UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Una plataforma para el uso combinado de procesadores en la demostración de la terminación de la reescritura

Trabajo Fin de Máster

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

AUTOR/A: Martínez López, Alejandro

Tutor/a: Lucas Alba, Salvador

CURSO ACADÉMICO: 2021/2022

Universidad Politécnica de Valencia

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

Trabajo Final de Máster

TOrch: una plataforma para el uso combinado de procesadores en la demostración de la terminación de la reescritura

Tutor:
Salvador Lucas

Alumno:
Alejandro Martínez

Curso Académico 2021/2022

Tabla de contenidos

Abstract

TOrch es una herramienta que demuestra la terminación de sistemas de reescritura utilizando el denominado Marco de Pares de Dependencia (*DP Framework*) a partir de procesadores aportados por usuarios externos. Dichos procesadores son evaluados antes de utilizarse para garantizar cierto grado de seguridad. La herramienta acepta sistemas de reescritura de términos en formato TPDB (Termination Problem Data Base) y permite la definición de una estrategia de terminación a partir de los procesadores disponibles. La demostración se realiza siguiendo la estrategia introducida y se presenta al usuario de forma uniforme a partir de los reportes parciales obtenidos de la ejecución de cada procesador. Dicha demostración incluye la procedencia de cada procesador utilizado.

Palabras clave: Análisis de programas, Sistemas de reescritura, Terminación de programas, Verificación automática, Pares de dependencia.

TOrch is a tool which can be used to prove the termination of term rewriting systems using the Dependency Pair Framework (*DP Framwork*) based on processors supplied by external users. These processors are evaluated before they are available in order to guarantee certain degree of security. This tool accepts term rewriting systems in TPDB (Termination Problem Data Base) format and it allows users the definition of a termination strategy by combining the available processors. The demonstration is built following the strategy entered, and it is uniformly shown to the user by collecting the partial reports generated by each processor. The demonstration also includes the origin of each processor used.

Keywords: Program analysis, Rewriting systems, Program termination, Automatic verification, Dependency pairs.

1 Introducción

La terminación, en sistemas de reescritura de términos, es la propiedad que asegura no existe ninguna secuencia de reescritura infinita. La mayoría de herramientas de demostración automática de la terminación de sistemas de reescritura (por ejemplo, AProVE [4], MU-TERM [7], T$_{\text{T}}$T$_2$ [14], etc., ver `http://www.jaist.ac.jp/~hirokawa/tool/?tag=termination`) utilizan el denominado Marco de Pares de Dependencia (*DP Framework*) [5]. Este permite combinar distintas técnicas de análisis de terminación mediante la descomposición y simplificación del problema de terminación inicial en subproblemas que son más fáciles de abordar y que a su vez también se pueden descomponer y simplificar. Un componente fundamental del marco son los denominados *procesadores* que se encargan de dicha descomposición y simplificación.

Un aspecto fundamental en las demostraciones de terminación en dicho marco es la *organización* apropiada del uso de los procesadores, que puede afectar mucho al éxito o rendimiento de las demostraciones. Dicha organización puede especificarse mediante una *estrategia* de demostración, que guiará la aplicación y uso de los procesadores.

En este trabajo se plantea la implementación de una herramienta, llamada TOrch (Termination Orchestrator), que permite verificar la terminación de la reescritura automáticamente utilizando el DP Framework. A diferencia de las herramientas de terminación ya existentes, TOrch permite al usuario definir la forma en la que se va a tratar de verificar la terminación (estrategia de terminación) a partir de módulos aportados por usuarios externos. Esta característica supone varias ventajas:

- Flexibilidad: como la terminación, en sistemas de reescritura, es una propiedad indecidible, la flexibilidad que se le proporciona al usuario hace posible que, siempre que se disponga de una estrategia y unos procesadores adecuados, se pueda determinar la terminación en un número mayor de casos respecto a una herramienta de terminación con estrategias predeterminadas.

- Colaboración: los módulos son aportados de forma independiente por usuarios o grupos de investigación externos, por lo que en una estrategia de terminación pueden intervenir módulos de distintos orígenes, favoreciendo una colaboración entre grupos de investigación que no se da en las herramientas de terminación tradiciones.

Los módulos se aportan en formato ejecutable de Linux. Esto implica un nivel de acoplamiento muy bajo, ya que no es necesario realizar ningún cambio en la aplicación para la inserción de nuevos módulos, algo que es imprescindible para que la herramienta sea escalable. Por otro lado, aceptar ejecutables aportados por usuarios externos entraña problemas evidentes de seguridad puesto que podrían contener código malicioso. Para ello, los módulos aportados no están disponibles inmediatamente, sino que deben ser verificados por un usuario autorizado a ello. Dicha verificación consiste en comprobar que el origen del módulo es una persona o grupo de investigación conocido, y si es así, probar localmente que el módulo cumple con los requisitos necesarios para funcionar en TOrch.

La herramienta es accesible a través de su interfaz web. Podemos encontrarla siguiendo el siguiente enlace:

`http://zenon.dsic.upv.es/torch/`

2 Preliminares

Los conceptos definidos a continuación siguen la notación del libro *Advanced Topics in Term Rewriting* [22].

Término: Sea $\mathcal{X}$ un conjunto contable de variables y una signatura $\mathcal{F}$ (que es un conjunto de símbolos de función, cada uno de los cuales con su correspondiente aridad fija dada por la función $ar : \mathcal{F} \to \mathbb{N}$), el conjunto de términos sobre la signatura $\mathcal{F}$ es el conjunto mínimo $\mathcal{T}(\mathcal{F}, \mathcal{X})$ que satisfaga:

1. Si $x \in \mathcal{X}$, entonces $x \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.
2. Si a es un símbolo constante (es decir, tal que $ar(a) = 0$), entonces $a \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.
3. Si f es un símbolo de función (es decir, tal que $k = ar(f) > 0$) y $t_1, ..., t_k \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, entonces $f(t_1, ..., t_k) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

Posición: Una posición p es una cadena (posiblemente vacía) de números enteros positivos:

$$p \in \mathcal{P}os = \{\Lambda\} \cup \{i.q \mid i \in N_{>0} \wedge q \in \mathcal{P}os\}$$

El conjunto de posiciones de un término t es:

$$\mathcal{P}os(t) = \begin{cases} \{\Lambda\} & \text{if } t \in \mathcal{X} \\ \{\Lambda\} \cup \bigcup_{1 \leqslant i \leqslant k} i.\mathcal{P}os(t_i) & \text{if } t = f(t_1, \ldots, t_k) \end{cases}$$

Subtérminos de un término:

1. El subtérmino de t a la posición p, donde $p \in Pos(t)$ es $t|_p$:

$$t|_\Lambda = t \qquad y \qquad f(t_1, \ldots, t_k)|_{i.p} = t_i|_p$$

2. La profundidad de un subtérmino es la longitud $|p|$ de p.
3. $\mathcal{P}os_{\mathcal{F}}(t)$ es el conjunto de posiciones de subtérminos no variables en t:

$$Pos_{\mathcal{F}}(t) = \{p \in Pos(t) \mid root(t|_p) \in \mathcal{F}\}$$

Decimos que t es un *subtérmino* de s (notación: $s \trianglerighteq t$) si $t = s|_p$ para alguna posición $p \in \mathcal{P}os(s)$. Si, además, $s \neq t$ entonces decimos que t es un

subtérmino *estricto* de s (notación: $s \rhd t$).

Reemplazamiento de subtérminos: Sea $t[s]_p$ el término t donde $t|_p$ ha sido reemplazado por s

$$t[s]_\Lambda = s \qquad y \qquad f(t_1, \ldots, t_i, \ldots, t_k)[s]_{i.p} = f(t_1, \ldots, t_i[s]_p, \ldots, t_k)$$

Sustitución: Una sustitución es una función $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ tal que $\sigma(x) \neq x$ para un conjunto finito de variables, llamado el *dominio* de la sustitución:

$$\mathcal{D}om(\sigma) = \{x \in \ | \ \sigma(x) \neq x\}.$$

Especificamos una sustitución σ mediante las funciones (no triviales) $x_i \mapsto t_i$ que corresponden a las variables $\{x_1, \ldots, x_n\}$ en $\mathcal{D}om(\sigma)$:

$$\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}.$$

La sustitución vacía (o identidad) es ε (nótese que $\mathcal{D}om(\varepsilon) = \varnothing$).

Unificación Un término ℓ unifica con t si existe una sustitución σ (el unificador de t sobre ℓ) tal que $t = \sigma(\ell)$. Es decir, si existe una sustitución para ℓ que hace ambos términos sintácticamente iguales.

Regla de reescritura: Una regla de reescritura es un par ordenado $\ell \to_r$ donde $\ell, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ lado izquierdo (abreviado lhs) y lado derecho (abreviado rhs) respectivamente y:

1. El lado izquierdo no es una variable, es decir, $\ell \notin \mathcal{X}$.
2. Todas las variables que aparecen en el lado derecho, también aparecen en el izquierdo. Es decir, $Var(r) \subseteq Var(\ell)$.

Sistema de reescritura de términos: Un sistema de reescritura de términos (SRT) es un par $\mathcal{R} = (\mathcal{F}, R)$ donde $\mathcal{F}$ es una signatura y R es un conjunto de reglas de reescritura sobre la signatura $\mathcal{F}$.

Una instancia $\sigma(\ell)$ del lado izquierdo ℓ de una regla $\ell \to r$ se denomina redex (*red*ucible *ex*pression) de la regla.

Símbolo constructor y símbolo definido: Dado un TRS $\mathcal{R} = (\mathcal{F}, R)$, podemos particionar su signatura $\mathcal{F}$ como sigue: $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$, donde

$$\mathcal{D} \ = \ \{f \in \mathcal{F} \mid f(\bar{l}) \to r \in R\} \qquad \text{y} \qquad \mathcal{C} \ = \ \mathcal{F} - \mathcal{D}$$

los símbolos $f \in \mathcal{D}$ se denominan *símbolos definidos* y los símbolos $c \in \mathcal{C}$ se denominan *símbolos constructores*.

Relación bien fundada: Dado un conjunto A. Una relación R sobre A (i.e., $R \subseteq \mathsf{A} \times \mathsf{A}$) es *bien fundada* si no existe una secuencia infinita:

$$a_1 \,\mathsf{R}\, a_2 \,\mathsf{R}\, a_3 \,\mathsf{R}\, \cdots \,\mathsf{R}\, a_n \,\mathsf{R}\, a_{n+1} \,\mathsf{R} \cdots$$

donde $a_i \in \mathsf{A}$ para todo $i \geqslant 1$.

Monotonía: Sea $\mathcal{F}$ una signatura. Una relación R sobre $\mathcal{T}(\mathcal{F}, \mathcal{X})$ es *monótona* (o compatible con la estructura de términos en $\mathcal{T}(\mathcal{F}, \mathcal{X})$) si, para todos los símbolos $f \in \mathcal{F}$, $1 \leqslant i \leqslant ar(f)$, y términos $s_1, \ldots, s_k, s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s \,\mathsf{R}\, t$ implica $f(s_1, \ldots, s_{i-1}, s, \ldots, s_k) \,\mathsf{R}\, f(s_1, \ldots, s_{i-1}, t, \ldots, s_k)$.

