de Galois. Supongamos que (α, γ) es una conexión de Galois, entonces:

1. $\gamma \circ \alpha \sqsupseteq id^{b}$, donde id^{b} es la función identidad en A^{b}. De otra forma, $\gamma \circ \alpha$ es *extensiva*.
2. $\alpha \circ \gamma \sqsubseteq id^{\#}$ donde$id^{\#}$ es la función identidad en $A^{\#}$. De otra forma, $\alpha \circ \gamma$ es *reductiva*.
3. α y γ son funciones monótonas.
4. (α, γ) conexión de Galois $\Leftrightarrow$ Se cumplen 1., 2. Y 3.

La propiedad 1. nos dice que la abstracción puede ocasionar pérdida de precisión de forma segura, mientras que la propiedad 2. nos dice que en la concrección nunca se produce pérdida de precisión.

En una conexión de Galois, α es sobreyectiva si y sólo si γ es inyectiva, si y sólo si $\alpha \circ \gamma = id^{\#}$, en cuyo caso se denomina *sobreyección o inserción de Galois*.

Como hemos comentado anteriormente, este marco matemático puede utilizarse para aproximar puntos fijos. Se aproximarán puntos fijos de la semántica concreta haciendo el cálculo de puntos fijos en la semántica abstracta. Sea $F^{b}: A^{b} \to A^{b}$ la función que calcula la semántica concreta. Queremos calcular el punto $X = F^{b}(X)$.

Supongamos $lfp(F^b) = \bigsqcup_{n\geq 0}^{b} {F^b}^n(\bot^b)$. Sea $A^\#(\sqsubseteq^\#, \bigsqcup^\#)$ el *cpo* abstracto. Queremos obtener $\bot^\#$ y $F^\#$ tales que $\alpha(lfp(F^b))= \bigsqcup_{n\geq 0}^{\#} {F^\#}^n(\bot^\#)$. Para ello haremos las siguientes suposiciones:

a) $\alpha(\bot^b) = \bot^\#$
b) $\alpha\Big(F^b(a^b)\Big) = F^\#\Big(\alpha(a^b)\Big)\ \forall a^b \in A^b$
c) $F^\# = \alpha \circ F^b \circ \gamma\ \wedge \forall a^b \in A^b: \gamma \circ \alpha(a^b) = a^b$, i.e. (α, γ) es una inyección de Galois.

Las siguientes propiedades nos aseguran aproximaciones de puntos fijos:

1. Sean $(A^b, \sqsubseteq^b, \bot^b, \top^b, \bigsqcup^b, \bigsqcap^b)$, $(A^\#, \sqsubseteq^\#, \bot^\#, \top^\#, \bigsqcup^\#, \bigsqcap^\#)$ retículos completos, (α, γ) una conexión de Galois y $F^b: A^b \to A^b$ una función monótona. Entonces se verifica $\alpha\Big(lfp(F^b)\Big) \sqsubseteq^\# lfp(\alpha \circ F^b \circ \gamma)$.
2. Sean $(A^b, \sqsubseteq^b, \bigsqcup^b)$, $(A^\#, \sqsubseteq^\#, \bigsqcup^\#)$ *cpos*, (α, γ) una conexión de Galois, $F^b: A^b \to A^b$ y $F^\#: A^\# \to A^\#$ funciones contínuas tales que
   i) $\alpha(\bot^b) = \bot^\#$
   ii) $\forall a^\# \in A^\#: \alpha \circ F^b \circ \gamma(a^\#) \sqsubseteq^\# F^\#(a^\#)$

   Entonces se verifica que $\alpha\Big(lfp(F^b)\Big) \sqsubseteq^\# lfp(F^\#)$

Deben demostrarse i) y ii) de la propiedad 2. para asegurar puntos fijos seguros. Tanto en el caso en el que la función semántica abstracta $F^\#$ se calcula a partir de la función semántica concreta F^b como en el caso en que es una aproximación superior, el punto fijo calculado en el dominio abstracto aproxima desde arriba al punto fijo calculado en el dominio concreto.

Otro concepto importante en interpretación abstracta es el de operador de ensanchamiento. Estos operadores se usan para acelerar el cálculo de puntos fijos en dominios muy grandes o para asegurar la terminación de los análisis cuando los dominios abstractos poseen cadenas ascendentes infinitas.

Un operador de ensanchamiento construye una secuencia ascendente que finalmente se estabiliza en un punto situado con seguridad por encima del mínimo punto fijo.

Definición 2.10: Un **operador de cota superior** $\breve{\bigsqcup}: L \times L \to L$ satisface que $\forall l_1, l_2 \in L: l_1 \sqsubseteq (l_1 \breve{\bigsqcup} l_2) \sqsupseteq l_2$

Partiendo de una secuencia cualquiera $\{l_n\}_{n\in\mathbb{N}}$ donde $\forall i \in \mathbb{N}: l_i \in L$ y de un operador $\breve{\sqcup}$, podemos construir la siguiente secuencia $\{l_n^{\breve{\sqcup}}\}_{n\in\mathbb{N}}$ de la siguiente manera:

$$l_0^{\breve{\sqcup}} = l_0$$

$$l_{n+1}^{\breve{\sqcup}} = l_n^{\breve{\sqcup}} \breve{\sqcup} l_{n+1}$$

Esta secuencia cumple las siguientes propiedades:

1. $\{l_n^{\breve{\sqcup}}\}_{n\in\mathbb{N}}$ es una secuencia ascendente.
2. $\forall n \in \mathbb{N}: l_n^{\breve{\sqcup}} \sqsupseteq \bigsqcup \{l_0, l_1, \dots, l_n\}$

Definición 2.11: Un **operador de ensanchamiento** $\nabla: L \times L \to L$ satisface:

1. ∇ es un operador de cota superior.
2. Para toda cadena ascendente $\{l_n\}_{n\in\mathbb{N}}$, la cadena $\{l_n^{\nabla}\}_{n\in\mathbb{N}}$ finalmente se estabiliza.

Sea $f: L \to L$ monótona, y sea $\nabla: L \times L \to L$ un operador de ensanchamiento. Construimos la secuencia ascendente $\{f_\nabla^{\mathrm{n}}\}_{n\in\mathbb{N}}$ de la siguiente manera:

$$f_\nabla^0 = \bot$$

$$f_\nabla^{n+1} = f_\nabla^n \; si \; f(f_\nabla^n) \sqsubseteq f_\nabla^n \; (i.e., si \; f \in f^{\sqsubseteq})$$

$$f_\nabla^{n+1} = f_\nabla^n \; \nabla \; \mathrm{f}(f_\nabla^n) \; en \; otro \; caso$$

Se verifican las siguientes propiedades:

1. La secuencia $\{f_\nabla^{\mathrm{n}}\}_{n\in\mathbb{N}}$ es ascendente
2. La secuencia $\{f_\nabla^{\mathrm{n}}\}_{n\in\mathbb{N}}$ finalmente se estabiliza
3. $\exists m > 0: f(f_\nabla^m) \sqsubseteq f_\nabla^m \; \wedge \; \forall n > m: \bigsqcup \{f_\nabla^{\mathrm{n}}\}_{n\in\mathbb{N}} = f_\nabla^m$
4. $\bigsqcup \{f_\nabla^{\mathrm{n}}\}_{n\in\mathbb{N}} \sqsupseteq lfp(f)$

Así pues, la propiedad 4. nos asegura que mediante un operador de ensanchamiento y trabajando con funciones monótonas y retículos completos, llegamos a una aproximación al mínimo punto fijo de una función f. Es más, sabemos que dicha aproximación es f_∇^m por la propiedad 3. Puesto que es f_∇^m es reductiva en ese punto ($f(f_\nabla^m) \sqsubseteq f_\nabla^m$), podemos construir la secuencia descendiente $\{f^n(f_\nabla^m)\}_{n\in\mathbb{N}}$ que por el teorema de Tarski, se mantiene siempre por encima de $lfp(f)$.

2.3. Conceptos básicos sobre Grafos

En este apartado presentaremos algunos de los conceptos más elementales sobre grafos que aparecerán a lo largo del trabajo, especialmente en el Capítulo 3, donde explicaremos la técnica *SCT* de análisis de terminación de programas, que hace uso de estas estructuras para representar llamadas de una función a otra. En primer lugar definiremos lo que es un *grafo*:

Definición 2.13: Un **grafo** es un par ordenado $G = (V, E)$ donde V es un conjunto no vacío de *nodos* o *vértices* y $E \subseteq V \times V$ es un conjunto de pares de nodos, denominados *arcos o arístas*. Cuando las arístas son pares ordenados, diremos que el **grafo** es **dirigido o** es un **digrafo**, mientras que cuando las arístas sean pares no-ordenados, diremos que el **grafo** es **no-dirigido**. Dado un conjunto T de *etiquetas*, si $E \subseteq V \times T \times V$, diremos que E es un conjunto de *arístas o arcos etiquetados* y que G es un **grafo etiquetado**.

Como veremos en el siguiente capítulo, la técnica *SCT* utiliza conjuntos de grafos para representar programas. Cada uno de estos grafos representará cada una de las llamadas del programa. Los nodos de estos grafos serán los parámetros de las llamadas externa e interna y las arístas nos indicarán si hay decrecimiento en el valor de cada uno de los parámetros entre ambas llamadas por lo que sólo existirán arístas entre los parámetros de la primera con los parámetros de la segunda. Formalmente, esto puede expresarse mediante el concepto de *grafo bipartito*:

Definición 2.14: Sea $G = (V, E)$ un **grafo**. Diremos que G es **bipartito** si se cumple que $V = V_1 \cup V_2, \quad V_1 \cap V_2 = \emptyset, \quad \forall x_1, x_2 \in V_1 \forall y_1, y_2 \in V_2 : (x_1, x_2) \notin E \wedge (y_1, y_2) \notin E$. Utilizaremos la notación $G = (V_1, V_2, E)$ para referirnos al grafo bipartito anterior.

En algunas ocasiones nos interesará estudiar solamente una parte de un grafo. Por esta razón debemos definir el concepto de *subgrafo*:

Definición 2.15: Sea $G = (V, E)$ un grafo. Dado un conjunto $\emptyset \neq W \subseteq V$, el **subgrafo** generado por W es $G_W = (W, E_W)$ donde $E_W = \{(e, e') \in E | \, e, e' \in W\}$.

El principal foco de problemas en el análisis de terminación de programas son los bucles de los mismos. Para los grafos usados por *SCT* para representar programas, los bucles se corresponden con sus componentes fuertemente conexas. Para formalizar este concepto de teoría de grafos requerimos definir previamente otros elementos:

Definición 2.16: Sea $G = (V, E)$ un grafo. Una **cadena** de longitud q en G es una sucesión de q arístas $\mu = \{e_1, e_2, \dots, e_q\}$ tales que $\forall k \in \{1,2, \dots q-1\}: e_k \neq e_{k+1}$ y $\forall k \in \{2,3, \dots q-1\}: (e_{k-1}, e_k) \in E \ \wedge (e_k, e_{k+1}) \in E$.

Definición 2.17: Se dice que un **grafo** $G = (V, E)$ es **conexo** si $\forall v_1, v_2 \in V$ tales que $v_1 \neq v_2$, existe una cadena que une v_1 y v_2.

Ahora ya podemos definir formalmente lo que es una componente fuertemente conexa de un grafo:

Definición 2.18: Una **componente fuertemente conexa** de un grafo G es un subgrafo conexo y maximal, i.e., no es subgrafo de ningún otro subgrafo de G.

Un concepto que aparecerá posteriormente es el de orden topológico de un grafo. En primer lugar damos una definición general de *orden topológico*:

Definición 2.19: Sea $(A, \sqsubseteq)$ un orden parcial. Diremos que $(A, \sqsubseteq_T)$ es un **orden topológico**, si es un orden total y si se cumple que $\forall x, y \in A: x \sqsubseteq y \Rightarrow x \sqsubseteq_T y$.

En general, el conjunto de vértices de los grafos suele ser finito, y en particular lo serán para los grafos con los que trabajaremos posteriormente. Además, trabajaremos con órdenes sobre los elementos de los vértices de los grafos. Por estas dos razones, el siguiente resultado nos justificará por qué podemos utilizar el orden topológico sobre los grafos que usaremos posteriormente:

Teorema 2.20: Si $(A, \sqsubseteq)$ un orden parcial y A es finito, entonces admite un orden topológico.

El orden topológico se aplica en los grafos sobre su conjunto de vértices V. El orden $\sqsubseteq$ de la definición anterior entre los vértices puede definirse como sigue:

$\forall v, v' \in V: v \sqsubseteq v' \Leftrightarrow$ existe una cadena que une a v con v'

2.4. Poliedros

En este apartado presentaremos conceptos básicos sobre poliedros puesto que aparecerán en el Capítulo 4 donde explicaremos una técnica de inferencia de invariantes basada en interpretación abstracta sobre ellos. Denotaremos por $\mathbb{R}^n$ al espacio n-dimensional sobre el conjunto de los números reales $\mathbb{R}$, y por $\bar{x} = (x_1, x_2, \ldots, x_n)$ a los elementos de $\mathbb{R}^n$. Los poliedros pueden verse como conjuntos de puntos de $\mathbb{R}^n$ tales que cumplen una serie de inecuaciones lineales por lo que utilizaremos la notación $\{\bar{x} \in \mathbb{R}^n \mid c_1 \wedge c_2 \wedge \ldots \wedge c_m\}$ para representarlos, donde cada c_i representa cada una de las inecuaciónes lineales que deben cumplir los puntos del poliedro. Más compactamente, podemos representar los poliedros simplemente por el conjunto de las inecuaciones lineales que deben cumplir sus puntos, i.e., $\{c_1 \wedge c_2 \wedge \ldots \wedge c_m\}$. Como veremos más adelante, los programas pueden verse de esta misma forma donde dichas restricciones lineales representan las relaciones de tamaño entre los parámetros. En general, se podrá representar los programas como conjuntos de poliedros con este significado. Por cada caso base y cada caso recursivo de un programa habrá un poliedro que represente las relaciones de tamaño de los parámetros correspondientes.

Una vez explicada de forma intuitiva de lo que entenderemos por un poliedro, pasaremos a dar una definición formal. Previamente requerimos de la siguiente colección de conceptos que pasamos a ver a continuación:

Definición 2.21: Dados $\bar{x}$, $\bar{x}' \in \mathbb{R}^n$ se define y denota su **producto escalar** como $< \bar{x}, \bar{x}' > = \sum_{i=1}^{n} x_i {x_i}'$.

Definición 2.22: Dado $\lambda > 0$, se define la **multiplicación escalar positiva de un conjunto de puntos** $S \subseteq \mathbb{R}^n$ como $\lambda S = \{\lambda \bar{x} \mid \bar{x} \in S\}$.

Definición 2.23: Un **conjunto de puntos** $S \subseteq \mathbb{R}^n$ es **afín** si y sólo si $\forall \bar{x}, \bar{y} \in S \; \forall \lambda \in \mathbb{R}: (1-\lambda)\bar{x} + \lambda\bar{y} \in S$

Definición 2.24: Sea $S \subseteq \mathbb{R}^n$. Diremos que la envolvente afín de S, denotada $aff(S)$ es el menor conjunto afín contenido en S.

Definición 2.25: Un **hiperplano** es un conjunto afín de dimensión $(n-1)$ en $\mathbb{R}^n$.

Definición 2.26: Diremos que la suma $\lambda_1\overline{x_1} + \lambda_2\overline{x_2} + \cdots + \lambda_k\overline{x_k}$ es una **combinación convexa** de $\overline{x_1}, \overline{x_2}, \ldots, \overline{x_k}$ si y sólo si $\forall i \in \{1,2,\ldots k\}: \lambda_i > 0 \wedge \sum_{i=1}^{k} \lambda_i = 1$.

Definición 2.27: Diremos que $S \subseteq \mathbb{R}^n$ es un **conjunto convexo** si y sólo si contiene a todas las combinaciones convexas de sus elementos.

Definición 2.28: Sea $S \subseteq \mathbb{R}^n$. Diremos que la **envolvente convexa (*convex hull*)** de S es $conv(S) = \cap \{S' \subseteq \mathbb{R}^n \mid S \subseteq S' \wedge S' \text{ es un conjunto convexo}\}$.

Definición 2.29: La **bola abierta** de centro $\bar{x}$ y radio $\delta \in \mathbb{R}$ se define y se denota como $\mathcal{B}(\bar{x}, \delta) = \{\bar{y} \in \mathbb{R}^n \mid d(\bar{x}, \bar{y}) < \delta\}$ donde $\delta > 0$ y $d(\bar{x}, \bar{y})$ es la distancia euclídea entre dos puntos de $\mathbb{R}^n$.

Definición 2.30: Sean $S \subseteq \mathbb{R}^n$, $\bar{x} \in S$. Diremos que $\bar{x}$ es un **punto interior** de S si y sólo si $\exists \delta > 0: \mathcal{B}(\bar{x}, \delta) \subset S$. El **conjunto** S es **abierto** si y sólo si todo $\bar{x} \in S$ es un punto interior de S.

Definición 2.31: Un **conjunto** S es **cerrado** si y sólo si su complementario es abierto. El cierre de un conjunto S se denota y define como $cl(S) = \cap \{S' \subseteq S \mid S' cerrado\}$.

Definición 2.32: Sean $\bar{x} \in \mathbb{R}^n$, $\bar{\mu} \in \mathbb{R}^n$ tal que $\bar{\mu} \neq \bar{0}$, y $\eta \in \mathbb{R}$. Un **semiespacio cerrado** en $\mathbb{R}^n$ es un conjunto de puntos $\{\bar{x} \mid < \bar{x}, \bar{\mu} > \leq \eta\}$.

Con todos estos conceptos ya estamos en disposición de definir formalmente lo que entenderemos por poliedro:

Definición 2.33: Un **poliedro** es un conjunto de puntos en $\mathbb{R}^n$ que puede expresarse como la intersección de una colección finita de semiespacios cerrados, o de otro modo, es la conjunto solución de algún sistema finito de inecuaciones lineales de la forma $\{\bar{x} \mid < \bar{x}, \bar{\bar{\mu}_i} > \leq \eta_i\}$, donde $i \in \{1,2, \dots, m\}$.

Notaremos por $Poly^n$ al conjunto de los poliedros de $\mathbb{R}^n$.

Como hemos comentado anteriormente, en el Capítulo 4 veremos una técnica de síntesis de invariantes basada en interpretación abstracta sobre el conjunto de los poliedros. El siguiente resultado nos justifica el uso de la técnica de interpretación abstracta en este contexto:

Proposición 2.34: $(Poly^n, \subseteq, \overline{\cup}, \cap)$ es un retículo donde $\overline{\cup}$ es el operador de envolvente convexa, el elemento mínimo es el conjunto vacío y el elemento máximo es $\mathbb{R}^n$.

Capítulo 3

Size-Change Termination (*SCT*)

En este apartado presentamos el marco de *terminación por cambio de tamaño* (en inglés, *Size-Change Termination*, abrev. *SCT*) que ha dado lugar a numerosas publicaciones sobre terminación de programas en los últimos años y ha servido como fundamento teórico de varias herramientas de detección de terminación.

La idea básica del método consiste en representar los programas mediante un conjunto de grafos, denominados *grafos de cambio de tamaño*, que nos ofrecen información relevante de cada una de las llamadas recursivas de un cierto programa. En particular, estos grafos nos informan sobre la relación de tamaño entre los parámetros de la llamada interna y la llamada externa, indicando en concreto si estos decrecen o no. Si para toda secuencia infinita de llamadas se cumple que existe al menos un parámetro que decrece infinitamente, suponiendo un orden bien fundado en el conjunto de valores de los datos, se puede garantizar la terminación de dicho programa.

En el apartado 3.1. presentaremos la técnica *SCT* original de Lee, Jones y Ben-Amram [POPL' 01] en la que se detallan un par de algoritmos de terminación correctos y completos pero ineficientes en coste computacional. En 3.2. hablaremos de un subconjunto importante de instancias *SCT* denomidado *SCP* [TOPLAS' 05] para el que existe un algoritmo de coste polinomial que presentaremos en pseudocódigo. El apartado 3.3. presenta una generalización de *SCT* denominada δSCT [TOPLAS' 07], que tiene la ventaja de extender la información que nos ofrecen los grafos de cambio de tamaño, pero que tiene pocas aplicaciones prácticas. Finalmente, presentaremos en 3.4. una técnica que infiere funciones de rango de programas a un conjunto de instancias *SCT* denominado *SCNP* [TACAS' 08]. Como explicaremos posteriormente, la existencia de estas funciones para un cierto programa nos asegurará la terminación del mismo.

3.1. *SCT*

La presentación original de *SCT* se encuentra en el artículo de Lee, Jones y Ben-Amram [POPL' 01], que se ha convertido en una referencia clásica en el ámbito de la terminación de programas.

Partiremos del lenguaje funcional de primer con la siguiente sintaxis:

$p \in Prog ::= def_1 \dots def_m$

$def \in Def ::= f(x_1, \dots, x_n) = e^f$

$e \in Expr ::= x$

$\quad | \; if\ e_1\ then\ e_2\ else\ e_3$

$\quad | \; op(e_1, \dots, e_n)$

$\quad | \; c{:}\, f(e_1, \dots, e_n)$

$x \in Parameter ::= identifier$

$f \in FcnName ::= identifier\ not\ in\ Parameter$

$op \in Op ::= primitive\ operator$

Denotaremos por $f_{initial}$ a la primera función de un programa p. Dada una función $f(x_1, \dots, x_n)$ denotaremos por $arity(f) = n$ a su aridad y como $Param(f) = \{f^{(1)}, \dots, f^{(n)}\}$ a sus parámetros dónde $f^{(i)}$ se corresponde con x_i para cada $i \in \{1,2, \dots, n\}$.

El operador semántico ε se define del modo siguiente: $\varepsilon[\![e]\!]\vec{v}$ es el valor de la expresión e en el entorno $\vec{v} = (v_1, \dots, v_n)$, con $\varepsilon{:}\, Expr \to Value^* \to Value^{\#}$ donde $Value$ es el dominio de los valores de entrada, $Value^*$ es el dominio de las secuencias finitas de valores de entrada y $Value^{\#} = Value \cup \{Err, \bot\}$ en donde Err modela errores en tiempo de ejecución y $\bot$ representa no-terminación. Diremos que el programa p es terminante con la entrada $\vec{v}$ si y sólo si $\varepsilon[\![e^{f_{initial}}]\!]\vec{v} \neq \bot$.

Se escribirá $c{:}\, f \to g$ o $f \xrightarrow{c} g$ para una **llamada** c **a una función** g dentro de e^f **en un** cierto **programa** p. Una **secuencia de llamadas** es una secuencia finita o infinita $cs = c_1 c_2 c_3 \dots$. Diremos que la **secuencia de llamadas** cs está **bien formada** si existe una secuencia de funciones (finita o infinita) $f_0, f_1, f_2, \dots$ tal que $f_0 \xrightarrow{c_1} f_1 \xrightarrow{c_2} f_2 \xrightarrow{c_3} \dots$. Denotaremos por $\mathcal{C}$ al conjunto de todas las llamadas del programa p.

La representación elegida para representar programas en *SCT* se basa, como ya hemos comentado, en conjuntos de grafos, llamados *grafos de cambio de tamaño* (en inglés, *Size-Change Graphs*, abrev. *SCG*), donde cada uno de los grafos representa una de las llamadas recursivas del programa y la relación de tamaños entre los parámetros entre dichas llamadas.

Definición 3.1: Sean f, g nombres de funciones de un programa p. Un **grafo de cambio de tamaño (size-change graph, *SCG*)** en una llamada de f a g denotado $G: f \to g$ es el grafo bipartito $G = (Param(f), Param(g), E)$ dónde $E \subseteq Param(f) \times \{>, \geq\} \times Param(g)$ tal que E no contiene al mismo tiempo los arcos $f^{(i)} \overset{>}{\to} g^{(j)}$ y $f^{(i)} \overset{\geq}{\to} g^{(j)}$.