Estabilidad: Sea una signatura $\mathcal{F}$. Una relación R sobre $\mathcal{T}(\mathcal{F}, \mathcal{X})$ es *estable* si, para toda sustitución $\sigma \in$ y términos $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s \,\mathsf{R}\, t$ implica $\sigma(s) \,\mathsf{R}\, \sigma(t)$.

Preorden: Un *preorden* $(\mathsf{A}, \gtrsim)$ es un par que consiste en un un conjunto A y una relación binaria reflexiva y transitiva $\gtrsim$ sobre A.

Relación de reescritura en un paso: Un término $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ se reescribe a t en la posición p (escrito $s \xrightarrow{p}_{\mathcal{R}} t$, $s \to_{\mathcal{R}} t$, o simplemente $s \to t$ y llamado *un paso de reescritura*) si existe una posición $p \in Pos(s)$ tal que

- $s|_p = \sigma(\ell)$ para algún $\ell \to r \in R$ y sustitución σ, y
- $t = s[\sigma(r)]_p$.

Reescritura de términos: Dado un SRT $\mathcal{R} = (\mathcal{F}, R)$, definimos la *relación de reescritura* $\to_{\mathcal{R}}$ (o simplemente $\to$ si R no tiene ambigüedades en el contexto) como el conjunto de todos los *pasos de reescritura* sobre los términos $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

3 Terminación de los sistemas de reescritura

En esta sección se describen los conceptos teóricos que son aplicados por la herramienta. Para llegar a hablar de terminación en la reescritura, hay ciertos conceptos que son necesarios describir previamente.

Un Sistema de Reescritura $\mathcal{R}$ es *terminante* si no hay ningún término t que inicie una secuencia de reescritura infinita $t = t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots t_n \to_{\mathcal{R}} \cdots$.

3.1 Pares de dependencia

Dado $t = f(t_1, ..., t_k)$, marcamos t de la siguiente forma: $t^\sharp = f^\#(t_1, ..., t_k)$ donde $f^\sharp$ (o simplemente F) es un nuevo símbolo llamado *tupla*.

Sea $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ un sistema de reescritura de términos, el conjunto de pares de dependencia (DPs) se define como:

$$DP(\mathcal{R}) = \{\ell^\# \to s^\# \mid \ell \to r \in R, r \trianglerighteq s, root(s) \in \mathcal{D}\}$$

Por ejemplo, dado el siguiente TRS:

$$f(f(x)) \quad \to \quad f(g((f(x))) \tag{1}$$

Obtendríamos los siguientes pares de dependencia:

$$F(f(x)) \quad \to \quad F(g((f(x))) \tag{2}$$
$$F(f(x)) \quad \to \quad F(x) \tag{3}$$

3.2 Cadenas de pares de dependencia

Sean $\mathcal{R}$ y $\mathcal{P}$ TRSs, una $(\mathcal{P}, \mathcal{R})$-cadena es una secuencia finita o infinita de pares (posiblemente renombrados) $u_i \to p_i \in \mathcal{P}$ tales que $Var(u_i) \cap Var(u_j) = \varnothing$ siempre que $i \neq j$, junto a una sustitución σ que satisfaga que, para todo $i \geqslant 1, \sigma(v_i) \to^*_{\mathcal{R}} \sigma(u_{i+1})$ y $\sigma(v_i)$ es $\mathcal{R}$-terminante.

3.3 Terminación con pares de dependencia

Dado un sistema de reescritura $\mathcal{R}$, la ausencia de $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$-cadenas de pares de dependencia infinitas caracteriza la terminación de $\mathcal{R}$. Es decir, un sistema de reescritura $\mathcal{R}$ es terminante si, y solo si, no existe una $(DP(\mathcal{R}), \mathcal{R})$-cadena infinita.

Si $\mathcal{P}$ es finito, entonces cualquier $(\mathcal{P}, \mathcal{R})$-cadena infinita involucra un número finito de pares. Por lo tanto, para todas las $(\mathcal{P}, \mathcal{R})$-cadenas infinitas existe un subconjunto finito $\mathcal{Q} \subseteq \mathcal{P}$ de pares que son utilizados de forma infinita.

Este hecho se utiliza en el Marco de Pares de Dependencia disponiendo los pares en un grafo y estudiando sus *ciclos*.

3.4 Grafo de dependencias

Sean $\mathcal{R}$ y $\mathcal{P}$ dos sistemas de reescritura, el grafo de pares $G(\mathcal{P}, \mathcal{R})$ asociado a $\mathcal{R}$ y a $\mathcal{P}$ tiene a $\mathcal{P}$ como conjunto de nodos. Entre dichos nodos, existe un arco desde $u \to v \in \mathcal{P}$ a $u' \to v' \in \mathcal{P}$ y existen sustituciones θ y θ' tales que $\theta(v) \to_{\mathcal{R}}^{*} \theta'(u')$.

El grafo de dependencias $DG(\mathcal{R})$ es un caso particular del grafo de pares, en el que $DG(\mathcal{R}) = G(DP(\mathcal{R}), \mathcal{R})$.

Contando con el grafo de dependencias, podemos redefinir la terminación: un TRS $\mathcal{R}$ es terminante si, y solo si, no hay una $(\mathcal{P}, \mathcal{R})$-cadena para ningún ciclo $\mathcal{P}$ en $DG(\mathcal{R})$. Esto implica que, a la hora de analizar la terminación, podemos centrarnos únicamente en los ciclos del grafo de dependencias.

El grafo de dependencias no es computable por ningún algoritmo ya que el problema de alcanzabilidad $s \to_{\mathcal{R}}^{*} t$ es indecidible. Por lo tanto, los grafos de dependencia calculados siempre consistirán en aproximaciones.

El procedimiento para calcular dichas aproximaciones se desarrollará más adelante.

4 DP Framework

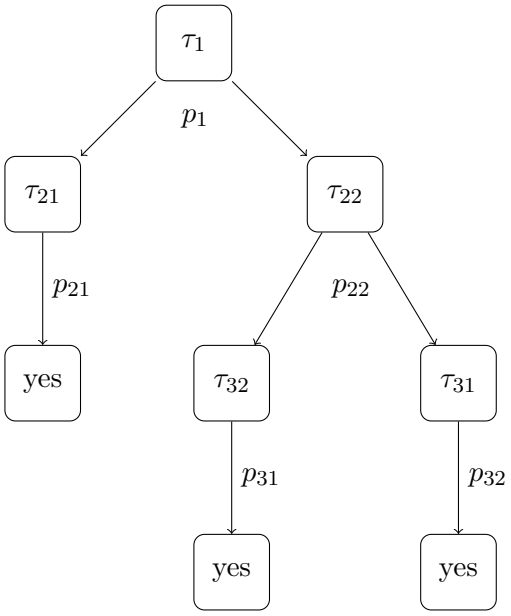
El DP Framework proporciona una serie de procedimientos y reglas que se utilizan para verificar la terminación en sistemas de reescritura.

Como se ha indicado antes, la mayoría de herramientas de demostración de la terminación de sistemas de reescritura están basadas en el Marco de Pares de Dependencia. A continuación describimos brevemente en qué consiste:

4.1 Árbol de terminación

La idea principal de este marco de trabajo se basa en la construcción de una prueba de terminación en forma de árbol, que representa la aplicación de distintos procesadores (que se definirán más adelante) sobre problemas de

terminación τ que se van simplificando. Un ejemplo de árbol de terminación sería el siguiente:



Un apunte importante sobre el árbol de terminación es que no se garantiza que se vaya a alcanzar un resultado concluyente, ya que la terminación en la reescritura es un problema indecidible. Es por eso que sus implementaciones necesitarán de tiempos máximos de ejecución (o *timeouts*).

4.2 DP Problems

Un DP-Problem es un par $\tau = (\mathcal{P}, \mathcal{R})$ donde $\mathcal{P}$ y $\mathcal{R}$ son SRTs.

- Un DP problem τ es finito si no hay ninguna cadena de dependencias $(\mathcal{P}, \mathcal{R})$ infinita.
- Un DP problem τ es infinito si $\mathcal{R}$ es no terminante, o si existe una cadena de dependencias $(\mathcal{P}, \mathcal{R})$ infinita.

Es decir, un SRT $\mathcal{R}$ es terminante si, y solo si, el DP problem $(DP(\mathcal{R}), \mathcal{R})$ es finito.

4.3 DP Processors

Un procesador de pares de dependencia (o DP processor) P consiste en una función que toma un DP problem y devuelve un conjunto de DP problems.

Alternativamente, el procesador también puede devolver *no*. Un procesador aplicado en un DP problem $(\mathcal{P}, \mathcal{R})$ donde $\mathcal{P} = \varnothing$ denota que ese DP problem es trivialmente finito, y se añade como hoja al árbol de terminación mediante una etiqueta *yes*.

Un procesador de pares de dependencia es:

- *Sound*: si para todos los DP problems τ, τ es finito siempre que $P(\tau) \neq no$ y $\forall \tau' \in P(\tau)$, τ' es finito. Esta característica es esencial para probar la terminación del sistema de reescritura, pues para poder concluir que un DP problem es finito (y por tanto, el SRT terminante), todas las hojas del árbol de terminación deben ser *yes*.
- *Complete*: si para todos los DP problems τ, τ es finito siempre que $P(\tau) = no$ o $\exists \tau' \in P(\tau)$, τ' es infinito. Esta característica es importante para demostrar la no terminación de un sistema de reescritura, ya que, con que una sola hoja del árbol de terminación esté etiquetada como *no*, queda demostrado que el DP problem es infinito y, por tanto, que el SRT es no terminante.

A continuación se introducen algunos procesadores del DP Framework. Estos procesadores se corresponden con los que van a existir inicialmente en la herramienta.

4.3.1 Procesador de componentes fuertemente conexos (SCC)

Un conjunto de pares $\mathfrak{C} \neq \varnothing$ constituye un ciclo en $G(\mathcal{P}, \mathcal{R})$ si para cualquier $s \to t$ y $u \to v$ en $\mathfrak{C}$ existe un camino no vacío de $s \to t$ a $u \to v$ en el grafo que solo recorre pares de $\mathfrak{C}$.

Un ciclo $\mathfrak{C}$ es un componente fuertemente conexo o SCC (*Strongly Connected Component*) de $G(\mathcal{P}, \mathcal{R})$ si $\mathfrak{C}$ es maximal, es decir, si $\mathfrak{C}$ no es un subconjunto de otro ciclo.

Haciendo uso de estos conceptos, ya podemos definir el procesador de componentes fuertemente conexos o SCC. Sean $\mathcal{R}$ y $\mathcal{P}$ SRTs, el procesador P_{SCC} se define:

$$P_{SCC}(\mathcal{P}, \mathcal{R}) = \{(\mathfrak{C}, \mathcal{R}) \mid \mathfrak{C} \text{ son todos los pares de un SCC en } G(\mathcal{P}, \mathcal{P})\}$$

que es *sound* y *complete*.

4.3.2 Procesador de subtérminos

Sea $\mathcal{R} = (\mathcal{F}, \mathcal{R})$ un SRT. El conjunto de símbolos raíz (o *root*) asociados a $\mathcal{R}$ es aquel que cumple:

$$Root(\mathcal{R}) = \{root(\ell) \mid \ell \to r \in R\} \cup \{root(r) \mid \ell \to r \in R, r \notin \mathcal{X}\}$$

Dado un SRT $\mathcal{R}$, una proyección simple de $\mathcal{R}$ es una función π que asigna a cada símbolo f donde $ar(f) > 0$ y $f \in Root(\mathcal{R})$ un argumento en la posición $i \in \{1, ..., k\}$. La función que asigna a cada término $t = (t_1, ..., t_k)$ con $f \in Root(\mathcal{R})$ su subtérmino $\pi(t) = t\ |_{\pi(f)}$ también se denota con π. $\pi(x) = x$ si $x \in \mathcal{X}$.

Extendemos la definición de π a SRTs $\mathcal{R} = (\mathcal{F}, R)$ como sigue: $\pi(\mathcal{R}) = (\mathcal{F}, \pi(R))$, donde

$$\pi(R) = \{\pi(\ell) \to \pi(r) \mid \ell \to r \in R\}$$

Teniendo en cuenta las definiciones anteriores, podemos definir el procesador de subtérminos. Sean $\mathcal{R} = (\mathfrak{C} \uplus \mathcal{D}, R)$ y $\mathcal{P}$ sistemas de reescritura de términos donde $\mathcal{P}$ no contiene ninguna regla colapsante (es decir, tal que $r \in \mathcal{X}$) y $Root(\mathcal{P}) \cap \mathcal{D} = \varnothing$. Sea π una proyección simple sobre $\mathcal{P}$ y $\mathcal{P}_{\pi,\rhd} = \{u \to v \in \mathcal{P} \mid \pi(u) \rhd \pi(v)\}$. El procesador $P_{subterm}$ se define como:

$$P_{subterm}(\mathcal{P}, \mathcal{R}) = \begin{cases} \{(\mathcal{P} - \mathcal{P}_{\pi,\rhd}, \mathcal{R})\} & \text{si } \pi(\mathcal{P}) \subseteq \trianglerighteq \\ \{(\mathcal{P}, (R))\} & \text{de lo contrario} \end{cases}$$

que es *sound* y *complete*.

Una característica destacable de este procesador es que no utiliza las reglas de reescritura definidas en $\mathcal{R}$, sino que solo los pares de $\mathcal{P}$ son tenidos en cuenta.

4.3.3 Procesador de infinitud

Sean $\mathcal{R}$ y $\mathcal{P}$ SRTs, el procesador de infinitud que se define como

$$P_{Inf}(\mathcal{P}, \mathcal{R}) = \begin{cases} no & \text{si } v = \theta(u) \text{ para alguna regla } u \to v \in \mathcal{P} \\ & \text{y substitución } \theta, \\ \{(\mathcal{P}, (R))\} & \text{de lo contrario} \end{cases}$$

Este procesador supone una herramienta básica para demostrar la no terminación.

4.3.4 Procesador de pares de reducción

Un par de reducción $(\gtrsim, \sqsupset)$ consta de un preorden estable y monótono (llamado componente no estricto) y un orden estable y bien fundado $\sqsupset$ (llamado componente estricto).

Un ejemplo de par de reducción sería el siguiente: $(\trianglerighteq_\varepsilon, \rhd_\varepsilon)$ donde $\trianglerighteq_\varepsilon$ es la relación de embebimiento (o *embedding*) inducido por el SRT

$$\varepsilon mb(\mathcal{F}) = \{f(x_1, ..., x_{ar(f)}) \to x_i \mid f \in \mathcal{F}, 1 \leqslant i \leqslant ar(f)\} : s \trianglerighteq_\varepsilon t$$

si $s \to^*_{\varepsilon mb(\mathcal{F})} t$ y

$$s \rhd_\varepsilon t$$

si $s \to^+_{\varepsilon mb(\mathcal{F})} t$.

Un filtrado de argumentos π sobre la signatura $\mathcal{F}$ es una función que asigna a cada símbolo de función $f \in \mathcal{F}, ar(f) > 0$ la posición de un argumento $i \in 1, ..., k$ o una lista (posiblemente vacía) de posiciones de argumentos $[i_1, ..., i_m]$ tales que $1 \leqslant i_1 < ... < i_m \leqslant k$.

Un filtrado de argumentos induce una función $\mathcal{T}(\mathcal{F}, \mathcal{X})$ a $\mathcal{T}(\mathcal{F}_\pi, \mathcal{X})$, también denotado por π:

$$\pi(t) = \begin{cases} t & \text{si } t \text{ es una variable} \\ \pi(t_i) & \text{si } t = f(t_i, ..., t_k) \text{ y } \pi(f) = i \\ f(\pi(t_{i_1}), ..., \pi(t_{i_m})) & \text{si } t = f(t_i, ..., t_k) \text{ y } \pi(f) = [i_1, ..., i_m] \end{cases}$$

La signatura $\mathcal{F}_\pi$ de términos filtrados $\pi(t)$ puede ser diferente de la signatura $\mathcal{F}$ de t.

Sean $\mathcal{R}$ y $\mathcal{P}$ sistemas de reescritura de términos, y$(\gtrsim, \sqsupset)$ un par de reducción tal que $\mathcal{R} \subseteq \gtrsim$ y $\mathcal{P} \subseteq \gtrsim \cup \sqsupset$. Sea $\mathcal{P}_\sqsupset = \{u \to v \in \mathcal{P} \mid u \sqsupset v\}$. El *procesador de pares de reducción* P_{RP} dado por

$$P_{RP}(\mathcal{P}, \mathcal{R}) = \{(\mathcal{P} - \mathcal{P}_\sqsupset, \mathcal{R})\}$$

es *sound* y *complete*.

5 Incorporación de procesadores

En la sección anterior se ha presentado el DP Framework, que es el medio por el que la herramienta presentada en este trabajo (TOrch) tratará de

demostrar la terminación de los sistemas de reescritura de términos. Esta herramienta hace el papel de integrador de DP processors, que podrán ser aportados por usuarios externos.

En esta sección se definen las características que deben cumplir los DP processors que se añadan a la plataforma para asegurar la compatibilidad con la herramienta. También se detalla el proceso que se ha seguido a la hora de extraer los procesadores de los que disponemos inicialmente.

5.1 Características

Una de las ventajas más importantes de esta herramienta es el bajo nivel de acoplamiento entre los procesadores y la plataforma. Esta característica dota de escalabilidad al sistema. Sin embargo, para que los procesadores puedan funcionar correctamente, tienen una serie de requisitos comunes que deben cumplir:

- El procesador debe ser un archivo ejecutable en Linux.
- El ejecutable debe aceptar un parámetro "-i" que permita establecer la ruta que apunta al fichero que contiene el DP problem. que deseamos suministrar al procesador. Este DP problem podrá expresarse en un formato extendido de TPDB. Esta extensión permite expresar un DP problem $(\mathcal{P}, \mathcal{R})$ de la siguiente forma:

```
(VAR x y)
(PAIRS
A -> B
)
(RULES
a -> b)
```

  Donde $\mathcal{P}$ y $\mathcal{R}$ son representados con las palabras reservadas *PAIRS* y *RULES* respectivamente.

- El ejecutable debe aceptar un parámetro "-o" que permita establecer la ruta en la que se va a depositar el fichero con el resultado de la ejecución. Este resultado también debe emitirse en el mismo formato que se usa para la entrada, es decir, la extensión de TPDB mencionada.

El formato TPDB admite comentarios. El procesador debe usar estos comentarios para proporcionar al orquestador información sobre la ejecución:

- Como ya sabemos, un procesador no es más que una función que toma un DP problem y devuelve un conjunto de DP problems (o el valor "no"). Dependiendo del número de DP problems en ese conjunto:
  - Si el conjunto es vacío ($\varnothing$) no se devolverá ningún DP problem, por lo tanto, en este caso, no habrá ningún DP problem en el fichero de salida. El orquestador inferirá la finitud del problema a partir de esta situación.
  - Si el conjunto contiene algún DP problem, cada uno de ellos será escrito secuencialmente, indicando antes de cada uno un comentario de tipo @dpproblem que etiquete el problema con un número natural único:

    ```
    (COMMENT @dpproblem #n)
    ```

    En caso de que solo un DP problem sea devuelto, este comentario es opcional.

- Un comentario de tipo @proofreport que contenga un reporte que detalle el proceso que se ha seguido en la ejecución del procesador. Es importante que dicho reporte tenga cierta calidad, ya que se utilizará en la construcción de la demostración que emitirá la herramienta:

  ```
  (COMMENT @proofreport ...)
  ```

  Este comentario es obligatorio y debe situarse en cualquier parte del documento.

- Un comentario de tipo @result que contenga la palabra "NO" en caso de que el procesador haya determinado la infinitud del DP problem introducido:

  ```
  (COMMENT @result NO)
  ```

  Esta es la única variante aceptada de este comentario, cualquier otro valor será ignorado.

5.2 Extracción de MU-TERM

Para poder tener una primera versión operativa de la herramienta, debemos disponer de un conjunto inicial de procesadores. Estos procesadores se han extraído de la herramienta llamada MU-TERM.

MU-TERM es una herramienta desarrollada en Haskell que sirve para probar la terminación en sistemas de reescritura. A nivel interno, también basa la terminación en el DP Framework, por lo que también usa procesadores que podremos aprovechar para TOrch. Aunque MU-TERM también dispone de distintas estrategias de terminación, el usuario no escoge cuál de ellas quiere utilizar, sino que el sistema escoge la más adecuada en función del problema que se le proporcione.

El trabajo a realizar, en esta fase del proyecto, consiste en aprovechar MU-TERM para extraer los procesadores que utiliza, y adaptarlos para que cumplan las características descritas en la sección anterior.

El proceso de extracción es simple. En MU-TERM, los procesadores ya están separados en módulos independientes. El trabajo de extracción consiste en adaptar estos módulos para que puedan ser ejecutados de forma independiente, de forma que el funcionamiento sea el siguiente:

1. Se obtiene el DP problem ubicado en el fichero cuya ruta se define en el parámetro "-i".
2. Si el DP problem obtenido no contiene pares es trivialmente finito. En caso contrario, se ejecuta el procesador para el DP problem leído.
3. Escribe en el fichero cuya ruta se define en el parámetro "-o"
    - Si el procesador se ha ejecutado y ha obtenido *NO*, este resultado es escrito. En caso contrario, se escribe el conjunto de DP problems obtenidos junto a la demostración del proceso.
    - Si el procesador no ha llegado a ejecutarse, simplemente se indica en la demostración que el DP problem es finito.