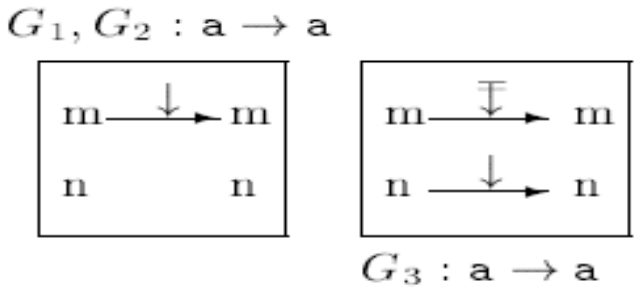
Denotaremos por $\mathcal{G} = \{G_c \mid c \in \mathcal{C}\}$ al conjunto de grafos de cambio de tamaño de un programa p. Un arco de la forma $f^{(i)} \overset{>}{\to} g^{(j)}$, denominado ***arco estricto***, indica que el valor del tamaño del argumento i-ésimo de f decrecerá estrictamente al convertirse en el argumento j-ésimo de g en la llamada, mientras que un arco de la forma $f^{(i)} \overset{\geq}{\to} g^{(j)}$, denominado ***arco no-estricto***, indica que el valor del tamaño del argumento i-ésimo de f no crecerá al convertirse en el argumento j-ésimo de g en la llamada, i.e., o decrecerá o se mantendrá igual. La ausencia de arco entre dos parámetros indica que no puede asegurarse ninguna de las dos relaciones anteriores.

Para ilustrar el concepto de grafo de cambio de tamaño, tomaremos el ejemplo de la función de Ackermann:

$$a(m,n) = if\ m = 0\ then\ n+1\ else$$
$$if\ n = 0\ then\ 1{:}\, a(m-1,1)$$
$$else\ \ 2{:}\, a(m-1, 3{:}\, a(m, n-1))$$

Los grafos de cambio de tamaño de las llamadas 1 y 2, $G_1, G_2 : a \to a$ tienen el mismo conjunto de aristas puesto que de ellos sólo puede inferirse que el argumento m decrece estrictamente, así pues, el citado conjunto de aristas es $\{m \overset{>}{\to} m\}$. En el caso del grafo de cambio de tamaño de la llamada 3, $G_3 : a \to a$, vemos que el argumento m se mantiene invariable mientras que el argumento n decrece estrictamente. Por tanto

su conjunto de aristas será $\{m \overset{\geq}{\rightarrow} m, n \overset{>}{\rightarrow} n\}$. Gráficamente podemos representar los grafos anteriores de la siguiente manera:



La construcción del *SCG* con la mayor información posible no siempre es computable por lo que no todos los programas pueden representarse de esta forma. No obstante, para aquellos programas que puedan representarse mediante conjuntos de grafos de cambio de tamaño, *SCT* nos da un método completo que nos detecta si son terminantes o no. En lo sucesivo supondremos que los *grafos de cambio de tamaños* son *seguros*, en el sentido de que toda la información que aparece en ellos se cumple en toda ejecución real.

La manera formal de representar secuencias finitas o infinitas de llamadas recursivas de un programa en *SCT* nos la da el concepto de *multicamino*:

Definición 3.2: Un **multicamino** $\mathcal{M}$ es una secuencia finita o infinita de grafos de cambio de tamaño $G_{c_1}, G_{c_2}, \ldots$

El concepto de *hebra* (o *thread*) nos permite saber de qué forma se van relacionando los tamaños de los parámetros a lo largo de una secuencia de llamadas:

Definición 3.3: Una **hebra** (o ***thread***) th en un multicamino $\mathcal{M} = G_{c_1}, G_{c_2}, \ldots$ es un camino conectado $th = f_t^{(i_t)} \overset{r_{t+1}}{\longrightarrow} f_{t+1}^{(i_{t+1})} \overset{r_{t+2}}{\longrightarrow} \ldots$. Una **hebra** es **maximal** si dicho camino es maximal en el multicamino. Una **hebra** es **descendente** si en la secuencia $r_{t+1}, r_{t+2}, \ldots$ aparece al menos una vez un $>$, y una **hebra** es **infinitamente descendente** si en dicha secuencia hay infinitas apariciones de $>$.

En el siguiente ejemplo intentaremos clarificar estos dos últimos conceptos de multicamino y hebra:

Programa p

$f(a, b, c) = 1{:}\, g(cons\ a\ b, tl\ c)$

$g(d, e) = \cdots\ 2{:}\, h([], tl\ e, d) \ldots 3{:}\, k(tl\ e)$

$h(u, v, w) = 3{:}\, g(u, tl\ w)$

$k(x) = \cdots$

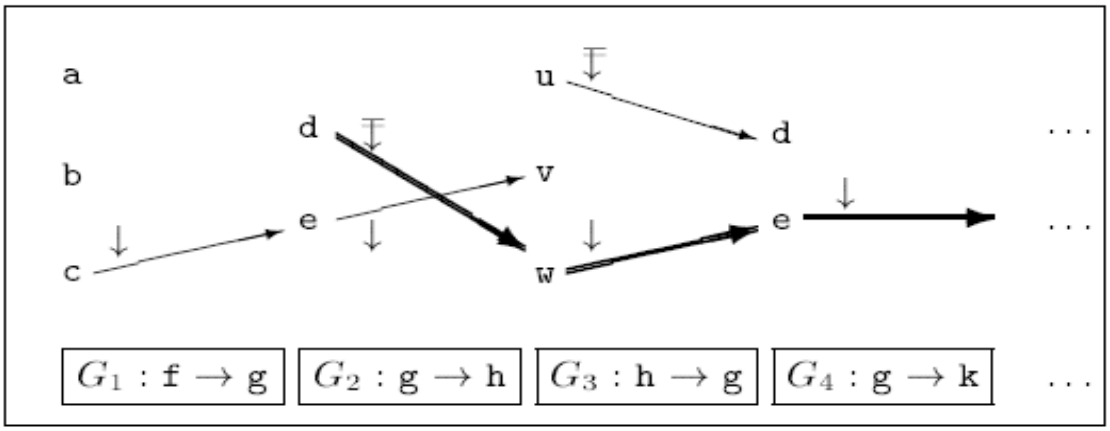
Un multicamino es una secuencia de los *SCGs* correspondientes a una secuencia de llamadas del programa. Un posible multicamino para el programa p anterior es el siguiente:

$$\mathcal{M} \equiv G_1{:}\, f \to g, G_2{:}\, g \to h, G_3{:}\, h \to g, G_4{:}\, g \to k, \ldots$$

El concepto de hebra dentro de un multicamino nos informa sobre de qué forma varía un cierto argumento en el transcurso de una secuencia de llamadas. Por ejemplo, una hebra del multicamino finito $G_1{:}\, f \to g, G_2{:}\, g \to h$ es:

$$c \overset{>}{\to} e \overset{>}{\to} v$$

En la siguiente figura damos más ejemplos de hebras de diferentes partes del multicamino anterior:



Dado $\mathcal{G} = \{G_c \mid c \in C\}$ para un cierto programa p, y dada una secuencia de llamadas $cs = c_1c_2c_3$ …, diremos que el $\mathcal{G}$-multicamino para cs se define por $\mathcal{M}^{\mathcal{G}}(cs) = G_{c_1}, G_{c_2}, G_{c_3}, \ldots$, el cuál muestra la información proporcionada por los grafos de cambio de tamaño de $\mathcal{G}$.

Una vez construidos los grafos de cambio de tamaño, se presentan dos métodos que deciden a partir de dichos grafos si el programa termina o no, entendiendo como criterio que un programa termina para todas sus entradas cuando toda secuencia infinita de llamadas provoca un descenso infinito del valor de alguno de los parámetros de las mismas.

El primer método presentado está basado en los denominados *autómatas de Büchi*. Estos autómatas son capaces de reconocer los conjuntos de las secuencias de llamadas infinitas potenciales del flujo de control de un programa, y en particular de todas aquellas secuencias de llamadas infinitas para las cuales uno de los parámetros siempre decrece en cada una de las llamadas sucesivas. Si se prueba que estos conjuntos son iguales, se demuestra que el programa termina por el criterio de terminación comentado anteriormente.

Los conjuntos de secuencias de llamadas que acabamos de mencionar los podemos formalizar de la siguiente manera:

$$FLOW^{\omega} = \{cs = cs = c_1c_2c_3 \ldots \in \mathcal{C}^{\omega} \mid cs\ bien\ formada\ \wedge\ c_1 : f_{initial} \rightarrow f_1\}$$
$$DESC^{\omega} = \{cs \in FLOW^{\omega} \mid \exists\ hebra\ infinitamente\ descendente\ en\ \mathcal{M}^{\mathcal{G}}(cs)\}$$

donde, dado un conjunto A denotaremos como A^{ω} al conjunto de secuencias infinitas sobre A.

Como hemos comentado más arriba, la igualdad de estos dos conjuntos nos asegura la terminación de un programa, lo cual queda reflejado en el siguiente resultado:

Teorema 3.4: $FLOW^{\omega} = DESC^{\omega} \Rightarrow p$ termina para todo valor de entrada

En las condiciones del teorema anterior se dice que el **programa** p es **terminante por cambio de tamaño**, o **satisface *SCT*** o **es *SCT***.

Los *autómatas de Büchi* son una extensión de los autómatas finitos que reconocen lenguajes con cadenas infinitas y son utilizados en el ámbito de la verificación formal y en los comprobadores de modelos. La idea de aceptación de cadenas infinitas consiste en que estos autómatas detecten que en dichas cadenas haya infinitas apariciones de un estado de aceptación.

Definición 3.5: Un **autómata de Büchi** $\mathcal{A} = (\Sigma, \mathrm{Q}, \mathrm{Q}_0, \delta, \mathrm{F})$ es una tupla dónde Σ es un *conjunto* finito *de símbolos de entrada*, Q es un *conjunto* finito *de estados*, $\mathrm{Q}_0 \subseteq \mathrm{Q}$ es el *conjunto de estados iniciales*, $\delta \subseteq \mathrm{Q} \times \Sigma \times \mathrm{Q}$ es la *relación de transición de estados* y $F \subseteq Q$ es el *conjunto de estados de aceptación*.

Diremos que una **traza** r del autómata de Büchi $\mathcal{A}$ sobre la palabra infinita $w = a_1 a_2 a_3 \ldots \in \sum^{\omega}$ es una secuencia $r = q_0 a_1 q_1 a_2 q_2 a_3 q_3 \ldots \in Q(\sum Q)^{\omega}$ tal que $q_0 \in Q_0$ y $(q_t\ ,\ a_{t+1}\ ,\ q_{t+1}) \in \delta$ donde $t = 0,1,2,3, \ldots$ La **traza** r es **aceptada por el autómata de Büchi** $\mathcal{A}$ si y sólo si existe $q \in F$ tal que q aparece infinitas veces en $q_0 q_1 q_2 q_3 \ldots$ Definimos y denotamos el **lenguaje reconocido por el autómata de Büchi** $\mathcal{A}$ como $\mathcal{L}_{\omega}(\mathcal{A}) = \{w \in \sum^{\omega} \mid \exists\ traza\ sobre\ w\ aceptada\ por\ \mathcal{A}\}$. Diremos que un **conjunto** $A \subseteq \sum^{\omega}$ es ***ω*-regular** si y sólo si es reconocido por algún autómata de Büchi.

En [POPL' 01] se demuestra el siguiente resultado que nos asegura la existencia de autómatas de Büchi para los dos conjuntos de secuencias de llamadas anteriores:

Lema 3.6: $FLOW^{\omega}$ y $DESC^{\omega}$ son subconjuntos ω-regulares de $\mathcal{C}^{\omega}$

Puesto que existe un procedimiento en *PSPACE* que dados dos autómatas de Büchi decide si reconocen exactamente las mismas secuencias, el método propuesto consiste en construir los autómatas de Büchi correspondientes a $FLOW^{\omega}$ y $DESC^{\omega}$, existentes por el Lema 3.6, y aplicar el procedimiento que acabamos de mencionar y que pasamos a formalizar:

Teorema 3.7: El siguiente problema es decidible en *PSPACE*: Dados dos autómatas de Büchi $\mathcal{A}$ y $\mathcal{A}'$, decidir si $\mathcal{L}_{\omega}(\mathcal{A}) = \mathcal{L}_{\omega}(\mathcal{A}')$.

La respuesta de este procedimiento de decisión nos responderá si el programa en cuestión es terminante o no lo es.

El segundo método propuesto opera directamente con los grafos de cambio de tamaño. En primer lugar construye el conjunto de todos los grafos de cambio de tamaño a partir de las secuencias de llamadas bien formadas del programa, y a cada uno de ellos les aplica el criterio expuesto en el Teorema 3.9 que veremos más adelante para comprobar que el programa termina o no. El punto débil de este método es el paso de construcción de dichos grafos de cambio de tamaño, basado en

un algoritmo de cierre transitivo, puesto que es exponencial con respecto al tamaño del programa.

Definición 3.8: La **composición de** dos **grafos de cambio de tamaño** $G : f \to g$ y $G' : g \to h$ es el grafo de cambio de tamaños $G; G' : f \to h$ cuyo conjunto de aristas viene definido por:

$$E = \{x \overset{>}{\to} z \mid \exists y, r: x \overset{>}{\to} y \overset{r}{\to} z \ \vee \ x \overset{r}{\to} y \overset{>}{\to} z\} \ \cup \ \{x \overset{\geq}{\to} z \mid \left(\exists y: x \overset{\geq}{\to} y \overset{\geq}{\to} z\right)$$
$$\wedge \ \forall y, r, r': x \overset{r}{\to} y \overset{r'}{\to} z \Rightarrow r = r' = \geq\}$$

La composición de grafos de cambio de tamaño es asociativa.

Dada una secuencia de llamadas no-vacía $cs = c_1 c_2 \dots c_n$ notamos y definimos su grafo de cambio de tamaño como $G_{cs} = G_{c_1}; G_{c_2}; \dots; G_{c_n}$, y definimos como sigue el conjunto $\mathcal{S} = \{G_{cs} \mid cs, cs_0 \ bien \ formadas \ \wedge \ f_{initial} \overset{cs_0}{\to} f \overset{cs}{\to} g\}$.

La principal idea del método se basa en el siguiente resultado que nos ofrece un condición de chequeo de terminación para cada uno de los grafos de cambio de tamaño:

Teorema 3.9: Un programa p es terminante por cambio de tamaño $\Leftrightarrow$ $\mathcal{S}$ contiene a $G: f \to f$ tal que $G = G; G$ y G contiene algún arco de la forma $x \overset{>}{\to} x$.

El algoritmo basado en este resultado consiste en construir el conjunto $\mathcal{S}$ y aplicar el criterio de este teorema para cada uno de sus elementos:

1. Construir el conjunto $\mathcal{S}$ mediante un procedimiento de cierre transitivo:

   a) Incluir en $\mathcal{S}$ todo $G_c: f \to g$ dónde $c: f \to g$ es una llamada en el programa p tal que f es alcanzable por alguna secuencia de llamadas bien formada $cs_0: f_{initial} \to g$.

   b) Para cada $G: f \to g$ y $H: g \to h$ en $\mathcal{S}$, incluir $G; H$ en $\mathcal{S}$.

2. Para cada $G: f \to f$ en $\mathcal{S}$, comprobar si $G = G; G$ y $x \overset{>}{\to} x \ \notin G \ \forall x \in Param(f)$

La construcción de $\mathcal{S}$ requiere tiempo exponencial por lo que para ambos métodos que acabamos de explicar, podemos concluir lo siguiente:

Teorema 3.10: La terminación por cambio de tamaño puede decidirse en espacio polinomial y tiempo exponencial en el tamaño del programa p, i.e., *SCT* es *PSPACE*-completo.

Cabe destacar que para la construcción de los grafos de cambio de tamaño, los autores usan un único grafo que abarca todas las llamadas a funciones del programa. Una posibilidad para mejorar la eficiencia del método podría ser manejar los grafos de cada función de manera independiente (salvo en casos de recursión mutua) y jerárquica por niveles, procediendo posteriormente a comprobar la terminación de las funciones situadas en los niveles más bajos para probar la terminación de las funciones de los niveles más altos que usan a las anteriores.

3.2. *SCP*

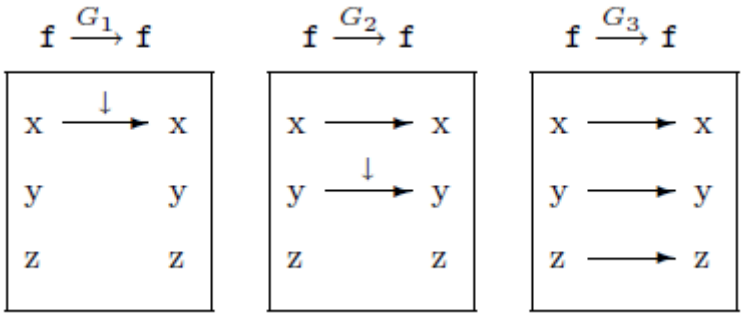
Los algoritmos presentados en [POPL' 01] son demasiado ineficientes y poco útiles en la práctica. Por esta razón, Ben-Amram y Lee [TOPLAS' 05] presentan un subconjunto de instancias *SCT*, denominado *SCP*, para las cuales se puede decidir terminación en tiempo polinomial. *SCP* no cubre en absoluto a todo *SCT*, pero abarca una gran parte, muy significativa, de él. Los autores presentan una tabla comparativa de los algoritmos SCP y SCT para diferentes programas en los que puede apreciarse la eficiencia del primero frente al segundo obteniendo, en general, la misma respuesta.

La construcción de *SCP* requiere de una serie de conceptos previos entre los que destaca el de *ancla de un conjunto de grafos de cambio de tamaño*:

Definición 3.11: El **conjunto infinito de una secuencia infinita** $s_1 s_2 \ldots$ es $\{s \mid \{i \mid s_i = s\}\ infinito\}$.

Definición 3.12: Sea $\mathcal{H}$ un conjunto de grafos de cambio de tamaño. Diremos que $G \in \mathcal{H}$ es un **ancla *SCT*** si todo $\mathcal{H}$-multicamino cuyo conjunto infinito contenga a G, tiene un descenso infinito, i.e., contiene infinitos arcos estrictos.

La idea intuitiva es que las *anclas* evitan que algunos bucles contenidos en el programa se ejecuten infinitamente. Sea el siguiente conjunto de *SCGs* $\mathcal{G} = \{G_1, G_2, G_3\}$:



Claramente G_1 es un ancla de $\mathcal{G}$ puesto que si aparece infinitas veces en cualquier $\mathcal{G}$-multicamino, induce a un descenso infinito. Análogamente, tomando $\mathcal{H} = \{G_2, G_3\}$, G_2 es un ancla para $\mathcal{H}$.

La forma de localizar los bucles de un programa representado por grafos de cambio de tamaño es calcular sus componentes fuertemente conexas:

Definición 3.13: Una **componente fuertemente conexa de un dígrafo** D es todo subgrafo maximal fuertemente conexo. Diremos que un **subgrafo** es **no-trivial** si contiene al menos un arco. Denotaremos por $SCC(D)$ al conjunto de componente fuertemente conexas no-triviales de D.

Los siguientes lemas justifican el algoritmo que presentaremos a continuación y que denominaremos *algoritmo SCP genérico*:

Lema 3.14: Si G es un ancla para $\mathcal{G}$, entonces G es un ancla para todo $\mathcal{H} \subseteq \mathcal{G}$, tal que $G \in \mathcal{H}$

Lema 3.15: $\mathcal{G}$ satisface *SCT* si y sólo si todos sus subgrafos fuertemente conexos y no-triviales tienen un ancla.

Con los ingredientes anteriores pasamos a presentar el algoritmo:

Algoritmo SCP genérico:

Llamar a $SCP(\mathcal{H})\ \forall \mathcal{H} \in SCC(\mathcal{G})$

Procedimiento $SCP(\mathcal{H})$:

1. Calcular el conjunto $\mathcal{A}$ de anclas de $\mathcal{H}$.
2. Si $\mathcal{A}$ es vacío, el algoritmo termina con fallo
3. Si $\mathcal{A}$ no es vacío, llamar a SCP con cada elemento de $SCC(\mathcal{H} \backslash \mathcal{A})$

Suponiendo que el procedimiento para calcular anclas (que veremos más adelante) termina, el algoritmo anterior siempre termina y además, si el algoritmo no finaliza con fallo para $\mathcal{G}$, entonces $\mathcal{G}$ satisface *SCT*. Por tanto, si $\mathcal{G}$ es seguro para un programa p, en dicho caso, se podrá concluir que p es terminante.

La cuestión de la que nos ocuparemos a continuación será sobre cómo detectar y calcular anclas de un conjunto de grafos de cambio de tamaño. El siguiente resultado, adaptación del Teorema 3.9, nos da una condición necesaria y suficiente para que un grafo sea un ancla *SCT*:

Teorema 3.16: Un *SCG* $G \in \mathcal{G}$ es un ancla *SCT* si y sólo si todo grafo de cambio de tamaño idempotente (con respecto a la composición de *SCGs*) $G^o = G_1; G_2; \dots; G_n$ con $G_i \in \mathcal{G}\ \forall i \in \{1,2,\dots,n\} \wedge G = G_1$ incluye un arco de la forma $x \overset{>}{\rightarrow} x$ para algún x.

Los autores identifican dos tipos de anclas *SCT* cuya caracterización depende del concepto de *thread preserver* y su detección puede computarse de manera eficiente. Intuitivamente, un *thread preserver* es un subconjunto de parámetros que nos darán la clave para decidir la terminación del programa. Antes de formalizar este concepto requerimos la definición de *fan-in* y *fan-out* en el contexto de los grafos de cambio de tamaño, análoga a la conocida en teoría de circuitos:

Definición 3.17: Sea G un **grafo de cambio de tamaño**:

1. G es **libre de fan-out** si cumple que el grado de salida de cualquier nodo de G es a lo sumo 1
2. G es **libre de fan-in** si cumple que el grado de entrada de cualquier nodo de G es a lo sumo 1
3. G tiene **fan-out estricto** si para cada nodo x en G con grado de salida mayor que 1 se tiene que todos los *arcos* $x \to y \in G$ son *estrictos* (i.e. $x \overset{>}{\to} y$)
4. G tiene **fan-in estricto** si para cada nodo y en G con grado de entrada mayor que 1 se tiene que todos los *arcos* $x \to y \in G$ son *estrictos* (i.e. $x \overset{>}{\to} y$)

Estas definiciones se aplican análogamente a un conjunto de grafos de cambio de tamaño cuando todos sus elementos son libres de fan-in/fan-out o tienen fan-in/fan-out estricto.

Sea $Par = \bigcup_f Param(f)$, definimos:

Definición 3.18: Sea $\mathcal{H}$ un conjunto de grafos de cambio de tamaño. Un conjunto $P \subseteq Par$ se llama ***thread preserver*** para $\mathcal{H}$ si para todo $G \in \mathcal{H}$ donde $G: f \to g$, se cumple que siempre que $x \in (Param(f) \cap P)$, existe $x \to y \in G$ para algún $y \in P$.

Denotamos por $MTP(\mathcal{H})$ al *thread preserver* maximal de $\mathcal{H}$, el cual siempre existe puesto que el conjunto de *thread preservers* es cerrado bajo la unión.

Definición 3.19: Dado un grafo de cambio de tamaño G y un conjunto de parámetros P, denotaremos por G^P (leído **G restringido a P**) al subgrafo de G inducido por los nodos de G correspondientes a los parámetros en P. Para un conjunto de grafos de cambio de tamaño $\mathcal{H}$, se tiene: $\mathcal{H}^P \overset{\text{def}}{=} \{G^P \mid G \in \mathcal{H}\}$.

Con estos elementos ya podemos definir el primer tipo de ancla *SCT*

Definición 3.20: Sea P un thread preserver para $\mathcal{H}$ tal que $\mathcal{H}^P$ tiene fan-in estricto. Un grafo de cambio de tamaño $G \in \mathcal{H}$ es un **ancla de tipo 1** con respecto a P si G^P contiene un arco estricto.

El método para obtener anclas de tipo 1 será calcular el MTP y comprobar si algún grafo de cambio de tamaño es un ancla de tipo 1 con respecto a él.