El formato en el que se realizan las anteriores escrituras cumplen los requisitos detallados en la sección 5.1.

5.3 Procesadores extraídos

Aunque muchos son variantes del mismo tipo, MU-TERM cuenta con cientos de procesadores ideados para adaptarse al problema que va a tratar de procesar. Para nuestra herramienta, inicialmente será suficiente con disponer de algunos de ellos.

Los procesadores que han sido seleccionados están destinados a resolver problemas de terminación de sistemas reescritura de términos estándar, por lo que ningún otro tipo de SRT estará contemplado inicialmente. En particular, los procesadores extraídos son:

- Procesador SCC (strongly connected components) o procesador de componentes fuertemente conexos.
- Procesador Inf (infiniteness) o procesador de infinitud.
- Procesador Sub (subterm) o procesador de subtérminos.
- Procesador RP (reduction pairs) o procesador de pares de reducción.

Estos procesadores se corresponden con aquellos cuyo funcionamiento ha sido descrito en la sección 4.3.

6 Orquestador

Esta sección describe la implementación y las características del programa que utiliza los procesadores aislados para construir estrategias y obtener resultados homogéneos. Dicho programa es el que hemos llamado *orquestador*. El orquestador está implementado en C#, concretamente en una biblioteca de clases [18], característica que hace al programa independiente de la tecnología que se usa. Es decir, no depende de ninguna infraestructura en particular como pueden ser marcos de trabajo, la interfaz web o bases de datos. Esto supone que, a pesar de que TOrch sea una aplicación web, toda su lógica es independiente, de forma que sería sencillo, por ejemplo, desarrollar una versión CLI (Command Line Interface) que se ejecute desde una terminal.

6.1 Componentes

En esta sección se describen los distintos componentes del orquestador que son percibidos por el usuario, que esencialmente consisten en la entrada/salida del programa.

6.1.1 Sistema de reescritura de términos

Se debe proporcionar un sistema de reescritura de términos en formato TPDB. De momento, este es el único formato compatible. Otras variantes de sistemas de reescritura de términos (como, por ejemplo, sistemas de reescritura sensibles al contexto) no son compatibles con el orquestador. Esta sería una característica interesante a implementar, de la que hablaremos en sección de trabajo futuro.

6.1.2 Estrategia de terminación

La estrategia consiste en una secuencia de texto que define cómo se van a combinar los procesadores disponibles a la hora de tratar de verificar la terminación. Esto se consigue mediante el uso de dos operadores:

- "&" combina dos procesadores de forma secuencial. Es decir, primero ejecuta el que se encuentra en el lado izquierdo del operador, y después, basándose en sus resultados, se ejecuta el que se encuentra en el lado derecho del operador.
- "|" combina dos procesadores de forma paralela. Es decir, ambos procesadores se ejecutan simultáneamente utilizando los mismos datos de entrada.

Sea $p_1, p_2, ..., p_n$ el listado de los procesadores disponibles, las propiedades semánticas que satisface cada combinador son las siguientes:

- Para el combinador "&":
  - Propiedad asociativa, según la cual, $(p_1 \& p_2) \& p_3 = p_1 \& (p_2 \& p_3) = p_1 \& p_2 \& p_3$. Su uso nos permite prescindir de los paréntesis innecesarios.
  - Propiedad distributiva (orientada de izquierda a derecha) según la cual $(p_1 \mid p_2) \& p_3 \rightarrow (p_1 \& p_3) \mid (p_2 \& p_3)$. Su uso nos permite realizar las transformaciones lógicas necesarias para obtener la representación que va a utilizar el orquestador cuando se ejecute. Esta propiedad se aplica únicamente de izquierda a derecha. Esto se debe a que, en nuestra semántica, $(p_1 \mid p_2) \& p_3$ significa ejecutar p_3 después de cada procesador anterior. En cambio, esta propiedad orientada de derecha a izquierda $(p_1 \mid p_2) \& p_3 \leftarrow (p_1 \& p_3) \mid (p_2 \& p_3)$ produce una transformación que nos aleja de la semántica deseada.

  - Este operador no cumple la propiedad conmutativa. Es decir $p_1 \& p_2 \neq p_2 \& p_1$. Esto se debe a que, como este operador indica secuencia, el orden de sus miembros es importante.

- Para el combinador "|":
  - Propiedad asociativa, según la cual, $(p_1 \mid p_2) \mid p_3 = p_1 \mid (p_2 \mid p_3) = p_1 \mid p_2 \mid p_3$. Su uso nos permite prescindir de los paréntesis innecesarios.
  - Propiedad conmutativa, es decir $p_1 \mid p_2 = p_2 \mid p_1$. En este caso sí se cumple, ya que, al ejecutar ambos procesadores en paralelo, el orden en el que se ejecuten no influye en el resultado.
  - Llama la atención que este operador no cumpla la propiedad distributiva, es decir $(p_1 \& p_2) \mid p_3 \neq (p_1 \mid p_3) \& (p_2 \mid p_3)$. Aplicar la propiedad distributiva en este combinador produciría que p_3 se aplique de forma redundante. $(p_1 \& p_2) \mid p_3$, en nuestra semántica, significa aplicar p_1 y p_2 secuencialmente y, en paralelo a esa secuencia, ejecutar (una sola vez) el procesador p_3.

Así pues, como hemos visto, el uso de paréntesis sirve para anidar expresiones y evitar ambigüedades, las cuales no están permitidas.

Tomando de nuevo $p_1, p_2, ..., p_n$ como el listado de los procesadores disponibles, algunas estrategias de terminación serían las siguientes:

- La estrategia $p_1 \& p_2 \mid p_3$ no sería una estrategia válida, puesto que existe una ambigüedad entre ambos operadores.

- La estrategia $(p_1 \& p_2) \mid p_3$ sería una estrategia válida, en la que p_1 y p_2 se ejecutan de forma secuencial y, de forma paralela, p_3. Tal y como se indica anteriromente, no se aplica la propiedad distributiva.

- La estrategia $(p_1 \mid p_2) \& p_3$ sería una estrategia válida, en la que p_1 y p_2 se ejecutan de forma paralela. Posteriormente se ejecutará p_3 después de cada resultado. Es decir, en esta ocasión se aplica la propiedad distributiva mencionada anteriormente: $(p_1 \mid p_2) \& p_3 \rightarrow (p_1 \& p_3) \mid (p_2 \& p_3)$.

Desde un nivel de abstracción mayor, en cada nivel de de anidamiento, podemos observar que, en realidad, el operador & permite definir la estrategia describiendo cada paso, mientras que el operador | permite definir la estrategia describiendo cada rama.

6.1.3 Tiempo de espera

El tiempo que tarda la herramienta en devolver el resultado es indeterminado (posiblemente infinito), ya que, como hemos descrito en la descripción inicial del DP Framework, no existe ninguna garantía de que se vaya a alcanzar un resultado concluyente. Es por eso que es necesario definir un tiempo de ejecución máximo en el que el programa debe ejecutarse. Transcurrido ese tiempo, si aún no se ha obtenido una respuesta, la ejecución es abortada.

6.1.4 Resultado

Llamamos resultado a la respuesta que obtiene el programa al ejecutar una estrategia de terminación determinada sobre un sistema de reescritura determinado. Podemos obtener tres resultados distintos:

1. YES: indica se ha podido demostrar la terminación del sistema proporcionado mediante el uso de la estrategia proporcionada.
2. NO: indica se ha podido demostrar la **no** terminación del sistema proporcionado mediante el uso de la estrategia proporcionada.
3. MAYBE: indica que no se ha podido determinar si el sistema proporcionado, mediante el uso de la estrategia proporcionada, es terminante o no. Esta respuesta pude darse por dos motivos:
   (a) El tiempo de espera máximo ha sido alcanzado, por lo que la ejecución ha sido abortada.
   (b) La ejecución ha terminado pero no se ha podido llegar a YES ni a NO mediante la estrategia suministrada. Esto puede ocurrir si dicha estrategia es incompleta, ya que, como hemos visto, no siempre podremos determinar si un DP problem es finito o no basándonos en el uso de un solo procesador.

6.1.5 Demostración

No solo necesitamos un resultado indicando si el sistema es terminante o no, también es necesaria una demostración que describa cómo se ha llegado a ese resultado. Dicha demostración no es más que la traza de ejecución de los procesadores utilizados, y que han llevado al programa a determinar el resultado al que ha llegado, que se construye mediante la concatenación de los informes parciales que ha emitido cada procesador ejecutado.

Además, junto a la ejecución de cada procesador, se indica la procedencia del mismo (si disponemos de ella), es decir: autor/es, grupo de investigación, versión del procesador... De esta forma, los colaboradores de la plataforma obtienen cierto reconocimiento por su trabajo y colaboración.

6.2 Implementación

En este apartado se describen los detalles de implementación que caracterizan el orquestador. Es importante destacar que esta sección se refiere únicamente a la implementación del núcleo del orquestador, con lo que no se describirán detalles de alto nivel como pueden ser la aplicación web o conceptos de infraestructura, como la base de datos. Dichos aspectos se cubrirán en secciones posteriores.

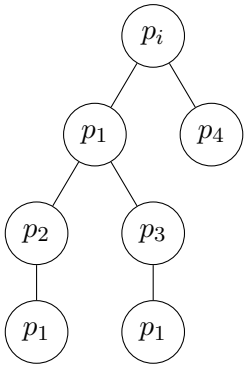
6.2.1 Funcionamiento

Como hemos descrito anteriormente, el orquestador necesita tres componentes de entrada para funcionar: sistema de reescritura, estrategia de terminación y tiempo de espera.

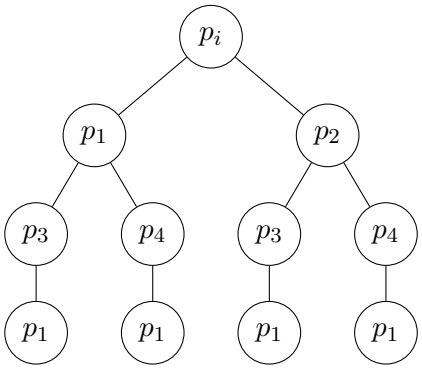
Cuando el sistema recibe estas tres entradas, internamente construye un árbol análogo a la estrategia proporcionada, en el que cada nodo representa un procesador. Como raíz del árbol se usa siempre un proceso predeterminado que debe existir en el sistema en todo momento. Realmente, este proceso no es un DP processor, pues no tiene como entrada un DP problem. La tarea de este proceso es convertir el sistema de reescritura introducido en un DP problem, que en esencia consiste en obtener sus pares de dependencia. Este procesador es el único que no es proporcionado por un usuario externo, además de ser el único que no aparece en la estrategia de terminación que introduce el usuario, pues su uso es implícito. Cada una de las ramas de dicho árbol representa una secuencia distinta, mediante la cual, tratará de verificar la terminación del sistema de reescritura.