Para formalizar el segundo tipo de ancla *SCT* introducimos las siguientes definiciones sobre grafos de cambio de tamaño:

Definición 3.21: El **grafo de relación de tamaños** debido a un conjunto de grafos de cambio de tamaño $\mathcal{H}$, denotado $SRG_{\mathcal{H}}$, es el dígrafo etiquetado cuyo conjunto de nodos es Par y su conjunto de arcos es $\left\{x \overset{r,G}{\rightarrow} y \;\middle|\; (x \overset{r}{\rightarrow} y \in G) \wedge G \in \mathcal{H}\right\}$.

Definición 3.22: Sea $\mathcal{H}$ un conjunto de grafos de cambio de tamaño. El **grafo no-descendente**, denotado $NDG_{\mathcal{H}}$ es el subgrafo de $SRG_{\mathcal{H}}$ consistente en su arcos no-estrictos (i.e. $x \overset{\geq}{\rightarrow} y$). El **interior del grafo no-descendente**, notado $NDG^0_{\mathcal{H}}$ es el subgrafo de $NDG_{\mathcal{H}}$ consistente en todos los arcos internos de una componente fuertemente conexa de $NDG_{\mathcal{H}}$.

La construcción de estos grafos puede computarse eficientemente, lo cual nos asegura que el cálculo del tipo de anclas *SCT* que pasamos a definir puede realizarse en un coste razonable:

Definición 3.23: Un grafo de cambio de tamaño $G \in \mathcal{H}$ se denomina **ancla de tipo 2** si se verifica que $(\mathcal{H} \backslash G) \cup \{G^{\downarrow}\}$ tiene un MTP no-vacío, donde $G^{\downarrow}$ es G menos los arcos de $NDG^0_{\mathcal{H}}$.

Algoritmo de Cálculo de $\boldsymbol{MTP}(\boldsymbol{\mathcal{H}})$

El cálculo del MTP se establece con el siguiente resultado, justificado por el algoritmo que le sigue:

Lema 3.24: Sea $\mathcal{H} \subseteq \mathcal{G}$ dado como un conjunto de punteros a los grafos de cambio de tamaño de $\mathcal{G}$, y sea N el tamaño de $\mathcal{H}$. Entonces $MTP(\mathcal{H})$ puede calcularse en $\mathcal{O}(N)$.

Pasamos a detallar el algoritmo:

Asociar un contador para cada nodo fuente de los grafos de cambio de tamaño de $\mathcal{H}$ igualándolo inicialmente a su grado de salida. Si dicho valor es 0 para algún grafo de cambio de tamaño, marcar el parámetro asociado a dicho nodo y construir una lista de trabajo con todos los parámetros marcados. Procesar los elementos de la lista de trabajo por turno de la siguiente manera: Al procesar el elemento x, chequear los arcos de la forma $y \to x$, y reduciendo el contador del nodo fuente por cada uno de ellos. Si dicho contador llega a 0 e y no esté marcado, marcarlo e introducirlo en la lista de trabajo. El algoritmo termina cuando la lista de trabajo sea vacía. Los parámetros sin marcar forman $MTP(\mathcal{H})$.

La corrección del anterior algoritmo y su complejidad se justifica en [TOPLAS' 05].

Algoritmo SCP completo

Presentamos aquí el algoritmo *SCP* basado en encontrar los tipos de anclas que definimos anteriormente. Una forma de encontrar más anclas es hacer una búsqueda en los grafos de cambio de tamaño traspuestos. La transposición de SCGs se define como vemos a continuación:

Definición 3.25: Sea $G = (Param(f), Param(g), A)$ un grafo de cambio de tamaño. Definimos su **transposición** como $G^t = (Param(g), Param(f), A^t)$, donde su conjunto de arcos es $A^t = \left\{ x \xrightarrow{r} y \;\middle|\; y \xrightarrow{r} x \in G \right\}$. Para un conjunto de grafos de cambio de tamaño $\mathcal{H}$, definimos $\mathcal{H}^t = \{G^t \mid G \in \mathcal{H}\}$.

La búsqueda de anclas en los grafos traspuestos se justifica en el siguiente lema:

Lema 3.26: SCT es cerrado bajo trasposición: G es un ancla para $\mathcal{H}$ si y sólo si G^t es un ancla para $\mathcal{H}^t$.

El algoritmo es el siguiente:

Llamar a $SCP(\mathcal{H})\ \forall \mathcal{H} \in SCC(\mathcal{G})$

Procedimiento $SCP(\mathcal{H})$:

1. Calcular $SRG_{\mathcal{H}}$ y dividirlo en sus componentes fuertemente conexas. Eliminar los arcos que no sean internos a una componente conexa C tal que $\mathcal{H}^C$ contiene al menos un arco estricto.
2. $\mathcal{A} \coloneqq AnchorFind(\mathcal{H})$, calcula el conjunto de anclas de $\mathcal{H}$.
3. Llamar a SCP con cada elemento de $SCC(\mathcal{H} \backslash \mathcal{A})$.

Procedimiento $AnchorFind(\mathcal{H})$:

1. Si $\mathcal{A} = Type1Anchors(\mathcal{H})$ es no-vacío, devolver $\mathcal{A}$
2. Si $\mathcal{A} = Type1Anchors(\mathcal{H}^t)$ es no-vacío, devolver $\mathcal{A}$
3. Si $\mathcal{A} = Type2Anchors(\mathcal{H})$ es no-vacío, devolver $\mathcal{A}$
4. Si $\mathcal{A} = Type2Anchors(\mathcal{H}^t)$ es no-vacío, devolver $\mathcal{A}$
5. Terminar con fallo (Esto termina con el procedimiento SCP)

Procedimiento $Type1Anchors(\mathcal{H})$:

1. Calcular $P = MTP(\mathcal{H})$
2. Si $\mathcal{H}^P$ tiene fan-in estricto, calcular $\mathcal{A} = \{G \in \mathcal{H} \mid \exists\, x \xrightarrow{\downarrow,G} y \in \mathcal{H}^P\}$, sino asignar $\mathcal{A} = \{\}$
3. Devolver $\mathcal{A}$

Procedimiento $Type2Anchors(\mathcal{H})$:

1. Calcular $P = MTP(\mathcal{H})$ y $D = NDG^0_{\mathcal{H}}$
2. Para cada $G \in \mathcal{H}$:
   a) Obtener $G^{\downarrow}$ a partir de G^P y D. Asignar $\mathcal{H}' = \mathcal{H}\backslash\{G\} \cup \{G^{\downarrow}\}$
   b) Calcular $MTP(\mathcal{H}')$. Si es no-vacío, devolver $\{G\}$
3. Devolver {}

Considerando que $\mathcal{G}$ tiene tamaño N y contiene n grafos de cambio de tamaños, el algoritmo SCP se computa en tiempo $\mathcal{O}(Nn^2)$. Suponiendo que $\mathcal{H}$ tiene tamaño N, y contiene n grafos de cambio de tamaño, $AnchorFind(\mathcal{H})$ se computa en $\mathcal{O}(Nn)$.

El algoritmo es capaz de detectar situaciones frecuentes de terminación de programas, como ordenes lexicográficos o de multiconjuntos, entre la relación de tamaños de las llamadas.

En [TOPLAS' 05] se adjunta una tabla comparativa de los algoritmos *SCT* y *SCP* en la que se puede contemplar que este último es mucho más eficiente que el primero. Todos los ejemplos expuestos en dicha tabla son detectados por *SCP*. En general, este no cubre todo *SCT*, pero si tiene éxito en un alto porcentaje de programas *SCT*.

3.3. δSCT

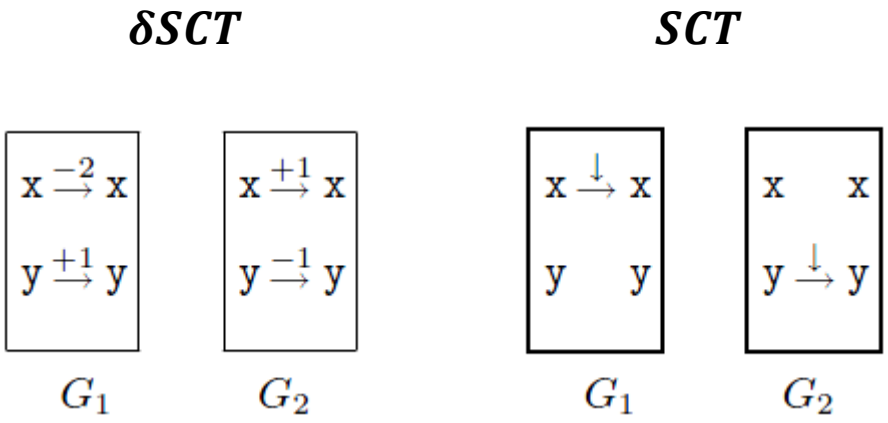
δSCT [TOPLAS' 07] es una generalización de *SCT* consistente en que los arcos de los grafos de cambio de tamaño vienen etiquetados por un número entero que representa en cuanto crece o decrece un parámetro en una cierta llamada. De nuevo, la ausencia de arco entre un par de parámetros indica que dicha información no puede conocerse con seguridad. Este planteamiento supone redefinir el concepto de grafo de cambio de tamaño que habíamos visto anteriormente:

Definición 3.27: Un **grafo de cambio de tamaño (*SCG*) $\boldsymbol{\delta SCT}$** G asociado al arco $f \to g$, denotado $G: f \to g$, es el grafo bipartito dirigido con conjunto fuente $A = Param(f)$, conjunto destino $B = Param(g)$, y conjunto de aristas $E = \{x \xrightarrow{\delta} y \mid x \in A \wedge y \in B \wedge \delta \in \mathbb{Z}\}$.

Podemos representar este grafo mediante una matriz $G_{|A|\times|B|}$ donde $G[i,j]$ representa la etiqueta del arco $f^{(i)} \to g^{(j)}$ si éste existe o ∞ en caso contrario.

Para aclarar ideas, veamos la representación δSCT frente a la representación SCT para el siguiente ejemplo, debido a A. Pnueli:

$$f(x,y) = if\, x < 2\ or\ y < 1\ then \ldots else\ if\ \ldots 1: f(x-2, y+1)$$
$$else \ldots 2: f(x+1, y-1)$$



Dada una hebra $t = x_0 \overset{\delta_1}{\rightarrow} x_1 \overset{\delta_2}{\rightarrow} \ldots$, usaremos la notación $\int_0^n t$ como abreviatura de $\sum_{i=1}^{n} \delta_i$, y usaremos la notación $\int t$ para referirnos a toda esa suma sobre t, cuando ésta sea finita. En estos términos diremos que una hebra finita es descendente cuando $\int t < 0$, y diremos que una hebra infinita es infinitamente descendente cuando se cumpla que $\lim_{n\to\infty} \int_0^n t = -\infty$.

Con los elementos anteriores ya estamos en disposición de definir formalmente la propiedad δSCT:

Definición 3.28: Un **conjunto de grafos de cambio de tamaño δSCT $\mathcal{G}$ satisface la condición δSCT** si todo multicamino infinito contiene al menos una hebra infinitamente descendente.

Intuitivamente, esto puede entenderse como que si todo cómputo del programa, hipotéticamente infinito, implica un descenso infinito sobre el tamaño de algún dato, entonces no pueden existir cómputos infinitos en dicho programa.

La condición SCT puede verse como el conjunto de ejemplares que satisfacen δSCT cuando no se incluyen etiquetas positivas, por lo que claramente δSCT es un problema más general que SCT.

Ben-Amram [TOPLAS' 07] demuestra por reducción al problema de la parada que en general δSCT es indecidible. Seguidamente razonaremos distintas restricciones al problema que lo convierten en decidible. Previo a ello, tenemos que presentar los

siguientes conceptos fundamentales, algunos, ya vistos atrás, pero adaptados a los nuevos grafos de cambio de tamaño con los que se trabaja en δSCT:

Definición 3.29: La **composición de dos grafos de cambio de tamaño δSCT** $G_1: f \to g$ y $G_2: g \to h$, es el grafo de cambio de tamaño δSCT $G = G_1; G_2$ con función fuente f, función destino h y conjunto de arcos etiquetados definido por $G[i,j] = \min_k\{G_1[i,k] + G_2[k,j]\}$.

Si G_1 y G_2 son libres de fan-in, entonces $G_1; G_2$es también libre de fan-in.

Dado un multicamino finito $\mathcal{M} = G_1 G_2 \dots G_n$, definimos $\overline{\mathcal{M}} = G_1; G_2; \dots; G_n$. El *multicamino vacío* que empieza y termina en la función f se denota Id_f, y se define $\overline{Id_f}$ como el grafo de cambio de tamaño δSCT con los arcos $x \xrightarrow{0} x\ \forall x \in Param(f)$. La *concatenación de multicaminos* se notará por yuxtaposición (i.e. $\mathcal{M}_1\mathcal{M}_2$ …). Diremos que dos *multicaminos* $\mathcal{M}_1$ y $\mathcal{M}_2$ son *equivalentes*, notado $\mathcal{M}_1 \equiv \mathcal{M}_2$, si se verifica que $\overline{\mathcal{M}_1} = \overline{\mathcal{M}_2}$. Dados dos grafos de cambio de tamaño δSCT, G_1 y G_2, escribiremos $G_1 \simeq G_2$ si ambos grafos son idénticos obviando sus etiquetas, y diremos que un *grafo de cambio de tamaño* δSCT G es *idempotente* si $G; G \simeq G$. Se dice que un *multicamino* $\mathcal{M}$ es *cíclico* si $\overline{\mathcal{M}}$ es idempotente.

Dado un grafo de cambio de tamaño G con idéntica fuente y destino, diremos que un arco de la forma $x \to x \in G$ es un *arco in-situ*, y denotamos por $\lfloor G \rfloor$ a la *parte in-situ* de G, i.e. a su subgrafo consistente en todos sus arcos in-situ.

Un *grafo de símbolos* es un grafo bipartito similar a un grafo de cambio de tamaños δSCT excepto en que sus etiquetas están en el conjunto $\{>, \geq, <\}$. Un grafo de este tipo abstrae un grafo de cambio de tamaño δSCT: Denotamos por α al *operador de abstracción*, definido como sigue: Dado un grafo de cambio de tamaño δSCT G, $\alpha(G)$ es la siguiente matriz:

$$\alpha(G)[i,j] = \begin{cases} > \ si\ G[i,j] < 0 \\ \geq \ si\ G[i,j] \leq 0 \\ < \ si\ G[i,j] > 0 \\ \infty \ si\ G[i,j] = \infty \end{cases}$$

Teorema 3.30: Un ACG fuertemente conexo con grafos de cambio de tamaño δSCT libres de fan-in satisface δSCT si y sólo si para todo multicamino cíclico $\mathcal{M}$ se verifica que $\overline{\mathcal{M}}$ tiene un arco in-situ con etiqueta negativa.

Una consecuencia de este teorema es que un algoritmo para δSCT puede proceder buscando un contraejemplo finito al criterio de terminación, es decir, un multicamino cíclico que no contenga arcos in-situ.

A continuación se presentan un par de restricciones que convierten el problema δSCT en decidible. La primera de ellas es un caso sencillo pero poco frecuente mientras que la segunda supone grafos de cambio de tamaño libres de fan-in que se presenta en la práctica con más frecuencia.

La primera restricción trabaja con grafos de cambio de tamaño δSCT in-situ, estos son aquellos que contienen arcos in-situ: Se supone una única función simple cuyos grafos de cambio de tamaño son in-situ. En este caso sea $\mathcal{G} = \{G_1, G_2, \dots, G_n\}$, puesto que los grafos in-situ conmutan (i.e. $\forall i, j \in \{1,2, \dots n\}: G_i; G_j = G_j; G_i$) entonces para cada $\mathcal{G}$-multicamino hay una "forma normal" $\mathcal{M} = G_1^{x_1} G_2^{x_2} \dots G_n^{x_n}$, donde la notación $G_i^{x_i}$ indica G_i repetido x_i veces. En este caso, la forma de encontrar un contraejemplo del criterio de terminación expuesto en el Teorema Básico es comprobar si el siguiente problema de Programación Lineal (PL) tiene alguna solución no-trivial:

$$\sum_{i=1}^{n} G_i[j,j] x_i \ \geq 0 \ \forall j \in \{1,2, \dots, |Param(f)|\}$$

$$\forall i \in \{1,2, \dots, n\}: x_i \geq 0$$

Por tanto, este subproblema sencillo de δSCT es decidible por PL.

La segunda restricción de la que se hace estudio en el artículo es la de centrarse solamente en grafos libres de fan-in. La idea antes mencionada de encontrar una "forma normal" de multicaminos se aplica de nuevo en este caso, presentando el concepto de reducción de multicaminos que no detallaremos aquí. Con este tipo de grafos, todo multicamino tendrá una "forma normal" construida como la concatenación de una parte irreducible del multicamino con otra parte con una forma similar a las "formas normales" presentadas para los grafos in-situ. A partir de esto se define lo siguiente:

Definición 3.31: Un **multicamino extendido** tiene la forma $\mathcal{M} G_1^{x_1} G_2^{x_2} \dots G_n^{x_n}$ donde $\mathcal{M}$ es un multicamino de fuente g y destino f, cada G_i es un grafo in-situ para f, y cada $x_i \in \mathbb{N}^+$

Un **multicamino extendido** es **normal** si verifica:

1. $\mathcal{M}$ es irreducible
2. $\forall i \in \{1,2, \dots, n\}: G_i = \lfloor \overline{B_\iota} \rfloor$ para algún multicamino irreducible B_i
3. $\forall i, j \in \{1,2, \dots, n\}: i \neq j \Rightarrow G_i \neq G_j$

Un **multicamino formal** es una expresión $E = \mathcal{M}G_1^{x_1}G_2^{x_2}\dots G_n^{x_n}$ en la que los exponentes son variables formales que cumplen las condiciones 1-3 anteriores y para $k_0, k_1, \dots, k_n \in \mathbb{N}$, $E(k_0, k_1, \dots, k_n) \stackrel{\text{def}}{=} \mathcal{M}^{k_0}G_1^{k_1}G_2^{k_2}\dots G_n^{k_n}$ donde $k_0 > 1$.

Lema 3.32: Para cada $\mathcal{G}$-multicamino cíclico $\mathcal{M}: f \to f$, existe un multicamino formal $E = \mathcal{M}G_1^{x_1}G_2^{x_2}\dots G_n^{x_n}$ tal que:

1. $\exists r_1, r_2, \dots, r_n \in \mathbb{N}^+: E(r_1, r_2, \dots, r_n) \equiv \mathcal{M}$
2. $\exists b_0, b_1, \dots, b_n \in \mathbb{N}$ tales que para toda secuencia $(l_0, l_1, \dots, l_n)$ que verifique $\forall i \in \{0,1, \dots, n\}: l_i \geq b_i$ se cumple que $E(l_0, l_1, \dots, l_n)$ es equivalente a algún $\mathcal{G}$-multicamino cíclico

Los multicaminos formales que cumplen el Lema anterior se denominan *representaciones escalares*. El conjunto de representaciones escalares es finito.

Teorema 3.33: Sea $\mathcal{G}$ un ACG fuertemente conexo con grafos de cambio de tamaño libres de fan-in. $\mathcal{G}$ satisface δSCT si y sólo si para cada función f y para cada representación escalable $E = \mathcal{M}G_1^{x_1}G_2^{x_2}\dots G_n^{x_n}: f \to f$ de un $\mathcal{G}$-multicamino cíclico, el siguiente problema de Programación Lineal Entera no tiene solución:

$$\sum_{i=0}^{n} G_i[j,j]x_i \geq 0 \; \forall j \in \{1,2, \dots, |Param(f)|\}$$

$$\forall i \in \{1,2, \dots, n\}: x_i > 0$$

donde $G_0 = \lfloor \overline{\mathcal{M}} \rfloor$

Un algoritmo para decidir δSCT para esta restricción es el siguiente:

1. Construir el conjunto de representaciones escalables de los $\mathcal{G}$-multicaminos
2. Chequear la condición del teorema anterior para cada una de ellas.

Se demuestra que δSCT para grafos libres de fan-in es un problema *PSPACE*-completo, y apoyándose en acotaciones dadas por resultados de la Programación Lineal (y que aquí no comentaremos), se presenta otro algoritmo mucho más eficiente que el anterior:

Algoritmo basado en cierre para decidir δSCT sobre grafos libres de fan-in

El algoritmo construye el conjunto $\mathcal{S}$ de todos los grafos δSCT $\overline{\mathcal{M}}$ donde $\mathcal{M}$ es un $\mathcal{G}$-multicamino de longitud acotada por una constante $B(\mathcal{G})$. Si no se encuentra un grafo idempotente sin un descendiente in-situ, entonces $\mathcal{G}$ satisface δSCT.

El algoritmo etiqueta cada grafo G con la longitud de su multicamino correspondiente por lo que se mantiene un conjunto $\mathcal{S}'$ de grafos con longitudes:

1. Inicializar $\mathcal{S}'$ incluyendo $(G, 1)$ $\forall G \in \mathcal{G}$
2. Para cada (G, i): $f \to g$ y (H, j): $g \to h$ en $\mathcal{S}'$, si $i + j \leq B(\mathcal{G})$ y no hay aún un par $(G; H, k)$ en $\mathcal{S}'$, incluir $(G; H, i + j)$ en $\mathcal{S}'$. Si $(G; H, k)$ está, reemplazar k por $i + j$ si $i + j < k$.
3. Repetir el paso anterior hasta que no haya cambios en $\mathcal{S}'$.

El principal inconveniente de δSCT es el que ya comentamos para *SCT*, la ineficiencia de estos algoritmos por ser exponenciales en espacio por lo que se trata de un marco esencialmente teórico con poca aplicación práctica. Esta abstracción de *SCT* tiene la ventaja de ofrecernos más información puesto que no sólo nos indica si un parámetro decrece o no crece en cada llamada, sino que nos da la cantidad concreta de crecimiento o decrecimiento.

3.4. *SCNP*

El método clásico de detección de terminación ha sido el de obtener una *función de rango* para un cierto programa. Estas funciones, que trataremos con más detalle en el Capítulo 4, tienen la cualidad de que su existencia asegura terminación. Ben-Amram y Codish [TACAS' 08] desarrollaron un método para inferir funciones de rango a partir del problema *SCT*. Los autores definen un subconjunto *NP* de instancias *SCT* al que denominan *SCNP* para el cuál valdría el método propuesto en este trabajo. Las funciones de rango obtenidas por el método son polinómicas con respecto al tamaño de su expresión vista como cadena de caractéres y sirven como testigo o certificado de la terminación de un programa, lo cual supone un resultado más eficiente que los presentados en los artículos [TOPLAS' 09] y [LMCS' 09] en los que se obtenía funciones de rango triplemente exponenciales y simplemente exponenciales con respecto a los tamaños de sus expresiones vistas como cadenas de caractéres, respectivamente. Puesto que *SCNP* está en *NP*, esta propiedad puede reducirse al problema *SAT*, este es, el de dada una expresión booleana con variables y sin cuantificadores, obtener una

asignación de valores de dichas variables que hagan cierta la expresión. Por este motivo, se puede usar un resolutor *SAT* para resolver el problema *SCNP*, por lo que el grueso de la implementación del algoritmo propuesto es la codificación del problema en fórmulas booleanas que posteriormente se proporcionará como entrada de un resolutor *SAT*.

En lo sucesivo, además de ver un programa como un conjunto de grafos de cambio de tamaños, le asociaremos un conjunto de *puntos de programa*, donde cada punto de programa p tiene asociada una lista fija de argumentos $Arg(p)$. Escribiremos los estados de un programa como términos de la forma $p(u_1, u_2, \dots, u_n)$ donde n es la aridad de p, denotado p/n.

Una notación alternativa para representar grafos de cambio de tamaños consiste en verlos como clausulas de restricciones de programas lógicos, de la forma $p(\bar{x}) :- \pi; q(\bar{y})$ donde $\bar{x} = x_1, x_2, \dots, x_n$, $\bar{y} = y_1, y_2, \dots, y_m$, y π es una conjunción de restricciones de la forma $x_i > y_j$ o $x_i \geq y_j$. Escribiremos $\pi \vDash \phi$ para indicar que la proposición ϕ sobre los valores $\bar{x}$ e $\bar{y}$, se cumple bajo las suposiciones π.