Por ejemplo, sea $p_1, p_2, ..., p_n$ la lista de procesadores disponibles y p_i el procesador que obtiene un DP-problem a partir del sistema de reescritura:

- La estrategia $(p_1 \& (p_2 \mid p_3) \& p_1) \mid p_4$ generaría el siguiente árbol:



- Otro ejemplo sería la estrategia $(p_1 \mid p_2)\&(p_3 \mid p_4)\&p_1$, que generaría el siguiente árbol:



Es importante no confundir este árbol de orquestación con el árbol de terminación del marco de pares de dependencias. En el que utilizamos en esta ocasión, cada nodo representa un procesador, mientras que en el árbol de terminación del marco de pares de dependencia, cada nodo representa un DP-problem que se transforma mediante el procesador indicado cada arco. En el árbol de orquestación, obtener una respuesta concluyente (*YES/NO*) en una de las ramas es suficiente para demostrar la terminación del SRT suministrado.

Observamos entonces, que cada rama del árbol de orquestación se corresponde con una posible demostración, y con ello, con un árbol de terminación. Esta característica se describirá con más detalle en la siguiente sección.

Una vez generado el árbol con la estructura de procesadores análoga a la estrategia, se ejecuta recursivamente desde la raíz, utilizando como entrada el sistema de reescritura a analizar. Cada nodo contiene la información necesaria para ejecutar el procesador que representa (nombre, ubicación en disco del ejecutable, autor...). En cada ejecución de cada nodo, se leen las entradas que necesita en un directorio específico y deposita el resultado parcial en el mismo directorio específico, donde el siguiente nodo podrá encontrar dicho resultado. Todos los nombres de fichero se generan mediante un sistema de identificadores únicos que asegura que no va a haber colisiones entre los nombres.

A la hora de recorrer el árbol, los procesadores se aplican sobre todos los DP-problems que se han obtenido en la ejecución del nodo anterior. Es así como, de forma implícita, simulamos el árbol de terminación del marco de pares de dependencia.

En cada nodo se comprueba si ya disponemos de un resultado concluyente:

- Un nodo del árbol de orquestación devuelve YES si los procesadores siguientes aplicados a todos los DP-problems generados por el nodo anterior devuelven YES.
- Un nodo del árbol de orquestación devuelve NO si los procesadores siguientes aplicados a alguno de los DP-problems generados por el nodo anterior devuelve NO.
- Si un nodo no llega a un resultado concluyente por si mismo, la ejecución continúa y queda a la espera de que las ejecuciones posteriores le propaguen su resultado, o a que se llegue a las hojas del árbol.

De esta manera cada rama del árbol de orquestación verifica las propiedades que ha de cumplir su árbol de terminación análogo. Si un nodo obtiene YES o NO, es porque alguna rama del árbol de orquestación ha demostrado la terminación o la no terminación, abortando así la ejecución en el resto de ramas.

Si se completa la ejecución de todos los nodos del árbol de orquestación y no se ha obtenido en YES ni NO en ninguna rama, se devuelve MAYBE, indicando que la estrategia de terminación está incompleta.

Una vez terminado el proceso, el sistema devuelve los valores de salida descritos anteriormente: el resultado, que es una de las posibles respuestas (YES, NO, MAYBE) y una cadena de caracteres que contiene la demostración obtenida.

6.2.2 Árboles de orquestación y terminación

Como se ha indicado en la sección anterior, cada rama del árbol de orquestación se corresponde con una posible demostración, y por tanto, con un posible árbol de terminación. Por lo tanto, en un árbol de orquestación puede haber tantas demostraciones distintas como ramas tenga. Por ejemplo, considerando el DP problem con pares de dependencia $\mathcal{P}$:

$$F(f(x)) \rightarrow F(g((f(x))) \quad (4)$$
$$F(f(x)) \rightarrow F(x) \quad (5)$$

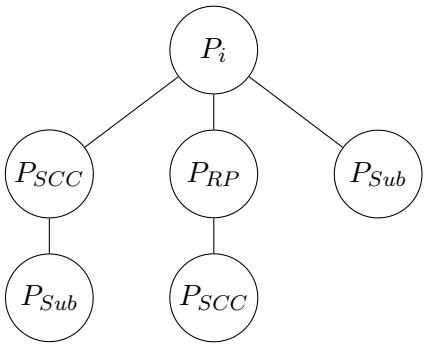
y reglas $\mathcal{R}$:

$$f(f(x)) \rightarrow f(g((f(x))) \quad (6)$$

Dada la estrategia:

$$(P_{SCC}\&P_{Sub})|(P_{RP}\&P_{SCC})|P_{Sub}$$

Obtendríamos el siguiente árbol de orquestación:



Donde, como se ha mencionado, cada rama se corresponde con una demostración distinta:

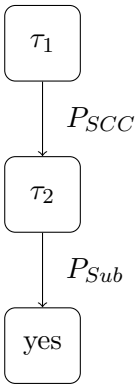
- P_{SCC}, P_{Sub}: al aplicar $P_{SCC}(\mathcal{P}, \mathcal{R})$ conseguimos eliminar el primer par de $\mathcal{P}$. El Dp problem resultante tendría los pares $\mathcal{P}$:

$$F(f(x)) \rightarrow F(x) \quad (7)$$

y reglas $\mathcal{R}$:

$$f(f(x)) \quad \rightarrow \quad f(g((f(x))) \tag{8}$$

Después de esto, si aplicamos P_{Sub} con la proyección $\pi(F) = 1$, eliminamos el único par que queda. Como un DP problem $(\mathcal{P}, \mathcal{R})$ donde $\mathcal{P} = \varnothing$ es trivialmente finito, obtendríamos el siguiente árbol de terminación:



Con lo que se demuestra que el SRT es terminante.

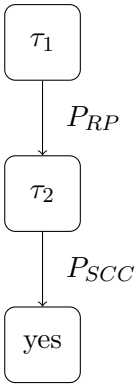
- P_{RP}, P_{SCC}: al aplicar $P_{RP}(\mathcal{P}, \mathcal{R})$ con la relación de embebimiento (o *embedding*) y el filtrado de argumentos $\pi(g) = 1$ conseguimos eliminar el segundo par. El DP problem resultante sería:

$$F(f(x)) \quad \rightarrow \quad F(g((f(x))) \tag{9}$$

y reglas $\mathcal{R}$:

$$f(f(x)) \quad \rightarrow \quad f(g((f(x))) \tag{10}$$

Posteriormente, al aplicar el procesador P_{SCC}, al no haber componentes fuertemente conectados, se elimina el primer par, quedando así un DP problem con un conjunto vacío de pares, que como hemos visto antes es trivialmente finito. El árbol de demostración sería:



Por lo que sería otra forma de demostrar que el SRT es terminante.

- P_{Sub}: No es posible reducir el DP problem mediante este procesador, pues no existe ninguna proyección válida que consiga eliminar ningún par de $\mathcal{P}$.

Por lo tanto, de las tres ramas del árbol de orquestación, dos nos han permitido demostrar la terminación de dos formas distintas, mientras que con la tercera no ha sido posible. Que esta última rama no haya tenido éxito (es decir, devolver *YES* o *NO*) no significa que el SRT no sea terminante, ya que con que una rama haya tenido éxito es suficiente.

Una opción que podría ser interesante, sobre todo en estudios sobre el comportamiento de los procesadores, consistiría en usar esta herramienta para obtener todas las posibles demostraciones de la terminación de un SRT dado. Más adelante se explica esto en detalle como posible desarrollo que realizar como trabajo futuro.

6.2.3 Concurrencia y asincronía

Dado que trabajamos con árboles (terminación, procesadores...) en los que las ramas son independientes, una optimización natural es ejecutar todo lo que sea posible en paralelo, aprovechando potencialmente la mayor cantidad de núcleos posibles de la máquina en la que se ejecute la herramienta. Éste también es uno de los motivos por el que se ha escogido C# como lenguaje de desarrollo: por las facilidades que ofrece a la hora de escribir código paralelo.

Para describir cómo se comporta el programa en la máquina, primero hay que definir algunos conceptos:

- Concurrencia: es la habilidad de un programa para descomponer su funcionamiento en distintos procesos y ejecutarlos en distintas unidades de trabajo de la máquina. En todo momento se debe conservar el determinismo del sistema, lo que supone que dichos procesos deben coordinarse y/o sincronizarse para asegurar la consistencia de la ejecución.

- Paralelismo: es el fenómeno que se da cuando un programa concurrente se ejecuta en una máquina preparada para ello (por ejemplo, con varias unidades de trabajo), es decir, cuando realmente la carga se trabajo se reparte entre dos o más unidades de trabajo. Si un sistema solo cuenta con una unidad de trabajo, el programa concurrente puede ejecutarse igualmente, pero en este caso no estamos hablando de paralelismo, pues toda la carga está siendo llevada a cabo por una sola unidad.

- Asincronía: el esquema concurrente planteado anteriormente es, en cierta forma, subóptimo: si un proceso a depende de otro proceso b para ejecutarse, el procesador físico encargado del proceso a quedará bloqueado, esperando a que el proceso b termine para llevar a cabo a. En este escenario, decimos que b es un proceso bloqueante. Este fenómeno puede darse tanto en sistemas concurrentes como no concurrentes.

  Para hacer un mejor uso de la máquina y obtener mayor rendimiento, debemos permitir que las unidades de trabajo encargadas de procesos que están en espera, sean capaces de aprovechar dicho tiempo de espera, realizando otras tareas mientras tanto. En esta situación, decimos que estamos ante procesos no bloqueantes, o **asíncronos**.

El esquema asíncrono sencillo de usar en C# y además también aplica concurrencia [19]. Es decir, los procesos asíncronos que generamos con C#, además de ser no bloqueantes, se distribuyen entre los procesadores físicos con los que cuente la máquina en la que se ejecuta el programa, abstrayendo las labores de sincronización de cara al programador.

Aunque el orquestador implementado aprovecha la asincronía de C# siempre que es posible, hay dos puntos donde realmente destaca este esquema:

- Cada nodo del árbol de orquestación ejecuta sus nodos hijo en paralelo haciendo uso de la asincronía.

- Como hemos visto, a cada nodo del árbol de ejecución se le puede aplicar n DP-problems. Cuando esto ocurre, cada DP-problem se procesa en paralelo, también haciendo uso de la asincronía.

7 Interfaz web

En la presente sección se describen los detalles de implementación de la aplicación web que hace uso del orquestador anteriormente descrito. Esta aplicación es la interfaz mediante la cual el orquestador se pone a disposición de los usuarios, que también podrá ser utilizada por parte de los colaboradores para aportar procesadores, y por el administrador, para verificar dichos procesadores.