Pasamos a ver la noción de terminación utilizada en *SCNP* y para esta notación alternativa de grafos de cambio de tamaños:

Definición 3.34: Un **estado de transición** es un par (s, s') de estados. Sea $g = p(\bar{x}) :- \pi; q(\bar{y})$ un grafo de cambio de tamaños. El **conjunto de transiciones** asociado a g es $T_g = \{(p(\bar{x}), q(\bar{y})) \mid \pi\}$. El **sistema de transiciones** asociado a un conjunto de grafos de cambio de tamaños $\mathcal{G}$ es $\mathcal{T}_\mathcal{G} = \bigcup_{g \in \mathcal{G}} T_g$.

En este marco, la terminación se define de la siguiente manera:

Definición 3.35: Sea $\mathcal{G}$ una instancia *SCT*. Una **traza** de $\mathcal{T}_\mathcal{G}$ es una secuencia finita o infinita de estados $s_0, s_1, s_2, \dots$ tal que $\forall i \in \mathbb{N}: (s_i, s_{i+1}) \in \mathcal{T}_\mathcal{G}$. Diremos que el **sistema de transiciones** $\mathcal{T}_\mathcal{G}$ es **uniformemente terminante** si no tiene trazas infinitas.

En consecuencia, una instancia *SCT* $\mathcal{G}$ es terminante si $\mathcal{T}_\mathcal{G}$ es uniformemente terminante.

Como hemos comentado anteriormente, la existencia de funciones de rango asegura la terminación de un programa y ellas mismas pueden utilizarse como testigos de terminación. Aunque en posteriores capítulos nos ocuparemos más en detalle de estas funciones y de qué manera inferirlas, pasamos a dar su definición formal adaptada a la notación alternativa que estamos utilizando en este apartado:

Definición 3.36: Sea $\mathcal{G}$ un conjunto de grafos de cambio de tamaño. Una función ρ de los estados de un programa a un preorden bien fundado $(\mathcal{D}, \succcurlyeq)$ es una **función de rango (global)** para $\mathcal{G}$ si $\forall p(\bar{x}) : -\pi ; q(\bar{y}) \in \mathcal{G} : \pi \vDash \rho(p(\bar{x})) \succ \rho(q(\bar{y}))$, donde $\succ$ es la parte estricta de $\succcurlyeq$.

Es decir, una función de rango es una función sobre el conjunto de estados de un programa a un conjunto bien fundado tal que decrece su valor para cada transición de estados. De esta forma, la terminación se deduce de forma trivial por la buena fundamentación del orden. El apelativo "global" se utiliza para diferenciar el concepto anterior del concepto de funciones de rango "locales" referentes a partes concretas de un programa.

La construcción de *SCNP* es un subconjunto de *SCT* basado en una serie de extensiones de los órdenes, del concepto de *level mapping* dado por los propios autores y de la definición anterior de función de rango. Los *level mapping* son, como veremos seguidamente, funciones sobre el conjunto de estados del programa a las extensiones de orden que pasamos a definir a continuación:

Definición 3.37: Sea $(\mathcal{D}, \succcurlyeq)$ un orden total y $(\mathcal{D}, \succ)$ su parte estricta. Sea $\mathcal{P}(\mathcal{D})$ el conjunto de multiconjuntos de $\mathcal{D}$ con al menos n elementos, donde n se fija por el contexto. El $\boldsymbol{\mu}$**-orden extendido** de $(\mathcal{D}, \succcurlyeq)$ para $\mu \in \{max, min, ms, dms\}$ es el preorden $(\mathcal{P}(\mathcal{D}), \succcurlyeq^{\mu})$ donde:

1. (max):

$$S \succcurlyeq^{max} T \Leftrightarrow max(S) \succcurlyeq max(T) \vee T = \emptyset$$
$$S \succ^{max} T \Leftrightarrow max(S) \succ max(T) \vee (T = \emptyset \wedge S \neq \emptyset)$$

2. (min):

   $S \succcurlyeq^{min} T \Leftrightarrow min(S) \succcurlyeq min(T) \vee S = \emptyset$
   $S \succ^{min} T \Leftrightarrow min(S) \succ min(T) \vee (S = \emptyset \wedge T \neq \emptyset)$

3. (ms):

   $S \succ^{ms} T \Leftrightarrow \exists U, V : \emptyset \neq U \subseteq S \wedge T = (S \backslash U) \cup V \wedge U \succ^{max} V$
   $S \succcurlyeq^{ms} T \Leftrightarrow S \succ^{ms} T \vee S = T$

4. (dms):

   $S \succ^{dms} T \Leftrightarrow \exists U, V : \emptyset \neq U \subseteq S \wedge V \neq \emptyset \wedge T = (S \backslash U) \cup V \wedge U \succ^{min} V$
   $S \succcurlyeq^{dms} T \Leftrightarrow S \succ^{dms} T \vee S = T$

El número n de la anterior definición será, en nuestro contexto, la mayor de las aridades en la instancia *SCT* $\mathcal{G}$.

Sean $S = \{4,3,3,0\}$ y $T = \{4,3,2,1,1\}$. Se tiene que $T >^{min} S, S \geq^{max} T$ y $T \geq^{max} S$. Además $S >^{ms} T$ tomando $U = \{3,0\}$ y $V = \{2,1,1\}$. No se tiene $S >^{dms} T$ pero si $T >^{dms} S$.

El siguiente resultado nos asegura que los μ-ordenes extendidos preservan la buena fundamentación:

Lema 3.38: Si $(\mathcal{D}, \succcurlyeq)$ es un orden total entonces $(\mathcal{P}(\mathcal{D}), \succcurlyeq^{\mu})$ es un orden total $\forall \mu \in \{max, min, ms, dms\}$. Si $(\mathcal{D}, \succcurlyeq)$ es bien fundado entonces $(\mathcal{P}(\mathcal{D}), \succcurlyeq^{\mu})$ es bien fundado $\forall \mu \in \{max, min, ms, dms\}$.

El concepto de *level mapping* lo introducen los autores como funciones sobre el conjunto de estados del programa a los anteriores conjuntos ordenados. Se tomaran tres tipos de funciones de este estilo que pasamos a ver:

Definición 3.39: Sea $\mathcal{G}$ un conjunto de grafos de cambio de tamaño que representa a un programa con N puntos de programa y donde M es la mayor de la aridades en $\mathcal{G}$. Un ***level mapping*** es una función f del conjunto de estados del programa a un cierto conjunto ordenado. En lo sucesivo, los *level mapping* sólo podrán ser de los siguientes tipos:

1. **Numéricos**: f asocia a cada estado del programa s un número natural $0 \leq f(s) < N$.
2. **Planos**: f asocia a cada estado del programa s un multiconjunto $\{v_1, v_2, \dots, v_k\} \in \mathcal{P}(Value)$.
3. **Etiquetados**: f asocia a cada estado del programa s un multiconjunto de pares $\{(v_1, n_1), (v_2, n_2), \dots, (v_k, n_k)\} \in \mathcal{P}(\mathcal{D} \times \mathbb{N})$ donde $n_1, n_2, \dots, n_k < M$ se denominan *etiquetas*.

Ya tenemos definidos todos los elementos necesarios para definir el conjunto *SCNP*:

Definición 3.40: Un conjunto de grafos de cambio de tamaños está en ***SCNP*** si tiene una función de rango tal que es una tupla de *level mappings*.

Tenemos que comprobar cuando una tupla de *level mappings* es en realidad una función de rango. En este sentido entra el concepto de *grafo orientado*.

Definición 3.41: Diremos que un ***level mapping*** f_μ **orienta a un SCG** $g = p(\bar{x}) :- \pi; q(\bar{y})$ si $\pi \vDash f(p(\bar{x})) \succcurlyeq^\mu f(q(\bar{y}))$. Diremos que f_μ **orienta estrictamente** a g si se cumple que $\pi \vDash f(p(\bar{x})) \succ^\mu f(q(\bar{y}))$. Diremos que f_μ orienta a un conjunto de grafos de cambio de tamaño $\mathcal{G}$ si orienta a todos sus elementos y orienta estrictamente a al menos uno de ellos.

El siguiente resultado nos da una condición necesaria y suficiente para verificar que una tupla de *level mappings* es una función de rango lexicográfica.

Lema 3.42: Sean $f^1, f^2, \dots, f^m$ *level mappings*. La siguiente función, tupla *de level mappings*, $\rho(s) =< f^1(s), f^2(s), \dots, f^m(s) >$ es una función de rango lexicográfica para el conjunto de grafos de cambio de tamaño $\mathcal{G}$ si y sólo si $\forall g \in \mathcal{G}\ \exists i \leq m$ tal que $f^1, f^2, \dots, f^i$ orientan a g, y f^i orienta estrictamente a g.

Para cada tipo de *level mapping* y para cada una de las extensiones de orden que hemos definido anteriormente se puede dar un algoritmo que verifique si un *level mapping* orienta (estrictamente) a una instancia *SCT*. En el caso de los *level mappings* numéricos, esta comprobación es trivial utilizando los órdenes $>$ y $\geq$ habituales de los enteros no negativos. Para los *level mappings* planos y etiquetados, supondremos que f_μ es el *level mapping* en cuestión y $g = p(\bar{x}) : -\pi; q(\bar{y})$ es el grafo al que se aplicará este test. Denotaremos por $S \subseteq \bar{x}$ al conjunto de posiciones de argumentos seleccionados por $f_\mu(p(\bar{x}))$ y de forma similar por $T \subseteq \bar{y}$ para los seleccionados por $f_\mu(q(\bar{y}))$.

Si f_μ es un *level mapping* plano, distinguimos casos según la extensión de orden:

1. (max): Comprobar $\forall y \in T\ \exists x \in S : \pi \vDash x \geq y$. Para comprobar orientación estricta es exactamente igual excepto $x > y$.
2. (min): Análogo al caso anterior tomando g^t, i.e., el grafo traspuesto.
3. (ms): Comprobar $\forall y \in T\ \exists x \in S : \pi \vDash x \geq y$. Para comprobar orientación estricta es exactamente igual excepto $x > y$.
4. (dms): Análogo al caso anterior tomando g^t.

Si f_μ es un *level mapping* etiquetado, la situación es análoga a la de los *level mappings* planos excepto que entran en juego las etiquetas. El test que debe realizarse se basa en las siguientes reglas:

$$\pi \vDash (x, i) \succ (y, j) \Leftrightarrow (\pi \vDash x \succcurlyeq y) \wedge (\pi \vDash x \succ y \vee i > j)$$

$$\pi \vDash (x, i) \succcurlyeq (y, j) \Leftrightarrow (\pi \vDash x \succ y) \vee (\pi \vDash x \succcurlyeq y \vee i \geq j)$$

La forma de calcular *level mappings* la veremos a continuación y adelantamos que deberá calcularse uno por cada componente fuertemente conexa no-trivial de la instancia *SCT* $\mathcal{G}$. Para cada uno estos *level mappings* se deberá comprobar el chequeo que acabamos de ver. Si finalmente existe la función de rango buscada, tupla de *level mappings*, deberá ser de la siguiente forma:

Definición 3.43: La **función**, $\rho(s) =< f^1(s), f^2(s), \ldots, f^m(s) >$, tupla de *level mappings*, es **irredundante** si $\forall i \leq m$: f^i orienta a todos los grafos no orientados estrictamente por f^j para algún $j < i$, y orienta estrictamente a al menos uno de ellos.

Comprobar la condición del Lema 3.42 es claramente polinomial y el tamaño de las funciones irredundantes también lo es puesto que como máximo pueden ser de longitud $|\mathcal{G}|$. Esto justifica el siguiente resultado:

Teorema 3.44: *SCNP* está en *NP*

Sabemos que todo problema en *NP* puede reducirse al problema *SAT*, este es, como dijimos anteriormente, el de dada una expresión booleana con variables y sin cuantificadores, obtener una asignación de valores de dichas variables que hagan cierta la expresión. Para poder aplicar un resolutor *SAT* se ha de codificar con variables booleanas todo el problema *SCNP*, para lo cual, remitimos al lector al artículo [TACAS' 08].

Puesto que es conocido que *SAT* es además, *NP*-completo, puede pensarse que su uso no sería adecuado en término de eficiencia, pero en [TACAS' 08] se demuestra que *SCNP* es también *NP*-completo por reducción al problema de *Set Covering (SC)* que también se sabe *NP*-completo. Por esta razón, la utilización de un resolutor *SAT* para resolver *SCNP* no supone un sobrecoste excesivo en la eficiencia.

Solo nos queda exponer el algoritmo *SCNP* que obtiene una función de rango de la forma que hemos comentado previamente:

Algoritmo *SCNP*

1. Inicializar ρ a la tupla vacía.
2. Desarrollar los siguientes pasos mientras $\mathcal{G}$ sea no vacío
3. Calcular las componentes fuertemente conexas (*SCCs*) de $\mathcal{G}$. Si $\mathcal{G}$ contiene transiciones entre sus componentes definir el level mapping numérico f a partir del orden topológico inverso de las *SCCs*. Extender ρ con f y eliminar dichas transiciones.
4. Si $\mathcal{G}$ tiene una *SCCs* no-triviales, aplicar los siguientes pasos para cada *SCC* $\mathcal{C}$:

a) Aplicar un resolutor *SAT* para calcular un *level mapping* f que oriente a $\mathcal{C}$. Si dicho *level mapping* no existe, finalizar el algoritmo con un mensaje de fallo.
b) Definir el valor de f como $\emptyset$ para todos los puntos de programas no contenidos en $\mathcal{C}$. Añadir f a ρ y eliminar las transiciones estrictamente orientadas por f.

Como hemos comentado, la codificación del problema para poder aplicar *SAT* puede encontrarse en [TACAS' 08]. A partir del valor de las variables booleanas usadas en dicha codificación puede obtenerse el *level mapping*, saber de qué tipo es y saber qué transiciones se orientan estrictamente por él.

Un resultado interesante que tenemos que destacar es la siguiente cadena de inclusiones:

$$SCP \subset SCNP \subset SCT$$

$SCNP \subset SCT$ se explica porque $SCNP$ es una condición correcta de terminación definida como subconjunto de SCT y porque hay instancias detectadas por SCT que no están en $SCNP$. Por otra parte, $SCNP$ incluye instancias no detectadas por SCP y $SCP \subseteq SCNP$ porque, intuitivamente, SCP no tiene en cuenta las extensiones de orden (max) y (min) que hemos visto anteriormente.

Esta técnica supone un avance puesto que cubre muchas de las instancias SCT, y en particular, como acabamos de ver, a todas las de *SCP*. Además de esto, $SCNP$ nos ofrece como resultado una función de rango del programa que nos sirve como testigo o certificado seguro de la terminación del programa, y con la cuál, pueden realizarse estudios sobre la complejidad del mismo, como se muestra en diversos trabajos entre los que destacamos el reciente artículo [SAS'10]. Los autores de esta técnica la han aplicado a dos *suites* de *benchmarks* obteniendo resultados bastante motivadores. Para el primer *suite*, formado por 123 programas, el algoritmo demuestra la terminación de 84 de ellos, dando sus correspondientes funciones de rango, en menos de 3 segundos. Para el segundo suite, formado por 4062 programas, el algoritmo demuestra la terminación de 3820 de ellos, dando su correspondiente función de rango, en aproximadamente 20 segundos.

Capítulo 4

Métodos Lineales

En este capítulo presentaremos técnicas de terminación de programas basados en métodos lineales. La base de los métodos que explicaremos es trabajar con una abstracción de los programas denominados *programas de tamaño* consistentes en identificar cada parámetro con su tamaño. El concepto de tamaño deberá ser definido, pero para los ejemplos que ofreceremos durante el capítulo, el lector puede pensar que el tamaño de una variable o constante entera es su valor, y el tamaño de una lista puede identificarse con su longitud. De estos programas de tamaño se pueden extraer las relaciones entre los tamaños de los parámetros. Las técnicas que explicaremos en este capítulo son útiles siempre que dichas relaciones sean lineales.

Los invariantes de un bucle nos proporcionan información sobre su comportamiento y en muchas ocasiones es útil para decidir terminación. De hecho, en trabajos como [SAS' 10] se utilizan como paso previo para decidir si un programa termina. En el apartado 4.1 presentaremos una técnica de inferencia de invariantes sobre las relaciones (lineales) de tamaño de los parámetros de un programa utilizando la abstracción de programas de tamaño. Esta técnica, presentada en varios trabajos como [BK' 96] o [Ben' 02], se basa en la técnica de interpretación abstracta sobre los programas de tamaño, utilizando como dominio abstracto el de los poliedros, o de otro modo, el de los conjuntos de restricciones lineales.

La existencia de funciones de rango para un programa nos asegura su terminación. Por esta razón, el método clásico de detección de terminación ha sido el de encontrar una función de rango para un cierto programa. Las funciones de rango pueden verse como testigos de la terminación de los programas y puede servir de utilidad en código con certificado. También podemos obtener información sobre la complejidad de los programas por medio de sus funciones de rango como muestra [SAS' 10]. En el apartado 4.2 hablaremos de diferentes técnicas de inferencia de funciones de rango lineales haciendo especial hincapié por la propuesta por Podelski y Rybalchenko [VMCAI' 04] por tratarse de un método completo con un coste computacional aceptable.

Las técnicas expuestas en ambos apartados las hemos adaptado para que puedan usarse en el lenguaje funcional de primer orden *Safe*, a pesar de que la técnica pensada para inferir invariantes esté pensada para lenguajes lógicos y la técnica de Podelski y Rybalchenko a la que hemos hecho referencia esté pensada para un tipo muy concreto de bucle. Esta adaptación la detallaremos en el Capítulo 5.

4.1. Inferencia de Invariantes

Estamos interesados en inferir invariantes que nos den información sobre de qué manera se relacionan los tamaños de los parámetros de los programas. Como antes hemos comentado, la forma de plantear esta cuestión es utilizar una abstracción de los programas o funciones consistente en identificar los parámetros por sus tamaños y que da como resultado lo que denominamos *programas o funciones de tamaño*. Para comprender mejor en qué consiste esta abstracción, daremos el ejemplo de la concatenación de dos listas. Para respetar la presentación de esta técnica dada en [BK' 96] escribiremos el ejemplo al estilo lógico:

$append([], s, s).$

$append([x|xs], s, [x|t]) : -append(xs, s, t).$

Tomaremos el convenio de identificar el tamaño de una lista por su longitud. Si f es la función original, denotamos por f^S a su versión de tamaño. Así pues, para este ejemplo, la función de tamaño de $append$ es la siguiente:

$append^S(o, s, s).$

$append^S(1 + r, s, 1 + t) : -append^S(r, s, t)$

Tras realizar la traducción de programas en programas de tamaño, esta técnica utiliza interpretación abstracta usando como dominio abstracto el de los poliedros. Los poliedros pueden ser vistos como conjuntos de puntos de $\mathbb{R}^n$ tales que cumplen una serie de restricciones lineales como vimos en el apartado 2.4. Cada caso base y recursivo de un programa de tamaño puede verse de la misma manera donde las

restricciones lineales son las relaciones de los tamaños de los parámetros. Por ejemplo, el caso base $append^S(o,s,s)$ podemos identificarlo con el poliedro $\{(r,s,t) \in \mathbb{R}^3 | \, r = 0 \wedge s = t\}$ el del caso recursivo de $append^S$ podemos identificarlo con el poliedro $\{(r+1, s, t+1) | (r,s,t) \in \mathbb{R}^3\}$.

El algoritmo utilizado para inferir invariantes se basa en el cálculo de una aproximación segura a un punto fijo que refleja el siguiente resultado:

Proposición 4.1: Sean $L(\sqsubseteq, \sqcup)$ un *cpo*, $F: L \to L$ contínua, $\perp \in L$ tal que $\perp \sqsubseteq F(\perp)$, y $\nabla: L \times L \to L$ un operador de ensanchamiento. Definimos la siguiente iteración como sigue donde $i, k \in \mathbb{N}$:

$$x_0 = \perp$$

$$x_{i+1} = x_i \qquad si\ F(x_i) \sqsubseteq x_i$$

$$x_{i+1} = F(x_i) \qquad si\ i \leq k\ \wedge\ F(x_i) \not\sqsubseteq x_i$$

$$x_{i+1} = x_i\ \nabla\ F(x_i) \qquad si\ i > k\ \wedge\ F(x_i) \not\sqsubseteq x_i$$

La anterior iteración converge, y su límite $\mathcal{A}$ es tal que $lfp(F) \sqsubseteq \mathcal{A}\ \wedge F(\mathcal{A}) \sqsubseteq \mathcal{A}$.

La entrada del algoritmo son los casos base y recursivos del programa de tamaño vistos como poliedros de la forma en la que hemos visto, y la función F del anterior resultado opera sobre ellos utilizando la envolvente convexa (*convex hull*) sobre ellos como veremos a continuación. Para nosotros el conjunto L de la Proposición será el conjunto de poliedros $Poly^n$, $\sqsubseteq$ será $\subseteq$, $\perp$ será el poliedro vacío $\emptyset$ y $\sqcup$ se corresponderá con la operación de envolvente convexa entre poliedros. Como pone de manifiesto la Proposición 2.34, esta estructura es un retículo.

La forma de operar de la función F puede explicarse del siguiente modo:

Notaremos por $\bar{x}, \bar{x}'$ a los parámetros de la llamada y a los parámetros modificados tras iterar respectivamente. Así pues, los casos base dependerán únicamente de $\bar{x}$ y los casos recursivos dependerán de $\bar{x}$ y $\bar{x}'$. Comprender esto tomando el ejemplo de $append$ para su caso base es obvio viendo su poliedro correspondiente. Para el caso recursivo, el poliedro dado puede verse como $\{(r', s', t') \in \mathbb{R}^3 | r' = r\ \wedge\ s' = s\ \wedge\ t' = t + 1\}$ por lo que depende de las $\bar{x}$ y de las $\bar{x}'$.

Supongamos que tenemos como entrada los poliedros correspondientes a los casos base $I_{B_1}(\bar{x}), I_{B_2}(\bar{x}), \ldots, I_{B_n}(\bar{x})$ y los poliedros correspondientes a los casos recursivos $I_{R_1}(\bar{x}, \bar{x}'), I_{R_2}(\bar{x}, \bar{x}'), \ldots, I_{R_m}(\bar{x}, \bar{x}')$. Los casos base hacen referencia al valor de los

parámetros $\bar{x}$ según el valor que tomen cada uno de ellos en cada caso base, y que deberá indicarse en las restricciones de cada uno de ellos, y los casos recursivos hacen referencia a $\bar{x}, \bar{x}'$ indicándonos de qué manera se modifican los parámetros ($\bar{x}'$) a partir de ellos mismos ($\bar{x}$) según las restricciones de cada poliedro como hemos visto en el ejemplo de $append$. Notaremos por $\overline{\cup}$ al operador de envolvente convexa entre poliedros, y denotaremos por $\mathcal{C}(\bar{y}) \downarrow \bar{z}$ al resultado de proyectar el conjunto de restricciones $\mathcal{C}$ sobre variables $\bar{y}$ en las variables $\bar{z}$, subconjunto de $\bar{y}$.