Por cuestiones de simplicidad e interoperabilidad con el orquestador, se ha decidido escribir la interfaz web en el mismo lenguaje que el orquestador: C#. En particular se ha utilizado el marco de trabajo *.NET Core 6.0 - Razor Pages* [23], que proporciona una un flujo de trabajo sencillo para implementar aplicaciones web sencillas.

7.1 Casos de uso

En esta sección se describen los distintos casos de uso de la interfaz web, es decir, los distintos procesos que los usuarios pueden llevar a cabo en dicha aplicación:

7.1.1 Uso de TOrch

El caso de uso más básico consiste en la utilización de la aplicación para verificar la terminación de un sistema de reescritura determinado. Es una funcionalidad que está abierta a cualquier usuario que pueda acceder a la plataforma.

7.1.2 Incorporación de un nuevo procesador

Este es el proceso que siguen los colaboradores para aportar sus propios procesadores. Esta tarea se lleva a cabo cumplimentando un formulario específico para ello, que consta de los siguientes campos:

- Nombre del autor: nombre completo de la persona o grupo de investigación que ha desarrollado el procesador.

- Correo electrónico de contacto: proporciona al administrador una vía para ponerse en contacto con la entidad que ha desarrollado el procesador, ya sea para comentar posibles errores o para confirmar la aceptación del mismo.
- Iniciales de la organización: iniciales de la entidad (Persona, departamento, organización...) que ha desarrollado el procesador. Estas iniciales se mostrarán junto al nombre del procesador para identificarlo una vez vaya a usarse.
- Archivo que contiene el procesador: debe ser un archivo ejecutable en Linux. El nombre del procesador se infiere a partir del nombre del archivo. Las características funcionales que debe cumplir dicho procesador se describen en la sección 5.1.

Una vez se envíe el formulario, se le indica al usuario que el procesador debe ser verificado antes de estar disponible para su uso en TOrch.

7.1.3 Verificación de un nuevo procesador

Este proceso, que solo puede ser llevado a cabo por un administrador (identificado mediante una clave), consiste en la verificación manual de que el procesador aportado es seguro, procede de una fuente fiable y cumple las características para interoperar con el resto de procesadores de la plataforma (descritas en la sección 5.1). Para ello, se muestran en una tabla los distintos procesadores pendientes de verificación, en la que se muestran los datos introducidos en el formulario descrito anteriormente, junto con un enlace de descarga del procesador a evaluar.

7.2 Diseño

En esta sección se describen los criterios de diseño utilizados para construir la aplicación web. Es un apartado que se ha cuidado bastante para mantener la simplicidad del código y favorecer la escalabilidad de cara a nuevas funcionalidades o cambios.

7.2.1 Patrones y principios

En ingeniería del software, uno de los patrones arquitectónicos más básicos es el diseño por capas, que consiste en agrupar el código en distintos niveles.

Este patrón se combina con otro, el patrón DAO (Data Access Object) [25], que consiste en aislar las partes del programa que hacen uso de la persistencia. Esto permite que la lógica de negocio de la aplicación no dependa de la infrastructura que se usa para persistir los datos.

Haciendo uso de ambos patrones, tenemos el código distribuido en los siguientes proyectos:

- TOrch.WebApp: es el punto de entrada de la aplicación. Contiene las vistas y los controladores a los que llaman dichas vistas. También, por simplicidad, contiene los servicios que orquestan los distintos casos de uso de la plataforma.

- TOrch.WebApp.Domain: es el punto que debe tener una baja acoplación con la tecnología, en el que se deben habitar los modelos que se utilicen y la lógica de negocio de la aplicación web. Dichos modelos y dicha lógica no son más que una versión extendida del orquestador, ya que este, en su implementación pura, no cuenta con funcionalidades que sí se incluyen en la aplicación web. Es por eso que estos conceptos (lógica de negocio, modelos) se encuentran distribuidos entre este y el siguiente proyecto de la lista.

- TOrch.Core: contiene la implementación mínima del orquestador. Se ubica en una librería de clases [18] con bajo acoplamiento tecnológico. Es decir, no depende de tecnologías volátiles como pueden ser lo marcos de trabajo o las infraestructuras de base de datos, depende únicamente del lenguaje en el que está escrito. Es el proyecto que se utiliza por debajo de cualquier variante que pueda existir de la aplicación (aplicación web, servicio web, aplicación de escritorio...)

- TOrch.WebApp.Persistence: proyecto dedicado a manejar el acceso a la persistencia, cumpliendo de esta forma el anteriormente mencionado patrón DAO. En este caso, la persistencia consiste en una base de datos PostgreSQL [6] que contiene una sola tabla, en la cual se indican los metadatos de los procesadores disponibles. Este punto se desarrollará más en secciones posteriores. Como tecnología usada en este proyecto de persistencia, se hace uso de un ORM (Object Relational Mapping) [15], que es un tipo de herramienta que sirve para abstraer al proyecto del lenguaje de la base de datos (en este caso, SQL). Utiliza los modelos definidos para crear una base de datos análoga, y se establece un mapeo entre estos dos componentes, de forma que cualquier cambio en el

modelo de la aplicación, puede reflejarse fácilmente en nuestra base de datos. El ORM pone a disposición del programador una serie de operaciones que se utilizan para manipular los datos almacenados en la persistencia. Este proyecto utiliza, en particular, el ORM más popular del entorno de desarrollo de C#: *Entity Framework Core* [21].

Más allá del diseño por capas implementado, se ha tenido en mente el principio de programación orientada a objetos conocido como inversión de dependencias [20]. Este concepto se explica en mayor detalle en la siguiente sección.

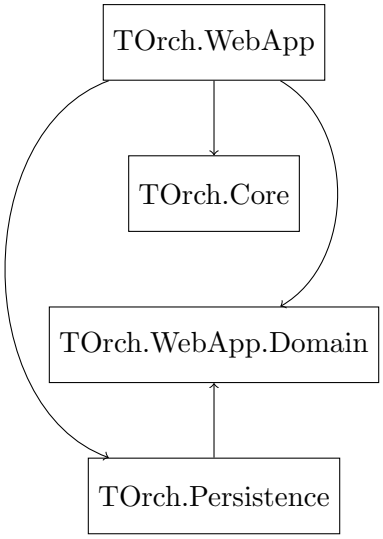
7.2.2 Gestión de las dependencias

Según el principio de inversión de dependencias, los componentes deben estar definidos por interfaces, que sirven como contratos de cara a los componentes que deban usarlos. Así se consigue que el componente en cuestión no dependa de la implementación de los componentes que utiliza, sino que simplemente se ciñe al contrato definido por la interfaz. Esto es de gran utilidad, ya que interfaz e implementación pueden estar ubicados en puntos completamente distintos de la aplicación, proporcionando al programador la capacidad de gestionar qué dependencias hay y en qué sentido. Estos contratos normalmente se ubican en el proyecto de Dominio, junto a la lógica de negocio (la cual podría utilizar estos contratos) y a los modelos. De esta forma conseguimos que las dependencias ocurran siempre hacia esta capa, evitando así dependencias tecnológicas más fuertes y volátiles.

El principio descrito produce una gran cantidad de interfaces e implementaciones. A continuación se describen los puntos necesarios para que los componentes puedan utilizar correctamente los servicios.

- Si un componente a depende de un componente b, el contrato (interfaz) de b debe ser inyectado como parámetro en el método constructor del componente a.
- El programa de entrada, que es el que desencadena la ejecución (en el caso de TOrch, TOrch.WebApp) debe registrar las dependencias, proceso que consiste en, dentro de su configuración de arranque, indicar qué implementación corresponde a qué interfaz.

Es de esta forma en la que se establecen las dependencias entre capas, o en este caso, entre proyectos. El grafo de dependencias de la aplicación web tiene el siguiente aspecto:



Donde cada nodo:

- TOrch.WebApp depnde de TOrch.Domain y de TOrch.Persistence porque, como hemos indicado anteriormente, es el encargado de relacionar interfaces con implementaciones. También depende de TOrch.Core, ya que los servicios que constituyen los casos de uso y que, por tanto, dependen del orquestador, también se ubican en este nodo.
- TOrch.WebApp.Persistence depende únicamente de TOrch.Domain. Esto sucede porque es dicho proyecto el encargado de definir las interfaces que implementan los componentes definidos en el nodo.
- TOrch.WebApp.Domain y TOrch.Core no dependen de nadie porque, como hemos descrito anteriormente, es buena práctica que la lógica de negocio no tenga dependencias que puedan compremeterla.

7.2.3 Modelo de datos

El modelo de datos en esta aplicación web es muy sencillo. Aún así, es importante que esté bien definido, pues es algo básico para el correcto funcionamiento de la plataforma. En este caso consiste en una sola entidad que almacena los metadatos de los procesadores disponibles. Dicha entidad contiene los siguientes campos:

- Identificador: es un valor numérico entero y único, que en términos de bases de datos, será la clave primaria de la tabla que se corresponde con la entidad.

- Nombre: consiste en una cadena de caracteres que se infiere de forma automática a partir del nombre del ejecutable subido.
- Ruta: indica la ubicación física del ejecutable, dato necesario para que el orquestador pueda ejecutar el procesador correctamente.
- Nombre del autor: hace referencia a la persona o grupo de investigación que ha implementado y subido el procesador.
- Correo electrónico del autor: tanto si el procesador termina siendo aceptado como si no, el correo electrónico proporciona al administrador una vía de comunicación con el autor del archivo ejecutable.
- Iniciales de la organización: se utiliza para una mayor precisión a la hora de identificar el origen del procesador. También será necesario para construir el siguiente elemento.
- Nombre único: utiliza las iniciales de la organización, el nombre del archivo y un identificador numérico incremental para crear un nombre único para el procesador. Este nombre único será que el que se le muestre al usuario en la construcción de la estrategia de terminación.
- Fecha de verificación: este atributo permanecerá vacío en caso de que el procesador no esté verificado y, en caso contrario, contendrá la fecha y hora en la que sucedió dicha verificación.
- Clave de verificación: este atributo indica la clave de administrador que se utilizó para verificar el presente procesador. Puede ser útil en caso de que se use una clave de administrador por usuario, ya que se podría identificar al administrador que verificó cada procesador.
- Fecha de creación: indica la fecha y hora en la que se dio la subida del procesador ejecutable.

7.3 Configuración

En el contexto de las aplicaciones web es frecuente encontrarnos con una serie de parámetros de configuración, que a menudo dependen del entorno de ejecución. Un ejemplo de esto, en el orquestador, es el directorio en el que se van a almacenar los procesadores. Es muy probable que no queramos utilizar la misma ruta para todos los entornos (local, producción...). Lo mismo se aplica a otros datos como las credenciales de la base de datos o la

clave que identifica al administrador, donde podría ser deseable que en producción estos valores sean algo más complejos para maximizar la seguridad.