Sea $IB(\bar{x}) = \overline{\cup}_{i=1}^{n} I_{B_i}(\bar{x})$ el poliedro resultante de aplicar la envolvente convexa a todos los poliedros que representan los casos base, y sea $IR(\bar{x}, \bar{x}') = \overline{\cup}_{i=1}^{m} I_{R_i}(\bar{x}, \bar{x}')$ el poliedro obtenido análogamente para los casos recursivos, los cuales pueden obtenerse sin importar el orden de aplicación puesto que $\overline{\cup}$ cumple las propiedades conmutativa y asociativa. Notaremos por $IP(\bar{x}')$ al conjunto de restricciones previas que se obtienen al aplicar una iteración del algoritmo de punto fijo de la Proposición 4.1, i.e., se corresponden con las x_i de dicho algoritmo, donde $\perp$ es inicialmente el conjunto vacío de restricciones, esto es, el poliedro vacío. Con todo esto, la función F opera del modo siguiente:

1. $IRP(\bar{x}, \bar{x}') \coloneqq IR(\bar{x}, \bar{x}') \cap IP(\bar{x}')$ i.e., añadir las restricciones previas a IR.
2. $IRPP(\bar{x}) \coloneqq IRP(\bar{x}, \bar{x}') \downarrow \bar{x}$
3. $IN(\bar{x}) \coloneqq IRPP(\bar{x}) \,\overline{\cup}\, IB(\bar{x})$
4. $IP(\bar{x}') = IN(\bar{x}')$ tras renombrar variables en IN
5. Iterar los pasos anteriores

El número de iteraciones viene dado por $k \in \mathbb{N}$ de la Proposición 4.1. Una vez alcanzado dicho número de iteraciones, entra en juego el operador de ensanchamiento sobre los poliedros. Este operador de ensanchamiento opera sobre el poliedro resultante de la última iteración, sea P, y sobre el poliedro resultante de aplicar nuevamente la función F a P y que llamaremos P'. El operador de ensanchamiento comprueba que ambos poliedros son distintos. Si son distintos, el invariante del programa es el poliedro cuyas restricciones son las de P que estén implicadas por las de P' y si P y P' son iguales, se ha alcanzado un punto fijo, P, que es el invariante que buscábamos.

Nótese que tal y como acabamos de exponer este algoritmo, damos la primera iteración en vez de comenzar con el poliedro vacío. Esto lo hacemos porque en la primera iteración siempre se obtiene IB ya que IP es el conjunto vacío y por tanto IRP e $IRPP$ también lo serán y en el paso 3. en IN nos queda IB.

Retomamos el ejemplo de $append$ para $k = 3$ iteraciones. Denotaremos por P_i al poliedro obtenido en la i-ésima iteración. Como la notación de poliedros utilizando $\bar{x}$ y $\bar{x}'$ solo es útil a efectos de implementación, los únicos pasos relevantes del algoritmo anterior son el 1 y el 3 puesto que por esta razón $IRP = IRPP$.

Como $append$ sólo tiene un caso base y un caso recursivo, los poliedros IB e IR son los que hemos visto anteriormente:

$IB = \{(r, s, t) \in \mathbb{R}^3 | \, r = 0 \wedge s = t\}$

$IR = \{(r + 1, s, t + 1) | (r, s, t) \in \mathbb{R}^3\}$

En la primera obtendríamos IB como acabamos de razonar, por tanto:

$P_1 = IB = \{(r, s, t) \in \mathbb{R}^3 | \, r = 0 \wedge s = t\}$

Para la segunda iteración, en el paso 1 obtendremos $IRP = IR \cap P_1 = \{(r + 1, s, t + 1 \, r=0 \wedge s=t\}$. Al realizar la envolvente convexa en el paso 3 obtenemos:

$P_2 = IRP \cup IB = \{(r + 1, s, t + 1) \in \mathbb{R}^3 | \, r = 0 \ \wedge s = t\} \cup \{(r, s, t) \in \mathbb{R}^3 | \, r = 0 \wedge s = t\} = \{(r, s, t) \in \mathbb{R}^3 | \, r = 1 \ \wedge s = t - 1\} \cup \{(r, s, t) \in \mathbb{R}^3 | \, r = 0 \wedge s = t\} = \{(r, s, t) \in \mathbb{R}^3 | \, 0 \leq r \leq 1 \wedge t = r + s\}$

En la tercera y última iteración ($k = 3$) que daremos, en el paso 1 obtenemos $IRP = IR \cap P_2 = \{(r + 1, s, t + 1) \in \mathbb{R}^3 | \, 0 \leq r \leq 1 \wedge t = r + s\}$. Y finalmente:

$P_3 = IRP \cup IB = \{(r + 1, s, t + 1) \in \mathbb{R}^3 | \, 0 \leq r \leq 1 \wedge t = r + s\} \cup \{(r, s, t) \in \mathbb{R}3 \ \ r=0 \wedge s=t\}=r,s,t \in \mathbb{R}3 \ \ 1 \leq r \leq 2 \ \wedge t=r+s\} \cup r,s,t \in \mathbb{R}3 \ \ r=0 \wedge s=t\}=r,s,t \in \mathbb{R}3$
$0 \leq r \leq 2 \wedge t = r + s\}$

Podemos darnos cuenta de que esta secuencia nos da como resultado una cadena infinita del retículo de poliedros y que en general obtendremos en cada iteración:

$$P_j = \{(r, s, t) \in \mathbb{R}^3 | \, 0 \leq r \leq j - 1 \wedge t = r + s\}$$

Al llegar a la cota de iteraciones $k = 3$ que hemos dado en este ejemplo, tenemos que aplicar el operador de ensanchamiento. Este operador se aplica al último poliedro obtenido, P_3 y al siguiente:

$P_4 = \{(r, s, t) \in \mathbb{R}^3 | \ 0 \leq r \leq 3 \wedge t = r + s\}$

Como hemos comentado, este operador debe comprobar si las restricciones de P_3 implican a las restricciones de P_4, lo cual es cierto pues:

$$0 \leq r \leq 2 \wedge t = r + s \Rightarrow 0 \leq r \leq 3 \wedge t = r + s$$

Y por último, se obtiene el invariante como aquel poliedro cuyas restricciones son las restricciones de P_3 que son implicadas por las de P_4. En este ejemplo como $r \leq 3 \not\Rightarrow r \leq 2$, la restricción $r \leq 3$ queda eliminada y obtenemos como resultado:

$$Inv = \{(r, s, t) \in \mathbb{R}^3 | \ r \geq 0 \ \wedge t = r + s\}$$

El invariante obtenido por el algoritmo nos dice que el tamaño de la lista resultado de la concatenación es la suma de los tamaños de las dos listas que concatenamos, lo cual es el resultado que esperábamos obtener.

4.2. Inferencia de Funciones de Rango

El método más clásico y el más utilizado para detectar la terminación de un programa consiste en obtener una *función de rango (ranking function)* del mismo como ya comentamos en el Capítulo 3 cuando hablamos de la técnica *SCNP*. Sin detenernos en formalismos matemáticos concretos como vimos en *SCNP*, podemos definir en general una función de rango como una función sobre el conjunto de estados de un programa a un conjunto bien fundado tal que decrece su valor para cada transición de estados.

La existencia de estas funciones para un programa nos asegura la terminación del mismo. La suposición de órdenes bien fundados nos asegura la existencia de un elemento mínimo en el conjunto y por analogía a lo que hemos explicado en el marco *SCT* no podrá haber hebras infinitamente descendentes puesto que al alcanzarse dicho elemento mínimo no habrá posibilidad de decrecimiento del tamaño de los datos y nos aseguraremos que el programa es terminante.

Esta forma de detección de la terminación de los programas nos ofrece ventajas muy relevantes. Por un lado, tiene interés en el ámbito de la certificación de código. En este campo, los programas incluyen una demostración matemática (certificado) de que el programa cumple una cierta propiedad. Puesto que la existencia de funciones de rango aseguran la terminación de los programas, una función de rango puede verse como testigo o certificado de la terminación de un programa e incluirse como prueba de que dicho programa cumple en efecto la propiedad de que es terminante. Aquella persona que se encargue de comprobar que en efecto esta información es válida, solamente tendrá que demostrar que la función incluida como certificado cumple en efecto con la definición de función de rango del programa, lo cual puede llevarse a cabo mediante algún demostrador de teoremas. Por otra parte, las funciones de rango nos ofrecen información acerca de la complejidad de los programas como muestran algunos trabajos como por ejemplo el reciente [SAS' 10].

Dentro de la literatura actual sobre este campo vamos a hacer especial hincapié en el trabajo [VMCAI' 04] de Podelski y Rybalchenko. En él se presenta un método completo de inferencia de funciones de rango lineales y por tanto de terminación, para un tipo muy concreto, pero muy frecuente, de programas que ellos denominan *LASW* que detallaremos a continuación. Por supuesto existen muchos otros métodos de terminación basados en métodos lineales, como por ejemplo el presentado en [CAV' 02], pero pensamos que la completitud del método de Podelski y Rybalchenko le da especial relevancia. Esta elección también se debe a que en otros trabajos como por ejemplo en [ICLP' 05] se presentan casos particulares ya cubiertos por la propuesta de Podelski y Rybalchenko.

Como veremos en el Capítulo 5, hemos implementado la técnica propuesta por Podelski y Rybalchenko pensando en aplicarla en programas escritos en un lenguaje funcional de primer orden denominado *Safe*. En [IFL' 10] presentamos los resultados obtenidos al aplicarlo.

Como hemos comentado atrás, el método de Podelski y Rybalchenko está pensado para un tipo de programas muy sencillos pero frecuentes denominados *LASW (linear arithmetic simple while)*. Partiendo de un programa de este estilo se construye un sistema de inecuaciones lineales como explicaremos más adelante. Aplicando a dicho sistema técnicas de programación lineal sobre números racionales se obtienen una serie de coeficientes que nos permitirán construir la función de rango buscada, demostrando consiguientemente la terminación de dicho programa. Un programa puede frecuentemente dividirse en varios bucles *LASW*, y la terminación del mismo vendrá implicada tras demostrar mediante esta técnica la terminación de cada bucle *LASW*.

Pasamos a describir como son los programas *LASW* con los que trabajaremos:

Definición 4.2: Un **programa *LASW (linear arithmetic simple while)*** sobre las variables $x = (x_1, x_2, \dots, x_n)$ es de la forma:

$$while\ B_1 \wedge B_2 \wedge \dots \wedge B_m\ do\ (x_{i_1}, x_{i_2}, \dots, x_{i_r}) \coloneqq (e_{i_1}, e_{i_2}, \dots, e_{i_r})$$

donde en el cuerpo del bucle solo se admite asignación múltiple, y donde se cumple lo siguiente:

1. Todas las variables x_i son enteras
2. Cada B_j es una inecuación lineal de la forma $c_1x_1 + c_2x_2 + \cdots + c_nx_n \leq c_0$
3. Cada asignación $x_{i_j} \coloneqq e_{i_j}$ puede traducirse a una inecuación lineal sobre x y sobre x', donde x' representa el valor de las x_i tras una o varias iteraciones, de la siguiente forma: $a'_1x'_1 + a'_2x'_2 + \cdots + a'_nx'_n \leq a_1x_1 + a_2x_2 + \cdots + a_nx_n + a_0$

Identificando a los estados de un programa de este estilo mediante la letra s, su relación de transición consiste en todos los pares (s, s') tales que s satisface la condición $B_1 \wedge B_2 \wedge \dots \wedge B_m$ y (s, s') satisface la asignación múltiple del cuerpo del programa *LASW*.

Por la forma de estos programas, podemos denotarlos mediante notación matricial de la siguiente manera:

$$(AA')\begin{pmatrix} x \\ x' \end{pmatrix} \leq b$$

donde la notación (AA') es la yuxtaposición de las matrices A y A' correspondientes a los coeficientes a_i y a'_i de la definición anterior. Para aclarar las ideas, veamos un ejemplo de programa *LASW* y su correspondiente representación matricial:

$$while\ i - j \geq 1\ do\ (i, j) \coloneqq (i - N, j + M)$$

dónde $N \geq 0$ y $M > 0$ son constantes.

Tendremos:

$$A = \begin{pmatrix} -1 & 1 \\ -1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$A' = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$b = \begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix}$$

El número de columnas de A y A' es el número de variables de x, es decir, n columnas, mientras que el número de filas, sea m, depende de la condición y del número de asignaciones dentro del bucle. Así pues, suponiendo que A y A' tienen dimensión $m \times n$ entonces su yuxtaposición tendrá dimensión $m \times 2n$ y el vector columna b tendrá dimensión $m \times 1$.

Con los elementos que llevamos presentados ya podemos definir el criterio de terminación de un programa de este tipo:

Definición 4.3: Un **programa *LASW*** es **terminante** si no existe una secuencia $\{s_i\}_{i=1}^{\infty}$ tal que $\forall i \geq 1$: (s_i, s_{i+1}) satisface:

$$(AA') \begin{pmatrix} x \\ x' \end{pmatrix} \leq b$$

Como hemos comentado anteriormente, una vez obtenido el sistema de inecuaciones lineales asociado a un programa *LASW*, la forma de de encontrar una función de rango para el programa se consigue comprobando la satisfactibilidad de un conjunto de inecuaciones. El siguiente resultado nos justifica que el uso de métodos lineales sobre estos sistemas de inecuaciones nos permite detectar su terminación. Además, señalamos cómo utilizar los resultados obtenidos al aplicar los mencionados métodos lineales para construir la función de rango buscada.

Teorema 4.4: Dado un programa *LASW* en notación matricial:

$$P \equiv (AA') \begin{pmatrix} x \\ x' \end{pmatrix} \leq b$$

Dicho programa es terminante si existen vectores $\lambda_1, \lambda_2 \geq \bar{0}$ tales que satisfacen el siguiente sistema:

1. $\lambda_1 A' = 0_{1\times n}$
2. $(\lambda_1 - \lambda_2)A = 0_{1\times n}$
3. $\lambda_2(A + A') = 0_{1\times n}$
4. $\lambda_2 b < 0_{1\times 1}$

Suponiendo que al aplicar este chequeo de terminación a un programa *LASW* concluimos que es terminante, sean λ_1, λ_2 una solución cualquiera del sistema anterior. La siguiente función de rango lineal nos asegura la terminación del programa:

$$\rho(x) \stackrel{\text{def}}{=} \begin{cases} rx & si\ \exists\ x'\ tal\ que\ cumple\ P \\ \delta_0 - \delta & en\ caso\ contrario \end{cases}$$

dónde se definen:

$$r \stackrel{\text{def}}{=} \lambda_2 A'$$

$$\delta_0 \stackrel{\text{def}}{=} -\lambda_1 b$$

$$\delta \stackrel{\text{def}}{=} -\lambda_2 b$$

El Teorema 4.4 nos establece una condición de terminación suficiente. El siguiente resultado nos asegurará que dicha condición también es necesaria con lo que estableceremos que este método es en efecto completo para sintetizar funciones de rango lineales en bucles *LASW* si estas existen:

Teorema 4.5: Si existe una función de rango lineal para un programa *LASW* entonces la condición de terminación del Teorema 4.4 se cumple.

En [VMCAI' 04] se demuestra este resultado haciendo uso de un resultado utilizado en Investigación Operativa conocido como Lema de Farkas.

De los teoremas 4.4 y 4.5 y dado que existen algoritmos polinomiales para resolver sistemas de inecuaciones lineales se deduce el siguiente corolario:

Corolario 4.6: La existencia de funciones de rango lineales para programas *LASW* es decidible en tiempo polinomial.

Aplicando el Teorema 4.4 al ejemplo visto anteriormente obtenemos los siguientes resultados. Sean $\lambda_1 = (\lambda_{11}, \lambda_{12}, \lambda_{13})$, $\lambda_2 = (\lambda_{21}, \lambda_{22}, \lambda_{23})$, entonces:

$$\lambda_{12} = \lambda_{13} = \lambda_{21} = 0$$

$$\lambda_{11} = \lambda_{22} = \lambda_{23} > 0$$

Tomando $\lambda_{11} = \lambda_{22} = \lambda_{23} = 1$, se obtiene lo siguiente:

$$r = (1, -1)$$

$$\delta = 1$$

$$\delta_0 = 1$$

Y en consecuencia, la función de rango lineal obtenida es la que mostramos a continuación:

$$\rho(i,j) = \begin{cases} i - j & si\ i - j \geq 1 \\ 0 & e.o.c \end{cases}$$

Los autores presentan algunos resultados obtenidos por el método propuesto al aplicarlo sobre ejemplos concretos. Cabe destacar el ejemplo de aplicación sobre un programa que calcula la descomposición de una matriz en sus valores singulares (*SVD*), en el que para demostrar su terminación, testean la terminación de 219 programas *LASW* tardando en total 800ms (3,6 ms/prog. *LASW*).

El principal inconveniente de la técnica de Podelski y Rybalchenko es que aunque los programas *LASW* aparezcan con frecuencia, se trata de un tipo de programas muy limitado. Por ejemplo, la condición del bucle debe ser una conjunción, o en el cuerpo del bucle solo se admiten asignaciones. Los propios Podelski y Rybalchenko nos presentan en [LICS' 04] los conceptos de invariantes de transición (*transition invariants*) y de relación disyuntivamente bien fundada (*disjunctively well-founded*) que permiten ampliar la gama de programas para los que se puede chequear su terminación.

Para explicar estos dos conceptos fundamentales presentados en [LICS' 04] describiremos como son los programas que allí se presentan:

Definición 4.7: Un **programa** es una tupla $P = < W, I, R >$ dónde W es el *conjunto de estados* del programa P, $I \subseteq W$ es el *conjunto de estados iniciales* del programa P y $R \subseteq W\ x\ W$ es la *relación de transición* del programa P.

Denotando por $\mathbb{N}^+ = \mathbb{N}\backslash\{0\}$ al conjunto de los enteros estrictamente positivos, damos la siguiente noción de cómputo de un programa:

Definición 4.8: Un **cómputo de un programa** $P = < W, I, R >$ es una secuencia de estados $s_1, s_2, \ldots$ dónde $\forall\, i \in \mathbb{N}^+ \colon s_i \in W \colon \quad s_1 \in I \wedge \forall\, i \in \mathbb{N}^+ \colon (s_i, s_{i+1}) \in R$, (e $i \leq n$ si se trata de una secuencia finita de longitud n).

Dado un programa P, se define su *conjunto de estados accesibles* Acc como aquel que contiene a todos los estados que aparecen en algún cómputo de P.

Definición 4.9: Un **invariante de transición (*transition invariant*) de un programa** $P = < W, I, R >$ es $T \subseteq W \times W$ tal que $R^+ \cap (Acc\ x\ Acc) \subseteq T$**.**

Definición 4.10: Una **relación** $T = \bigcup_{i=1}^{n} T_i$ es **disyuntivamente bien fundada (*disjunctively well-founded*)** si $\forall\, i \in \{1,2, \ldots, n\} \colon T_i$ es bien fundada.

Una vez presentados estos conceptos los autores dan una caracterización de los programas terminantes a partir del siguiente teorema:

Teorema 4.11: Un programa P es terminante si y sólo si existe un invariante de transición T para P disyuntivamente bien fundado.

El principal aporte dado en [LICS' 04] es la introducción de los conceptos de invariante de transición y de relación disyuntivamente bien fundada, a partir de los cuáles se da una caracterización general de los programas terminantes. No obstante la técnica presentada está pensada para verificar propiedades de vitalidad de programas en general por lo que no se centra únicamente en su terminación, objeto del presente trabajo.

En [ESOP'08] se utiliza el concepto de relación disyuntivamente bien fundada para desarrollar la técnica de terminación de programas ahí presentada. En la técnica presentada en dicho artículo los autores obtienen un conjunto finito de funciones de rango mediante cómputos iterados de cálculo de puntos fijos utilizando interpretación abstracta, de forma que una vez alcanzado el punto fijo se garantiza que la unión (finita) de dichas funciones de rango es disyuntivamente bien fundada, y por lo tanto el programa termina por el Teorema 4.11.

Capítulo 5

Una aplicación a la Programación Funcional

En este Capítulo explicaremos de manera resumida la implementación que hemos realizado de la técnica de inferencia de invariantes de relaciones de tamaño expuesta en [BK' 96] y que hemos comentado en el apartado 4.1 y del método de síntesis de funciones de rango lineales propuesto por Podelski y Rybalchenko en [VMCAI' 04] que hemos presentado en el apartado 4.2. El objetivo de esta implementación es aplicarla al lenguaje funcional de primer orden *Safe* y en particular, en una versión reducida de este lenguaje, denominada *Core-Safe*, en la cual ya se aplican otro tipo de análisis.

Para implementar estas técnicas hemos escogido el lenguaje de programación Prolog por varias razones. En primer lugar, Prolog es capaz de manejar restricciones lineales, usadas en ambas técnicas. En segundo lugar, Prolog incluye la librería *simplex* con herramientas de programación lineal, que nos ha sido de utilidad para resolver el sistema de inecuaciones propuesto por Podelski y Rybalchenko y que hemos visto en el Teorema 4.4.

En el apartado 5.1 explicaremos brevemente el lenguaje de programación *Safe*, y en particular una versión simplificada del mismo denominada *Core-Safe* de la que daremos su sintaxis. En el apartado 5.2 explicaremos la implementación de la técnica de inferencia de invariantes propuesta por [BK' 96]. El algoritmo es exactamente el mismo que explicamos en el apartado 4.1 pero necesita una transformación previa del programa *Core-Safe*. En el apartado 5.3 explicaremos la implementación que hemos realizado del método completo de síntesis de funciones de rango lineales de Podelski y Rybalchenko [VMCAI' 04] que hemos explicado con anterioridad. Como veremos en este apartado, hemos adaptado este método para obtener la mínima función de rango de una función recursiva.

Los resultados expuestos en este capítulo los hemos plasmado en el artículo [IFL' 10] en el que hemos presentado algunos ejemplos de las respuestas ofrecidas por nuestro programa. El artículo completo [IFL' 10] se añade en el Apéndice B de este trabajo.

El lector puede consultar el código fuente de nuestra implementación en el Apéndice A de la presente memoria para que pueda ver con más detalle los diseños de las funciones a las que haremos referencia en los diferentes apartados.

5.1. El lenguaje funcional *Safe*

El principal objetivo de este apartado es describir brevemente el lenguaje *Safe* y su versión reducida *Core-Safe* sobre la que se utilizará nuestra implementación. Explicaremos a grandes rasgos las principales características del lenguaje y daremos la sintaxis de los programas escritos en *Core-Safe*.

Safe es un lenguaje funcional impaciente de primer orden con una sintaxis similar a Haskell y con la particularidad de que incluye facilidades para la destrucción explícita de memoria y la copia de estructuras de datos. Como característica adicional, *Safe* proporciona *regiones* (partes disjuntas de la memoria) en las que el programador puede almacenar estructuras de datos. *Safe* puede enmarcarse en el área de investigación de código con certificado o código con demostración asociada (*Proof Carrying Code* o *PCC*). El lenguaje *Safe* forma parte del proyecto *SELF* (*Software Engineering and Lightweight Formalisms*) llevado a cabo de 2005 a 2008 por las Universidades de Castilla-La Mancha (UCLM), Politécnica de Valencia (UPV), de Málaga (UMA) y la Complutense de Madrid (UCM) de la cual participan miembros del Departamento de Sistemas Informáticos y Computación (DSIC).

Core-Safe es una versión reducida del lenguaje *Safe*. Puede verse como el lenguaje núcleo de *Safe* en el que se definen diferentes análisis. Algunos de estos análisis se refieren a la inferencia de tipos seguros o a cuestiones referentes a la memoria como por ejemplo, la ausencia de punteros descolgados durante la ejecución.

Las implementaciones que presentaremos en los apartados 5.2 y 5.3 están pensadas para programas *Core-Safe*. A continuación presentamos la sintaxis del lenguaje:

$$prog \rightarrow \overline{data_i}; \overline{dec_j}; e \qquad \{Core - Safe\ program\}$$

$$dec \rightarrow f\,\overline{x_i} = e \quad \{recursive, polymorphic\ function\ definition\}$$

$$e \rightarrow a \quad \{atom\ a\text{: either a literal c or a variable } x\}$$

$$\mid a_1 \oplus a_2 \quad \{primitive\ operator\ application\}$$

$$\mid f\,\overline{a_i} \qquad \{function\ application\}$$

$$\mid C\,\overline{a_i} \qquad \{constructor\ application\}$$

$$\mid \textbf{\textit{let}}\ x_1 = e_1\ \textbf{\textit{in}}\ e_2\ \{non - recursive, monomorphic\ \textbf{\textit{let}}\}$$

$$| \ \boldsymbol{case}\ x\ \boldsymbol{of}\ \overline{alt_i} \qquad \{case\ expression\}$$

$$alt \rightarrow C\overline{x_i} \rightarrow e \qquad \{case\ alternative\}$$

Un programa es una secuencia de datos polimórficos posiblemente recursivos seguidos por una serie de definiciones de funciones recursivas y por una expresión *e* cuyo valor será el resultado del programa.