Tener estos valores de configuración escritos directamente en el código se considera una mala práctica por varios motivos:

- Un componente que utiliza uno de estos valores tendría lógica referida al entorno en el que se ejecuta. Es deseable que los componentes sean agnósticos al entorno de ejecución en el que habitan, pues incrementa la claridad y la simplicidad del código, haciéndolo menos propenso a posibles fallos.
- Estos valores pueden cambiar con el tiempo, bien porque el programa se va a ejecutar en un nuevo entorno, o bien porque alguno de estos valores realmente ha cambiado para un entorno dado. Si dichos valores están escritos en el código, al modificarlos tendríamos que compilar de nuevo toda la aplicación, cosa que, en según qué contextos, sería poco deseable.

En su lugar, estos valores de configuración se encuentran, de forma centralizada, en un archivo de configuración [2] en formato JSON (JavaScript Object Notation), existiendo así un archivo de configuración por cada entorno de ejecución. Después, en el arranque de la aplicación, se determina qué archivo de configuración utiliza basándose en el entorno en el que está. El entorno de ejecución se puede especificar mediante las variables de entorno del sistema operativo de la máquina que ejecuta la aplicación.

8 Resultados

En esta sección se describen los resultados obtenidos en la primera versión de la herramienta.

Evaluar esta plataforma de la misma forma que las demás herramientas de terminación no es adecuado, ya que su rendimiento depende en gran medida de cómo rindan los procesadores subidos. La mejor forma de explotar esta herramienta es ver cómo los distintos DP problems son resueltos de una forma u otra mediante el uso de distintas estrategias.

Por ello, planteamos una serie de SRTs que hemos extraído de la base de datos conocida como COPS (Confluence Problems Database). En esta base de datos, para cada SRT, podemos observar en sus etiquetas si es

terminante o no, con lo que sabremos de antemano el resultado que debe obtener la herramienta. Los procesadores disponibles con los que se han realizado las pruebas son los siguientes: $P_{SCC}, P_{Inf}, P_{Sub}$ y P_{RP}. Vamos a escoger un total de seis problemas y a comentar el comportamiento de TOrch en cada caso. Intentaremos también ir construyendo una estrategia que cubra incrementalmente distintos casos, llegando así a una estrategia que sirva para demostrar la terminación de todos los SRTs escogidos.

1. (SRT *1.trs* en COPS) Empezamos por un SRT no terminante. Inicialmente, el único procesador del que disponemos que demuestra la no terminación de un DP problem es P_{Inf}. Dado el SRT:

```
(VAR x y)
(RULES
  f(x,y) -> x
  f(x,y) -> f(x,g(y))
  g(x) -> h(x)
  F(g(x),x) -> F(x,g(x))
  F(h(x),x) -> F(x,h(x))
)
```

   y la estrategia:

   $$@P_{Inf}$$

   Obtenemos ***NO*** como respuesta, ya que P_{Inf} ha tenido éxito. Si, en cambio, usamos cualquier otro procesador obtenemos, como es lógico, ***MAYBE***, pues ningún otro procesador de los iniciales puede verificar la no terminación de un DP problem. Llegados a este punto, parece lógico incluir siempre este procesador en nuestras estrategias combinado con el operador |.

2. (SRT *1650.trs* en COPS) Continuamos con un SRT terminante:

```
(VAR x)
(RULES
 f(f(x)) -> f(g(f(x)))
 f(x) -> x
)
```

y la estrategia:

$$@P_{Inf} \mid @P_{SCC}$$

Analizando únicamente al estrategia, observamos que se obtendrá ***NO*** si el SRT fuera no terminante, ***YES*** si el DP problem asociado no tuviera componentes fuertemente conexos (y por lo tanto, el DP problem es finito y el SRT terminante) o ***MAYBE*** en caso de que se alcance el tiempo máximo de ejecución, o bien si el DP problem asociado si tuviera componentes fuertemente conexos.

El resultado es ***YES***. En la demostración podemos verificar que, efectivamente, no tiene componentes fuertemente conexos.

3. (SRT *1155.trs* en COPS) Dado el SRT:

```
(VAR x y)
(RULES
  plus(x,0) -> x
  plus(0,1) -> 1
  plus(x,plus(y,1)) -> plus(plus(x,y),1)
  times(x,0) -> 0
  times(x,1) -> x
  times(x,plus(y,1)) -> plus(times(x,y),x)
  m(0) -> 0
  plus(m(1),1) -> 0
  plus(m(plus(x,1)),1) -> m(x)
  m(m(x)) -> x
  plus(x,m(y)) -> m(plus(m(x),y))
  times(x,m(y)) -> m(times(x,y))
)
```

y la estrategia:

$$@P_{Inf} \mid (@P_{SCC} \ \& \ @P_{Sub} \ \& \ @P_{SCC})$$

En este caso, necesitamos el procesador de subtérminos P_{Sub} para demostrar la taerminación, pues el DP problem asociado a este SRT sí cuenta con componentes fuertemente conexos. El resultado obtenido es ***YES***.

4. (SRT *1279.trs* en COPS) Dado el SRT:

```
(VAR x)
(RULES
 f(g(x)) -> g(f(f(x)))
 g(x) -> x
)
```

y la estrategia:

$$@P_{Inf} \mid (@P_{SCC} \;\&\; @P_{RP} \;\&\; @P_{Sub} \;\&\; @P_{SCC})$$

Obtenemos ***YES*** como respuesta. En esta estrategia ya utilizamos todos los procesadores disponibles de la plataforma una o más veces.

5. (SRT *1155.trs* en COPS) Dado el SRT:

```
(VAR x)
(RULES
  a(a(x)) -> a(b(a(x)))
  b(a(b(x))) -> a(c(a(x)))
)
```

y la estrategia:

$$@P_{Inf} \mid (@P_{SCC} \;\&\; @P_{RP} \;\&\; @P_{RP} \;\&\; @P_{Sub} \;\&\; @P_{SCC})$$

Obtenemos ***YES*** como respuesta. Observamos en la demostración que se ha obtenido un resultado concluyente gracias a que P_{RP} ha sido capaz de ejecutarse en varias configuraciones distintas. Esto pone en manifiesto hasta qué punto TOrch depende del funcionamiento de los procesadores para poder llevar a cabo una demostración. Si P_{RP} solo hubiera admitido una configuración, la terminación no podría haberse verificado. Otra alternativa habría consistido en tener dos variantes de P_{RP} disponibles en TOrch, donde cada una supondría una configuración distinta. La terminación podría haberse verificado mediante el uso de ambas variantes.

6. (SRT *984.trs* en COPS) Dado el SRT:

```
(VAR x)
(RULES
  a(s(x)) -> s(a(x))
  b(a(b(s(x)))) -> a(b(s(a(x))))
  b(a(b(b(x)))) -> a(b(a(b(x))))
  a(b(a(a(x)))) -> b(a(b(a(x))))
)
```

y la estrategia:

$$@P_{Inf} \mid (@P_{SCC} \;\&\; @P_{RP} \;\&\; @P_{RP} \;\&\; @P_{RP} \;\&\; @P_{RP} \;\&\; @P_{RP} \;\&\; @P_{RP} \;\&\; @P_{RP} \;\&\; @P_{Sub} \;\&\; @P_{SCC})$$

Obtenemos ***YES*** como respuesta. Sin embargo, hay algo que llama la atención en la estrategia: hemos tenido que aplicar el procesador P_{RP} siete veces consecutivas. Esto supone un problema evidente en cuanto a escalabilidad, ya que, para llevar a cabo una demostración, es posible que un procesador deba aplicarse consecutivamente un número indeterminado de veces. En ese caso, para que la demostración sea correcta, el usuario debería saber cuántas veces debe aplicar cada procesador o ir incrementando el número de usos según los resultados obtenidos, algo que podría no ser factible. En una sección posterior se describe cómo esta situación podría resolverse en el futuro.

9 Trabajo relacionado

Un sistema que comparte algunas características con TOrch es la aplicación web CoCoWeb [11] que puede utilizarse aquí

`http://colo7-c703.uibk.ac.at/cocoweb/index.php`

Este sistema ha sido desarrollado a instancias de la Competición Internacional de Confluencia (CoCo[1]), organizada para animar el desarrollo de herramientas automáticas de análisis de propiedades relacionadas con la confluencia de (variantes de) los sistemas de reescritura.[2] En los últimos 10 años, coincidiendo con el inicio de la competición en 2012, se han desarrollado un buen número de herramientas de análisis de la confluencia. La plataforma

[1] `http://project-coco.uibk.ac.at/`

[2] Un sistema de reescritura $\mathcal{R}$ se denomina *confluente* si para todo término s y para todo par de términos t y t' tales que $s \to_{\mathcal{R}}^* t$ y $s \to_{\mathcal{R}}^* t'$, siempre existe un término u tal que $t \to_{\mathcal{R}}^* u$ y $t' \to_{\mathcal{R}}^* u$.

CoCoWeb permite utilizar todas ellas de forma combinada para demostrar la confluencia de un mismo sistema de reescritura. Cada herramienta se aplica de forma independiente al problema de entrada y sus resultados son presentados por la plataforma de manera que el usuario puede saber cuál ha sido la conclusión obtenida y también puede visualizar los informes producidos, en su caso. Las herramientas son invocadas por la plataforma utilizando los ficheros ejecutables registrados para participar en la competición CoCo. Es posible utilizar las distintas versiones de las herramientas enviadas cada año.

No existe nada parecido en la comunidad de desarrollo de herramientas de terminación, entre las cuales hay más de 20 herramientas, ver, por ejemplo,

`https://www.jaist.ac.jp/~hirokawa/tool/?tag=termination`

También existen otros trabajos relacionados donde se presentan lenguajes de estrategias que comparten, en cierta medida, la misma filosofía que el desarrollado en este trabajo. Estos lenguajes podrían ser tomados como referencia para extender el nuestro. Por ejemplo, Stratego [3] (para transformación de programas) o el el lenguaje de estrategias de Maude, para dirigir la reducción de expresiones [24].

En este trabajo se ha dado un paso más allá del modelo seguido en CoCoWeb (integración de herramientas de confluencia) aprovechando que las herramientas modernas para la demostración de la terminación de sistemas de reescritura están basadas en el DP Framework. Por ese motivo, es posible ir más allá del uso de múltiples herramientas en una única plataforma para lograr definir una única herramienta sobre la base de la integración y orquestración de los procesadores utilizados en las distintas herramientas de terminación existentes. Este es un aspecto que, hasta donde nos ha sido dado investigar, no ha sido explorado con anterioridad a nuestro trabajo.

10 Trabajo futuro

TOrch es una prueba de concepto que supone un paradigma alternativo a las herramientas de terminación tradicionales. Este concepto, como hasta el momento no se había planteado, todavía podría tener mucho desarrollo por delante.

Inicialmente, habría que presentar la herramienta a los grupos de investigación interesados en esta temática. De esta forma, podríamos tener

feedback sobre la misma y comprobar hasta qué punto esta herramienta resulta interesante a la comunidad.