A modo de ejemplo, mostraremos la versión *Core-Safe* del algoritmo *mergesort* dando por supuesto que se dispone de las funciones auxiliares *split* y *merge*:

```
msort x = case x of
            [] -> []
            ex:x' -> case x' of
                       [] -> ex:[]
                       _:_ -> let n = length x in
                              let n2 = n/2 in
                              let (x1,x2) = split n2 x in
                              let z1 = msort x1 in
                              let z2 = msort x2 in
                              merge z1 z2
```

5.2. Síntesis de Invariantes

El algoritmo que hemos implementado es una adaptación del explicado en el apartado 4.1 de este trabajo y que puede encontrarse en [BK' 96] o en [Ben' 02]. Como hemos visto anteriormente, el dominio utilizado en este análisis es el de los poliedros, o de otro modo, el de los conjuntos de restricciones lineales. El manejo que hace Prolog de las restricciones lineales gracias a sus librerías *clpq* y *clpr* ha sido una de las razones por las que nos hemos decantado por este lenguaje de programación a la hora de implementar.

Como hemos explicado en 4.1, el método requiere que pasemos como entrada los casos base y los casos recursivos del programa. El siguiente algoritmo, escrito en pseudo-código al estilo Haskell, extrae los casos base y los casos recursivos de una expresión *Core-Safe* identificando los primeros mediante una etiqueta B y a los segundos mediante una etiqueta R.

$$\boldsymbol{data}\ QExp = B\ Exp \mid R\ Exp \mid Var \coloneqq [QExp]$$

$$seqs_f :: Exp \to [[QExp]]$$

$$seqs_f\ e = \left[[Be]\right] \quad -- \ si\ e \in \{c, x, a_1 \oplus a_2, g\overline{a_i}, C\overline{a_i}\}$$

$$seqs_f(f\overline{a_i}) = [[R\ f\overline{a_i}]]$$

$$seqs_f(\boldsymbol{case}\ e\ \boldsymbol{of}\ alts) = concat\ [seqs_f\ e\ |(C\overline{x_i} \to e) \in alts]$$

$$seqs_f(\boldsymbol{let}\ x_1 = e_1\ \boldsymbol{in}\ e_2) = [(\ x_1 \coloneqq s_1) : s_2 \mid s_1 \in seqs_f e_1,\ s_2 \in seqs_f e_2]$$

Un paso previo que hemos de realizar para hacer este análisis es una traducción de los programas *Core-Safe* a unos programas abstractos en los que las estructuras de datos se sustituyen por sus *tamaños*. En *Core-Safe* el concepto de *tamaño* se toma de su modelo de memoria y se basa en el número de celdas que ocupa cada estructura de datos. Por ejemplo, para una lista de n elementos, tiene tamaño $n + 1$ puesto que la lista vacía ocupa una celda de memoria. Para constantes o variables enteras, su tamaño será su valor, y para constantes o variables booleanas se tomará el tamaño 0.

Así pues, tomando el ejemplo visto anteriormente del algoritmo *mergesort* , el resultado de la traducción será el siguiente:

```
msort^S x = case x of
              x = 1 -> 1
              x _ 2 -> case x of
                         x = 2 ->2
                         x ≥3 -> let n = length x in
                                 let n2 = n=2 in
                                 let (x1; x2) = split n2 x in
                                 let z1 = msort x1 in
                                 let z2 = msort x2 in
                                 merge z1 z2
```

Denominaremos *programas de tamaño* o *funciones de tamaño* a estos programas o funciones abstractos resultantes de la transformación que acabamos de explicar. Si f es la función original, denotamos por f^S a su versión de tamaño.

El siguiente paso consiste en aplicar una interpretación abstracta sobre las funciones de tamaño. Tomaremos como dominio abstracto el retículo de poliedros $(Poly^n, \subseteq, \overline{\cup}, \cap)$ que vimos en el apartado 2.4. La implementación del cálculo de envolventes convexas y de proyección de un conjunto de restricciones lineales sobre un

subconjunto de variables sobre las que están definidas, realizada en Prolog, está tomada del trabajo [TPLP' 05] de F. Benoy y A. King. La forma de obtener los poliedros asociados a los casos base y los casos recursivos de los programas consiste en aplicar un algoritmo semejante a $seqs_f$ sobre la función de tamaño.

A continuación explicaremos los predicados Prolog más relevantes. Seguiremos el convenio habitual en programación lógica de indicar con un signo "-" los parámetros de entrada y con un signo "+" a los parámetros de salida.

En nuestra implementación aplicamos el algoritmo explicado en 4.1 obviando los pasos previos que acabamos de explicar de traducción del programa original a su programa de tamaño asociado y obtención de los poliedros correspondientes a los casos base y los casos recursivos. La función que aplica la iteración de la Proposición 4.1 es la siguiente:

fixpoint(-CBase,-CRecursivos,-K,+Poliedro)

Donde *CBase* y *CRecursivos* son listas que tienen como componentes los poliedros correspondientes a cada caso base y a casa caso recursivo respectivamente. Estos poliedros son a su vez listas de restricciones lineales. *K* es el coeficiente que nos marca el número de iteraciones antes de empezar a aplicar el operador de ensanchamiento. Por último, *Poliedro* representa el invariante inferido por este algoritmo de punto fijo.

En la sección 4.1 enumeramos las operaciones que tiene que realizar la función F de la Proposición 4.1, que tiene como función definir un paso de la iteración del algoritmo. La función que implementa las operaciones a las que hacemos referencia es la siguiente:

operador(-(VP,RP),+(VN,RN))

(VP,RP) representa al poliedro obtenido durante la iteración anterior, donde *RP* son las restricciones lineales sobre las variables *VP* que describen al poliedro. La función *operador* aplica las operaciones que detallamos en la sección 4.1 dando lugar al nuevo poliedro *(VN,RN)* donde *RN* son las restricciones lineales que describen al poliedro y que vienen definidas sobre las variables *VN*.

Una vez agotadas el número de iteraciones, el algoritmo aplica un operador de ensanchamiento sobre poliedros. Este operador opera sobre el último poliedro obtenido durante las iteraciones, sea P, y sobre el poliedro resultante de aplicar al primero la función *operador*, sea P'. El operador comprueba si ambos poliedros son iguales. En ese caso, se ha alcanzado un punto fijo y se devuelve P como el invariante buscado. Si P y P' son distintos, el invariante inferido es el poliedro cuyas restricciones son aquellas de P que están implicadas por P', y por la Proposición 4.1, sabemos que es una aproximación segura del punto fijo. La comprobación de implicación lógica

entre restricciones lineales nos la proporciona el predicado *entailed/1* incluido en la librería *clpq* de Prolog.

A continuación vemos la respuesta de nuestro programa para el ejemplo de la concatenación de dos listas que vimos en 4.1, obteniendo el resultado esperado:

```
fixpoint([([R1,S1,T1],[R1=0,S1=T1])],
        [([R2,S2,T2,R3,S3,T3],[R3=R2+1,S3=S2,T3=1+T2])]
        ,3,
        Poliedro).
```

Poliedro = ([R1, S1, T1], [R1>=0, S1= -R1+T1]),

5.3. Síntesis de Funciones de Rango

Hemos implementado el método completo de inferencia de funciones de rango lineales para bucles *LASW* propuesto por Podelski y Rybalchenko en [VMCAI' 04]. Como explicaremos seguidamente, nuestro propósito es más ambicioso que el de demostrar únicamente la terminación de un programa obteniendo como certificado una función de rango. Por esta razón, hemos hecho una adaptación del algoritmo de Podelski y Rybalchenko que detallaremos más abajo.

El *resolutor* de programación lineal sobre racionales usado por los autores para resolver los sistemas de inecuaciones es el presentado en el artículo [Hol' 95]. En la implementación de esta técnica que hemos realizado hemos utilizado la librería *simplex* incluida en el lenguaje de programación lógico Prolog.

Como vimos en 4.2, los bucles con los que trabaja este método son de la forma $\boldsymbol{while}\ B\ \boldsymbol{do}\ S$, donde $B(\bar{x})$ es una conjunción de restricciones lineales sobre las variables $\bar{x}$ que aparecen en el cuerpo del bucle y $S(\bar{x}, \bar{x}')$ es una relación de transición expresada mediante una conjunción de restricciones lineales sobre los valores de las variables antes y después de ejecutar el cuerpo del bucle, los cuales representaremos mediante $\bar{x}$ y $\bar{x}'$ respectivamente.

Usando las restricciones antes mencionadas, en la implementación generamos el sistema de inecuaciones que aparece en el Teorema 4.4 sobre los vectores $\overline{\lambda_1}$ y $\overline{\lambda_2}$.

Como ya sabemos por el citado teorema, este sistema de inecuaciones es factible si existe una función de rango lineal para el bucle, y como también vimos, podemos sintetizar dicha función de rango a partir de los valores obtenidos de $\overline{\lambda_1}$ y $\overline{\lambda_2}$ tras resolver el sistema. Más en concreto, el método de Podelski y Rybalchenko sintetiza un vector de números reales $\bar{r}$ y dos constantes $\delta > 0$ y δ_0 de forma que verifican:

$\bar{r}\bar{x} \geq \delta_0 \qquad \forall \bar{x}: B(\bar{x})$

$\bar{r}\bar{x}' \leq \bar{r}\bar{x} - \delta \qquad \forall \bar{x}, \bar{x}': B(\bar{x}) \wedge S(\bar{x}, \bar{x}')$

Estas dos condiciones aseguran la terminación del bucle.

Además de asegurar la terminación de un programa dando como certificado una función de rango, nos proponemos el objetivo adicional de buscar la *menor cota superior* de la longitud de la cadena de llamadas en el caso peor para una función recursiva *Core-Safe* f. Para ello hemos realizado dos adaptaciones al método:

En primer lugar, hemos reemplazado la restricción $\delta > 0$ por $\delta \geq 1$. De esta forma, cada transición se cuenta al menos como una llamada interna a la función f por lo que obtendremos una cota superior al número de llamadas internas en la cadena.

En segundo lugar, hemos reformulado el problema como un problema de minimización en el que pedimos minimizar la siguiente función objetivo:

$$Obj \stackrel{\text{def}}{=} \sum \overline{\lambda_1} + \sum \overline{\lambda_2} - \delta_0$$

Minimizar $-\delta_0$ equivale a maximizar δ_0 con lo que conseguimos expresar que buscamos el menor valor de $\bar{r}\bar{x}$ que es cota superior. Minimizamos los valores de los vectores $\overline{\lambda_1}$ y $\overline{\lambda_2}$ puesto que hemos observado que si solo minimizamos $-\delta_0$ solemos obtener problemas sin acotación y con infinitas soluciones en las que alcanza el mínimo con algún valor λ_{ij} infinito.

La función que tomaremos como la cota buscada será $\bar{r}\bar{x} - \delta_0 + 2$ puesto que $\bar{r}\bar{x} - \delta_0 \geq 0$ es una cota superior del número de iteraciones (cada una, correspondiendo con una llamada interna a f) y añadimos 2 para tener en cuenta las llamadas de antes y después de ejecutar el bucle, estas son, las llamadas inicial y final de la cadena. Por supuesto, esta cota es válida siempre que se cumpla $B(\bar{x})$. En otro caso, la longitud de la cadena es 1.

Nos falta por explicar cómo adaptar los programas *Core-Safe* a los bucles *LASW* para los que está definida la técnica de Podelski y Rybalchenko ya que en *Core-Safe* no existe la estructura de bucle. La idea es interpretar las llamadas recursivas como bucles, y para ello utilizamos la aproximación dada por la envolvente convexa de las restricciones proporcionadas por las llamadas internas. Estas restricciones expresan la variación de tamaño de los argumentos con respecto a los de la llamada externa. De esa envolvente convexa, las restricciones que dependan solo de $\bar{x}$ se interpretan como la guarda $B(\bar{x})$ y las que dependan de $\bar{x}$ y $\bar{x}'$ se interpretan como la relación de transición $S(\bar{x}, \bar{x}')$.

Para más detalles y ejemplos de aplicación, véase el texto completo del artículo [IFL' 10] incluido en el Apéndice B.

Capítulo 6

Conclusiones y Trabajo Futuro

El objetivo de este capítulo es dar algunas conclusiones que podemos extraer de todo lo expuesto anteriormente y dar algunas pautas sobre el trabajo futuro a seguir de la implementación explicada en el Capítulo 5.

En los últimos años se han publicado un número muy considerable de trabajos en torno a la terminación de programas. Los trabajos más relevantes en este campo podemos dividirlos en dos familias bien diferenciadas. Por un lado, aquellos trabajos que utilizan métodos lineales para detectar terminación y por otro lado aquellos trabajos basados en la técnica *SCT* presentada originalmente en [POPL' 01]. Ambas visiones del problema no son incompatibles puesto que si aquellas técnicas basadas en métodos lineales por lo general buscan inferir funciones de rango, como hemos visto en el apartado dedicado a *SCNP*, este objetivo puede conseguirse partiendo también de la técnica *SCT*.

Muchas de estas técnicas se han llevado a la práctica obteniendo resultados prometedores para un amplio espectro de programas que aparecen con bastante frecuencia y con costes computacionales aceptables.

Hemos visto distintas formas de abordar la terminación mediante diversas representaciones de los programas y utilizando diferentes marcos matemáticos como la interpretación abstracta o los grafos. No obstante, la terminación puede tratarse desde otros contextos como por ejemplo los sistemas de reescritura. El lector interesado puede consultar trabajos como [AAECC' 05] en donde se estudia la aplicación de la técnica *SCT* para detectar terminación en sistemas de reescritura.

Hemos implementado un algoritmo de punto fijo que infiere los invariantes de un programa según la técnica expuesta en [BK' 96] y el método de inferencia de funciones de rango propuesto por Podelski y Rybalchenko en [VMCAI' 04]. La intención es utilizarlo en el lenguaje funcional de primer orden *Safe*, en su versión *Core-Safe*, para realizar análisis de terminación y análisis del número de instrucciones ejecutadas por un programa. Ambas técnicas están implementadas por separado y como trabajo futuro se propone unir ambas implementaciones de modo que del resultado obtenido al inferir invariantes podamos obtener la entrada del algoritmo de Podelski y Rybalchenko. Por otra parte, para el algoritmo de inferencia de invariantes, queda

pendiente la implementación de la traducción de programas *Core-Safe* a programas de tamaño, y la obtención de los poliedros correspondientes a los casos bases y recursivos a través de estos.

Estos algoritmos se implementarán más adelante en el compilador de *Safe* una vez haya quedado comprobada su factibilidad. Una posible extensión de este trabajo es obtener funciones de rango lineales para funciones *Core-Safe* mutuamente recursivas. Otra posible mejora sería encontrar funciones de rango más generales que las lineales aunque para este caso no se conocen algoritmos completos.

Bibliografía

[AAECC' 05] Thiemann, R. and Giesl, J. 2005. *The size-change principle and dependency pairs for termination of term rewriting*. Applicable Algebra in Engineering, Communication and Computing 16, 4 (Sept.), 229–270.

[BK' 96] F. Benoy and A.M. King. *Inferring Argument Size Relationships with* $CLP(\mathcal{R})$. In 6th International Workshop on Logic Program Synthesis and Transformation. Springer-Verlag, 1996.

[CAV' 02] Colón, M., and Sipma, H. *Practical methods for proving program termination*. In *CAV* (2002).

[ESOP'08] Chawdhary, A., et al. *Ranking Abstractions*. In *ESOP* (2008).

[Hol' 95] Christian Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[ICLP' 05] M. Codish, V. Lagoon, and P. J. Stuckey. *Testing for termination with monotonicity constraints*. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP* 2005,volume 3668 of *LNCS*, pages 326–340. Springer, 2005.

[IFL' 10] Peña, R., Delgado, Agustín D., *Size Invariants and Ranking Functions Synthesis in a Functional Language*. In *IFL* (2010).

[LICS' 04] Andreas Podelski and Andrey Rybalchenko. *Transition Invariants*. In LICS, pages 32-41. *IEEE* Computer Society, 2004.

[LMCS' 09] Amir M. Ben-Amram and Chin Soon Lee. *Ranking functions for size-change termination II*. In *LMCS* (2009).

[POPL' 01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. *The size-change principle for program termination*. In *POPL*, pages 81-92, 2001.

[SAS' 10] Alias, C. et al. *Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs*. In *SAS* (2010).

[TACAS' 08] Amir M. Ben-Amram and Michael Codish. *A SAT-Based Approach to Size Change Termination with Global Ranking Functions*. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 218-232. Springer, 2008.

[Ben' 02] Benoy, P.M. *Polyhedral Domains for Abstract Interpretation in Logic Programming*. PhD. Thesis. University of Kent at Canterbury, 2002.

[TOPLAS' 05] Amir M. Ben-Amram and Chin Soon Lee. *Size-change analysis in polynomial time*. *ACM* Transactions on Programming Languages and Systems, 29(1), 2007.

[TOPLAS' 07] Ben-Amram, A.M.: *Size-change termination with difference constraints*. *ACM* Transactions on Programming Languages and Systems, 2007.

[TOPLAS' 09] Chin Soon Lee. *Ranking functions for size-change termination*. *ACM* Transactions on Programming Languages and Systems, 2009.

[TPLP' 05] Florence Benoy, Andy King, and Frédéric Mesnard. *Computing convex hulls with a linear solver*. *TPLP*, 5(1-2):259-271, 2005.

[VMCAI' 04] Andreas Podelski and Andrey Rybalchenko. *A Complete Method for the Synthesis of Linear Ranking Functions*. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of LNCS, pages 239-251. Springer, 2004.

Autorización

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Avances recientes en análisis estáticos de terminación", realizado durante el curso académico 2009-2010 bajo la dirección de Ricardo Peña Marí en el Departamento de Sistemas Informáticos y Computación (DSIC), y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fdo. Agustín Daniel Delgado Muñoz

Apéndice A

Código fuente de la aplicación

```
:- use_module(library(clpq)).    %Parte Construcción de Invariantes
:- use_module(library(simplex)). %Parte Podelski & Rybalchenko
:- use_module(library(clp_distinct)).
:- dynamic a1/1,    % Matriz A Podelski & Rybalchenko
        a2/1,    % Matriz A' Podelski & Rybalchenko
        l1/1,    % Vector lambda 1 Podelski & Rybalchenko
        l2/1,    % Vector lambda 2 Podelski & Rybalchenko
        b/1,     % Vector de coeficientes b Podelski & Rybalchenko
        x1/1,    % Incognitas x Podelski & Rybalchenko
        x2/1,    % Incognitas x' Podelski & Rybalchenko
        chbase/1, %Convex Hull de los casos base
        chrecursivo/1. %Convex Hull de los casos recursivos
set(X,Y) :- nb_setval(X,Y).
get(X,Y) :- nb_getval(X,Y).

:- set(a1,[]).
:- set(a2,[]).
:- set(l1,[]).
:- set(l2,[]).
:- set(b,[]).
:- set(x1,[]).
```

```
:- set(x2,[]).

:- set(chbase,([],[])).

:- set(chrecursivo,([],[])).



/* CÓDIGO CÁLCULO DE ENVOLVENTES CONVEXAS */


project(Xs, Cxs, ProjectCxs) :-
   call_residue_vars(copy_term(Xs-Cxs, CpyXs-CpyCxs), _),
   tell_cs(CpyCxs),
   prepare_dump(CpyXs, Xs, Zs, DumpCxs, ProjectCxs),
   dump(Zs, Vs, DumpCxs), Xs = Vs.


tell_cs([]).
tell_cs([C|Cs]) :-  {C}, tell_cs(Cs).



prepare_dump([], [], [], Cs, Cs).
prepare_dump([X|Xs], YsIn, ZsOut, CsIn, CsOut) :-
   (ground(X) ->
      YsIn  = [Y|Ys],
      ZsOut = [_|Zs],
      CsOut = [Y=X|Cs]
   ;
      YsIn  = [_|Ys],
      ZsOut = [X|Zs],
```

```
        CsOut = Cs
    ),
    prepare_dump(Xs, Ys, Zs, CsIn, Cs).

convex_hull(Xs, Cxs, Ys, Cys, Zs, Czs) :-
    scale(Cxs, Sig1, [], C1s),
    scale(Cys, Sig2, C1s, C2s),
    add_vect(Xs, Ys, Zs, C2s, C3s),
    project(Zs, [Sig1 >= 0, Sig2 >= 0, Sig1+Sig2 = 1|C3s], Czs).

scale([], _, Cs, Cs).
scale([C1|C1s], Sig, C2s, C3s) :-
    C1 =.. [RelOp, A1, B1],
    C2 =.. [RelOp, A2, B2],
    mul_exp(A1, Sig, A2),
    mul_exp(B1, Sig, B2),
    scale(C1s, Sig, [C2|C2s], C3s).
mul_exp(E1, Sigma, E2) :- once(mulexp(E1, Sigma, E2)).
mulexp(  X,   _,     X) :- var(X).
mulexp(N*X,   _,   N*X) :- ground(N), var(X).
mulexp( -X, Sig,    -Y) :- mulexp(X, Sig, Y).
mulexp(A+B, Sig,   C+D) :- mulexp(A, Sig, C), mulexp(B, Sig, D).
mulexp(A-B, Sig,   C-D) :- mulexp(A, Sig, C), mulexp(B, Sig, D).
mulexp(  N, Sig, N*Sig) :- ground(N).
```

```
add_vect([], [], [], Cs, Cs).

add_vect([U|Us], [V|Vs], [W|Ws], C1s, C2s) :-

    add_vect(Us, Vs, Ws, [W = U+V|C1s], C2s).


/*
**********************************************************************
*/


/* CÓDIGO CÁLCULO DE INVARIANTES*/


widening(C1,C2,CS1):- tell_cs(C2), selected(C1,CS1).

selected([],[]).

selected([C|Cs],[C|Is]):- entailed(C), selected(Cs,Is).

selected([_|Cs], Is):- selected(Cs,Is).


%Listas Casos Base y Recursivos de la forma [(Var1,Res1),(Var2,Res2),...(VarN,ResN)] donde

%VarI son listas de variables y ResI son restricciones sobre VarI con I=1,2,...,N


%i-ésimo par (VarI,ResI)


ith([X|_],1,X).

ith([_|L],I,X):-I1 is I-1, ith(L,I1,X).


%Obtiene VarI

var_ith([(V,_)|_],1,V).

var_ith([_|L],I,X):-I1 is I-1, var_ith(L,I1,X).
```

```
%Obtiene ResI

res_ith([(_,R)|_],1,R).
res_ith([_|L],I,X):-I1 is I-1, res_ith(L,I1,X).

%Obtiene lista de listas de variables VarI, I=1,2,...,N

lista_vars([],[]).
lista_vars([(V,_)|L],[V|R]):-lista_vars(L,R).

%Obtiene lista de listas de restricciones ResI, I=1,2,...,N

lista_restr([],[]).
lista_restr([(_,R)|L],[R|R1]):-lista_restr(L,R1).

%Aplanar una lista de listas

aplanar([],[]).
aplanar([X|L],S):-
        is_list(X),
        aplanar(X,S1),
        aplanar(L,S2),
        append(S1,S2,S),
        !.

aplanar([X|L],S):-
```

```
        not(is_list(X)),
        aplanar(L,S1),
        S=[X|S1].

%Obtiene lista de variables VarI, I=1,2,...,N

vars([],[]).
vars(L,V):-
        lista_vars(L,R),
        aplanar(R,V).

%Obtiene lista de restricciones ResI, I=1,2,...,N

restr([],[]).
restr(L,V):-
        lista_restr(L,R),
        aplanar(R,V).

%Convex Hull casos base

convex_hull_base([],_,[]).

convex_hull_base([(V,R)],V,R).

convex_hull_base([(V1,R1),(V2,R2)],Z,CH):-
            convex_hull(V1,R1,V2,R2,Z,CH),
```

```
                !.
convex_hull_base([(V1,R1),(V2,R2)|L],Z,CH):-
                    convex_hull(V1,R1,V2,R2,VCH1,CH1),
                    append([(VCH1,CH1)],L,L1),
                    convex_hull_base(L1,Z,CH).

extender_variables(V,0,V):-!.
extender_variables(V,N,NV):- append([_],V,V2),
                    N1 is N-1,
                    extender_variables(V2,N1,NV).

take([],_,[]).
take(_,0,[]):-!.
take([C|R],N,[C|L]):- N1 is N-1, take(R,N1,L).

take_ultimas([],_,[]).
take_ultimas(L,0,L).
take_ultimas([_|R],N,L):-N1 is N-1, take_ultimas(R,N1,L),!.

divide_lista(L,N,L1,L2):- take(L,N,L1), take_ultimas(L,N,L2).

operador((VP,RP),(VN,RN)):-
                get(chrecursivo,(VR,RR)),
                length(VP,N),
                divide_lista(VR,N,VRnoprimas,VRprimas),
                VP=VRnoprimas,
```

```
                append(RR,RP,IRP),
                project(VRprimas,IRP,IRPP),
                get(chbase,(VB,RB)),
                convex_hull(VRprimas,IRPP,VB,RB,VN,RN).

%Algoritmo de punto fijo

%base_fixpoint comprueba si las restricciones RP implican a todas las
%restricciones RR

parar_fixpoint([]).
parar_fixpoint([R|L]):- entailed(R),parar_fixpoint(L).
parar_fixpoint([_|_]):- fail.
base_fixpoint((_,RR),(_,RP)):-
            tell_cs(RP),
            parar_fixpoint(RR).

fixpoint_aux(0,RestricPrevias,Poliedro):-
            operador(RestricPrevias,R2),
            RestricPrevias=(VRRP,RRP),
            R2=(VR2,RR2),
            %Poliedro=(VRP,RP),
            VR2=VRRP,
            widening(RRP,RR2,RP),
            (base_fixpoint(R2,RestricPrevias)
              -> Poliedro=(VR2,RP),!
```

```
                ;
                fixpoint_aux(0,R2,Poliedro)
                ).

fixpoint_aux(K,RestricPrevias,Poliedro):-
        operador(RestricPrevias,RestricNuevas),
        K1 is K-1,
        fixpoint_aux(K1,RestricNuevas,Poliedro).

reset_fixpoint:- set(chbase,([],[])),
          set(chrecursivo,([],[])).

fixpoint(CBase,CRecursivos,K,Poliedro):-
        reset_fixpoint,
        convex_hull_base(CBase,VB,RB),
        set(chbase,(VB,RB)),
        convex_hull_base(CRecursivos,VR,RR),
        set(chrecursivo,(VR,RR)),
        fixpoint_aux(K,(VB,RB),Poliedro).

/* CÓDIGO PARTE PODELSKI & RYBALCHENKO: OBTENCIÓN DE FUNCIONES DE RANGO
*/

/*Matrices representadas como listas de listas (filas de la matriz)
[[A11,A12...,A1M],[A21,A22,...,A2M],..., [AN1,AN2,...,ANM]]  */
```

```
ponA1(A1):-set(a1,A1).

ponA2(A2):-set(a2,A2).

ponB(B):-set(b,B).

%Calcula el numero de filas de una matriz

num_filas(A,N):-length(A,N).

%Calcula el numero de columnas de una matriz

num_columnas([],0).

num_columnas([R|_],M):-length(R,M).


%Dadas las matrices A y A', de igual dimensión, genera su yuxtaposicion (AA')

yuxtaponer_matrices([],[],[]).

yuxtaponer_matrices([F1|M1],[F2|M2],[F|M]):-

                append(F1,F2,F),

                yuxtaponer_matrices(M1,M2,M).


%Suma de dos listas

suma_filas([],[],[]).

suma_filas([E1|F1],[E2|F2],[E|F]):-

            E is E1+E2,

            suma_filas(F1,F2,F).

%Suma de dos matrices

sumar_matrices([],[],[]).

sumar_matrices([F1|M1],[F2|M2],[F|M]):-

            suma_filas(F1,F2,F),

            sumar_matrices(M1,M2,M).
```

```
%Producto escalar de dos vectores
prod_escalar([],[],0).
prod_escalar([V1|L1],[V2|L2],V):-
        Vaux = V1*V2,
        prod_escalar(L1,L2,Vaux2),
        V = Vaux + Vaux2.


%Producto escalar de dos vectores
prod_escalar2([],[],[]).
prod_escalar2([V1|L1],[V2|L2],[Vaux|L]):-
        Vaux = V1*V2,
        prod_escalar2(L1,L2,L).
       % V = Vaux + Vaux2.

%Devuelve la i-ésima columna de una matriz
ith_columna([],_,[]).
ith_columna([F|M],I,[C|RC]):-
        ith(F,I,C),
        ith_columna(M,I,RC),!.

%Multiplica un vector por una matriz, ambos de dimensiones adecuadas
vector_por_matriz(_,[],[]):-!.
vector_por_matriz(V,M,R):- cambiar_filas_por_columnas(M,M2),
            vector_por_matriz_aux(V,M2,R).
```

```
vector_por_matriz_aux(_,[],[]):-!.
vector_por_matriz_aux(V,[F|M],[C|R]):-
                    prod_escalar2(F,V,C),
                    vector_por_matriz_aux(V,M,R).
                   % append(C,R,R2).


%Cambia filas por columnas de una matriz
cambiar_filas_por_columnas([],[]).
cambiar_filas_por_columnas(M,R2):-
                     num_columnas(M,N),
                     cambiar_filas_por_columnas_aux(M,1,N,R),
                     take(R,N,R2),!.

cambiar_filas_por_columnas_aux(M,K,N,[C|R]):-
                      K=<N,
                      ith_columna(M,K,C),
                      K1 is K+1,
                      cambiar_filas_por_columnas_aux(M,K1,N,R),!.
cambiar_filas_por_columnas_aux(_,_,_,_).
```

```
%Calcula el vector resta de dos vectores
restar_vectores([],[],[]).
restar_vectores([E1|V1],[E2|V2],[E|V]):-
        E = E1-E2,
        restar_vectores(V1,V2,V).

%Cambia de signo los elementos de un vector
cambia_signo_vector([],[]).
cambia_signo_vector([E|V],[NE|NV]):- NE is 0-E,
                cambia_signo_vector(V,NV).

%Cambia de signo los elementos de una matriz
cambia_signo_matriz([],[]).
cambia_signo_matriz([F|M],[NF|NM]):- cambia_signo_vector(F,NF),
                cambia_signo_matriz(M,NM).

%Pone las restricciones de que los valores lambda sean no negativos
restricciones_lambda(S0,S):-
        get(l1,L1),
        get(l2,L2),
        lambdas_positivos(L1,L2,S0,S).
```

```
%Carga una lista de restricciones al sistema
cargar_restricciones([],S,S).
cargar_restricciones([R|L],S0,S):-
        constraint(R = 0,S0,S1),
        cargar_restricciones(L,S1,S).

lambdas_positivos([],[],S,S).
lambdas_positivos([E1|V1],[E2|V2],S0,S):-
        constraint([E1]>=0,S0,S1),
        constraint([E2]>=0,S1,S2),
        lambdas_positivos(V1,V2,S2,S).

%Da el formato exigido a una de las restricciones del sistema
mezcla_listas([],[],[]).
mezcla_listas([L1|R1],[L2|R2],[L|R]):-
            append(L1,L2,L),
            mezcla_listas(R1,R2,R).

%Genera las restricciones del sistema propuesto por Podelski y Rybalchenko
%y lo resuelve
generar_restricciones(S):-
    gen_state(S0),
    get(l1,L1),
    get(l2,L2),
    get(a1,A1),
    get(a2,A2),
```

```
get(b,B),
restricciones_lambda(S0,S1),
 %L1*A2=0
vector_por_matriz(L1,A2,C1),
cargar_restricciones(C1,S1,S2),
%(L1-L2)*A1=0
cambia_signo_matriz(A1,NA1),
vector_por_matriz(L1,A1,Laux1),
vector_por_matriz(L2,NA1,Laux2),
mezcla_listas(Laux1,Laux2,C2),
cargar_restricciones(C2,S2,S3),

%L2*(A1+A2)=0
sumar_matrices(A1,A2,Asuma),
vector_por_matriz(L2,Asuma,C3),
cargar_restricciones(C3,S3,S4),

%L2*B<0 ---> L2 * B <= -1
%No deja poner restricciones con lados derechos negativos
%--> -L2*B >= 1 ---> L2*(-B) >=  1
cambia_signo_vector(B,NB),
prod_escalar2(NB,L2,C4),
%Delta is C4,
constraint(C4 = 1, S4,S5),
prod_escalar2(B,L1,Delta0), % -Delta0
append(L1,L2,SumaLambdas),
```

```
        mete_dieces(SumaLambdas,SumaLambdasAux),
        ajustarFO(SumaLambdasAux,Delta0,FO),
        minimize(FO,S5,S).


%Da el formato exigido a la función objetivo
mete_dieces([],[]).
mete_dieces([E|R],[E1|R1]):- E1=10*E,
                    mete_dieces(R,R1).

%Da el formato exigido a la función objetivo
ajustarFO(L,[],L).
ajustarFO([C1*L1|R1],[C2*L1|R2],[C*L1|R]):-
                    C is C1+C2,
                    ajustarFO(R1,R2,R),!.

%Inicia las variables globales del programa
reset:- set(a1,[]),
      set(a2,[]),
      set(b,[]),
      set(l1,[]),
      set(l2,[]).
```

```
%Generación de vectores lambda L1 y L2
 vectores_lambda(0,[],[]).
 vectores_lambda(N,[E1|V1],[E2|V2]):- E1=l1(N),
                                      E2=l2(N),
                                      N1 is N-1,
                                      vectores_lambda(N1,V1,V2),!.
%Calcula la lista inversa
inversa([],[]).
inversa([E|L],R):-
                  inversa(L,LI),
                  append(LI,[E],R),!.

%Genera los vectores lambda del sistema
genera_lambdas:-
                   get(a1,A1),
                   num_filas(A1,M),
                   vectores_lambda(M,V1,V2),
                   inversa(V1,IV1),
                   inversa(V2,IV2),
                   set(l1,IV1),
                   set(l2,IV2).
```

```
%Obtenención de los valores de los vectores lambda una vez obtenidos
valores_lambda(S,Val1,Val2):-
        get(l1,L1),
        get(l2,L2),
        lista_valores_lambda(S,L1,L2,Val1,Val2).


lista_valores_lambda(_,[],[],[],[]).
lista_valores_lambda(S,[E1|L1],[E2|L2],[V1|Val1],[V2|Val2]):-
         variable_value(S,E1,V1),
         variable_value(S,E2,V2),
         lista_valores_lambda(S,L1,L2,Val1,Val2).

%Dadas A1, A2 y B resuelve el problema de Programación Lineal propuesto
%por Podelski y Rybalchenko
resolver_sistema(A1,A2,B,S,Val1,Val2):-
    reset,
    set(a1,A1),
    set(a2,A2),
    set(b,B),
    genera_lambdas,
    generar_restricciones(S),
    valores_lambda(S,Val1,Val2).
```

Apéndice B

Artículo presentado en [IFL' 10]

Este artículo titulado *Size Invariants and Ranking Functions Synthesis in a Functional Language*, presentado en el congreso IFL 2010 (*Implementation and Application of Functional Languages*) contiene la version completa de las técnicas explicadas en el Capítulo 5. Ha surgido como aplicación práctica de este proyecto fin de Máster.

Size Invariants and Ranking Functions Synthesis in a Functional Language *

Ricardo Peña
ricardo@sip.ucm.es

Agustin D. Delgado
elsmda@gmail.com

Universidad Complutense de Madrid, Spain

Abstract. The paper presents preliminary results in automatic inference of size invariants, and of ranking functions proving termination of functional programs, by adapting linear techniques developed for other languages. The results are promising and allow to solve some problems left open in previous works on automatic inference of safe memory bounds.
Keywords: functional languages, linear techniques, abstract interpretation, size analysis, ranking functions.

1 Introduction

In a previous work [17] we presented static analysis-based algorithms for inferring upper bounds to memory consumption of programs written in a first-order functional language. The technique used was abstract interpretation with the abstract domain being the complete lattice of monotonic functions $f : \mathbb{R}^{+^n} \to \mathbb{R}^+$ ordered by point-wise $\leq$. We showed some examples of applications and the bounds obtained were rather good in simple programs. For instance, we got precise linear heap and stack bounds for functions such as *merge*, *append*, and *split*, and quadratic over-approximations for the heap consumption of functions such as *mergesort* and *quicksort*. The algorithms were even able to infer a constant stack space for tail recursive functions.

A remarkable feature of the algorithms was that in some circumstances the abstract interpretation was *reductive* in the lattice, meaning that iterating the interpretation by introducing as hypothesis the previously inferred bound, obtained a tighter bound, and still a correct one.

Unfortunately, the work was incomplete because it needed some information to be introduced by hand for every particular program, namely the size of some local variables and an upper bound to the length of the longest call chain of recursive functions. These two problems deserve independent and complex analyses by themselves, and we decided to defer their solution. The algorithms were proved correct provided correct functions for this figures were given.

In [14] we approached one of these problems —inferring the longest recursive call chain— by translating our functional programs into a term rewriting

* Work partially funded by the projects TIN2008-06622-C03-01/TIN (STAMP), and S2009/TIC-1465 (PROMETIDOS).

system (TRS) and then using termination proofs of TRSs, based on dependency pairs and polynomial synthesis, for computing this bound. The first results were encouraging but the approach could not prove any bound for simple algorithms such as *mergesort* and *quicksort*. We felt that having a previous size analysis could probably improve the termination proofs.

In this paper we approach both size analysis and termination proofs by using linear techniques, which have been proved successful in analysing other kind of languages such as imperative, and even bytecode programs, and also logic ones. The main contribution of the paper is showing that these techniques have the same power in a first-order functional language than in any other paradigm, and that the only adaptations needed are applying some transformations to the source programs, and having some care when applying the techniques in a context different to that they were conceived in.

The plan of the paper is as follows: In Sec. 2 we do a brief survey of linear techniques applied to both size analysis and ranking function synthesis. Then, Sec. 3 presents the aspects of our functional language *Safe* relevant to this paper. Sec. 4 is devoted to obtaining size invariants of *Safe* programs, while Sec. 5 applies the well-known ranking function synthesis method by Podelski and Rybalchenko [18] to computing our bound to the longest call chain. Finally, Sec. 6 provides a short conclusion.

2 Linear Constraints Techniques

Abstract interpretation [10] is a very powerful static analysis technique able to infer a variety of properties in programs written in virtually any programming language. In functional languages it has been successfully applied to strictness and update avoidance analyses of lazy languages, to sharing and binding time analyses, and many others. The abstract domains are usually finite small ones when abstracting non-functional values, but they tend to grow exponentially when they are extended to higher-order values.

Polyhedral abstract domains have been extensively used in logic and imperative languages, but not so frequently in functional ones. These domains are useful when quantitative rather than qualitative properties are sought for, as it is the case of size or cost relations between variables. Since the seminal work of Cousot and Halbwachs [11], polyhedra have been used to analyse arithmetical linear relations between program variables. A convex polyhedron is a subset of $\mathbb{R}^n$ limited by hyperplanes. There exist at least two finite representations of convex polyhedra:

- By three sets, namely of vertexes, rays, and lines in a n-dimensional space.
- By a conjunction of linear constraints between n variables.

There are algorithms for translating these representations into each other, although their cost is exponential. A frequent (and also costly) operation is to compute the convex hull of two polyhedra, which is the minimum convex polyhedron containing both. It is associative and commutative. The advantage of

using linear constraints is that most of the interesting problems involving them are decidable. For instance, to know whether a set of constraints is satisfiable, or whether a constraint is implied by a set of other ones, to project a set of constraints over a subset of their variables, to compute the convex hull of two sets of constraints, to maximise or minimise a linear function with respect to a set of constraints, and some others.

Invariant synthesis In this context, an invariant is a linear relation between the variables involved in a loop, holding at the beginning of each iteration. An abstract interpretation for synthesising loop invariants starts by computing a polyhedron with the relations known at the beginning of the loop, and iterates calculating the convex hull between the polyhedron coming from the previous iteration and the one obtained by the transition relation of the loop body. After a few iterations, some relations will stabilise while some others will not. The first ones constitute the invariant. Several tools have been developed for obtaining these invariants (for instance ASPIC, see [12]), or giving the necessary infrastructure to compute them (as e.g. [4]).

In the logic programming field, Benoy and King [7] applied a similar technique to the inference of size relations between the arguments of a logic predicate. In a first step, the logic program is transformed into an abstract one on arithmetic variables, by replacing the original predicate arguments by their corresponding sizes. An abstract interpretation of the transformed program infers the invariant size relations between the arguments of the original program. The ascending chain (in the sense of set inclusion) of polyhedra obtained by the fixpoint algorithm may in principle be infinite, because some relations do not stabilise. A *widening* technique is used to eliminate these variant relations while the invariant ones are retained. Of course, if the invariant relations are not linear the algorithm does not obtain anything meaningful.

Ranking functions synthesis Detecting termination statically has attracted the attention of much research work. Given that this is an undecidable problem in general, the algorithms try to cover as many particular decidable cases as possible. One successful approach has been the work by Ben-Amram and his group, starting in the seminal paper [13], where in a first phase the program being analysed is transformed into a so-called *size-change graph*. This is the program call flow graph enriched with information about the arguments that strictly decrease at a call and those that may decrease or remain equal. This part of the analysis is outside of the proposed termination algorithms, and may be done by hand or by a previous size analysis. What is nice in the approach is that termination of size-change graphs is decidable, although the algorithm is exponential in the worst case (these programs are called *size-change terminating*, or SCT). However, by using benchmarks the authors convincingly show that this case is very unusual and that most of the time a polynomial algorithm suffices [6]. Moreover, in these cases they can synthesise a global ranking function ensuring that the program terminates, which can be checked (i.e. proved that it decreases

at each transition) in polynomial time. Synthesising such a function is however a NP-complete problem which they decide by using a SAT-solver [5].

Another successful line of research has been the synthesis of linear ranking functions. In [18], Podelski and Rybalchenko give a complete method to synthesise this kind of functions for simple while-loops in which the loop guard is a conjunction on linear expressions, and the loop body is a multiple assignment of linear expressions to variables. The kernel of the method is solving a set of linear constraints. This small piece can be the basis for inferring termination of more complex programs. In [19] they show that the union of simple well-founded relations (which in general is not well-founded) can prove termination of a program with nested loops provided this union is an invariant of the transition relation (in this case, the union is called *disjunctively well-founded*). Another successful method for analysing complex loops and synthesising linear ranking functions is [9]. In a recent work [2] the authors present a complete method for synthesising a global ranking function for a complex nested control flow structure, provided the ranking function has the form of a lexicographically ordered tuple containing linear expressions.

In [1], the authors claim to have used —although not many details are given— the Podelski and Rybalchenko's method as one of the steps for solving recurrence relations obtained by analysing Java bytecode programs. The idea is to use the ranking function as an upper bound to the recurrence depth (i.e. to the number of unfoldings required in the worst case to reach a non-recursive case). We will pursue this idea here for inferring an upper bound to the depth of the call tree of a recursive *Safe* function.

3 The *Safe* Functional Language

Safe is a first-order eager polymorphic functional language with a syntax similar to that of (first-order) Haskell, and with an unusual memory management system. Its main aim is to facilitate the compile-time inference of safe upper bounds to memory consumption. Its memory model is based on disjoint heap regions where data structures are built. The compiler infers the optimal way to assign data to regions, so that their lifetimes are as short as possible, compatible with allocating and deallocating regions by following a stack-based strategy. The region-based model has two benefits: (1) a garbage collector is not needed; (2) the compiler may compute an upper bound to the size of each region and to the whole heap. More information about *Safe*, its type system, and its memory inference algorithms can be found at [15–17].

The *Safe* front-end desugars *Full-Safe* and produces a bare-bones functional language called *Core-Safe*. The transformation starts with region inference and follows with Hindley-Milner type inference, desugaring pattern matching into **case** expressions, **where** clauses into **let** expressions, collapsing several function-defining equations into a single one, and some other transformations.

As regions are not relevant to this paper, in Fig. 1 we show a simplified *Core-Safe*'s syntax where regions have been deleted. A program *prog* is a sequence of

$$
\begin{array}{lll}
prog \rightarrow \overline{data_i}; \overline{dec_j}; e & & \{\textit{Core-Safe} \text{ program}\} \\
dec \rightarrow f\ \overline{x_i}\ = e & & \{\text{recursive, polymorphic function definition}\} \\
e \rightarrow a & & \{\text{atom } a\text{: either a literal } c \text{ or a variable } x\} \\
\quad \mid a_1 \oplus a_2 & & \{\text{primitive operator application}\} \\
\quad \mid f\ \overline{a_i} & & \{\text{function application}\} \\
\quad \mid C\ \overline{a_i} & & \{\text{constructor application}\} \\
\quad \mid \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 & & \{\text{non-recursive, monomorphic } \textbf{let}\} \\
\quad \mid \textbf{case}\ x\ \textbf{of}\ \overline{alt_i} & & \{\text{case expression}\} \\
alt \rightarrow C\ \overline{x_i} \rightarrow e & & \{\text{case alternative}\}
\end{array}
$$

Fig. 1. Simplified *Core-Safe* syntax

```
split 0 ys      = ([], ys)
split n []      = ([], [])
split n (y:ys)  = (y:ys1,ys2)    where (ys1, ys2) = split (n-1) ys

merge [] ys  = ys
merge xs []  = xs
merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys

msort []  = []
msort [x] = [x]
msort xs  = merge (msort xs1) (msort xs2)
            where (xs1,xs2) = split (length n / 2) xs
```

Fig. 2. *mergesort* algorithm in *Full-Safe*

possibly recursive polymorphic data and function definitions followed by a main expression e whose value is the program result. The abbreviation $\overline{x_i}$ stands for $x_1 \cdots x_n$, for some n.

In Fig. 2 we show a *Full-Safe* version of the *mergesort* algorithm, which we will use as running example throughout the paper. In Fig. 3 we show the translation to *Core-Safe* of two of its functions.

Our purpose is to analyse *Core-Safe* programs in order to infer invariant size relations between the arguments and results of each function, and upper bounds to the runtime size of the call tree unfolded when invoking a recursive function. As part of this process, it will be important to discover which sub-expressions contribute to the non-recursive (or base) cases of a recursive function, and which ones contribute to the recursive ones. Moreover, we will distinguish between mutually exclusive recursive calls, i.e. those which will never execute together at runtime (e.g. those occurring in different alternatives of a **case** expression), and sequential recursive calls (those with may execute one after the other).

An approximation to this property can be obtained by an algorithm separating textual calls occurring in different branches of a **case**, and putting together textual calls occurring in the sub-expressions e_1 and e_2 of a **let**. In Fig. 4 we show the algorithm $seqs_f$, written in a Haskell-like language, analysing this property for the *Core-Safe* expression of function-f's body. It returns a list of lists of qualified expressions, meaning this that a tag B (for base cases), or R (for recursive

```
merge x y = case x of
             []    -> y
             ex:x' -> case y of
                        []    -> x
                        ey:y' -> case ex <= ey of
                                   True  -> let z1 = merge x' y in ex:z1
                                   False -> let z2 = merge x y' in ey:z2
msort x = case x of
           []    -> []
           ex:x' -> case x' of
                     []  -> ex:[]
                     _:_ -> let n  = length x        in
                            let n2 = n/2             in
                            let (x1,x2) = split n2 x in
                            let z1 = msort x1        in
                            let z2 = msort x2        in
                            merge z1 z2
```

Fig. 3. functions *merge* and *msort* in *Core-Safe*

$$\textbf{data}\ \mathit{QExp} = B\ \mathit{Exp} \mid R\ \mathit{Exp} \mid \mathit{Var} := [\mathit{QExp}]$$

$$\begin{aligned}
&\mathit{seqs}_f :: \mathit{Exp} \rightarrow [[\mathit{QExp}]] \\
&\mathit{seqs}_f\ e = [[B\ e]] && \text{-- if } e \in \{c, x, a_1 \oplus a_2, g\ \overline{a_i}, C\ \overline{a_i}\} \\
&\mathit{seqs}_f\ (f\ \overline{a_i}) = [[R\ (f\ \overline{a_i})]] \\
&\mathit{seqs}_f\ (\textbf{case of}\ \mathit{alts}) = \mathit{concat}\ [\mathit{seqs}_f\ e \mid (C\ \overline{x_i} \rightarrow e) \in \mathit{alts}] \\
&\mathit{seqs}_f\ (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2) = [(x_1 :=\ s_1) : s_2 \mid s_1 \in \mathit{seqs}_f\ e_1,\ s_2 \in \mathit{seqs}_f\ e_2]
\end{aligned}$$

Fig. 4. Algorithm for extracting the base and recursive cases of a *Core-Safe* expression

ones) is added to each individual sub-expression. Each internal list represents a mutually exclusive way of executing the function, while the sub-expressions inside a list indicate a possible sequential execution of them.