A nivel técnico, hay una serie de desarrollos que mejorarían notablemente la herramienta. La implementación de estas características también podría repercutir beneficiosamente en la acogida de la herramienta:

- Permitir que un procesador determinado se ejecute varias veces seguidas, hasta alcanzar un problema "normalizado" bajo ese procesador. Como hemos visto en la sección de resultados, hay ocasiones en las que tendremos que aplicar varias veces consecutivas el mismo procesador para que el DP problem siga simplificándose. En la versión actual, los procesadores se ejecutan tantas veces como indique la estrategia, pero no hay ningún mecanismo para indicar que un procesador debe ser ejecutado hasta alcanzar una forma normalizada. Esto puede tener cierta relevancia, ya que, a priori, el usuario no sabe cuántas veces debe aplicar un procesador para llegar a dicha forma normalizada que le permitiría seguir avanzando con la demostración.

- TOrch podría extenderse fácilmente a la demostración de propiedades de terminación de otras variantes de los sistemas de reescritura, como la reescritura sensible al contexto [16], la reescritura condicional [12], la reescritura ecuacional [13], etc., para todos los cuales existen métodos de demostración basados en extensiones del Marco de Pares de Dependencia [1, 9, 17, 26].

- TOrch también podría extenderse fácilmente a otros marcos recientemente introducidos para la demostración de otras propiedades de los sistemas de reescritura que también están basados en las ideas del DP Framework.

  Ejemplos de esos marcos son el *Feasibility Framework* [8] y el *Confluence Framework* [10].

- Mejoras en la presentación de los resultados. Esta herramienta se basa en el DP Framework. Tal y como se ha explicado, el DP Framework presenta un árbol como prueba de terminación. Lo ideal sería que la plataforma representara gráficamente el árbol de terminación en caso de que se haya podido alcanzar un resultado concluyente.

- Construir una estrategia, dependiendo del SRT que se plantee, puede ser complejo. En muchos casos, el usuario deberá confeccionarla incrementalmente a partir de los resultados que va obteniendo en la

herramienta. Dada esta situación, sería de gran utilidad disponer de un conjunto de estrategias guardadas a disposición de los usuarios.

- Un posible uso también podría darse si se implementa, para los procesadores, datos como la frecuencia de éxito, el nivel de uso o el tiempo medio que toma su ejecución. Esto favorecería el desarrollo de estrategias más eficaces.
- Un caso de uso interesante sería obtener, mediante la estrategia suministrada, las distintas formas en las que se podría demostrar la terminación de un SRT determinado. A nivel técnico consistiría obtener todas aquellas ramas del árbol de orquestación que han obtenido *YES* como respuesta. Para ello, habría que continuar con la ejecución de la demostración aunque ya se haya demostrado la terminación del SRT. Esta opción tendría que ser activada desde la interfaz web, pues podría suponer un gran aumento en el coste computacional de la demostración, y por lo tanto, no es algo que deba hacerse por defecto.

Los puntos anteriores son solo algunos ejemplos de mejoras que podrían darse en la plataforma. También es posible que, además de estos puntos, haya sugerencias por parte de los usuarios que colaboren, no solo con el funcionamiento, sino también con al desarrollo de la herramienta.

11 Conclusión

TOrch es una plataforma que supone una visión distinta respecto a las demás herramientas de verificación de la terminación. Su concepto no se había puesto en práctica hasta este momento, por lo que esta plataforma es una prueba de concepto que se expondrá a otros grupos de investigación y posteriormente se analizará su acogida.

Un concepto colaborativo como el que subyace en TOrch podría favorecer el interés de otras entidades (personas, grupos de investigación...) en el campo, ya que una entidad podría estar interesada en desarrollar un procesador de terminación pero no una herramienta de terminación completa. También, al no depender del lenguaje de programación en el que está escrito el orquestador, los posibles colaboradores podrían escribir sus procesadores en el lenguaje que quisieran.

También, como contrapartida, se da un incremento en el coste de mantenibilidad de la herramienta, ya que depende de un administrador que se en-

cargue de evaluar los procesadores. Estos procesadores suponen una fuente potencial de errores, los cuales podrían pasar inadvertidos. En un caso como este, el administrador tendría que contactar con el autor del procesador y retirarlo de la plataforma, algo que no es común en herramientas similares.

Probar la herramienta y verificar que todo funciona correctamente tampoco es tarea fácil: habría que verificar que, dado un SRT, se devuelve el resultado esperado en muchas estrategias distintas. Esto es considerablemente más costoso que probar una herramienta de terminación tradicional con, simplemente, un conjunto de SRTs.

Desde el punto de vista del usuario, también hay un incremento en el trabajo que debe realizar en comparación con las herramientas de terminación tradicionales, pues debe construir una estrategia válida. Además, la estrategia usada, en muchos casos, se tendrá que ir confeccionando incrementalmente a partir de los resultados que vayan obteniendo de la herramienta, cosa que puede suponer una barrera para muchos usuarios a la hora de utilizar la plataforma.

Bibliografía

[1] Beatriz Alarcón, Salvador Lucas, and José Meseguer. "A Dependency Pair Framework for A OR C-Termination". In: *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*. Ed. by Peter Csaba Ölveczky. Vol. 6381. Lecture Notes in Computer Science. 2010, pp. 35–51. DOI: 10.1007/978-3-642-16310-4_4.

[2] Rick Anderson and Kirk Larkin. *Configuration in ASP.NET Core*. updated on 2022-07-22. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-6.0.

[3] Martin Bravenboer et al. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Sci. Comput. Program.* 72.1-2 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003. URL: https://doi.org/10.1016/j.scico.2007.11.003.

[4] Jürgen Giesl et al. "Analyzing Program Termination and Complexity Automatically with AProVE". In: *J. Autom. Reasoning* 58.1 (2017), pp. 3–31. DOI: 10.1007/s10817-016-9388-y.

[5] Jürgen Giesl et al. "Mechanizing and Improving Dependency Pairs". In: *J. Autom. Reasoning* 37.3 (2006), pp. 155–203. DOI: 10.1007/s10817-006-9057-7.

[6] PostgreSQL Global Development Group. *PostgreSQL Docs*. visited on 2022-08-29. URL: https://www.postgresql.org/docs/current/index.html.

[7] Raúl Gutiérrez and Salvador Lucas. "MU-TERM: Verify Termination Properties Automatically (System Description)". In: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12167. Lecture Notes in Computer Science. Springer, 2020, pp. 436–447. DOI: 10.1007/978-3-030-51054-1_28.

[8] Raúl Gutiérrez and Salvador Lucas. "Automatically Proving and Disproving Feasibility Conditions". In: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*. Ed. by Nicolas Peltier and Viorica Sofronie-

Stokkermans. Vol. 12167. Lecture Notes in Computer Science. Springer, 2020, pp. 416–435. DOI: 10.1007/978-3-030-51054-1_27.

[9] Raúl Gutiérrez and Salvador Lucas. "Proving Termination in the Context-Sensitive Dependency Pair Framework". In: *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*. Ed. by Peter Csaba Ölveczky. Vol. 6381. Lecture Notes in Computer Science. Springer, 2010, pp. 18–34. DOI: 10.1007/978-3-642-16310-4_3.

[10] Raúl Gutiérrez, Miguel Vítores, and Salvador Lucas. "Confluence Framework: Proving Confluence with CONFident". In: *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2022, Tbilisi, Georgia, September 9-11, 2014. Revised Selected Papers*. Ed. by Alicia Villanueva. Vol. 13474. Lecture Notes in Computer Science. Springer, 2022, pp. 24–43. DOI: 10.1007/978-3-031-16767-6_2.

[11] Nao Hirokawa, Julian Nagele, and Aart Middeldorp. "Cops and CoCoWeb: Infrastructure for Confluence Tools". In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*. Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Vol. 10900. Lecture Notes in Computer Science. Springer, 2018, pp. 346–353. DOI: 10.1007/978-3-319-94205-6_23.

[12] Stéphane Kaplan. "Conditional Rewrite Rules". In: *Theor. Comput. Sci.* 33 (1984), pp. 175–193. DOI: 10.1016/0304-3975(84)90087-2.

[13] Claude Kirchner and Hélène Kirchner. "Equational Logic and Rewriting". In: *Computational Logic*. Ed. by Jörg H. Siekmann. Vol. 9. Handbook of the History of Logic. Elsevier, 2014, pp. 255–282. ISBN: 978-0-444-51624-4. DOI: 10.1016/B978-0-444-51624-4.50006-X.

[14] M. Korp et al. "Tyrolean Termination Tool 2". In: *Proc. of the 20th International Conference on Rewriting Techniques and Applications, RTA'09*. Ed. by R. Treinen. Vol. 5595. LNCS. Springer, 2009, pp. 295–304. ISBN: 978-3-642-02347-7.

[15] Mia Liang. *Understanding Object-Relational Mapping: Pros, Cons, and Types*. updated on 2021-03-11. URL: https://www.altexsoft.com/blog/object-relational-mapping/.

[16] Salvador Lucas. "Context-sensitive Rewriting". In: *ACM Comput. Surv.* 53.4 (2020), 78:1–78:36. DOI: 10.1145/3397677.

[17] Salvador Lucas, José Meseguer, and Raúl Gutiérrez. "The 2D Dependency Pair Framework for conditional rewrite systems. Part I: Definition and basic processors". In: *J. Comput. Syst. Sci.* 96 (2018), pp. 74–106. DOI: 10.1016/j.jcss.2018.04.002.

[18] Microsoft. *.NET class libraries*. updated on 2022-01-12. URL: https://docs.microsoft.com/En-us/dotnet/standard/class-libraries.

[19] Microsoft. *Asynchronous programming with async and await*. updated on 2022-08-04. URL: https://docs.microsoft.com/EN-us/dotnet/csharp/programming-guide/concepts/async/.

[20] Microsoft. *Dependency inversion*. updated on 2022-04-14. URL: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion.

[21] Microsoft. *Entity Framework Core*. updated on 2021-05-25. URL: https://docs.microsoft.com/en-us/ef/core/.

[22] Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002. ISBN: 978-0-387-95250-5.

[23] Dave Brock Rick Anderson and Kirk Larkin. *Introduction to Razor Pages in ASP.NET Core*. updated on 2022-08-08. 2022. URL: https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-6.0&tabs=visual-studio.

[24] Rubén Rubio et al. "Parameterized Strategies Specification in Maude". In: *Recent Trends in Algebraic Development Techniques - 24th IFIP WG 1.3 International Workshop, WADT 2018, Egham, UK, July 2-5, 2018, Revised Selected Papers*. Ed. by José Luiz Fiadeiro and Ionut Tutu. Vol. 11563. Lecture Notes in Computer Science. Springer, 2018, pp. 27–44. DOI: 10.1007/978-3-030-23220-7_2.

[25] Shubham. *DAO Design Pattern*. updated on 2022-08-03. URL: https://www.digitalocean.com/community/tutorials/dao-design-pattern.

[26] Akihisa Yamada et al. "AC Dependency Pairs Revisited". In: *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*. Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 8:1–8:16. DOI: 10.4230/LIPIcs.CSL.2016.8.