In Fig. 5 we show the application of the algorithm to *merge* and *msort*. For *merge* x y we obtain four internal lists illustrating that there are two mutually exclusive base cases, and that the two recursive calls exclude each other. When applied to $msort(x)$ we obtain three internal lists illustrating that there are two base cases, and that the two recursive calls may be sequentially executed.

4 Size Invariants Inference

Following [7], in order to reason about sizes, the original program must first be translated into an abstract program in which data structures have been replaced by their corresponding sizes, and previously to that, a notion of *size* must be defined. For logic programs, the more frequently used ones are the list length for list arguments, the value for integer arguments, zero for any other atom, and the term size for the rest of variables. In our memory model we have a precise notion of size in terms of memory cells occupied by a data structure: each constructor application fits exactly in one cell. So, the size of a list is its length plus one because the empty list constructor needs an additional cell. Moreover, we have a

$$
\begin{aligned}
seqs_{merge} = {} & [[B\ y], [B\ x], \\
& [z_1 := [R\ (merge\ x'\ y)], B\ (e_x : z_1)], [z_2 := [R\ (merge\ x\ y'))], B\ (e_y : z_2)]]
\end{aligned}
$$

$$
\begin{aligned}
seqs_{msort} = {} & [[B\ [\,]], [B\ (e_x : [\,])], \\
& [n := [B\ (length\ x)], n_2 := [B\ (n/2)], (x_1, x_2) := [B\ (split\ n_2\ x)], \\
& z_1 := [R\ (msort\ x_1)], z_2 := [R\ (msort\ x_2)], B\ (merge\ z_1\ z_2)]]
\end{aligned}
$$

Fig. 5. Decomposition of *merge* and *msort* into base and recursive cases

$$
\begin{array}{l}
merge^S\ x\ y = \textbf{case}\ x\ \textbf{of} \\
\quad x = 1 \rightarrow\ y \\
\quad x \geq 2 \rightarrow\ \textbf{case}\ y\ \textbf{of} \\
\qquad y = 1 \rightarrow x \\
\qquad y \geq 2 \rightarrow \textbf{case}\ ?\ \textbf{of} \\
\qquad\quad T \rightarrow \textbf{let}\ z_1 = merge^S\ (x-1)\ y\ \textbf{in}\ z_1 + 1 \\
\qquad\quad F \rightarrow \textbf{let}\ z_2 = merge^S\ x\ (y-1)\ \textbf{in}\ z_2 + 1 \\
msort^S\ x = \quad \textbf{case}\ x\ \textbf{of} \\
\quad x = 1 \rightarrow\ 1 \\
\quad x \geq 2 \rightarrow\ \textbf{case}\ x\ \textbf{of} \\
\qquad x = 2 \rightarrow 2 \\
\qquad x \geq 3 \rightarrow\ \textbf{let}\ n = length^S\ x\ \textbf{in} \\
\qquad\quad \textbf{let}\ n_2 = n/2\ \textbf{in} \\
\qquad\quad \textbf{let}\ (x_1, x_2) = split^S\ n_2\ x\ \textbf{in} \\
\qquad\quad \textbf{let}\ z_1 = msort^S\ x_1\ \textbf{in} \\
\qquad\quad \textbf{let}\ z_2 = msort^S\ x_2\ \textbf{in} \\
\qquad\quad merge^S\ z_1\ z_2
\end{array}
$$

Fig. 6. Abstract size functions *merge* and *msort*

precise notion of *data structure*: it comprises the set of cells corresponding to its recursive spine. In this way, a list of n lists constitutes $n + 1$ independent data structures. Additionally we define:

- The size of an integer constant or variable is its value n.
- The size of a Boolean constant or variable is zero.

We will call *size programs*, or size functions, to the abstract programs or functions resulting from the size translation. If f is the original function, f^S will denote its size version. The size functions resulting from the translation of *merge* and *msort* of Fig. 3 are shown in Fig 6.

The next step consists of performing an abstract interpretation of the size functions. The abstract domain will be that of convex polyhedra ordered by set inclusion, represented in this paper by conjunctions of linear constraints. The *meet* operation, or greatest lower bound $\sqcap$, is the intersection of two polyhedra, and consists of just putting together the two sets of constraints. The *join* operation, or least upper bound $\sqcup$, is the convex hull of the two polyhedra. We have used the algorithm developed by Andy King et al [8] to compute convex hulls.

The algorithm we propose has been inspired by Benoy and King's algorithm [7] for analysing logic programs. It consists of the following steps:

1. The size functions obtained by translating a *Core-Safe* program are analysed in the order they occur in the file: from low-level ones not using any other function, to higher-level ones using those previously defined in the file.
2. For each size function, a set of invariant size relations between input arguments and output results are inferred.
3. These relations are kept in a global environment. When analysing the current function, say f^S, these relations are instantiated with the sizes of the actual arguments at each site where an already analysed function is called. These relations, together with the rest of relations inferred for f^S, are used to infer f^S's invariant relations.

Analysing the current function f^S consists of the following steps. In order to fix ideas, we will call $\overline{x_i}$ to f^S's formal arguments and z to its result (or $\overline{z_j}$ if it is a tuple).

1. Function $seqs_f$ of Fig. 4 is applied to the size function f^S in a similar way as it was applied to *Core-Safe* expressions.
2. As a result, a separate set of constraints for each of the code sequences is inferred. The base sequences are then separated from the recursive ones (recursive sequences can be identified by the presence of at least a recursive call to f^S).
3. The constraints of each base sequence are expressed in terms of $\overline{x_i}$ and $\overline{z_j}$. This can always be done by projecting a set of constraints with more variables to variables $\overline{x_i}$ and $\overline{z_j}$.
4. The constraints of each recursive sequence are expressed in terms of $\overline{x_i}$, $\overline{z_j}$, and of two sets of variables $\overline{x_i}^k$ and $\overline{z_j}^k$ for each recursive call k in the sequence. The variables $\overline{x_i}^k$ represent the input arguments sizes of that call, while the $\overline{z_j}^k$ represent its output sizes.

Then, a fixpoint algorithm for f^S is launched having the following steps:

1. The initial polyhedron is $P_{next} = B_1 \sqcup \ldots \sqcup B_n$ where $B_l(\overline{x_i}, \overline{z_j})$, $1 \leq l \leq n$, are the polyhedra of the base cases.
2. At each iteration, the variables of polyhedron $P_{next}(\overline{x_i}, \overline{z_j})$ are renamed, obtaining $Q_{prev} = P_{next}[\overline{x'_i}/\overline{x_i}, \overline{z'_j}/\overline{z_j}]$. The idea is that $Q_{prev}(\overline{x'_i}, \overline{z'_j})$ represents the constraints coming from the previous iterations.
3. Now, for each recursive sequence l, its constraints are enriched by adding the constraints coming from Q_{prev}, as many times n_l as recursive calls are in the sequence, by previously substituting the $\overline{x'_i}$ for the $\overline{x_i}^k$ and the $\overline{z'_j}$ for the $\overline{z_j}^k$, $1 \leq k \leq n_l$. Let us call $R_l(\overline{x_i}, \overline{z_j}, \overline{x_i}^1, \overline{z_j}^1, \ldots, \overline{x_i}^{n_l}, \overline{z_j}^{n_l})$ to the polyhedron resulting from the l-th sequence.
4. Each R_l is projected over the variables $\overline{x_i}, \overline{z_j}$ obtaining $RP_l(\overline{x_i}, \overline{z_j})$. If there are m recursive sequences, then the following polyhedron is computed:

$$P_{next}(\overline{x_i}, \overline{z_j}) = RP_1 \sqcup \ldots \sqcup RP_m \sqcup B_1 \sqcup \ldots \sqcup B_n$$

5. If $P_{next}[\overline{x'_i}/\overline{x_i}, \overline{z'_j}/\overline{z_j}] = Q_{prev}$ then stop; else go to (2).

$$
\begin{array}{l}
B_1^{merge} = \{x = 1,\ z = y\} \\
B_2^{merge} = \{x \geq 2,\ y = 1,\ z = x\} \\
R_1^{merge} = \{x \geq 2,\ y \geq 2,\ x' = x - 1,\ y' = y,\ z = 1 + z'\} \\
R_2^{merge} = \{x \geq 2,\ y \geq 2,\ x' = x,\ y' = y - 1,\ z = 1 + z'\}
\end{array}
$$

$$
\begin{array}{l}
B_1^{msort} = \{x = 1,\ z = 1\} \\
B_2^{msort} = \{x = 2,\ z = 2\} \\
R_1^{msort} = \{x \geq 3,\ x + 1 = x'_1 + x'_2,\ z + 1 = z'_1 + z'_2\}
\end{array}
$$

Fig. 7. Restrictions corresponding to the base and recursive cases of *merge* and *msort*

Iter.	$\mathbf{P}_{next}$	$\mathbf{R}_i$
1_{merge}	$\{x \geq 1,\ z + 1 = x + y\}$	$R_1 = \{x \geq 2,\ y \geq 2,\ x' = x - 1,\ y' = y,$ $z = 1 + z',\ x' \geq 1,\ z' + 1 = x' + y'\}$
		$R_2 = \{x \geq 2,\ y \geq 2,\ x' = x,\ y' = y - 1,$ $z = 1 + z',\ x' \geq 1,\ z' + 1 = x' + y'\}$
2_{merge}	$\{x \geq 1,\ z + 1 = x + y\}$	
1_{msort}	$\{x \geq 1,\ x \leq 2,\ z = x\}$	$R_1 = \{x \geq 3,\ x + 1 = x'_1 + x'_2,\ z + 1 =$ $z'_1 + z'_2,\ x'_1 \geq 1,\ x'_1 \leq 2,\ z'_1 = x'_1,$ $x'_2 \geq 1,\ x'_2 \leq 2,\ z'_2 = x'_2\}$
2_{msort}	$\{x \geq 1,\ x \leq 3,\ z = x\}$	$R_1 = \{x \geq 3,\ x + 1 = x'_1 + x'_2,\ z + 1 =$ $z'_1 + z'_2,\ x'_1 \geq 1,\ x'_1 \leq 3,\ z'_1 = x'_1,$ $x'_2 \geq 1,\ x'_2 \leq 3,\ z'_2 = x'_2\}$
3_{msort}	$\{x \geq 1,\ x \leq 4,\ z = x\}$	

Fig. 8. Fixpoint iterations for *merge* an *msort*

In Fig. 7 we show the base and recursive sets of constraints inferred for *merge* and *msort* before launching the fixpoint algorithm. In the *merge* case, the constraints are easily obtained from the sequences resulting from $seqs_{merge}\ (merge^S)$. In the *msort* case, the following relations obtained from the *length, split* and *merge* invariants, are additionally needed:

$n + 1 = x$	$n := length\ x$
$n_2 = 0.5x$	$n_2 := n/2$
$x + 1 = x_1 + x_2,\ x \geq x_2,\ x \leq n_2 + x_2$	$(x_1, x_2) := split\ n_2\ x$
$z + 1 = z_1 + z_2$	$z := merge\ z_1\ z_2$

In Fig. 8 we show the polyhedra obtained for *merge* and *msort* after a few iterations of the fixpoint algorithm. For *merge*, the fixpoint is reached after the first iteration. For *msort*, the ascendant chain is infinite because of the restrictions $x \leq 2,\ x \leq 3,\ x \leq 4, \ldots$ The widening technique used in [7], and original from [11], eliminates this just by keeping as P_{next} the restrictions of iteration i implied by the ones of iteration $i + 1$. This leads to $\{x \geq 1,\ z = x\}$ as the size invariant of *msort*.

5 Ranking Functions Synthesis

In [17] we presented several inference algorithms to statically obtain upper bounds to the memory consumption of a *Core-Safe* program. One of them was for inferring resident heap memory, a second one for *peak* heap memory, and a last one for peak stack memory. There, we used several functions assumed correct for the following data:

- $nr_f(\overline{x})$ and $nb_f(\overline{x})$ respectively gave upper bounds to the number of non-base and base calls of the runtime call tree unfolded by function f when it is called with arguments sizes $\overline{x}$. A non-base call is one recursively calling f again, and a base call is one ending a chain of recursive calls to f.
- $len_f(\overline{x})$ gave an upper bound to the length of the longest chain of recursive calls to f.
- $|\,y\,|_f\ (\overline{x})$ gave an upper bound to the size of variable y (assumed to belong to f's body) as a function of fs' argument sizes $\overline{x}$.

The algorithms were proved correct assuming that we have correct functions for the above figures, but we left open how to infer them. A first step for inferring size functions $|\,y\,|_f$ has been given with the inference of size invariants presented in Sec. 4. By following a similar idea to that of [1], nr_f and nb_f can be approximated should we have a correct function for $len_f(\overline{x})$. In effect, having $len_f(\overline{x})$ and a bound n_f to the maximum number of calls issued from an invocation to f, we can compute the above functions as follows:

$$nb_f(\overline{x}) = \begin{cases} 1 & \text{if } n_f = 1 \\ n_f^{len_f(\overline{x})-1} & \text{if } n_f > 1 \end{cases} \qquad nr_f(\overline{x}) = \begin{cases} len_f(\overline{x}) - 1 & \text{if } n_f = 1 \\ \frac{n_f^{len_f(\overline{x})-1}-1}{n_f-1} & \text{if } n_f > 1 \end{cases}$$

This figures correspond to the internal and leaf nodes of a complete tree of branching factor n_f and height $len_f(\overline{x})$. The branching factor n_f is a static quantity easily computed by taking the maximum number of consecutive calls in the sequences returned by function $seqs_f$ of Fig. 4. For instance, $n_{merge} = 1$ and $n_{msort} = 2$.

So, it suffices to approximate the function $len_f(\overline{x})$. To this aim we will use Podelski and Rybalchenko's method [18]. It is complete for linear ranking functions of loops of the form **while** B **do** S, where $B(\overline{x})$ is a conjunction of linear constraints over the variables $\overline{x}$ involved in the loop, and $S(\overline{x}, \overline{x}')$ is a transition relation expressed as a conjunction of linear constraints over the variable values respectively before and after executing the loop body. Using these constraints over $\overline{x}$ and $\overline{x}'$, the method creates another set of constraints over $\overline{\lambda_1}$ and $\overline{\lambda_2}$, two lists of m non-negative variables, being m the number of restrictions contained in the conjunction of $B(\overline{x})$ and $S(\overline{x}, \overline{x}')$. This set is satisfiable if and only if a linear ranking function exists for the **while**, and it can be synthesised from the values of $\overline{\lambda_1}$ and $\overline{\lambda_2}$. More precisely, the method synthesises a vector $\overline{r}$ of real

numbers and two constants $\delta > 0$ and δ_0 such that:

$$\begin{cases} \overline{r}.\overline{x} \geq \delta_0 & \forall \overline{x} \,.\, B(\overline{x}) \\ \overline{r}.\overline{x}' \leq \overline{r}.\overline{x} - \delta & \forall \overline{x},\, \overline{x}' \,.\, B(\overline{x}) \wedge S(\overline{x}, \overline{x}') \end{cases}$$

These conditions guarantee the termination of the loop. The aim of [18] is proving termination and to exhibit a certificate of the proof. In this respect, *any* ranking function is a valid certificate. Our aim is slightly different: we seek for the *least* upper bound to the length of the worst case call chain to a recursive *Core-Safe* function f^S. Then, we introduce two variations to [18]:

- We replace the restriction $\delta > 0$ by $\delta \geq 1$. In this way, each transition counts at least as an internal call to f^S and so we will get an upper bound to the number of internal calls in the chain (this would not be true if $0 < \delta < 1$).
- We reformulate the problem as a *minimisation* one. We ask for the solution giving the minimum value of the following objective function:

$$Obj \stackrel{\text{def}}{=} \sum \overline{\lambda_1} + \sum \overline{\lambda_2} - \delta_0$$

Minimising $-\delta_0$ is equivalent to maximising δ_0, expressing that we look for the minimum value of $\overline{r}.\overline{x}$ that is still an upper bound. Minimising the values of $\overline{\lambda_1}$ and $\overline{\lambda_2}$ is needed because we have seen that requiring only the first condition frequently leads to unbounded linear problems with an infinite number of solutions and the minimum one is in the infinite for some λ_i.

The only remaining task is to codify our abstract size functions as **while** loops. In this respect, the only meaningful information is the size change between the arguments of an external call to f^S and the arguments of its internal calls. The result sizes are not relevant for termination of the call chains. But we must decide what to do when there are more than one internal call, either excluding each other (as in $merge^S$), or executed in sequence (as in $msort^S$). Our approach has been to compute the convex hull of the restrictions coming from all the internal calls, and to use this polyhedron both as the guard $B(\overline{x})$ —by collecting all restrictions depending only on $\overline{x}$—, and as the transition relation $S(\overline{x}, \overline{x}')$ —by collecting all restrictions depending both on $\overline{x}$ and $\overline{x}'$. The justification of this decision in the case of excluding calls is clear: at each 'iteration' the function may decide to take a possible different branch, so the convex hull amounts to compute the logical 'or' of the restrictions coming for all the branches. In the case of consecutive calls, the reasoning is different: at each internal node of the call tree, the function will take all the children branches and we seek for a bound to the worst case path. The convex hull collects in this case the minimum set of restrictions applicable to all the branches. It is like having a loop that non-deterministically decides at each iteration which branch will follow in the tree. A bound to the iterations of this 'loop' is then a bound to the longest path in the call tree.

In Fig. 9 we show the restrictions of each internal call for $split^S$, $merge^S$, and $msort^S$, and their respective convex hulls. When introducing this data to the

Function	Internal call 1	Internal call 2	Convex hull
split n x	$\{n \geq 1,\ x \geq 2,\ n' = n-1,\ x' = x-1\}$		$\{n \geq 1,\ x \geq 2,\ n' = n-1,\ x' = x-1\}$
merge x y	$\{x \geq 2,\ y \geq 2,\ x' = x-1,\ y' = y\}$	$\{x \geq 2,\ y \geq 2,\ x' = x,\ y' = y-1\}$	$\{x \geq 2,\ y \geq 2,\ x+y = x'+y'+1\}$
msort x	$\{x \geq 3,\ x' = \frac{1}{2}x\}$	$\{x \geq 3,\ x' = \frac{1}{2}x+1\}$	$\{x \geq 3,\ x' \geq \frac{1}{2}x,\ x' \leq \frac{1}{2}x+1\}$

Fig. 9. Termination restrictions of *split*, *merge* and *msort*

above formulation of Podelski and Rybalchenkos's method, we got the following ranking functions:

Function	$\overline{r}$	δ_0	$len_f(\overline{x})$	$B(\overline{x})$
split n x	$[0,1]$	2	x	$n \geq 1 \wedge x \geq 2$
merge x y	$[1,1]$	4	$x+y-2$	$x \geq 2 \wedge y \geq 2$
msort x	$[2]$	2	$2x$	$x \geq 3$

We take as $len_f(\overline{x})$ the expression $\overline{r}.\overline{x} - \delta_0 + 2$, because $\overline{r}.\overline{x} - \delta_0 + 1$ is a bound to the number of 'iterations', each one corresponding to an internal call to f^S, and we add 1 for taking into account the code before the loop representing the initial call of the chain. Of course, this length is valid when $B(\overline{x})$ holds at the beginning. Otherwise, the length is just 1. Notice that the bounds for *split* and *merge* are tight, while the one for *msort* is not (a tight bound would be $\log_2 x + 1$). An obvious limitation of the method is that it only can give a linear function as a result.

6 Conclusions

We have shown that linear techniques can be successfully applied to a first-order functional language in order to infer size invariants between the arguments and results of functions, and upper bounds to the longest call chain of recursive functions. To this respect, some previous transformations of the program may be needed in order to distinguish between internal recursive calls related by 'or' (i.e. excluding each other), from those related by 'and' (i.e. executed in a sequence). This distinction comes for free in Prolog programs, but not in functional ones.

Linear techniques —namely abstract interpretation on polyhedral domains and linear ranking function synthesis— have been extensively used in logic and imperative languages (see [3] for a broad bibliography), but apparently there have been no much interest in applying them to functional languages. An exception regarding termination analysis (not necessarily a linear one) is Sereni and Jones work [20] applying the SCT criterion to the termination of ML programs.

This paper should be considered as a proof-of-concept one in the sense that we still have neither a complete implementation, nor a full-size benchmark of test cases. The algorithms presented here have been implemented in SWI-Prolog[1], by using its CLP(Q) and simplex libraries. We have adapted to this Prolog system Andy King's algorithm [8] for computing convex hulls. We are grateful to the implementers of these tools and algorithms, and also to our colleague Samir Genaim for putting us on the track of Andy King's works.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis Symposium, SAS'08*, pages 221–237. LNCS 5079, Springer, 2008.
2. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis Symposium, SAS'10*, LNCS (to appear), pages 1–16. Springer, 2010.
3. R. Bagnara, P. M. Hill, E. Ricci 0002, and E. Zaffanella. Precise widening operators for convex polyhedra. *Sci. Computer Programming*, 58(1-2):28–56, 2005.
4. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. The Parma Polyhedra Library, User's Manual. Dept. of Mathematics, Univ. of Parma, Italy, Available at: `http://www.cs.unipr.it/ppl/`, July 2002.
5. Amir M. Ben-Amram and Michael Codish. A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 218–232. Springer, 2008.
6. Amir M. Ben-Amram and Chin Soon Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst.*, 29(1), 2007.
7. Florence Benoy and Andy King. Inferring Argument Size Relationships with CLP(R). In John P. Gallagher, editor, *LOPSTR*, volume 1207 of *LNCS*, pages 204–223. Springer, 1996.
8. Florence Benoy, Andy King, and Frédéric Mesnard. Computing convex hulls with a linear solver. *TPLP*, 5(1-2):259–271, 2005.
9. Michael Colón and Henny Sipma. Practical Methods for Proving Program Termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
11. Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.
12. Laure Gonnord and Nicolas Halbwachs. Combining Widening and Acceleration in Linear Relation Analysis. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 144–160. Springer, 2006.
13. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.
14. S. Lucas and R. Peña. Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In *Proc. Logic-Based Program Synthesis and Transformation, LOPSTR'08, Valencia, Spain*, pages 43–57, July 2008.

[1] Available at `http://www.swi-prolog.org/`.

15. M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *ACM Principles and Practice of Declarative Programming, PPDP'08, Valencia, Spain, July. 2008*, pages 152–162, 2008.
16. M. Montenegro, R. Peña, and C. Segura. A Simple Region Inference Algorithm for a First-order Functional Language. In S. Escobar, editor, *Work. on Functional and Logic Programming, WFLP 2009, Brasilia*, pages 145–161. LNCS 5979, 2009.
17. M. Montenegro, R. Peña, and C. Segura. A space consumption analysis by abstract interpretation. In *Selected papers of Foundational and Practical Aspects of Resource Analysis, FOPARA'09, Eindhoven*, pages 34–50. LNCS 6324, Nov. 2009.
18. Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.
19. Andreas Podelski and Andrey Rybalchenko. Transition Invariants. In *LICS*, pages 32–41. IEEE Computer Society, 2004.
20. Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2005.